Automating Multi-Tenancy Performance Evaluation on Edge Compute Nodes

Joanna Georgiou, Moysis Symeonides, George Pallis, Marios D. Dikaiakos
Department of Computer Science, University of Cyprus
Email: {jgeorg02, msymeo03, gpallis, mdd}@ucy.ac.cy

***Abstract*—Edge Computing emerges as a promising alternative of Cloud Computing, with scalable compute resources and services deployed in the path between IoT devices and Cloud. Since virtualization techniques can be applied on Edge compute nodes, administrators can share their Edge infrastructures among multiple users, providing the so-called multi-tenancy. Even though multi-tenancy is unavoidable, it raises concerns about security and performance degradation due to resource contention in Edge Computing. For that, administrators need to deploy services with non-antagonizing profiles and explore workload co-location scenarios to enhance performance and energy consumption. Achieving this, however, requires extensive configuration, deployment, iterative testing, and analysis, an effort-intensive and time-consuming process. To address this challenge, we introduce an auto-benchmarking framework designed to streamline the analysis of multi-tenancy performance in Edge environments. Our framework includes a built-in monitoring stack and integrates with widely used benchmarking workloads, such as streaming analytics, database operations, machine learning applications, and component-based stress testing. We perform a case-driven analysis and provide valuable insights into the impact of multi-tenancy on Edge environments with different hardware configurations and diverse workloads. Finally, the implementation of our framework, along with the containerized workloads used for experimentation, is publicly available.**

***Index Terms*—Benchmarking, Edge Computing, Multi-Tenancy**

I. Introduction

The rapid expansion of interconnected embedded devices, collectively known as the Internet of Things (IoT), is transforming our daily lives. The number of IoT devices is expected to reach 500 billion by 2030 [1], with each device generating vast amounts of data that require analysis. However, processing data on an IoT device is often impractical due to its limited computational capacity and reliability [2]. Meanwhile, continuously offloading data to centralized cloud infrastructures introduces challenges, including bandwidth constraints [3].

To bridge this gap, Multi-access Edge Computing (MEC) has emerged as a standardized architecture that, like traditional edge computing, brings processing closer to the IoT devices to enhance efficiency and reduce latency. What distinguishes MEC, however, is its tight integration with the Radio Access Network (RAN), enabling more effective utilization of network resources and further optimizing performance. [4]. Building on top of this, the 5th/6th generations of mobile networks (5G and 6G) advance the concept of Network Slicing, enabling the creation of multiple isolated virtual networks over a shared physical infrastructure. This allows operators to support multi-tenancy by leasing tailored "slices" of their networks, similarly to how Cloud providers rent out compute resources [5].With multi-tenancy, high-speed and performance applications like autonomous driving, drone-based surveillance, agentic workflows and LLM integration can be supported more effectively, as these types of workloads often require multiple machine learning, deep learning and LLM tasks to be executed in parallel [6]–[9].

Multi-tenancy refers to a system's ability to simultaneously serve multiple tenants (users or applications), while ensuring strict isolation of their data and resources [10]. A key enabler technology for multi-tenancy on Edge nodes is a lightweight virtualization (containerization) which bundles services into artifacts and runs them as isolated processes with negligible computational overhead [11]. Generally, service consolidation reduces the operational and maintenance costs of the infrastructure, while it also decreases energy consumption [12].

Unfortunately, even in cloud infrastructures, multi-tenancy raises concerns about security and performance deterioration of co-located services. Workloads that share the same resources may suffer quality-of-service degradation due to contention for CPU, memory, and network bandwidth. The constrained and heterogeneous nature of Edge devices, along with the diverse profiles of the deployed services, worsens these challenges.

For instance, running a set of services on an edge server can deliver high performance, but executing the same services on a resource-constrained device, such as a Raspberry Pi, may significantly increase service latency. This can be problematic for applications that require real-time responsiveness while performing multiple deep learning tasks simultaneously, such as object detection and face recognition for visually impaired users [6]. To maintain efficiency, edge infrastructure administrators and cluster orchestrators must co-locate services with compatible resource profiles on the same machine. This enables them to (i) assess whether local resources can sustain the workload and (ii) analyze trade-offs associated with multi-tenancy. These considerations are central to informed orchestration decisions. Therefore, it is essential to identify *which deployment scenarios or configurations enhance the performance of new or existing services and how to achieve objectives, such as minimizing energy consumption.*

Addressing this challenge requires the repeated deployment of diverse workloads across a range of configurations, to explore the full spectrum of co-location scenarios, and the continuous collection of performance metrics and in-

frastructure utilization data, followed by rigorous analysis to identify optimal deployment strategies [13]. However, as the number of potential configurations grows exponentially with the number of target workloads [14], the process quickly becomes intractable. Even though there are many benchmarks that prototype workloads like big data streaming [15] and machine learning (ML) inference [16], very few studies [6], [17] actually examine the effects of multi-tenancy in edge environments. Moreover, although tools for automating repeatable performance evaluations are available [13], [18], they lack support for executing concurrent workloads. This paper addresses the critical gap: the absence of a low-cost, scalable and reproducible approach for evaluating performance trade-offs introduced by workload co-location in real-world edge environments. The main contributions of this paper are:

- An open-source benchmarking framework [19], that facilitates the automated deployment of benchmarking experiments and ensures reproducible performance evaluation of co-located workloads on top of real edge infrastructures. It incorporates comprehensive monitoring for both performance and infrastructure metrics, supports detailed post-experiment analysis, and addresses common challenges in performance evaluation. Our framework leverages cloud-native technologies, including continuous deployment and containerization, to streamline benchmark configuration and deployment, whilst abstracting the hardware heterogeneity across the edge continuum.
- A set of publicly accessible containerized workloads [20], integrated with our framework to evaluate the performance of edge nodes. These workloads are derived from widely adopted benchmarks [15], [16], [21] and component-based stressors, and each has been modified to support parameterization and easy configuration. As the workloads are independent of the framework, researchers can utilize, extend, or adapt them to their requirements.
- A comprehensive, use-case-driven analysis of workload co-location on various edge compute nodes (from Raspberry Pi to edge Servers). Our analysis allows users to assess: (i) ML service performance across various compute nodes, showing that larger Edge servers offer better performance at higher power costs, while smaller devices like Raspberry Pi are more energy-efficient but slower; (ii) the impact of workload co-location, revealing that memory-intensive workloads increase cold start duration of ML services by 1.5x, and CPU or disk I/O stressors reduce CPU availability, degrading performance; (iii) the effects of different co-location configurations, comparing ML inference in two scenarios: (a) processing images locally vs streaming them, with performance varying by up to 3x, and (b) using different execution backends, where ONNX (Open Neural Network Exchange) outperforms NCNN (inference framework optimized for mobile platforms) by 9x and TensorFlow by 24x in throughput.

The remainder of the paper is structured as follows: Sec.II outlines the motivating scenario; Sec.III reviews related work; Sec.IV introduces our framework; Sec.V details the workloads and metrics used for evaluating multi-tenancy on Edge devices in Sec.VI; and Sec.VII concludes the paper.

II. Motivating Example

To illustrate the applicability of our framework, consider a scenario where an edge infrastructure provider, such as a mobile operator, serves multiple clients deploying applications on a shared Edge environment. The infrastructure consists of various heterogeneous devices, from high-performance edge servers to resource-constrained micro-edge devices like single-board computers. To meet client demands, the provider enables containerized services to run in parallel across these nodes while ensuring a specified Quality of Service (QoS).

Co-locating applications, even on constrained devices, is often more practical despite potential performance degradation from resource contention, as dedicating a node to each workload is usually impractical due to power, cost, and resource limitations. [22]. Operators, researchers, and system administrators who want to study multi-tenancy effects, optimize deployments, and benchmark their infrastructure must carefully assess the impact and trade-offs of workload co-location to determine optimal deployment strategies. Factors like energy efficiency must also be considered, requiring extensive experimentation to identify configurations that minimize energy consumption while maintaining performance [23]. Hence, researchers must either develop custom workloads, an effort requiring significant time and expertise, or use existing workloads, which still involve extensive software and hardware setup, dependency management, and monitoring configurations. Then they must execute a benchmarking pipeline, which traditionally involves manually deploying various workloads, monitoring resources, and analyzing service performance. This process on its own is time-consuming and shifts the focus away from the user's objective, which is the system's performance evaluation [13]. However, in the case of assessing multi-tenancy, this task becomes even more challenging, as it requires testing all workload combinations with diverse configurations. Additionally, as the number of target workloads and configurations increases, the number of scenario combinations grows exponentially, lengthening the process even further.

To address these challenges, our open-source framework streamlines the benchmarking of co-located services, enabling users to assess infrastructure performance, fine-tune service parameters, and analyze co-location trade-offs, such as identifying antagonistic workload metrics, without the overhead of managing complex setup and configuration tasks. The methodology automates (i) the installation and configuration of essential software components, including containers and monitoring agents on edge nodes, and (ii) the deployment of both isolated and co-located, parameterizable services (e.g., database workloads, ML inference) across diverse scenarios.

III. Related Work

Analyzing interference between applications in data centers is a well-known issue, requiring workload analysis through in-

frastructure metrics. Towards that, Ilager et al. [24] conducted a data-driven study on cloud datacenter workloads, energy, and thermal characteristics using nine months of machine-level metrics, developing predictive models for efficiency planning. Similarly, [25] analyzed workload interference in cloud providers (Google, Alibaba), quantifying co-location effects at micro-architecture and app-level. In [26], virtualization techniques (containers, VMs) were assessed, showing higher interference in containers, while an experimental study in [27] examined performance interference in containerized microservices, measuring the effects of co-located services. However, *these studies focus solely on workload co-location in cloud environments, omitting potential issues in MEC deployments.*

Recognizing the unique constraints of edge environments, edge co-location research has primarily focused on the deployment of ML inference workloads [6], [7]. For example, Edge-MultiAI enhances the multitenancy of ML inference while meeting latency and accuracy goals [6], while [7] introduces concurrent Deep Learning model inference with dynamic placement, improving throughput on Jetson TX2 using various ML frameworks. However, beyond ML-specific workloads, only a few systems consider general-purpose co-location on the edge, mostly from a scheduling perspective. PolarisProfiler [28] optimizes resource management through metadata-based profiling, while Edge Federation [29] introduces a dependency-aware scheduler for federated containerized workloads. Mathematical approaches like Co-Approximator [14] estimate co-location performance but lack automation and scalability considerations. Similarly, efforts to optimize network co-location (cellular base stations and MECs), such as ColoMEC [30], focus on orchestration but do not address general-purpose workload evaluation. Consequently, *the gap in automating the performance evaluation of co-located workloads on edge infrastructures still remains*.

To address the evaluation of edge deployments, experimentation tools are widely used by Edge Computing engineers to compare frameworks, applications, and infrastructures. Fog and Edge emulators [2], [31]–[34] facilitate topology emulation, application deployment, and metric extraction but do not support automated deployment on physical infrastructures. Although some frameworks attempt to automate evaluation, their focus remains limited. For instance, Plug and Play Bench (PAPB) [35] enables big data benchmarking using containerized execution, while Frisbee [36] performs chaos engineering on Kubernetes-deployed applications with a declarative approach to failure injection. Similarly, BenchPilot [13], although it provides declarative experimentation descriptions and benchmarking functionalities, supports only a single type of workload and, more importantly, does not support co-location. While these systems demonstrate promising advancements, they *do not provide mechanisms for automating the performance evaluation and analysis of co-located workloads across diverse edge environments.*

In summary, although several existing works support benchmarking or workload orchestration in cloud/edge settings, they either lack the capability to evaluate multi-tenancy, especially on real edge nodes, or focus on a specific type of workloads. Our work distinguishes itself by enabling fully automated deployment and in-depth analysis of co-located workloads. This empowers operators and researchers to explore interference patterns and deployment trade-offs across real-world, diverse, edge environments in a reproducible manner.

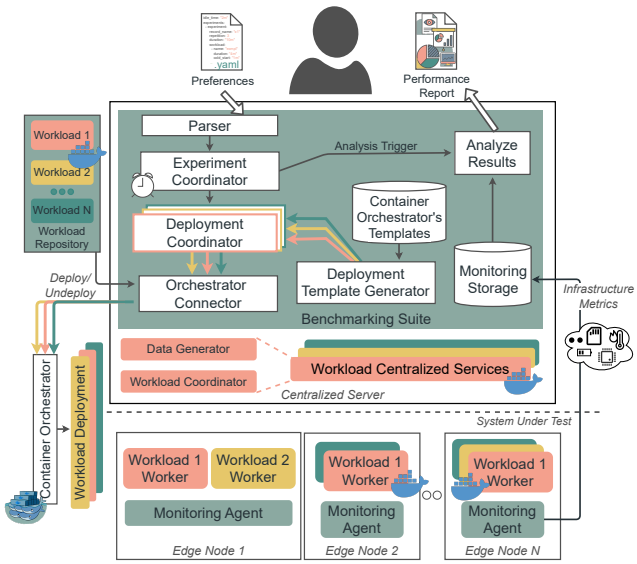

Fig. 1: The Overview of the Auto-benchmarking Framework

IV. Benchmarking Framework

A. System Overview

To address the complexity of evaluating and the deployment of co-located workloads, our system initially undergoes a bootstrapping phase to automatically install and configure all required dependencies across the evaluation nodes if necessary. This process ensures that all nodes are properly set up based on the user's configuration file that defines nodes' roles and access credentials (see Section IV-B). Once the bootstrapping phase is complete, the user can proceed with submitting experiment descriptions, as depicted in Fig. 1.

Specifically, the experiment description includes a list of co-located workloads, their configurations, execution duration, potential starting delays, and more, all structured according to the framework's high-level YAML model (see Section IV-C). Having a submitted description, the *Parser* is responsible for parsing it and evaluating whether the description of the experiments is syntactically correct and valid. If the description is correct, it is propagated to the *Experiment Coordinator*. This module translates the given information into a list of experiment objects and organizes their execution.

For each experiment object, the *Deployment Coordinator* will be spawned to create the workload(s) configurations. Each workload supported by the framework has a corresponding *Container Orchestrator Template* that defines the required services, a set of default parameters (which can be overridden), and their respective container images. The *Deployment Template Generator* retrieves the appropriate template, populates it based on the user's submitted variables, generates the final

workload description, and forwards it to the Deployment Coordinator. Once all of the experiment and workload objects have been prepared, the *Orchestrator Connector* creates deployment descriptions compatible with the user's desired container orchestrator. The orchestrator is responsible for managing the deployment and networking of containerized services and is materialized by frameworks like Docker Swarm.

After the benchmarking process setup is completed, the *Experiment Coordinator* will coordinate additional actions: (i) recording the starting and ending timestamps of the experiment, (ii) giving instruction to trigger workloads that were meant to start *delayed*, (iii) monitoring all workloads to ensure they are healthy, up, and running, and (iv) upon completion of an experiment, either repeating it as many times as defined by the user or starting the next experiment. To accomplish this, the *Experiment Coordinator* maintains constant communication with the *Deployment Coordinator*, which is responsible for invoking, monitoring, and stopping each deployment (workload). Once all experiments are complete, the user can retrieve the collected metrics and focus on the analysis. Specifically, our framework collects metrics–such as CPU usage, memory consumption, network traffic, and energy consumption–for both the underlying hardware and each deployed service. Additionally, it extracts workload-specific metrics, like latency or throughput, and stores all data as CSV files.

B. Bootstrapping Process

To streamline the benchmarking setup, our framework automates the installation, configuration, and deployment of all required software and dependencies, including the monitoring stack. Before benchmarking begins, it connects to each node based on a bootstrapping configuration where users define the cluster setup, specifying the manager and worker nodes along with their access credentials. Description 1 presents an example configuration file, where users specify: (i) the *manager* node, responsible for centralized services, such as data generators, and (ii) the *worker* nodes to be evaluated. Each node entry includes an accessible IP (reachable from the node where the framework is set up), hostname, and login credentials, supporting authentication via (i) username-password or (ii) username-SSH key. This file only needs to be set up once unless the user wishes to introduce new nodes. When the user initiates a benchmarking process, the framework utilizes this information to prepare the nodes for the evaluation, by installing all necessary software and dependencies, and then proceed with the benchmarking process.

C. Experimentation Modeling

Our framework offers users a YAML-based representation for defining comprehensive and repeatable experiment descriptions at a higher abstraction level. Users can specify a collection of experiment elements within the *experiments* section of the YAML file. Each experiment is identified by a *record_name*, which serves as the name of the log file that the system will generate for the experiment. Then, users specify the number of times the experiment should be executed (*repetition*) and, its execution *duration*. Once these parameters are specified, they define a set of *workloads* that the system will deploy concurrently. Description 2 presents a representative example of our modeling configuration, where two distinct workloads are defined: a database and a streaming analytics workload. Focusing on the first workload, users declare its *name* according to the framework's supported workloads. Then, they designate the set of nodes (*cluster*) where the workload will be deployed and provide the relevant workload parameters (e.g., *db: "mongodb"*). Depending on the workload type, users can also specify additional parameters, such as the data generation rate (e.g., *tuples_per_second*), the underlying system type (e.g., *"storm"*, *"mongodb"*, etc.), and other configuration settings. Users can also introduce a delay (*shift*) for one or more workloads, allowing them to start a few seconds or minutes after the experiment begins. Once the experiment definitions are completed, they can configure additional parameters, such as the idle duration between experiments (*idle_between_experiments*) and the chosen connector for managing the cluster (*orchestrator*, e.g., *"docker swarm"*). It is important to highlight that, since our framework operates directly on the underlying infrastructure without imposing additional constraints on system resources, users do not need to specify any limitations, such as memory and network settings.

```
cluster:
  manager:
      ip: "0.0.0.0" # manager's IP
      hostname: "manager"
      ssh_key_path: "/conf/ssh_keys/ssh_key.pem"
  nodes: # system under test
    - ip: "10.10.10.10" # using password
      hostname: "raspberrypi"
      username: "pi"
      password: "raspberrypi"
    ...
```

Description 1: Example of the Bootstraping Configuration

D. Containerization of Workloads

Containerization is a lightweight virtualization technique that abstracts hardware, enabling efficient application execution with minimal performance overhead [11]. To achieve an efficient and dynamic workload deployment, our framework utilizes Docker as its containerization engine. In Docker, users define services using Dockerfiles, which specify configurations, dependencies, and environment variables before being built into images and deployed as running containers. By leveraging Docker's inheritance mechanism, we introduce a base image, containing all essential dependencies, to ensure consistency across workloads. To support diverse edge infrastructures, including x64 and ARM architectures, our framework generates multi-architecture Docker images, automatically compiling them for the target hardware. For seamless distribution, we publish all image artifacts on DockerHub [20], making them accessible to the research and industry communities. Finally, containerizing the workloads enables our system to seamlessly integrate with various Docker orchestrators,

```
experiments:
  - experiment:
      record_name: "streaming_with_db"
      repetition: 2
      duration: "20m"
      workloads:
        - name: "database"
          cluster: [ "rpi", "small_server" ]
          parameters:
            db: "mongodb"
        - name: "marketing-campaign"
          cluster: [ "rpi", "small_server" ]
          parameters:
            engine: "storm"
            enging_parameters:
                tuples_per_second: 1000
                capacity_per_window: 10
          shift: 5m
  - experiment:
          ...
idle_between_experiments: "2m"
orchestrator: "docker swarm"
```

Description 2: Example of the Experiment Model

such as Docker Swarm or Kubernetes, without requiring any modifications to its core implementation components.

E. Infrastructure Monitoring Stack & Metrics

To monitor workloads and the system under test, our framework employs a monitoring stack that collects infrastructure metrics such as CPU and memory usage. This is enabled by deploying a containerized Netdata [37] agent on each node to gather metrics non-intrusively via multiple probes targeting specific sub-components (e.g., cgroup, OS). The agent exposes an API that a centralized monitoring server uses to periodically collect and store data. In our testbed, we expose two probes: one for a Meross Wi-Fi Smart Plug [38] for IoT devices, and another for a Smart Power Distribution Unit (SPDU) that uses SNMP to retrieve server power consumption. All metrics are collected at 5s intervals and stored in Prometheus [39], a widely used open-source monitoring server.

The key metrics used in our evaluation are: (i) average *CPU Utilization* (excluding idle time, in %), as high usage can lead to performance issues and crashes; (ii) total *Memory Usage* (in MiB), to ensure sufficient memory for system stability; (iii) total *Disk I/O* (in KiB), which affects data-intensive task performance; (iv) total *Network I/O* (in bytes), as high usage may cause delays; (v) average *Power Consumption* (in watts), a key cost and environmental factor; and (vi) average *CPU Temperature* (in °C), crucial for preventing hardware damage. All metrics are collected at the system level, while CPU, memory, disk, and network are collected on service level.

V. Applications & Workloads

Our framework offers a range of workloads to simulate diverse scenarios and stress resource components individually or in combination. These workloads are: (i) ***Component-based Stressors*** for targeting specific resources, (ii) ***Streaming Analytic Workload*** using popular distributed processing engines, (iii) ***ML Inference Application*** with a well-known model and various backends, and (iv) ***NoSQL Database Operations***. As

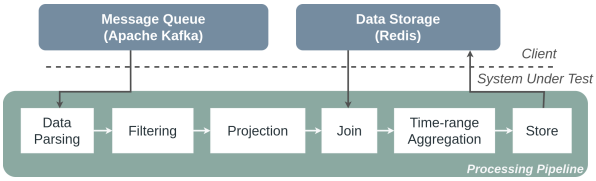

Fig. 2: Overview of Yahoo Streaming Workload

previously noted, all workloads are containerized and extended to expose application metrics, support parameterization, and enable customization via *Framework Templates*, eliminating the need for code modification or recompilation.

A. Component-based Stressors

To evaluate individual compute resources or analyze workload co-location, we selected targeted performance tests for each component. For CPU, memory, and disk I/O, we use the *Linux stress command* [40], which applies adjustable stress to components either individually or concurrently. Users can specify 1 to N CPU threads, where N is the CPUs thread limit, determining the number of workers spawned to simulate different levels of utilization of simple applications.

For network testing, we use *iperf3* [41], which transmits data between two devices, acting either as a client or a server. Devices must be on the same local network (wired or wireless) or use public IPs. This workload includes: (i) the *generator*, sending packets at a set rate and measuring outbound traffic, and (ii) the *receiver*, which monitors inbound traffic.

In regards to application-level metrics, one can collect the **number of packets** exchanged when using the iperf3 tool. As for the stress tool, it does not export any application statistics for direct use, as its purpose is solely to apply stress.

B. Streaming Analytics Workload

For this workload, we have employed the widely known *Yahoo Streaming Benchmark* [15], which is designed to simulate a data processing pipeline for extracting insights from marketing campaigns. As shown in Fig. 2, the pipeline runs on the edge device, performing tasks such as receiving advertising traffic data, filtering and cleaning it, merging it with existing key-value store information, and storing the final results. All data produced by the data generator is pushed and extracted through a message queue (Apache Kafka [42]), while intermediate data and final results are stored in an in-memory database (Redis [43]). This workload can be executed using any of the following distributed streaming processing engines: *Apache Storm*, *Flink*, or *Spark*. Additionally, we can adjust the data rate, the number of campaigns, workload duration, and engine specific parameters, such as worker partitions, etc.

To evaluate the performance of this application, we extract two measurements: the **total number of tuples** processed during execution and the **total latency** of the application, based on the statistics provided by the selected underlying processing engine for each deployed task.

C. Database Operations

To compare NoSQL database performance under different conditions, we have utilized the *Yahoo! Cloud Serving Bench-*

mark (YCSB) [21]. YCSB applies heavy load to perform basic operations such as reading, updating and inserting records using a single key. It is important to highlight that each run of the workload does not perform only one basic operation per time, but a round of all of them, based on the operation rate that the user defined, until the experimentation time is over. It supports multiple databases, like MongoDB and Redis, and offers three workload distributions: *Zipfian* (frequent access to some items), *Latest* (favoring recent records), and *Uniform* (random access). YCSB also provides adjustable parameters and a flexible schema. Users can change the number of records to be processed, the total number of operations to be performed, the load distribution, the rate of operations, and the experiment's duration. If all operations are completed before the time limit, the experiment ends early; otherwise, it stops when the time expires. Users can increase database stress by specifying the number of threads for asynchronous operations.

YCSB provides statistics for each operation it performs, informing the user of the **count**, **min**, **max**, and **average operations per second** for each minute of the experiment.

D. Machine Learning Inference Applications

To evaluate the performance of ML inference tasks, we use the MLPerf [16], a well-known benchmark for ML training and inference. While MLPerf includes numerous tasks, we have focused solely on image classification to create a more targeted workload. This task is widely used in commercial applications to evaluate ML performance. It involves a classifier network that assigns an image to a class. MLPerf uses the ImageNet 2012 dataset, resizing images to 224x224 and measuring Top-1 accuracy. It employs two models: ResNet-50 v1.5, known for high computational demands and strong classification performance, and RetinaNet, effective in object detection with accurate bounding box predictions.

Except for the pre-provided models from MLPerf, we modified its codebase to serve models over the network, enabling measurement of network overhead in ML inference. Specifically, we created two separate services: (i) a lightweight server that loads models and exposes a RESTful API, and (ii) a workload generator that loads and sends images to the server one-by-one. We refer to this as "*streaming mode*", while the default MLPerf setup, which loads images locally, is called "*local mode*". MLPerf allows users to adjust various workload parameters, including input dataset, max latency, batch size, duration, thread count for asynchronous execution, and the inference framework which is currently limited to CPU-based options: ONNX [44], NCNN [45], or TensorFlow [46].

After the workload execution, the retrieved statistics include the model's **accuracy percentage**, the **average batches per second** (where each query represents the processing batch of images), the **total completed queries**, and the **mean latency**.

VI. Evaluation

This section comprehensively evaluates our framework's capabilities via experiments based on the use case in Section II, demonstrating its ability to assess ML inference services across various configurations and co-located workloads.

A. Experimental Setup & Devices Under Test

To realistically replicate MEC nodes and their typical heterogeneity, common across the Multi-access Edge Computing continuum, we selected a range of compute devices under test:
Single-board Computer (SBC): A Raspberry Pi 4 Model B with a quad-core ARM Cortex-A72 (4 threads, 1.5GHz) and 4GB RAM. As the most well-known SBC, we chose one of its most popular variants, capable of running multiple workloads.
SMALL server, that is equipped with a six-core Intel(R) Xeon(R) CPU X5690, which has 12 threads @3.47GHz, 12MB CPU cache, 6.4 GT/s bus speed, and Thermal Design Power (TDP) 130W. Additionally, it comprises a 12GB RAM.
MEDIUM server, which features a 71GB RAM, and two Intel(R) Xeon(R) CPU E5620 processors. Each CPU has 4 cores and 8 threads at a 2.4GHz frequency, 5.86 GT/s bus speed, 12MB CPU cache, and 80W TDP.
LARGE server, has 173GB RAM and two Intel(R) Xeon(R) CPU E5-2680v3 CPU processors. Each processor has 12 cores and 24 threads @2.5GHz frequency, 9.6 GT/s bus speed, 30MB cache, and 120W TDP.

All servers feature dual power supply plugs to ensure uninterrupted operation. The **SMALL**, **MEDIUM**, and **LARGE** servers have dynamic frequency scaling (CPU throttling) enabled, a standard Intel CPU feature that improves power efficiency and reduces heat during CPU-intensive workloads.

Lastly, we use a server for (i) the experiments' orchestration, (ii) the deployment of data generators, and (ii) the collection of utilization and application metrics. The latter server has 71 GB RAM, and a 12-core CPU with 24 threads.

B. Workload Parameterization

For our analysis we have employed the workloads that were previously discussed with specific parameters, namely:
Component-based stressors, using them for stressing the: (i) CPU; (ii) memory; (iii) I/O; and (iv) network. The primary objective of this workload is to create an intense co-located workload that occupies the device's resources. In all stress scenarios, the number of workers assigned to each device was based on the maximum number of threads supported. We spawned 4 workers for the Raspberry Pi, 12 for the small server, 16 for the medium server, and 48 for the large server.
Streaming workload focuses on Apache Storm as the streaming processing engine, even though our experiments could be easily done using any of the other two engines, Spark and Flink. In regards to workload parameters, we have used 1000M tuples per second data generation rate and 100M campaigns.
Database workload employs MongoDB as the underlying database, using a Zipfian load distribution and the default YCSB workload configuration values, with two exceptions: the number of records and the number of operations, which for both we used a value of 2.5M. Additionally, we chose to run this benchmark in asynchronous mode, using 12 threads, in order to stress our cluster to its maximum capacity.
ML workload uses the image classification for both local and streaming mode, utilizing the following inference frameworks:

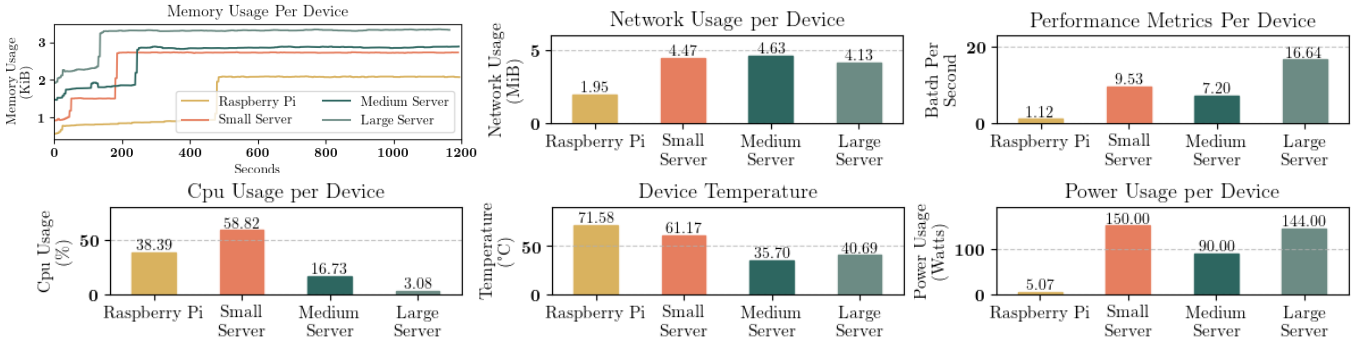

Fig. 3: Utilization and Performance Metrics Across Devices Using an ML Inference Application

ONNX, NCNN, and TensorFlow. Again, we deployed this workload using asynchronous processing with 12 threads.

For all the aforementioned workloads, *the selection of parameters was made based on trial and error*, where we tested a variety of values until we identified the silver lining of load to stress our cluster without overwhelming it.

C. Results and Discussion

In this section, we explore three key research questions related to the use case scenario outlined in Section II. These questions not only demonstrate the usability of our system but also provide valuable insights into service co-location on the edge, benefiting both researchers and practitioners. All scenarios are executed for 20 minutes utilizing our framework and capturing utilization and application-level metrics. Additionally, all of our utilization plots represent the mean value over the entire execution (e.g., power consumption, measured in watts, represents the average rate of power usage).

RQ1: Given an edge cluster, which device is best suited for execution of a streaming ML inference application in terms of performance, resource utilization and energy efficiency?

Let us consider a scenario, where a group of administrators have at their disposal a set of unused devices and want to deploy a streaming ML inference application. So, they are interested in evaluating which device performs best with this workload. To replicate this scenario, we deployed the ML inference application in streaming mode across all of the devices under test. It is important to highlight that the workload deployment was chosen to be isolated at this stage, allowing it to serve later as a baseline for comparison.

The first plot (upper-left corner of Fig. 3) shows the memory usage for the different nodes that follow a similar pattern. Initially, memory utilization is low across all nodes since the model has not yet been loaded. At a certain point, we notice a distinct ”step”, where the model occupies a significant portion of memory, maintaining this level until the end. However, the delay in memory allocation (cold start effect) differs depending on the machine. The *Large Server* loads the model fastest, followed by the *Small* and *Medium Servers*, while the *Raspberry Pi* is the slowest. Additionally, baseline memory usage and the devices' storage differences also impact the loading time. To ensure fair analysis, the results in the following plots and sections consider only the period after model loading.

Next, we examined CPU usage, power needs, and temperature metrics (Fig. 3, second series of plots), as these factors are closely correlated. We observe that both the *Raspberry Pi* and *Small Server* have a higher CPU usage, contrary to the *Medium* and *Large Servers*, where the CPUs remain underutilized. Moreover, the *Small Server* has the highest power demand during inference, consuming 150 watts, followed by the *Large Server* (144 watts) and the *Medium Server* (90 watts). The *Raspberry Pi*, due to its low-power profile, requires only 5 watts. However, when analyzing temperature, the *Raspberry Pi* reaches the highest levels at 71.5°C, followed by the Small Server at 61°C. This indicates that both devices are operating under significant load. In contrast, the *Medium* and *Large Servers* maintain lower temperatures (35°C - 40°C).

Moreover, we examine the performance and the overall network traffic of each device (Fig. 3 first line middle and right plots). Starting with throughput (processed batch of images per second), *Raspberry Pi* exhibits the lowest performance by a significant margin, processing only 1.12 batches per second, while *Small*, *Medium*, and *Large Servers* process 9.5, 7.2, and 16.6 batches per second, respectively. Interestingly, *Small Server* achieves higher throughput than *Medium Server*, but the latter offers a better balance between performance and CPU utilization, resulting in lower power consumption. In network traffic, due to its limited processing throughput, the *Raspberry Pi* generates approximately half of the network traffic of the other deployments. For the remaining nodes, data transfer remains relatively consistent, with minor differences related to the randomness of batching images before transmission.

Finally, we evaluated the energy efficiency of each node by calculating the average energy consumption per processed batch. Our results show that the *Raspberry Pi* is the most energy-efficient node, requiring only 4.52 joules per batch. Among the servers, the *Large Server* performs best, consuming 8.65 joules per batch, followed by the *Medium Server* and the *Small Server* at 12.5 and 15.73 joules per batch, respectively. ***Key Takeaways:*** To this end, small edge devices (e.g., Raspberry Pi) seem to have lower performance in model loading and throughput, but are the most energy-efficient options (6.67J per batch). Small servers are not always energy efficient and may have temperature issues, yet may match or surpass the medium servers in throughput. Large servers deliver the best performance and may provide energy efficiency but has high static power demands, while medium servers balance power

needs, energy efficiency and throughput.

RQ2: Given an edge device which already has deployed a streaming ML inference application, how can other different co-located services affect its performance?

After evaluating the first scenario, the administrator has chosen to deploy the ML inference algorithm on the *Medium Server*. However, with no other servers available, the user must co-locate the ML workload with other tasks, such as component-intensive stressors (e.g., CPU, memory, disk, and network) or general-purpose workloads (e.g., databases or streaming analytics). To determine whether this setup is feasible and how it impacts performance, the user leverages our framework again, modifying only the input parameters.

Our first observation is that cold start duration varies across different co-location strategies. Specifically, when deploying the ML streaming inference workload alongside a network stressor (iperf), the impact on cold start time is minimal (less than 10%). In contrast, collocating the service with CPU and disk I/O stressors increases cold start duration by approximately 35-40 %. Most notably, introducing a memory-intensive stressor doubles the cold start time, rising from 244 seconds to 505 seconds. This underscores the significant impact of memory pressure on ML model initialization.

Moreover, Fig. 4 (first plot) illustrates the CPU utilization of each workload across different experiments. Our key observation is that component-based stressors significantly reduce the CPU utilization of the ML streaming service. In an isolated execution, the service utilizes 16.73% of the CPU (Fig. 3, right-bottom plot). However, when co-located with disk I/O, CPU, and memory stressors, its utilization drops to approximately 8%, as these stressors consume a substantial share of CPU resources: 76.82% (disk I/O), 89.93% (CPU), and 90.10% (memory). When the co-located workloads require less CPU, such as the network stressor (0.29%), database (27.98%), and streaming analytics (14.14%), the CPU allocation for the ML workload nearly doubles to approximately 14-15%, which is closer to the 17.67% of isolated execution scenario.

Regarding memory and network utilization (Fig. 4, last two plots), we observe minimal impact on the corresponding metrics of the target workload (ML streaming). Interestingly, when workloads introduce network stress, such as the network stressor or streaming analytics, the target workload generates more network traffic. Although this may seem counterintuitive, it can be explained by considering CPU utilization and throughput. Specifically, we observe that more data points are processed and transmitted over the network when the CPU usage of the co-located workloads is low. Moreover, the memory allocation of the memory stressor and the database appears to have minimal impact on the target workload, except for the cold start period, as previously discussed.

Additionally, first and second plots of Fig. 5 show the server's temperature and power consumption, respectively. While workload co-location inevitably increases both metrics, different workload types contribute to varying levels of impact. For instance, CPU-intensive stressors significantly elevate energy demand, with power consumption rising from 88 watts

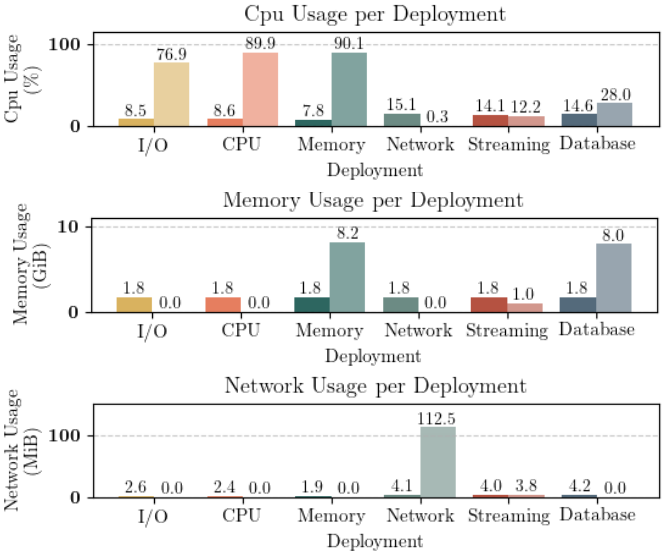

Fig. 4: CPU, memory & network usage during co-location of an ML inference application with other workloads. Darker shades represent the ML workload's resource consumption, while lighter indicate the co-located workload's usage.

(baseline) to 114 watts, 130 watts, and 132 watts for I/O, CPU, and memory stressors, respectively. In contrast, real-world workloads, such as database operations (105 watts) and streaming analytics (100 watts) lead to more moderate increases. Moreover, power consumption and temperature exhibit a strong correlation, with the highest temperature observed under CPU stress, followed by memory and I/O stressors. Notably, network stress has negligible effects on power and temperature, reevaluating the relationship between CPU usage and system heat generation.

Finally, last plot of Fig. 5 illustrates the throughput (Batches per second) of the ML streaming workload and the impact of service co-location. The green dashed line represents the throughput without co-location (7.29 batches per second). When co-located with disk I/O, network, database, and streaming workloads, performance degradation occurs, reducing throughput to approximately 4.5–5.5 Batches per second. The workloads that affect the ML application are the CPU-intensive workload and the memory-intensive workload. Examining the utilization metrics in Fig. 4, we observe that the CPU-intensive workload primarily consumes CPU resources, while the memory stressor heavily utilizes both CPU and memory. The latter has a greater negative impact on the ML workload, reducing its throughput to 2.54 Batches per second, whereas the CPU-intensive workload allows for a higher throughput of 4.04 Batches per second. The plot highlights a strong correlation between available CPU resources and the ML streaming service's throughput, as well as the influence of other compute resources, such as memory.

Key Takeaways: In this scenario, we highlighted that memory-intensive stressors significantly increase the cold start time of ML services, which is important to consider, especially when there is a need to restart the workload frequently. Furthermore, CPU and disk I/O stressors substantially reduce the CPU allocation for the ML workload, leading to degraded performance.

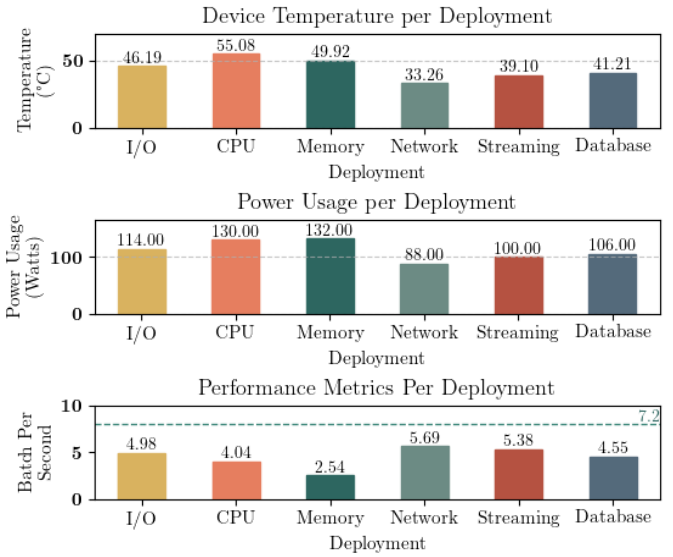

Fig. 5: Temperature, Power Usage, and Performance Metrics for co-located workloads (ML+Workload)

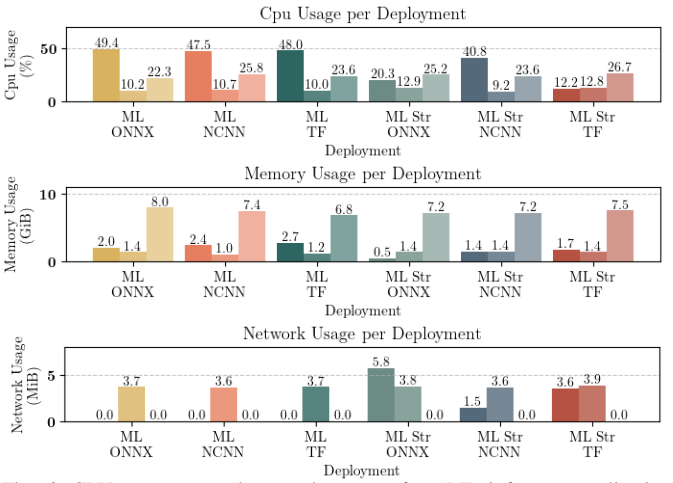

Fig. 6: CPU, memory, and network usage of an ML inference application co-located with other workloads. In each group, the first shade represents the ML workload, the middle the Streaming Analytic workload, and the right the Databases resource consumption.

Additionally, when co-located workloads demand less CPU, the ML service retains higher CPU utilization, improving throughput, while memory and network stressors have minimal direct impact. Moreover, CPU-intensive stressors significantly increase power consumption and temperature, while real-world workloads have a more moderate impact, and network stressors exhibit negligible effects, reinforcing the strong correlation between CPU usage and system heat generation.

RQ3: Given an Edge device which already has deployed multiple workloads, like ML, databases, and streaming analytics, how can different configurations of them affect the overall performance and utilization metrics?

In this session, we examine a scenario where the user has already deployed workloads on a selected node (*Medium Server*) and intends to execute an ML workload alongside them. The user also aims to compare different execution backends in various modes to find the best option for throughput and energy efficiency, while also identifying any potential uncharted trade-offs. For this, we co-locate the ML workload with streaming analytics and database workloads while also modifying the benchmarking configuration of the ML workload. We assess three different execution engines (backends) – ONNX, NCNN, and TensorFlow (TF) – under two execution modes: (i) streaming (remote), where image batches are sent over the network from a workload generator to the stressed node; and (ii) local execution, where images exist on the same machine, allowing inference without network involvement.

Examining the cold start period across different deployments, we observed that the ONNX and NCNN engines load the model significantly faster (30–40 seconds) compared to TensorFlow (356–375 seconds) in all cases. So, specialized ML engines optimized for CPU compute resources can substantially reduce the startup time in such deployments. Moreover, we have to note that the variations on co-located services (at least for database and streaming analytic process) do not influence the starting period of ML workloads.

Fig. 6 depicts the utilization of CPU, memory, and network resources when an ML inference application runs alongside other workloads. In each group, the first shaded section (left bar) represents the ML workload, the middle one corresponds to the Streaming Analytics workload, and the rightmost bar indicates the resource consumption of the Database workload. Firstly, we observe that local execution (ML ONNX / NCNN / TF) leads to higher CPU utilization for ML workloads, reaching just below 50%, while the database and streaming analytics workloads remain at approximately 23% and 10%, respectively. In contrast, when the ML workload processes batches of images transmitted over the network (ML Str ONNX / NCNN / TF), the CPU utilization decreases to 20.3% for ONNX, 40.8% for NCNN, and 12.2% for TF. This reduction is primarily due to network-related operations (e.g., encoding/decoding, HTTP handshake, etc.) handled by the network connector, during which the CPU remains idle. As a result, more CPU capacity becomes available for the co-located services, leading to a slight increase in their CPU utilization, ranging from 23.6% to 26.7% for the database workload and from 9.2% to 12.8% for streaming analytics.

For memory usage, we saw almost no differences in the database and streaming analytics workloads across execution modes, with the database workload consuming the largest portion of the server's memory (7–8GB). In contrast, the ML shows differences in memory allocation between local (2–3GB) and remote execution (1.4–1.7GB), as in local execution, the workload loads the entire dataset into memory, whereas in remote execution, it only fetches the model's weights.

Regarding network usage, streaming analytics dominate, generating approximately 3.7 MB of network traffic during the experiment. Neither database workload nor local ML execution involve network data transfers. Interestingly, ML streaming (remote) workloads consume a significant amount of network resources, with ML Str ONNX surpassing the streaming analytics workload and ML Str TF showing similar network usage. To understand this behavior, one must consider the

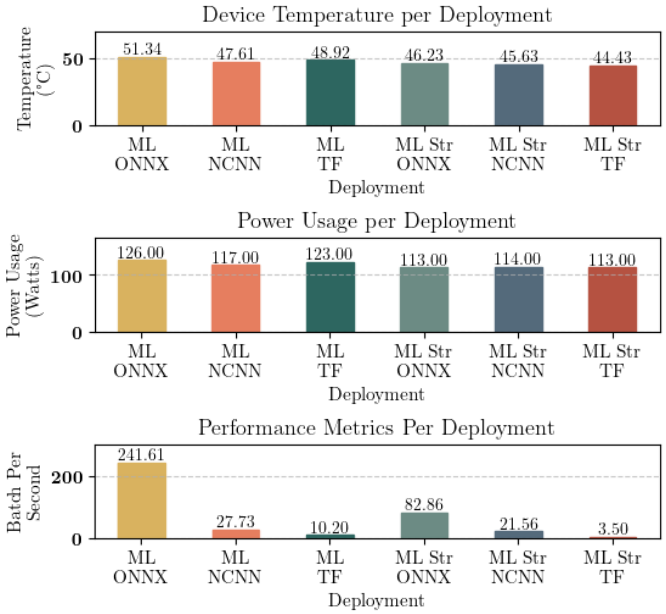

Fig. 7: Temperature, Power Usage, and Performance Metrics for co-located workloads (ML+Streaming+Database)

rate and size of the input data for streaming workloads (both analytics and ML). Streaming analytics handle numeric data at a higher throughput than ML services, whereas ML workloads process batches of images. Although images are larger in size compared to numeric data, their processing takes longer, usually resulting in lower throughput. However, since ONNX achieves a higher processing rate, it requires more network bandwidth compared to the streaming analytics workload.

Fig. 7 presents the server's temperature, power consumption, and the performance metrics of the ML workload. The temperature during local ML execution (ML ONNX / NCNN / TF) is slightly higher, ranging from 47.61°C to 51.19°C, compared to streaming ML execution (ML Str ONNX / NCNN / TF), which varies between 44.43°C and 46.23°C. Moreover, local execution requires more power, consuming between 117–127 watts, whereas remote execution operates with a lower power demand of 113–114 watts.

The last plot of Fig. 7 shows the throughput (batch per second) of ML services. In all cases, local ML execution provides much better results than the same workload running remotely. Specifically, the best performance is observed for local ONNX with 241.61 batches/second. The second position is also allocated by the ONNX but for its streaming execution with 82.86 batches/second. The latter highlights the superiority of ONNX execution engine for CPU inference. NCNN handles 27.73 batches/second and 21.56 batches/second, for local and remote execution, respectively, while the worst performance observed for TensorFlow (10.2 and 3.5 batches/second).

Key Takeaways: Our analysis highlights that ONNX and NCNN achieve significantly faster cold start times than TensorFlow, due to the benefits of CPU-optimized ML engines, which is particularly advantageous when someone wants to change models frequently. Local execution results in higher CPU usage but reduces network traffic, while remote execution lowers ML CPU usage, allowing co-located services to utilize more resources. ML workloads require more memory during local execution (2–3GB) than remote (1.4–1.7GB), and the latter workloads consume notable network bandwidth, with ONNX demanding the most due to its high throughput. Local ML execution leads to slightly higher temperatures (47.61°C–51.19°C) and power consumption (117–127W) than remote execution (44.43°C–46.23°C, 113–114W). Performance-wise, ONNX outperforms other backends, achieving the highest throughput (241.61 batches/sec locally, 82.86 remotely), while TF performs the worst (10.2 and 3.5 batches/sec).

VII. Conclusion

In this work, we introduced an automated benchmarking framework to evaluate the impact of multi-tenancy in Edge Computing environments. It simplifies the deployment, monitoring, and analysis of co-located workloads, enabling reproducible performance evaluation across diverse hardware setups. Through a series of experiments, we showed how workload co-location impacts key performance metrics, including cold start duration, CPU utilization, and network behavior. Specifically, our *key findings* include: (i) small edge devices, while energy-efficient, have slower model loading and throughput compared to larger servers, which offer the best performance but come with significantly higher power demands; (ii) memory-intensive workloads increase the cold start time of machine learning applications by nearly 1.5x, while CPU and disk I/O stressors degrade performance through heavy CPU usage; and (iii) CPU-optimized ML engines, such as ONNX and NCNN, achieve significantly faster cold starts (up to 11), with ONNX achieving the highest throughput, surpassing NCNN by 9x and TensorFlow by 24x. Building on these insights, our tool allows researchers to generate profiles of utilization metrics such as memory, CPU, disk, and energy, for each workload combination and configuration.These profiles are valuable for co-location-aware strategies, while practitioners can use the benchmarking loops to monitor and adapt placement policies based on specific target metrics.

For **future work**, we plan to evaluate the impact of the multitenancy of modern ML workloads, deployed on GPUs or edge accelerators. Additionally, we will extend the framework to support the creation of customizable edge network topologies, by using programmable network devices or emulators, in order for our framework to be capable of changing not only workload parameters but also infrastructure configurations. Most importantly, we aim to enhance the framework with the capability to automatically identify optimal workload parameters and configurations, apply them, and allocate them to edge nodes. To this end, we will explore automated methods, such as Bayesian optimization, ML techniques, and reinforcement learning, to determine deployment parameters based on specified performance indicators, including energy consumption, latency, and computational footprint.

Acknowledgement. This work is part of AdaptoFlow that has indirectly received funding from the European Unions Horizon Europe research and innovation action programme, via the TRIALSNET Open Call issued and executed under the TrialsNet project (Grant Agreement no. 101017141).

REFERENCES

[1] "Powering an inclusive, digital future for all, cisco," https://newsroom.cisco.com/c/r/newsroom/en/us/a/y2023/m01/powering-an-inclusive-digital-future-for-all.html/.

[2] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Fogify: A fog computing emulation framework," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2020, pp. 42–54.

[3] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[4] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on multi-access edge computing for internet of things realization," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 2961–2991, 2018.

[5] S. Wijethilaka and M. Liyanage, "Survey on network slicing for internet of things realization in 5g networks," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 957–994, 2021.

[6] S. Zobaed, A. Mokhtari, J. Champati, M. Kourouma, and M. Salehi, "Edge-MultiAI: Multi-tenancy of latency-sensitive deep learning applications on edge," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2022, pp. 11–20. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/UCC56403.2022.00012

[7] P. Subedi, J. Hao, I. Kim, and L. Ramaswamy, "Ai multi-tenancy on edge: Concurrent deep learning model executions and dynamic model placements on edge devices," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. Los Alamitos, CA, USA: IEEE Computer Society, sep, pp. 31–42. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CLOUD53861.2021.00016

[8] I. Alla, H. B. Olou, V. Loscri, and M. Levorato, "From sound to sight: Audio-visual fusion and deep learning for drone detection," in *Proceedings of the 17th acm conference on security and privacy in wireless and mobile networks*, 2024, pp. 123–133.

[9] K. Krasovitskiy, S. Christou, and D. Zeinalipour-Yazti, "LLM-MS: A multi-model llm search engine," in *First Intl. Workshop on Coupling of Large Language Models with Vector Data Management (LLM+Vector Data) (LLMVDB'25), collocated with the 40th IEEE International Conference on Data Engineering (IEEE ICDE'25)*, 2025.

[10] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and Virtual Machines at Scale: A Comparative Study," in *Proceedings of the 17th International Middleware Conference*. Trento Italy: ACM, Nov. 2016, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/2988336.2988337

[11] R. Morabito, "Virtualization on internet of things edge devices with container technologies: A performance evaluation," *IEEE Access*, 2017.

[12] G. Premsankar and B. Ghaddar, "Energy-efficient service placement for latency-sensitive applications in edge computing," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 17 926–17 937, 2022.

[13] J. Georgiou, M. Symeonides, M. Kasioulis, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Benchpilot: Repeatable & reproducible benchmarking for edge micro-dcs," in *2022 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2022, pp. 1–6.

[14] R. Mohammad, S. Gopalakrishnan, and K. Pattabiraman, "Co-approximator: Enabling performance prediction in colocated applications." *ACM Trans. Embed. Comput. Syst.*, vol. 24, no. 1, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3677180

[15] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 2016, pp. 1789–1792.

[16] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.

[17] P. Subedi, J. Hao, I. K. Kim, and L. Ramaswamy, "Ai multi-tenancy on edge: Concurrent deep learning model executions and dynamic model placements on edge devices," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021, pp. 31–42.

[18] M. Jansen, L. Wagner, A. Trivedi, and A. Iosup, "Continuum: Automate infrastructure deployment and benchmarking in the compute continuum," in *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, 2023, pp. 181–188.

[19] Joanna Georgiou, "Github repository," https://tinyurl.com/GithubFramework, 2025, accessed: 2025-05-28.

[20] ——, "Framework docker images repository," https://tinyurl.com/DockerHubFramework, 2025, accessed: 2025-05-28.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[22] N. Wang, M. Matthaiou, D. S. Nikolopoulos, and B. Varghese, "Dyverse: Dynamic vertical scaling in multi-tenant edge environments," *Future Generation Computer Systems*, vol. 108, pp. 598–612, 2020.

[23] D. Trihinas, M. Symeonides, J. Georgiou, G. Pallis, and M. D. Dikaiakos, "Energy-aware streaming analytics job scheduling for edge computing," in *2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2023, pp. 161–168.

[24] S. Ilager, A. N. Toosi, M. R. Jha, I. Brandic, and R. Buyya, "A data-driven analysis of a cloud data center: Statistical characterization of workload, energy and temperature," in *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*, ser. UCC '23. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3603166.3632137

[25] W. Chen, K. Ye, and C.-Z. Xu, "Co-locating online workload and offline workload in the cloud: An interference analysis," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 2278–2283.

[26] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2988336.2988337

[27] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan, "A holistic evaluation of docker containers for interfering microservices," in *2018 IEEE International Conference on Services Computing (SCC)*, 2018, pp. 33–40.

[28] A. Morichetta, V. Pujol, S. Nastic, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, "Polarisprofiler: A novel metadata-based profiling approach for optimizing resource management in the edge-cloud continnum," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2023, pp. 27–36. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SOSE58276.2023.00010

[29] U. Awada and J. Zhang, "Edge federation: A dependency-aware multi-task dispatching and co-location in federated edge container-instances," in *2020 IEEE International Conference on Edge Computing (EDGE)*, 2020, pp. 91–98.

[30] M. N. H. Nguyen, C. W. Zaw, K. Kim, N. H. Tran, and C. S. Hong, "Let's share the resource when were co-located: Colocation edge computing," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 5, pp. 5618–5633, 2020.

[31] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran, "Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures," in *2017 IEEE Fog World Congress (FWC)*, 2017, pp. 1–6.

[32] J. Hasenburg, M. Grambow, and D. Bermbach, " MockFog 2.0: Automated Execution of Fog Application Experiments in the Cloud ," *IEEE Transactions on Cloud Computing*, vol. 11, no. 01, pp. 58–70, Jan. 2023. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TCC.2021.3074988

[33] M. Jansen, L. Wagner, A. Trivedi, and A. Iosup, "Continuum: Automate infrastructure deployment and benchmarking in the compute continuum," in *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '23 Companion. New York, NY, USA: Association for Computing Machinery, 2023, p. 181188. [Online]. Available: https://doi.org/10.1145/3578245.3584936

[34] O. Naman, H. Qadi, M. Karsten, and S. Al-Kiswany, "Mecbench: A framework for benchmarking multi-access edge computing platforms," in *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*. IEEE, 2023, pp. 85–95.

[35] S. Ceesay, A. Barker, and B. Varghese, "Plug and play bench: Simplifying big data benchmarking using containers," in *IEEE BigData*, 2017.

[36] F. Nikolaidis, A. Chazapis, M. Marazakis, and A. Bilas, “Frisbee: A suite for benchmarking systems recovery,” in *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, ser. HAOC, 2021.
[37] Netdata, Inc., “Netdata cloud,” https://www.netdata.cloud/, 2025, accessed: 2025-05-30.
[38] Meross Technology Ltd., “Smart plug - meross wifi plug,” https://www.meross.com/product/3/article/, 2025, accessed: 2025-05-30.
[39] Prometheus Authors, “Prometheus: Monitoring system & time series database,” https://www.prometheus.io/, 2025, accessed: 2025-05-30.
[40] Linux Die.net, “stress(1) - linux manual page,” https://linux.die.net/man/1/stress, 2025, accessed: 2025-05-30.
[41] iPerf Team, “iperf download page,” https://iperf.fr/iperf-download.php, 2025, accessed: 2025-05-30.
[42] Apache Software Foundation, “Apache kafka,” https://kafka.apache.org/, 2025, accessed: 2025-05-30.
[43] Redis Labs, “Redis,” https://redis.io/, 2025, accessed: 2025-05-30.
[44] ONNX Contributors, “Open neural network exchange (onnx),” https://github.com/onnx/onnx, 2019, accessed: 2025-05-30.
[45] Tencent, “ncnn: High-performance neural network inference framework optimized for mobile platforms,” https://github.com/Tencent/ncnn, 2023, accessed: 2025-05-30.
[46] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Vigas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “Tensorflow: Large-scale machine learning on heterogeneous distributed systems,” in *OSDI Workshop on Machine Learning Systems (MLSys)*, 2015, https://www.tensorflow.org/about/bib.