future internet

MDPI

Article

Towards Efficient Resource Allocation for Federated Learning in Virtualized Managed Environments

Fotis Nikolaidis [1,*], Moysis Symeonides [2] and Demetris Trihinas [3]

[1] Institute of Computer Science, Foundation for Research and Technology Hellas, 70013 Heraklion, Greece
[2] Department of Computer Science, University of Cyprus, 1678 Nicosia, Cyprus; symeonidis.moysis@ucy.ac.cy
[3] Department of Computer Science, School of Sciences and Engineering, University of Nicosia, 2417 Nicosia, Cyprus; trihinas.d@unic.ac.cy
* Correspondence: fnikol@ics.forth.gr

Abstract: Federated learning (FL) is a transformative approach to Machine Learning that enables the training of a shared model without transferring private data to a central location. This decentralized training paradigm has found particular applicability in edge computing, where IoT devices and edge nodes often possess limited computational power, network bandwidth, and energy resources. While various techniques have been developed to optimize the FL training process, an important question remains unanswered: how should resources be allocated in the training workflow? To address this question, it is crucial to understand the nature of these resources. In physical environments, the allocation is typically performed at the node level, with the entire node dedicated to executing a single workload. In contrast, virtualized environments allow for the dynamic partitioning of a node into containerized units that can adapt to changing workloads. Consequently, the new question that arises is: how can a physical node be partitioned into virtual resources to maximize the efficiency of the FL process? To answer this, we investigate various resource allocation methods that consider factors such as computational and network capabilities, the complexity of datasets, as well as the specific characteristics of the FL workflow and ML backend. We explore two scenarios: (i) running FL over a finite number of testbed nodes and (ii) hosting multiple parallel FL workflows on the same set of testbed nodes. Our findings reveal that the default configurations of state-of-the-art cloud orchestrators are sub-optimal when orchestrating FL workflows. Additionally, we demonstrate that different libraries and ML models exhibit diverse computational footprints. Building upon these insights, we discuss methods to mitigate computational interferences and enhance the overall performance of the FL pipeline execution.

Keywords: federated learning; machine learning; edge computing; Internet of Things

Citation: Nikolaidis, F.; Symeonides, M.; Trihinas, D. Towards Efficient Resource Allocation for Federated Learning in Virtualized Managed Environments. *Future Internet* **2023**, *15*, 261. https://doi.org/10.3390/fi15080261

Academic Editors: Qiang Duan and Zhihu Lu

Received: 3 July 2023
Revised: 19 July 2023
Accepted: 25 July 2023
Published: 31 July 2023

1. Introduction

Federated Learning (FL) is transforming the field of Artificial Intelligence (AI) by allowing collaborative training of statistical models in decentralized environments [1]. It enables multiple devices or entities to contribute their local data for training a shared model without requiring the raw data to be transferred to a central server [2]. Instead, the models are trained locally on each device, and only the model updates, which are lightweight and privacy-preserving, are exchanged with a central server or coordinator [3]. This approach ensures data privacy, as sensitive information remains on the local devices, and it also addresses challenges associated with data silos and data transmission costs. FL is particularly useful for organizations operating in collaborative and cross-border settings, such as health and financial institutions, as they are subject to regulatory and legal frameworks like the EU's General Data Protection Regulation (GDPR) [4]. These frameworks mandate strict data governance policies for managing, processing, and exchanging data among different administrative sites [5].

Federated learning workloads, which involve communication, synchronization, and aggregation of model updates, are typically implemented using specialized frameworks like Flower [6]. These frameworks not only handle the complexities of the FL process but also establish standardized APIs, protocols, and conventions, promoting interoperability and compatibility among different components of the FL ecosystem. However, migrating ML training from centralized cloud data centers to distributed environments presents significant challenges that frameworks alone cannot easily address [7]. In these distributed settings, heterogeneity is common, with federated clients run on hardware with varying levels of capabilities and availability [8]. Additionally, clients may differ in network capabilities and geographic distance from the server [9], which can cause bottlenecks that affect the overall duration of the FL training process [10].

Moreover, as deployments scale up and organizations add more clients and administrative sites, managing the system becomes increasingly complex. This can become a nightmare for organizations. Cloud-native technologies such as Docker (a virtualized container execution runtime) and Kubernetes [11] (a cluster management service) are frequently considered to tackle these challenges [12]. Virtualization offers two main benefits: (i) it abstracts physical resources like compute, memory, and storage, allowing them to be shaped according to deployment needs; and (ii) it provides essential isolation when multiple models are trained simultaneously on the same set of nodes. The cluster management service, also known as the cluster orchestrator, automates the deployment of FL clients on the available compute nodes and provides a global view of the deployment's operating conditions [13,14].

However, the different objectives and requirements between cluster orchestrators and FL frameworks can lead to conflicting resource management schemes. Cluster orchestrators rely on predefined specifications for resource requests and limits, prioritizing fairness and fault tolerance in their task-scheduling algorithms. Their aim is to minimize resource wastage and enhance the Quality Of Service (QoS) [15]. In contrast, FL frameworks have their own scheduling mechanisms for distributed model training, considering factors like dataset size, data freshness, and model synchronization requirements. Typically, their goal is to minimize model loss and achieve convergence in fewer training rounds [3]. This misalignment between cluster orchestrators and FL frameworks can lead to inefficient resource allocation, impacting the timely execution and convergence of FL workflows.

Despite the existence of several FL frameworks (i.e., Flower, FATE) and plethora of studies in relevance to FL performance [16,17], the impact of the underlying computing system to FL deployments is significantly overlooked. Our work fills this gap by providing a comprehensive overview of the impact of resource allocation schemes on FL deployments. The focus is on a single organization that manages multiple administrative sites through a shared control plane. For this scenario, we investigate two challenge vectors related to the Quality of Service (QoS) of FL deployments: *"resource fitting"* and *"multi-tenancy"*. Resource fitting involves aligning available resources with the specific requirements of FL clients, while multi-tenancy focuses on running multiple FL workloads concurrently on a shared cluster of resources. To study the scenario in real-world settings, we utilize two widely used open-source frameworks: Kubernetes for cluster orchestration and Flower for FL execution. For the observability of experiments, we extend Flower's codebase by introducing a monitoring module that captures performance metrics from physical clients, containerized services, and FL workloads. To enhance reproducibility and streamline experimentation, we containerize Flower and provide abstractions for configuring FL deployments, eliminating the need for rebuilding containers.

Towards this, the main contributions of our work are as follows:

- We present an FL use-case for a healthcare provider with geographically distributed branches, highlighting the challenge vectors of resource fitting and deploying multiple FL workflows. These challenges are fundamental and remain relevant across numerous other applications considering FL adoption.
- We document the experimental setup designed for rapid and reproducible benchmarking of the FL infrastructure and deployment configuration. This includes con-

tainerizing the FL framework (Flower), using Kubernetes as the cluster orchestrator, employing a testing toolkit (Frisbee) to adjust Kubernetes' placement policies, and extensively monitoring infrastructure, virtual, and FL execution metrics.
- We conduct a comprehensive experimentation study, systematically analyzing performance, scalability, and resource efficiency. We evaluate different combinations of orchestrator policies and FL configurations using popular ML/FL datasets (MNIST [18] and CIFAR [19]) and frameworks (PyTorch [20] and TensorFlow [21]). We explore various resource configurations, server/client co-location, pod placement policies, and investigate the effects of running multiple FL training workloads simultaneously.

The remainder of this paper is organized as follows: In Section 2, we provide a short introduction to Federated Learning. In Section 3, we describe a high-level problem statement, including a critical evaluation of the associated challenges that exist nowadays. In Section 4, we provide an overview of our testing infrastructure, tools, and experiment methodology. In Section 5 we provide an in-depth performance analysis for a single workflow, whereas in Section 6 we provide an analysis for two concurrently executed workflows. Section 7 presents a literature review including a critical evaluation of the strengths and limitations of existing approaches. Finally, in Section 8, we summarize the key findings of our experiments, discuss their implications on Federated Learning deployments, and suggest future research directions.

2. Background

2.1. Federated Learning Applicability

The origins of Federated Learning date back to a set of seminal papers in 2015–2016 [22,23] with Google's GBoard (Android keyboard) one of the early and prominent production systems that FL was tested and still in use today [24]. There are several real-world cases where FL is an appealing setting. First, as the premises of FL is to avoid overwhelming, and often sensitive, volumes of data moving from the edge to the cloud for training, organisations such as hospitals and financial institutes can take advantage of distributed learning without sharing patient (i.e., bio-signals, imaging) and customer data (i.e., credit scores) with other branches and third-party organisations [9]. Second, model training is feasible on data collected from geo-distributed devices (i.e., smart phones, IoTs, wearable, voice assistants) where the generation rate and high communication cost makes it impractical to send data to centralized data centers for processing [16]. Third, it allows for the training of models to be achieved on larger datasets (the "big data multiplier"), since the data from multiple parties are aggregated and used to achieve intelligence far greater than what a single entity could achieve [25]. Fourth, the central server does not exercise control over the clients and hence, clients can go offline willingly or due to unreliable network links, and participate in the model training process only if they desire with the server selecting clients based on availability and resource capacity criteria.

2.2. The Federated Learning Process

Figure 1 depicts a typical FL training flow where a FL central server obtains a set of available clients $(c_1, c_2, \ldots, c_k)$ that meet certain criteria (i.e., availability) and subsequently broadcasts to the clients the training program and an initial model parameterization $\mathbf{W}_{t_0}$ with t_0 denoting the initial training round. At this point, the clients update the model locally $\mathbf{w}_{c_i,t_1}$, based on their data and current knowledge without exchanging (sensitive) data among themselves. The local data of each client and the number of samples $s_k \in S$ used during local training can differ per client. When finished, the server collects an aggregate of the client updates, creating a new global model $\mathbf{W}_{t_1}$. This process is repeated for several rounds until it reaches a max number of rounds or the convergence exceeds a certain accuracy/loss for early termination.

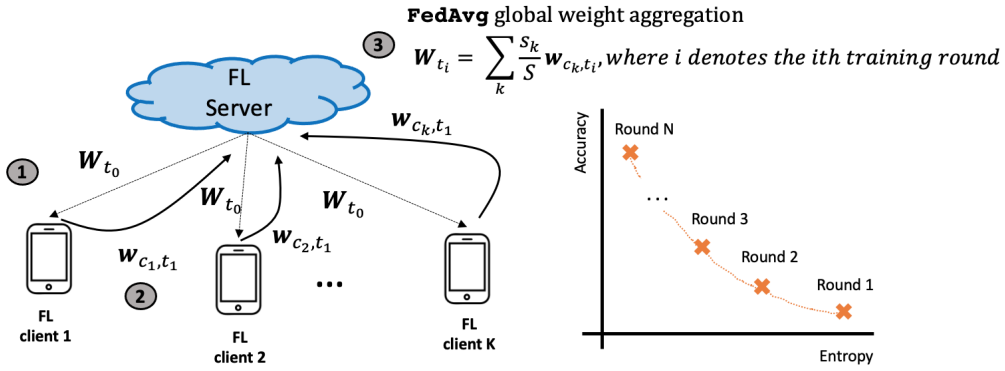


Figure 1. FL example where initial model W_{t_0} is disseminated to clients for round t_0 (step 1), clients update their local model state at t_1 (step 2) and afterward the central FL server employs a global aggregation to infer a new model state W_{t_1} (step 3) where in this case a weighted average is used.

2.3. Federated Learning Algorithms

Algorithm 1 provides an overview of FL, where at a high level, FL boils down to the coordination process overseeing the distributed training of a model on data that never leaves its origins. That said, both the aggregation function applied by the server and the client selection can take many forms.

Algorithm 1 High-Level Federated Learning Process

Input: Training rounds T, Clients K, local training epochs E, public dataset D for initial training
Output: Trained model W_T
Ensure: Central `Server` is running
1: $W_0 \leftarrow$ `initialize(D)`
2: **for** `each training round` t `in` T **do**
3: **for** `each` c_k `in` K **in parallel do**
4: $w_{t,c_k} \leftarrow$ `ClientUpdate`(W_{t-1}, E)
5: **end for**
6: $W_t \leftarrow$ `Aggregate`$(w_{t,c_1}, \ldots, w_{t,c_K})$
7: **end for**
8: **return** W_T

As an example, one may consider FedAvg [1], the most well-known FL algorithm and often considered the baseline for FL. After model initialization, FedAvg embraces local training where each client employs, in parallel, E epochs of Stochastic Gradient Descent (SGD), where the weights of the local model are updated to optimize the model loss based on the local data. At the end of the round, the central server collects the derived model weights per client. Aggregation is then performed using a weighted average where s_k is the number of samples used by each client during local training and $S = \sum_k s_k$:

$$W_t = \sum_k \frac{s_k}{S} w_{t,c_k} \tag{1}$$

With this, clients that have used more samples have a larger influence on the new state of the model. Hence, despite the simplicity of FedAvg, the literature has shown that for non-IID data, there are no convergence guarantees [26]. To compensate, FedProx [3] has been proposed as a generalization of FedAvg where the clients extend the SGD process so that clients optimize a regulated loss with a proximal term that enforces the local optimization of the loss in the vicinity of the global model per training round. Similarly, SCAFFOLD is a FL algorithm that attempts to optimize the training process for non-IID data by providing a "correction" mechanism for the client-drift problem during local training [27]. In brief, SCAFFOLD estimates the update direction for the global model at the FL server and the update direction for each client with the difference used to correct the local model versions.

To reduce the significant communication overhead imposed by FL, FedDyn is an algorithm that "pushes" to the client level more processing and optimization to reduce the overall communication burden and the number of training rounds [28]. For this, FedDyn adopts a regularization optimizer per round that dynamically modifies the client side objective with a penalty term so that, in the limit, when model parameters converge, they do so to stationary points of the global empirical loss.

In terms of client selection, the aforementioned algorithms employ a common strategy where the FL server opts for a random selection from the pool of available clients via a uniform distribution. On the other hand, two notable studies show that performing a biased selection of clients can yield faster global model convergence and reduce communication overhead. For example, after a few initial training rounds, the Power-of-Choice [29] opts for a biased selection towards clients with higher local loss. This way, the algorithm achieves faster error convergence by 3× and 10% higher test performance. In turn, FedCS [30] requires clients to send updates of their resource availability (i.e., computational power, network bandwidth) to estimate the time required to complete a training round that fits a large sample of clients. Subsequently, it only considers local model updates from those that actually meet the estimated deadline, penalizing the stragglers.

3. Motivation

This section introduces a use case and challenges that motivate the experimentation part of this article. The use case involves a large-scale Healthcare Provider with multiple branches (such as hospitals and clinics) spread across an association of countries (i.e., the European Union). Each branch of the Healthcare Provider is the bearer of its patients' data, with data governance obeying regional legislation. The goal is to develop medical AI applications trained on the available data. Researchers focus on various areas like cancer tumor detection in MRIs and abnormalities detection in patient electrocardiograms (ECGs). Although each branch can initiate ML model training using its own data, training complex medical models requires abundant data and computational resources. While computational resources can be shared among branches, data are unshareable due to regulatory restrictions.

Therefore, federated learning (FL) is the preferred training paradigm, where models are trained collaboratively on distributed data without sharing it. To facilitate FL implementation, the healthcare provider's IT team adopts a Kubernetes cluster management service. This service manages the diverse computational resources available at each branch, linking them through a common control plane (Figure 2). Kubernetes enables provisioning of the FL workflow by containerizing Flower server and client instances for distributed training. By establishing a single administrative domain, the healthcare provider can effectively offer an FL system to multiple researchers, providing them with sufficient computational resources to train complex models on their localized data. However, as more researchers adopt FL and resources are divided among them, resource allocation and workload distribution become critical challenges. Hence, from the Healthcare Provider's perspective, the interest is in reducing the FL model training time when utilizing a fixed set of (computational) resources. Subsequently, the throughput for multiple FL workloads will increase by reducing the training completion time.

Next, we highlight two main challenges faced in this deployment:resource fitting and parallel FL workflow execution.

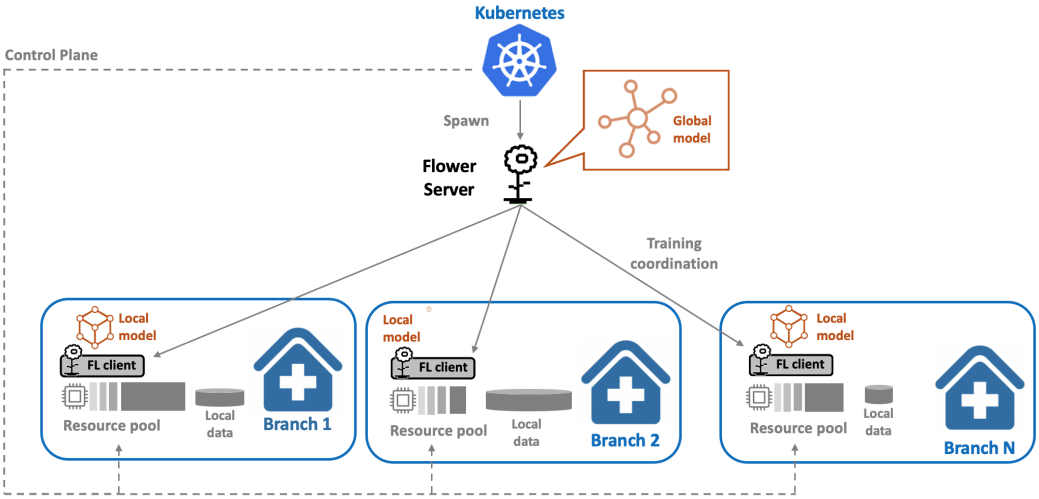


Figure 2. Healthcare provider use-case with containerized FL (Flower server and clients) employed over Kubernetes managed cluster administering common control plane and resource pool.

3.1. Challenge 1: Resource Fitting

Resource fitting refers to the process of mapping the available resource pool to the application services so that the use of the available resources meets the specific application demands as expressed by the user requirements. This process requires evaluating resource characteristics, capabilities, and availability, and assigning them to appropriate tasks [31]. To achieve effective resource fitting in FL, users must undergo an extensive and rigorous process that includes training with different properties, evaluating their impact on resource utilization and training performance, and revealing the inter-component dependencies.

Defining Application Requirements. While orchestrators like Kubernetes can automate the deployment of containerized FL workflows, they are generally agnostic to the specific requirements of each application. Users are responsible for specifying the desired amount of resources for their application to run smoothly. However, different FL workflows have varying computational requirements. For example, a time series prediction model may require significant memory but fewer computations, while a cancer image detection model relies heavily on computational power. Furthermore, ML models can have diverse parameters, structures, and libraries such as TensorFlow [21] and PyTorch [20]. To finely tune the resource requests, researchers need to extensively profile each component of the FL workflows and identify the application's actual needs.

Increasing Resource Utilization And Minimizing Wastage. Even when users understand the workflow profile and resource requirements, they may face limitations in allocating the desired resources due to quotas or allocation requests from other users in a multi-tenant environment. It is therefore important to consider the effects of resource saturation and starvation on submitted FL workflows. Such effects may also be caused by opportunistic schedulers who allocate fewer resources than is necessary, thus causing bottlenecks. Oppositely, conservative schedulers tend to allocate more resources than is necessary (over-provision), thus wasting precious resources and minimizing the potential for multi-tenancy.

Revealing Inter-Component Dependencies. Despite the decentralized nature of FL, the training is still a synchronous process that requires all clients to synchronize with the server at the end of each training round. This synchronization introduces a dependency among participating clusters, as rigorous clients have to wait for latent ones, introducing straggling task effects. To address this, users need to consider the computation- and network-aware placement of FL clients across managed nodes. By optimizing the placement of clients, the system can mitigate the impact of stragglers and improve overall efficiency, even with varying numbers of clients and data sizes.

3.2. Challenge 2: Parallel Workflows

Multi-tenancy and the implications of parallel FL workflow execution on a single administrative domain is another point of consideration for our healthcare provider use case. When users opt to run parallel FL workflows on the same physical resource pool, complexities arise that can impact the convergence speed and overall efficiency of the FL process or even cause unexpected system failures. To alleviate these threats, users need to consider the Resource Contention of parallel workloads, the Communication overhead that may be caused due to the repeatable patterns of FL workloads, and possible optimization opportunities for Workflows Synchronization.

Resource Contention. When multiple users choose to run parallel FL workflows on the same physical resource pool, resource contention issues arise. In limited resources, lightweight virtualization technologies like containers are used to share the resource pool among multiple users. However, when multiple FL workflows run simultaneously, the capacity of the nodes to handle the resource requirements becomes a bottleneck. If multiple workflows request the same constrained resources, they face resource starvation and contention problems. This conflict over shared resources (i.e., CPU, memory) hampers performance, impacts training efficiency, and hinders convergence rates.

Communication Overhead On Synchronization. FL involves frequent communication between the central server and the distributed client services. This interaction becomes critical when training models with a large number of parameters. Communication overhead and contention must be carefully examined to identify potential bottlenecks during FL cycles. For example, in a scenario where FL clients from different workloads attempt to update the global model simultaneously, the network remains idle while the clients are training the model locally, with network spikes occurring at the end of the round, during which the clients submit the training results to the central server. This strain on the network impacts overall performance and saturates the network bandwidth. By analyzing the communication overhead, optimization opportunities can be identified. For instance, shifting the starting point of one FL workflow or isolating FL servers and clients on different nodes may help avoid bandwidth congestion and minimize peak network traffic.

4. Testbed Design

FL experiments face challenges due to their distributed nature and reliance on multiple components and backend systems. These challenges hinder experiment reproducibility, making it difficult to evaluate performance under different configurations, assess scalability, and capture the dynamic behavior of FL algorithms. To address these issues, a testbed is necessary. This testbed should allow experiment replication, facilitate the comparison of approaches, and streamline the identification of performance bottlenecks and issues. In this section, we present the design of a Kubernetes-based testbed that provides a controlled environment for researchers and developers to conduct reproducible FL experiments with consistent settings, configurations, and data.

4.1. Building Parameterizable Containers for Federated Learning

Containers have proven to be effective in providing lightweight and isolated environments for testing distributed applications [12,32]. However, incorporating FL workflow components into containers presents challenges due to the need for dynamic parameterization, dependency injection, and complex networking requirements between clients and a central server, which can be difficult to manage within the scope of the containerized environment. These obstacles hinder the direct usability of FL frameworks within containerized environments, limiting their portability and scalability benefits. Therefore, there is a need for an FL framework able to interact with an external cluster orchestrator responsible for managing the execution order, network connectivity, and resource allocation to ensure the effective management of the distributed training process while maintaining the portability and scalability benefits offered by containerization in a cloud-edge continuum.

To address these challenges, we built a tool based on the Flower FL execution framework to interact with an external cluster orchestration system, out-of-the-box models, parametrizable configurations, and fine-grained monitoring. Our approach wraps Flower's configuration into a generic frontend script. This script serves two purposes: (i) defining the container's runtime environment, including model type, number of rounds, aggregation algorithm, etc., and (ii) configuring Flower's parameters accordingly and invoking the appropriate backend implementation. Currently, we support two deep-learning backends, PyTorch and TensorFlow, and two dataset types, MNIST and CIFAR-10. More information about the ML datasets and backends can be found in Sections 4.5 and 4.6, respectively. We also introduced code breakpoints for measuring the execution duration of ML training, evaluation, round duration, and code for capturing overall accuracy and loss of the FL pipeline. For building the containers, we utilized Dockerfiles [33]. These Dockerfiles include the code of the Flower FL framework and provide all the necessary instructions and environment variables for running the FL services. The resulting Docker image contains the required service codes, dependencies, and files and is uploaded to a private Docker repository. This allows us to download the image at runtime on the compute node, which can function as either an FL client or server.

4.2. Multi-Node Container Orchestration

Kubernetes is an open-source platform that streamlines the deployment, scaling, and management of containerized applications, relieving users from infrastructure concerns. It comprises two main components: the centralized control plane (Kubernetes master) and the decentralized data plane. The control plane consists of modular services like the K8s API, scheduler, and controller. The data plane is made up of distributed node agents called Kubelets [34], which act as local executors following commands from the master, such as starting or stopping containers. This separation allows users to define their desired application state, with Kubernetes striving to maintain that state. Figure 3 provides a high-level overview of our Kubernetes-enabled testing environment.

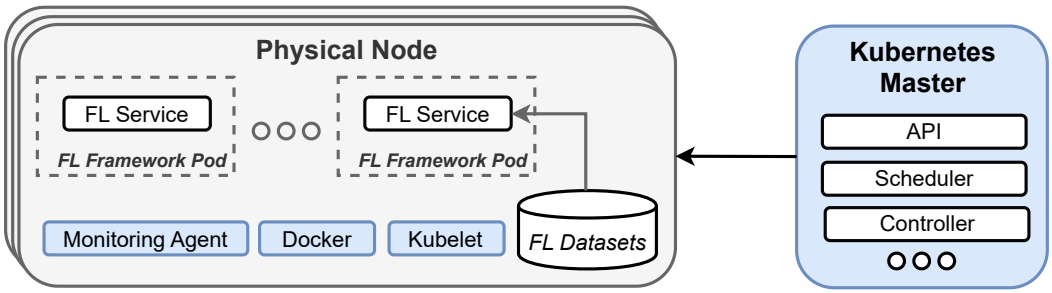


Figure 3. High-level overview of multi-host Kubernetes deployment.

A notable feature of Kubernetes is its ability to schedule the deployment and execution of containers across a node cluster using a scheduling algorithm. However, the default scheduler is designed to optimize node utilization and remain application-agnostic. For example, it may not consider data locality in scenarios like federated learning, where training data are distributed across edge devices or clusters. This can increase network traffic and latency due to unnecessary data movement. Moreover, federated learning requires frequent communication between edge devices/clusters and the central server for model updates. The default scheduler does not optimize scheduling decisions with regard to communication overhead, potentially increasing latency and impacting the training process. Additionally, each edge device or cluster in federated learning has its own resource constraints, such as limited computation power, memory, and network bandwidth.

In our analysis, we examine the default policies of Kubernetes for federated learning applications. We specifically assess the performance of the default scheduler under two scenarios: one with user-defined constraints that cater to the application's specific require-

ments, and another without any user-defined constraints. Our analysis revolves around three key user-based constraints:

- *Placement constraints:* dictate the selection of nodes for scheduling FL clients, specifying which nodes should be chosen.
- *Resource constraints:* determine the minimum and maximum amount of resources, such as CPU and memory, that an FL client can consume.
- *Timing constraints:* define when a new training workflow will be instantiated.

Our analysis aims to deliver insights into the impact of user-defined constraints and the potential for automatically inferring these constraints, rather than proposing a new scheduling algorithm.

4.3. Testing Workflows in Kubernetes

Kubernetes is a powerful platform, but it has a steep learning curve. Testers need to acquire knowledge of its concepts, configuration, and management, which can be challenging. Moreover, the dynamic nature of Kubernetes deployments can pose challenges when executing specific testing scenarios, especially those involving conditional executions.

To simplify testing on Kubernetes, we utilized the Frisbee platform, a Kubernetes-native framework specifically designed for testing distributed systems [35,36]. Frisbee offers several advantages over vanilla Kubernetes, such as orchestrating the testing actions and providing abstractions for managing containers as logical groups (e.g., servers, clients). This way, we can easily create complex placement schemes, as shown in Listing 1. Additionally, we take advantage of Frisbee's volume-sharing feature to enhance dataset acquisition from clients. Instead of each client downloading the dataset locally, we create a virtual shareable folder and pass it to the virtualized environment. FL clients are then programmed to read the datasets from this shared folder, reducing network pressure as the number of clients increases. Furthermore, to ensure reproducibility and flexibility for future experiments, we have incorporated the testing patterns for Federated Learning into the GitHub repository of the Frisbee platform [37]. This allows researchers to easily repeat and configure the experiments for different frameworks and datasets.

4.4. Monitoring Stack

Our testing environment incorporates a transparent monitoring stack (Figure 4) capable of extracting various utilization metrics from the system under test. These metrics encompass CPU, memory, network utilization, and FL-level metrics, including model accuracy/loss and training time per round (Table 1). To achieve this, we deploy a containerized monitoring agent on each node that follows a probe-based multi-threaded paradigm [38]. The agent, namely cAdvisor [39], collects performance metrics by initiating different probes for each sub-component, such as the cgroup probe and OS probe. It then exposes an http endpoint for Prometheus [40] to periodically retrieve and store the metrics in its embedded timeseries databases. Prometheus is integrated with Grafana [41] to provide real-time inspection of our experiments through its dashboard-as-a-service software.

As for the performance and FL monitoring metrics, the challenging part is to expose fine-grained metrics from the running FL workloads. To do that, we enhance the existing codebase of Flower to record accuracy, loss, and duration data into well-organized files, which are stored locally on each client. To capture performance metrics, we employ timers that intersect with the existing methods to extract the duration of training time. As for accuracy and loss, we extract these metrics from the FL master. At the end of the experiment, these files are automatically transferred into a central repository and combined with the performance metrics collected by Prometheus.

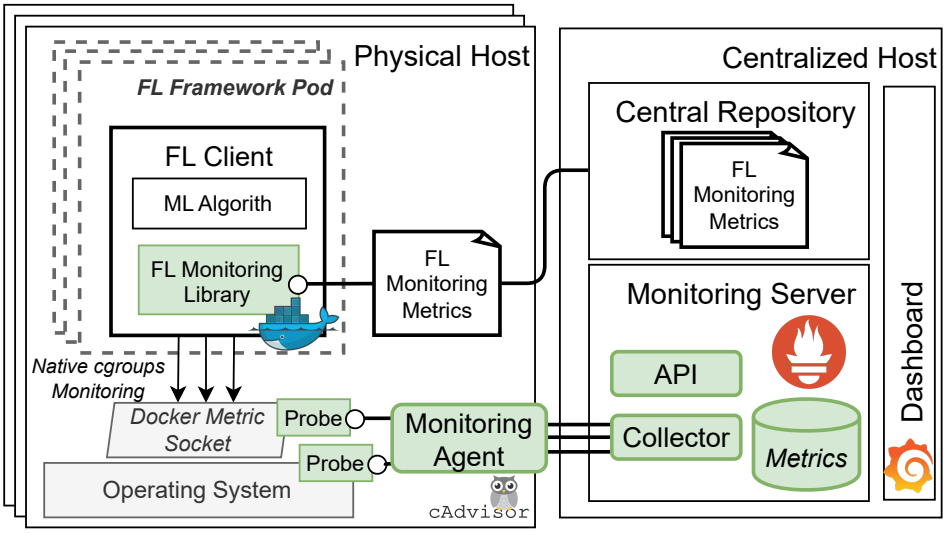


Figure 4. Monitoring stack for repeatable FL experimentation.

Listing 1. Snippet of a Frisbee scenario showing placement policies for groups of clients.

```
---
apiVersion: frisbee.dev/v1alpha1
kind: Scenario
metadata:
  name: node-placement
spec:
  actions:
    # Step 1: Create FedBed server
    - action: Service
      name: server
      service:
        templateRef: frisbee.apps.fedbed.server
        inputs:
          - { min_fit_clients: 20, min_available_clients: 20 }

    # Step 2: Place clients[0,4] to Node-1
    - action: Cluster
      name: group-a
      depends: { running: [ server ] }
      cluster:
        placement:
          nodes: [ k8s-node1 ]  # Change values here
        templateRef: frisbee.apps.fedbed.client
        inputs:
          - { fl_server: server,  total_nodes: 20, node_id: 0 }
          - { fl_server: server, total_nodes: 20, node_id: 1 }
          - { fl_server: server,  total_nodes: 20, node_id: 2 }
          - { fl_server: server, total_nodes: 20, node_id: 3 }
          - { fl_server: server, total_nodes: 20, node_id: 4 }

    # Step 2: Place clients[5,9] to Node-2
    - action: Cluster
      name: group-b
      depends: { running: [ server ] }
      cluster:
        placement:
          nodes: [ k8s-node2 ]  # Change values here
        templateRef: frisbee.apps.fedbed.client
        inputs:
          - { fl_server: server, total_nodes: 20, node_id: 5 }
          - { fl_server: server, total_nodes: 20, node_id: 6 }
          - { fl_server: server, total_nodes: 20, node_id: 7 }
          - { fl_server: server,  total_nodes: 20, node_id: 8 }
          - { fl_server: server,  total_nodes: 20, node_id: 9 }
```

Table 1. Experimental Testbed Metric Description.

Metric	Category	Description
`Accuracy`	FL/ML	The model accuracy per round
`Loss`	FL/ML	The model loss per round
`Round Duration`	Performance	The overall round duration
`Overall Duration`	Performance	The overall FL duration
`CPU Utilization`	Utilization	The CPU utilization of FL client or server
`Memory Utilization`	Utilization	The memory utilization of FL client or server
`Network I/O`	Utilization	The network data (both incoming and outgoing) of FL client or server in bytes
`Disk I/O`	Utilization	Disk I/O of FL client or server in bytes

4.5. ML Datasets for Experimentation

Driven by our motivating example of a healthcare provider in the field of imaging diagnostics (i.e., MRI-based research), we have based our evaluation on two widely recognized databases in computer vision: MNIST [18] and CIFAR-10 [19].

The fundamental difference between them lies in the complexity and characteristics of the images they contain. MNIST is a simple database that consists of grayscale images of handwritten digits. It includes 60,000 training images and 10,000 test images, each measuring 28 $\times$ 28 pixels. The dataset is divided into 10 different classes representing digits from 0 to 9. CIFAR10 is more complex and contains colored images of everyday objects like airplanes, cars, and animals. These images are larger, measuring 32 $\times$ 32 pixels, and contain three color channels (RGB). CIFAR-10 consists of 50,000 training images and 10,000 test images, spanning 10 distinct object classes.

Albeit both databases are common in the benchmarking of computer vision algorithms, the choice between MNIST and CIFAR-10 depends on the complexity of the task and the specific application scope. MNIST, with its simplicity and small image size, serves as a foundational dataset for evaluating and prototyping machine learning algorithms. It is commonly used to explore different classification techniques and benchmark the performance of new models. On the other hand, CIFAR-10 presents a more challenging task for algorithms due to its higher complexity and inclusion of color images. It serves as a stepping stone to more advanced computer vision tasks and is often used to assess the performance of deep learning architectures. Working with CIFAR-10 allows researchers to tackle the challenges posed by color information and gain insights into developing more robust and accurate models for image classification.

4.6. ML Backends and Models

PyTorch and TensorFlow are popular deep learning frameworks with distinct features capable of serving as the ML backend for FL deployments. Both libraries are open-source and utilize data flow graphs, where nodes represent mathematical operations and edges represent tensors (multi-dimensional arrays) carrying data. However, they differ in their programming interfaces. TensorFlow, developed by Google, initially adopted a declarative programming model with a static computational graph. On the other hand, PyTorch follows an imperative programming model, allowing computations to be defined and executed dynamically. Despite this distinction, both frameworks enable the seamless deployment of models on CPUs or GPUs without needing code modification.

To assess their performance on the MNIST dataset, we construct a Convolutional Neural Network (CNN) with connected layers using both PyTorch and TensorFlow. The network architecture includes multiple layers, with six trainable layers: two 2D convolutional layers, two 2D dropout layers, and two linear layers. Additionally, there are non-trainable

layers for activation and transformation operations, such as a ReLU activation layer after the first and second convolutional layers, a MaxPool2D layer following the second ReLU layer, and a Log-Softmax activation function for generating the final output. For CIFAR datasets, we implement the well-known MobileNetV2 model architecture in both TensorFlow and PyTorch. In brief, MobileNetV2 adopts a CNN model commonly used for image classification, comprising 53 layers. More details about the MobileNetV2 architecture can be found in [42].

4.7. Experimental Testbed

For the experimentation, we consider a bare-metal Kubernetes cluster with four server-grade physical nodes, each equipped with 24 cores over 2 Intel Xeon E5-2630v3 processors, 128 GB of DDR3 ECC main memory, and 250 GB of locally attached NVMe storage. The four nodes are connected via 1Gbps links to the top-of-rack switch. On top of the physical infrastructure, we deploy the rest of our end-to-end evaluation software stack, which operates as our main evaluation tool and automates the submission of FL systems, algorithms and ML models.

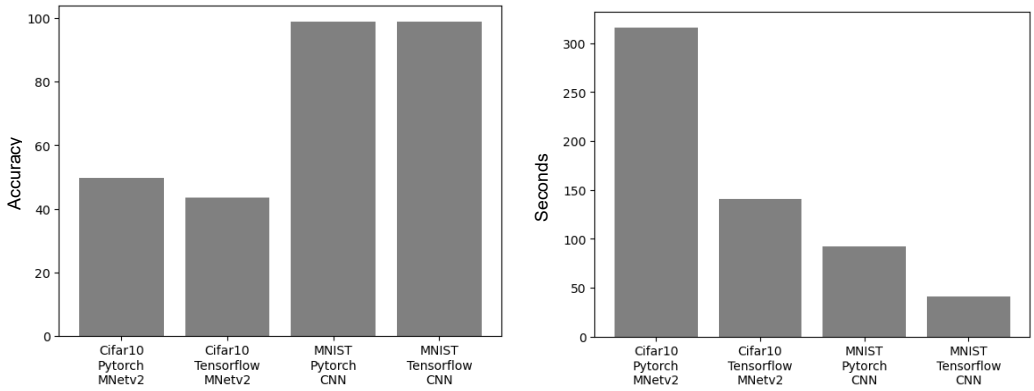
5. Case-Study: Resource Fitting

5.1. Experiment 1: Comparison of Native Performance of ML Backends

This experiment compares the performance and resource utilization of different machine learning (ML) libraries and models within a Federated Learning (FL) pipeline. We focus on TensorFlow and PyTorch as ML backends and evaluate their performance using the MNIST and CIFAR-10 datasets. We consider various factors such as model accuracy, completion time per training round, and resource utilization, including CPUs, memory, and network traffic. The FL workloads are executed for five rounds, utilizing an FL client and server with unrestricted resources. We determine the round duration by measuring the average duration of the last five rounds, while the accuracy of the fifth round serves as the representative value. To calculate CPU utilization we average the rate of change of cumulative CPU time consumed over a sampling window. Equally, we use the same formula for calculating the network throughput. In this case, the cumulative count of bytes that is transmitted and received. The term "window" refers to a specific time interval or data range over which the rate of change is calculated. To minimize the impact of monitoring on actual performance, we set the sampling interval to 15 s.

5.1.1. Training Round Duration and Model Accuracy

Firstly, we investigate if there is a significant variance in the duration of training rounds and the accuracy of the ML model, depending only on the dataset and ML backend. Figure 5a shows that both TensorFlow and PyTorch have comparable accuracy on the MNIST dataset, while PyTorch outperforms TensorFlow by 5% on the CIFAR-10 dataset. However, the average round duration varies significantly between the two ML backends and is an aspect for consideration (Figure 5b). Despite the 5% accuracy gain, training the MobileNetV2 model on PyTorch has a round duration of slightly over 5 min, while TensorFlow takes up to 140 s. Similarly, TensorFlow outperforms PyTorch, in terms of training round completion, by a large margin (around 60 s) for the CNN model on the MNIST dataset. Hence, *the absence of a dominant implementation across all cases, highlights the need to finely tune the ML backend and model architecture on a per-case basis, in order to improve the duration and accuracy of the FL pipeline.*

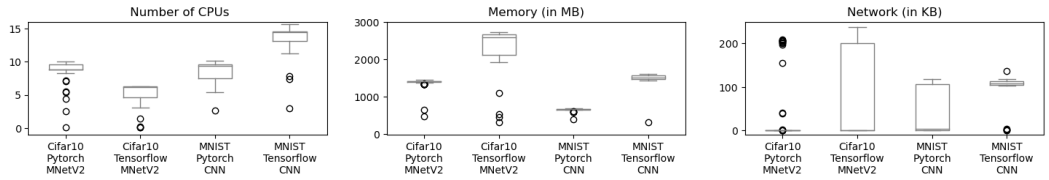


(**a**) (**b**)

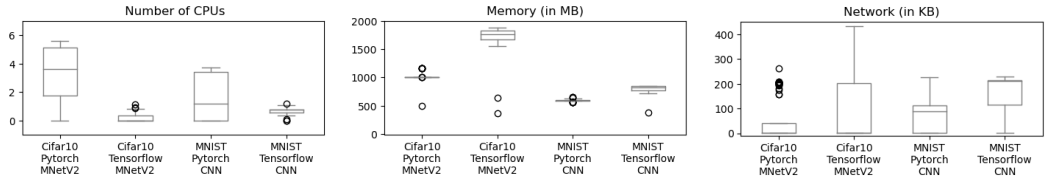
Figure 5. FL-level performance overview for different ML backends and datasets. (**a**) Model accuracy after 5 training rounds. (**b**) Mean training round duration after the completion of 5 rounds.

5.1.2. Resource Utilization for FL Clients and Server

Next, we elaborate on the reasons for the differences in the per-round time duration by analyzing the resource utilization of FL clients and servers for each ML backend. In Figure 6a, we compare the CPU utilization, memory allocation, and network traffic for different models, frameworks, and datasets on the client services. For clarity, we note that for the network, we account only for the outgoing traffic from the clients to filter noise, such as downloading the dataset. Interestingly, all examples utilize a comparable number of CPUs, with the only exception being the TensorFlow CNN model for the MNIST dataset. We attribute this behavior to the simple data layout of MNIST and the asynchronous capabilities of TensorFlow, which aims to load up the CPUs with as much work as possible and process a larger number of images in a parallel and vectorized fashion.



(**a**) Resource reservation as requested by PyTorch and TensorFlow clients.



(**b**) Resource reservation as requested by PyTorch and TensorFlow server.

Figure 6. Resource reservation for different Datasets/Framework/Models.

Moving now to the FL server resource consumption (Figure 6b), we first observe that PyTorch features a slightly higher CPU utilization than TensorFlow. Another observation is that TensorFlow consistently requires more memory and network bandwidth than PyTorch. This suggests that TensorFlow may not be suitable for FL training on extremely constrained IoT or edge devices. Additionally, the TensorFlow deployment exhibits higher memory utilization and network traffic on the server side than PyTorch, as shown in Figure 6b. These findings indicate that *TensorFlow-based FL pipelines are memory- and network-intensive, while PyTorch-based FL servers require more processing power*.

5.1.3. FL Network Usage Pattern

Although the previous remark effectively highlights the divergence in performance profiles between ML implementations, an important question arises regarding the dominance of recurrent or episodic events in these profiles. The network traffic patterns depicted in Figure 7a reveal that both ML backends exhibit recurring patterns, characterized by alternating periods of (a) relative inactivity, attributed to the initiation of local training on clients; and (b) high activity, attributed to the communication among clients and the server for the exchange of model updates and parameterization between rounds.

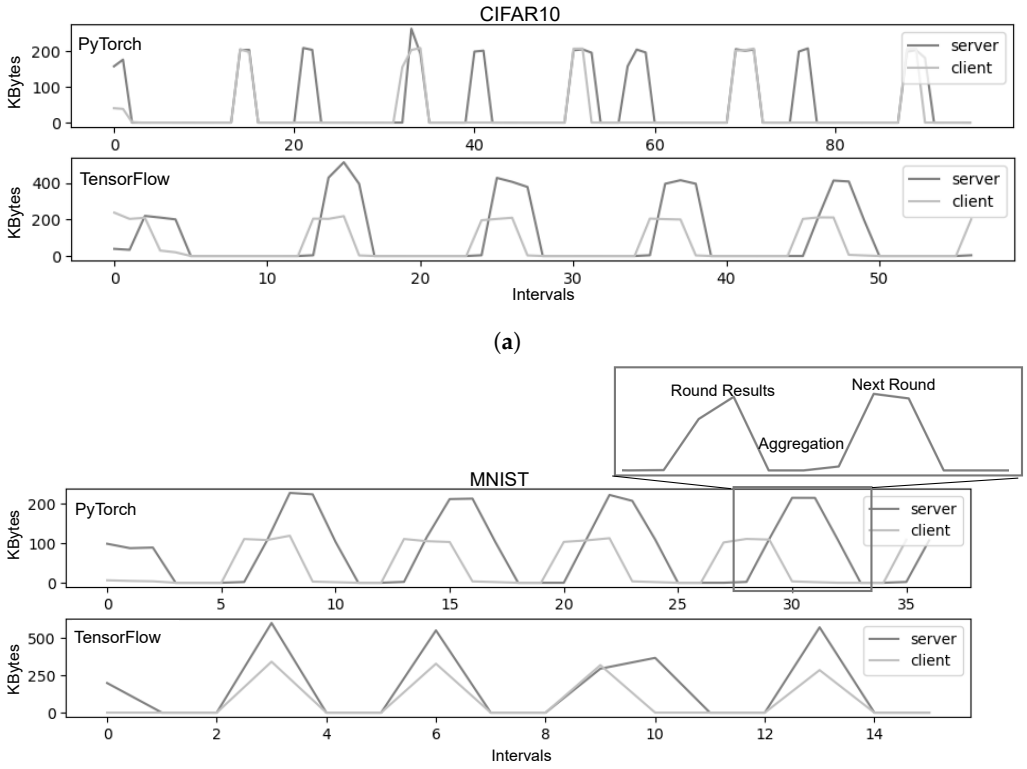


Figure 7. Network profiles profile for different framework/datasets/models. (**a**) Outbound Network Traffic for the CIFAR10 dataset. (**b**) Outbound Network Traffic for the MNIST dataset. The ratio between the sampling interval and the local processing can affect the visualization.

Specifically, after the initial model dissemination, the communication among clients and the server unfolds in three distinct steps. In the first step, the client transmits the local training results (model weights) to the server. In the second step, the server aggregates the outcomes from all clients and computes performance metrics (i.e., entropy) to assess if additional rounds are required. Finally, in the third step, the server sends the updated weights back to the clients to initiate a new round of local training.

As illustrated in the PyTorch plot of Figure 7b and in the zoom box of Figure 7b, the interval between the two peaks depends on the duration required for the server to aggregate and evaluate the results. A shorter duration results in the peaks appearing closer to each other. If the distance between peaks is less than twice the sampling interval, the synchronization step exhibits a "rectangular" shape. If the distance is less than one sampling interval, the peaks are calculated within the same interval, creating a "triangle" shape (as observed in TensorFlow-MNIST). In addition to the visualization artifacts, it is worth noting that *TensorFlow demonstrates steeper peaks and valleys compared to PyTorch, indicating larger communication messages and consequently higher demands for network bandwidth.*

5.2. Experiment 2: FL Clients' and Server Collocated vs. Isolated

In this experiment, we investigate the performance implications when collocating the FL server with the client services. For this, we explore two configurations: (i) the client and server share the same physical node and resources; and (ii) the client and server run on separate nodes. The purpose of this experiment is to establish a baseline for subsequent experiments that involve multiple clients. Our goal is to determine whether it is feasible to schedule the server on the same testbed node as the clients without affecting their performance, or if a dedicated physical node for the server is necessary.

5.2.1. Collocation Impact on Training Round Duration

We found that the difference in per-round duration between the co-located and isolated configurations was negligible. For TensorFlow, the co-located execution took about 40.7 s per round, while the isolated execution took 41.4 s per round, resulting in a mere 1.6% difference. Similarly, for PyTorch, the co-located execution took 90.8 s per round, and the isolated execution took 92.4 s per round, resulting in a 1.7% difference. Hence, *the duration of a training round is not impacted by the collocation of the FL server and clients*.

5.2.2. Collocation Impact on Resource Utilization

We also examined the resource utilization for the two setups. The results in Figure 8a present a similar negligible difference in compute and memory footprint. However, we observed a significant difference in network traffic for the TensorFlow framework. When the clients and server are deployed on different machines, the network throughput is higher and more stable than the collocated configuration. At the same time, we can notice a relationship between memory and network. The faster the network, the less the allocated memory. This relation can be attributed to buffering effects on the clients, who need to store packets locally before sending them to the server. Additionally, this observation suggests a correlation between TensorFlow as a network-intensive framework and the overhead of network virtualization in Kubernetes. In summary, our findings indicate that *the co-location and isolation of client and server do not significantly impact the deployment's performance or resource utilization metrics, except for the network traffic of network-intensive ML models/backends like TensorFlow.*

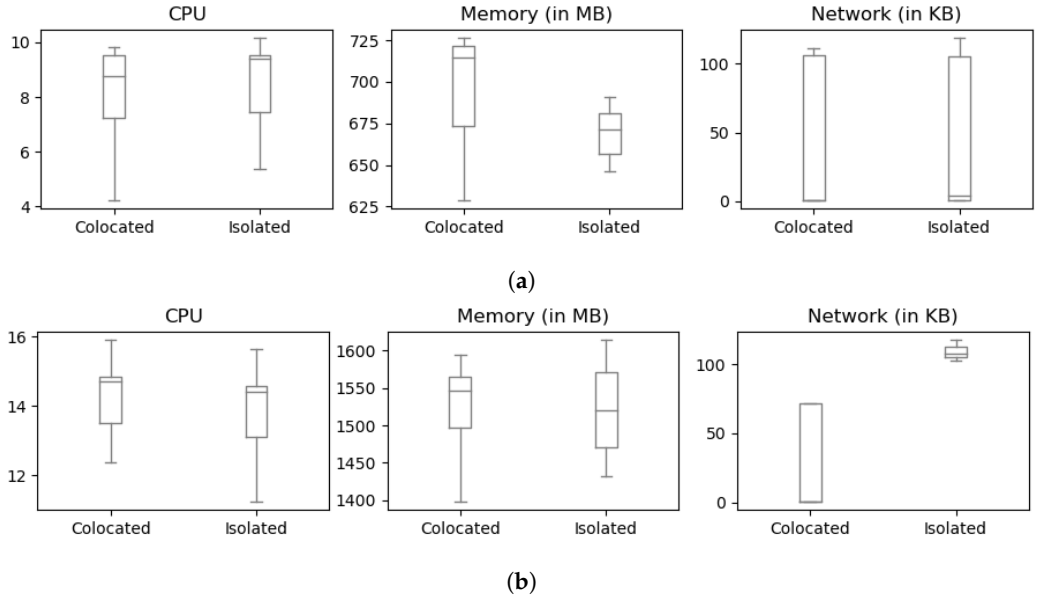


Figure 8. Analysis of collocations effects for PyTorch and TensorFlow. (**a**) Resource utilization for PyTorch client, in collocated and isolated setups. No apparent variation between the two setups. (**b**) Resource utilization for TensorFlow client, in collocated and isolated setups. Insignificant variation between the two setups.

5.3. Experiment 3: Performance of ML Backends under Constrained Resources

In previous experiments, we assessed the performance of FL deployments on fully allocated "server-grade" machines. In such a configuration, the containerized FL clients can use all of the CPU resources available on the node where the container is running. However, this scenario is uncommon at the edge, where devices have limited resources or share them among multiple applications. To understand how resource allocation can affect FL performance, we now focus on investigating FL deployments under different CPU allocation policies.

While resource management (reservation or limitation) can apply to other resources like memory and network, there are some distinctions. Memory management requires a static definition of the desired capacity, as dynamically shrinking or expanding memory can lead to application crashes (due to invalid memory addresses causing segmentation faults) or frequent swapping to disk, which harms overall performance. Network bandwidth can be limited, but end-to-end reservation is impossible without complete access to the network's switches and routers. Therefore, our investigation focuses solely on CPU adjustments, which can be made without affecting the application's correctness. For our FL deployment, we specifically consider the PyTorch CNN model applied to the MNIST dataset, and study the effects CPU policies may have on the training performance and the utilization of memory and network resources.

5.3.1. Resource Configurations for Containerized Services

Figure 9 presents the results of studying the round duration and average CPU utilization for different resource reservation policies. These policies represent various scenarios that a user may encounter in an FL deployment:

- *Native:* This configuration serves as the baseline, where no resource capping policies are applied. Because this represents what a non-expert user might consider first, we use it interchangeably with the term "Native performance".
- *Extreme over-provisioning:* Users aim for optimal training time without much concern for resource spending. This involves reserving more CPUs than what the ML frameworks require. For our setup, this configuration entails reserving 16 CPUs.
- *Best Resource Fitting:* This configuration reserves the number of CPUs that the ML framework can fully utilize without significantly extending the training time. In our setup, this is eight CPUs.
- *Resource Constrained Environments:* This configuration examines how FL frameworks perform in execution environments with limited resources. This can be motivated by cost reduction or resource-constrained edge devices (like single-board computers). The available CPUs for the client are limited to four or two.

We note that in all configurations, the accuracy remains consistent (as shown in Figure 5a) and thus, we omit the respective plots.

Starting from the Native and Extreme Over-Provisioning configurations, we observe that the FL workflow only utilizes nine CPUs, demonstrating that the training time matches the native performance in both cases (left plot). However, for Extreme Over-Provisioning, six CPUs remain unused (right plot). This highlights that FL/ML systems may not fully utilize available resources, leading to unnecessary resource spending. Next, we examine the Best Resource Fitting scenario. Compared to Extreme Over-Provisioning, this configuration reduces the number of required resources by 50% (from 16 to 8) but with a modest 22% increase in training time. Therefore, *it becomes crucial to develop a Kubernetes scheduling mechanism that dynamically adjusts the capping to match the native value of ML frameworks.*

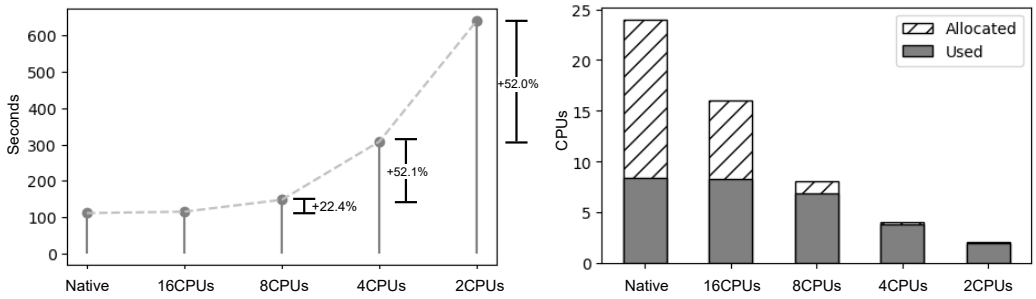


Figure 9. Correlation between training time (**left**) and resource allocation strategies (**right**).

Now, for Resource Constrained environments, there is a trade-off between increased training time and reduced processing power. Reserving four CPUs increases the training time by a factor of three (from 100 s to 300 s) but reduces the CPU allocation by a factor of eight compared to the native performance (from 24 to 4). In the extreme case of two CPUs, the training duration is doubled compared to four CPUs and is six times higher than the native performance. This indicates that *the training time increases inversely exponential to the available processing power*.

5.3.2. Inter-Winded resources

Figure 10 examines the impact of capped CPUs on memory and network metrics. In terms of memory, we observe minimal impact as the used memory remains unchanged. However, the network is significantly affected. While the total number of transferred bytes remains the same, the transmission of can vary significantly. In particular, the outliers are caused by the fewer reserved CPUs (four CPUs and two CPUs), resulting in increased per-round training time as now the processor becomes more strained due to handling both the computations and the network I/O coordination. We also observe a significant reduction in the number of transmitted packets when reservation (capping) policies are applied. In conclusion, *CPU capping can influence resource utilization of other uncapped resources, such as network traffic or packets.*

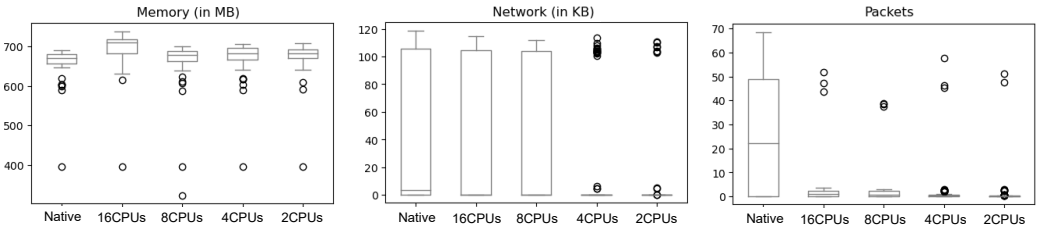


Figure 10. Correlation between CPU allocation strategies (x-axis) and utilization of memory and network resources (y-axis).

5.4. Experiment 4: Service Placement for Multi-Client Deployments

In this experiment, we build upon the previous experiment and investigate the FL performance using a single server and multiple clients. We employ 20 clients with no resource capping and evaluate how the distribution of clients across physical nodes can affect the performance of an FL deployment. As a baseline, we employ the default placement policy of Kubernetes, and compare it against a manual placement, where five client containers are fairly distributed to each physical node. For thoroughness, we repeat this experiment three times to assess the determinism of the Kubernetes placement strategy along with its impact to the predictability of the FL training time.

Default Placement Is Unbalanced and Unpredictable

Figure 11a depicts the distribution of client containers across the physical nodes of the Kubernetes cluster when adopting a manual placement compared to the default placement

strategy employed by Kubernetes. From this, we immediately observe that the default Kubernetes placement strategy exhibits two undesirable characteristics. First, it results in an uneven load distribution among nodes. Second, the distribution is unpredictable because clients may be assigned to different physical nodes during each execution. In contrast, the manual strategy ensures an equal and deterministic distribution of client containers among the physical nodes. It is important to note that with 20 clients and 4 physical nodes, one would expect the manual distribution to be split, with each node hosting 25% of the total clients. However, we see a <23.8%, 23.8%, 23.8%, 28.6%> split. The difference is attributed to the fact that 1 physical node must also run the FL server, effectively deploying 21 containers. As demonstrated in the experiment referenced in Section 5.2, the collocation of clients and the server does not impact overall performance.

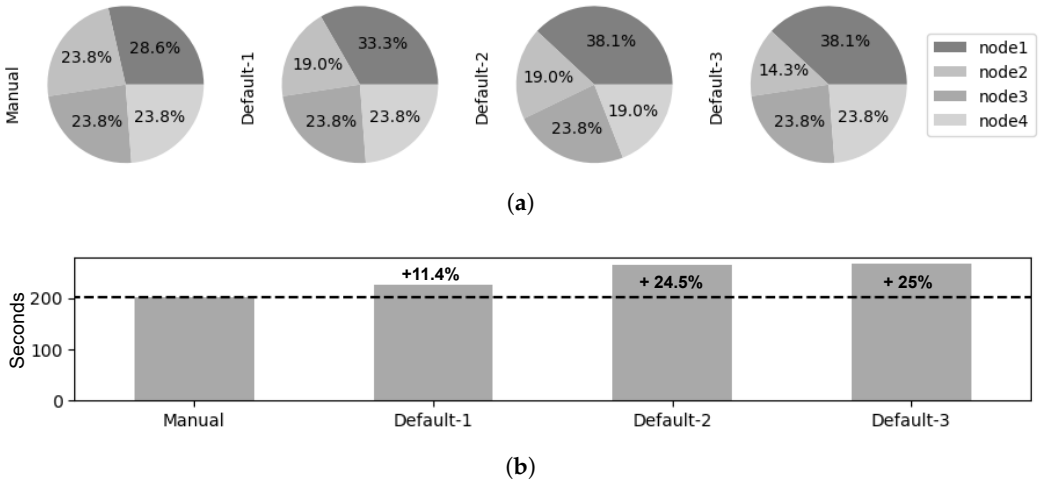


Figure 11. Training performance for different placement strategies. (**a**) Pod Placement strategies. The default strategy causes an unbalanced load among nodes. (**b**) FL round duration for different FL clients' placement strategies. The default strategy is non-deterministic and less performant than the manual strategy.

To assess the impact of the placement strategies on the FL training performance, we measure the duration of the FL rounds for each strategy, as depicted in Figure 11b. The manual strategy achieves the best performance, with a round duration of approximately 200 s. The default strategy performs worse than the manual strategy and is less deterministic. The first trial of the default strategy exhibits an 11.4% longer duration, while the subsequent trials (Default-2 and 3) with more uneven distributions increase the training time by about 25%. We observe that over-provisioned nodes, where multiple clients share limited resources, resulting in each client receiving fewer resources compared to clients on under-provisioned nodes. To further investigate this issue, we analyze the CPU profiles of the clients in Figure 12. The clients assigned through the manual strategy consume roughly the same CPU resources. In contrast, the default placement strategy of Kubernetes creates imbalances among the clients, which leads to straggling nodes [43] that become bottlenecks and prolong the training time. Examples of straggling clients are c1, c9, and c18 of the Default-3 run.

Based on our analysis, we draw two key takeaways: *(i) Evenly distributed FL clients contribute to improved performance (reduced training time) and predictable resource profiles of FL clients; (ii) the default Kubernetes scheduler creates an unbalanced FL deployment and may potentially introduce stragglers.*

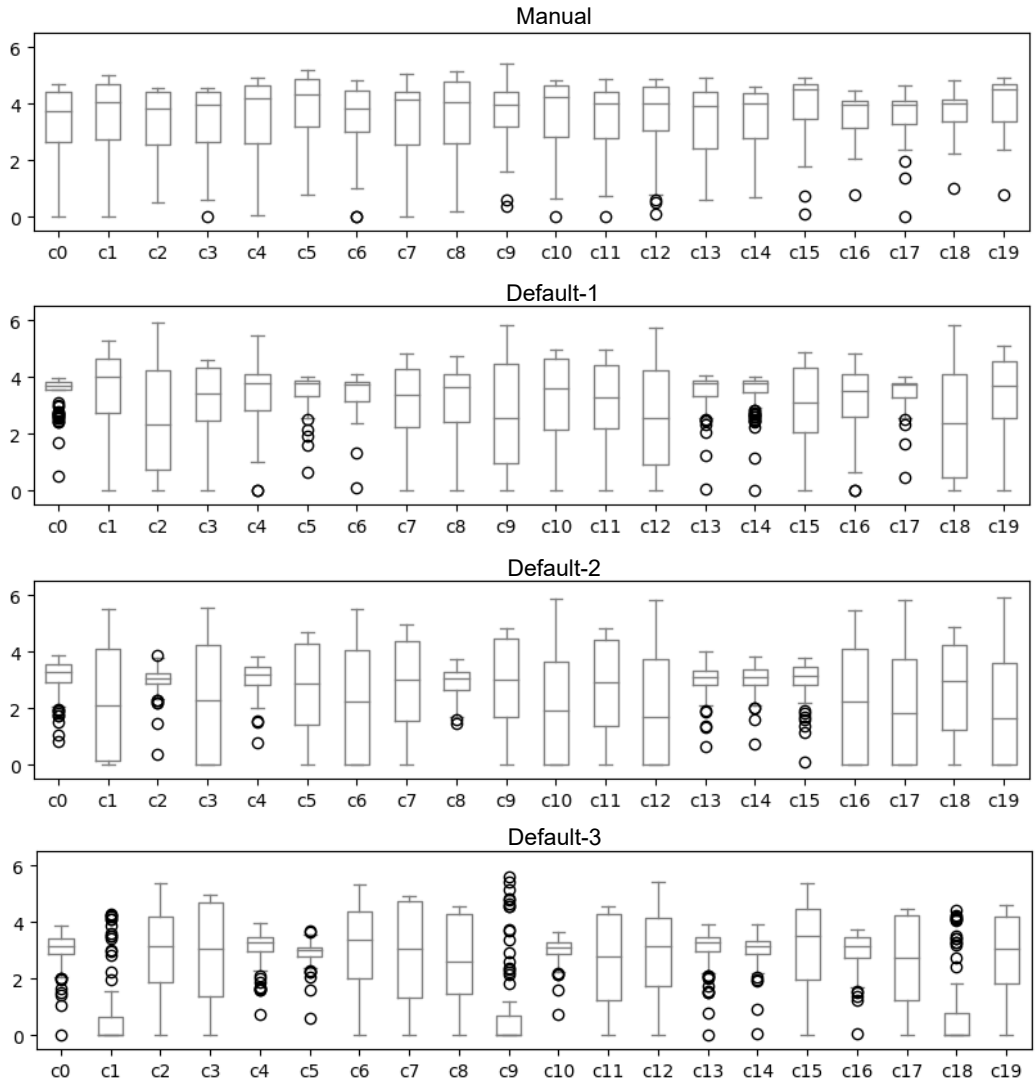


Figure 12. CPU utilization for different FL clients' (c0–c19) placement strategies. The default strategy is non-deterministic (different CPU per execution) and with significant variation among clients.

6. Case-Study: Parallel Workflows

6.1. Experiment 1: Evaluating Performance of Parallel Workflows

This experiment evaluates the overall performance when employing two FL workflows in parallel, meaning both are scheduled for execution concurrently on the same physical resources. By studying multiple parallel workflows instead of just one, we can differentiate factors specific to individual workflows from those with broader implications. Two different configurations are employed to conduct this experiment. At first, the workflows run unrestricted, allowing them to utilize their full native performance capacity. Next, each client is limited to two CPUs, thus enabling workflows with a larger number of clients while maintaining the training time within a reasonable timespan (as seen in Figure 6).

6.1.1. Enhanced Performance Stability with Capping

In the absence of resource capping, clients consume resources on a physical node arbitrarily, resulting in undesirable consequences. Firstly, the chaotic CPU usage (see Figure 13) leads to frequent context switching and cache invalidation at the operating system level, resulting in increased overheads and longer training times. Secondly, without explicit placement or resource requirement hints, the Kubernetes scheduler tries to maximize CPU utilization by fitting as many containers as possible onto a single physical machine.

However, as demonstrated in previous experiments, this approach creates a significantly imbalanced load among nodes. As a result, the performance profile becomes highly unstable (see Figure 14), making it challenging to differentiate between different training phases (e.g., local training and client-to-server communication) within the same workflow. On the other hand, *enforcing resource-capping policies brings about several benefits. It leads to more regular CPU utilization and improves the placement of containers on nodes, thereby enhancing the stability and performance of workflows.*

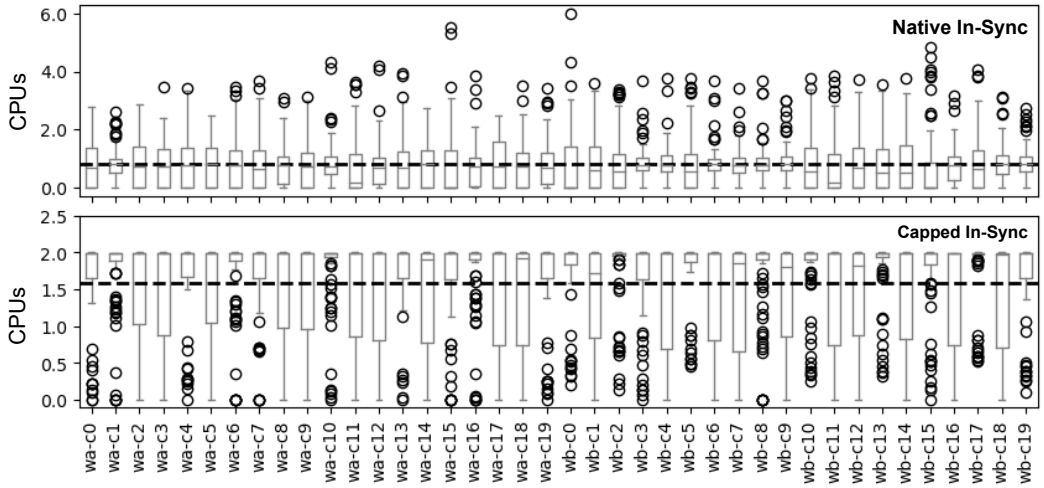


Figure 13. Average CPU utilization across all clients of parallel workloads. Properly capping the CPU can yield more predictable and stable utilization profiles.

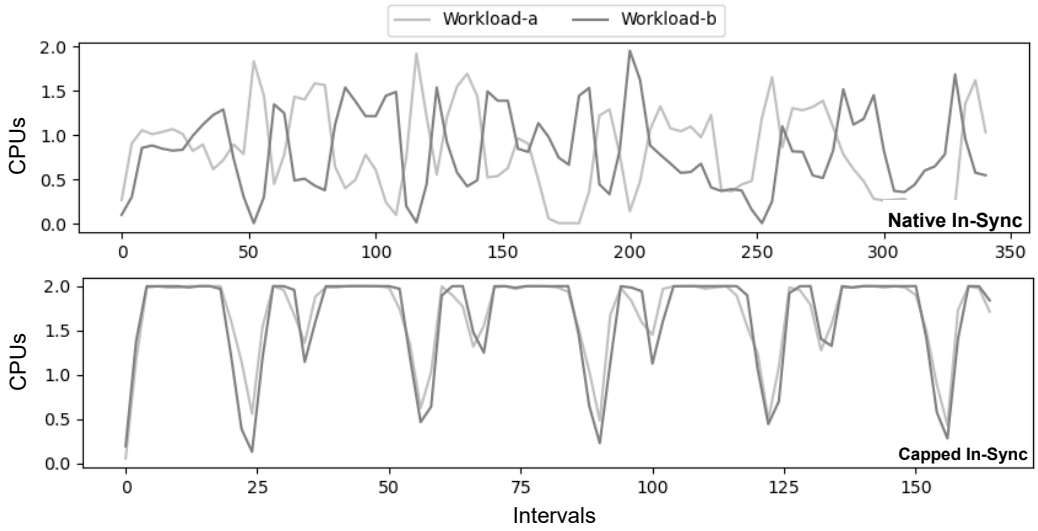


Figure 14. Per-client CPU utilization for parallel workloads.

6.1.2. Potential Deadlocks Caused by Capping

While resource capping offers benefits, there is a potential drawback in the context of Federated Learning: the possibility of deadlocks. This arises from the combination of three factors. First, training rounds commence only when all clients have joined the server. Second, resource reservation employs "thick-provisioning", allocating the full requested resources upon creation. Third, each workflow requires the total available resources from the node. The problem occurs when the shared physical resources become insufficient to serve both workflows simultaneously. If a node from workflow B interferes with the scheduling of workflow A, the resources on the node may deplete before all clients of workflow A are scheduled. Consequently, *the federated learning server will not dispatch jobs to*

the scheduled clients, rendering them idle or wasting resources, and mitigation of the deadlock risks needs careful resource management and allocation strategies.

6.2. Experiment 2: Analyzing Network Behavior of Parallel Workflows

In Figure 14 we observe that, when two capped workflows are started simultaneously, their performance patterns align perfectly, resulting in synchronized peaks and valleys. However, the synchronization of communication phases (valleys) doubles the requirements for network bandwidth, which is ironic given that the network remains largely idle for most of the training time. Unfortunately, dividing network bandwidth is more challenging than dividing CPUs or memory because network operations are influenced by external factors such as congestion and latency.

In this experiment, we investigate the possibility of reducing peak demand for bandwidth by spreading out the network operations over time. To do so, we examine the behavior of parallel workflows in two scenarios: (i) simultaneous execution, denoted as *in-sync*; and (ii) the execution with a time offset, denoted as *shifted*. These scenarios are compared against a baseline where workflows are started simultaneously without any capping (*native*). We note that to determine an optimal time offset, we leverage the predictable performance profile of capped workflows and attempt to find an offset that minimizes overlap during communication phases. In our setup, a time offset of 3 min proved to be near-optimal.

Predicting Workload Shifts for Reduced Bandwidth

Figure 15 demonstrates the highly predictable performance profiles of capped workflows, with actual CPU utilization closely matching the theoretical estimation. Furthermore, Figure 16 reveals that by carefully scheduling the start times of workflows, we can achieve the same training time while utilizing only half of the network bandwidth compared to the in-sync execution. Though somewhat arbitrary, the 3 min offset between workflows effectively reduces bandwidth requirements by minimizing overlap during communication phases. Consequently, *an important area for future investigation involves exploring techniques to determine the optimal shift between workflows, aiming to further minimize communication phase overlaps.*

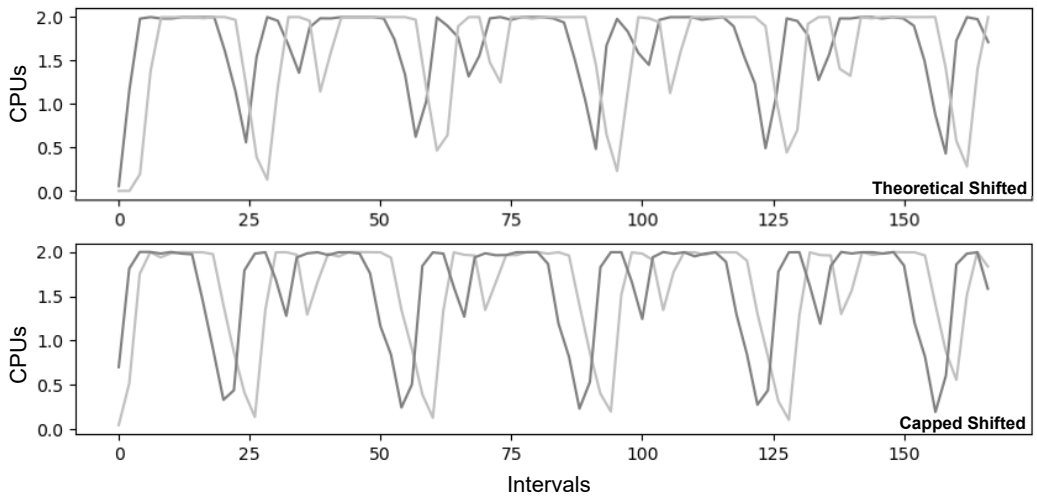


Figure 15. Performance prediction and performance upon actual deployments. Federated learning workloads can be highly predictable.

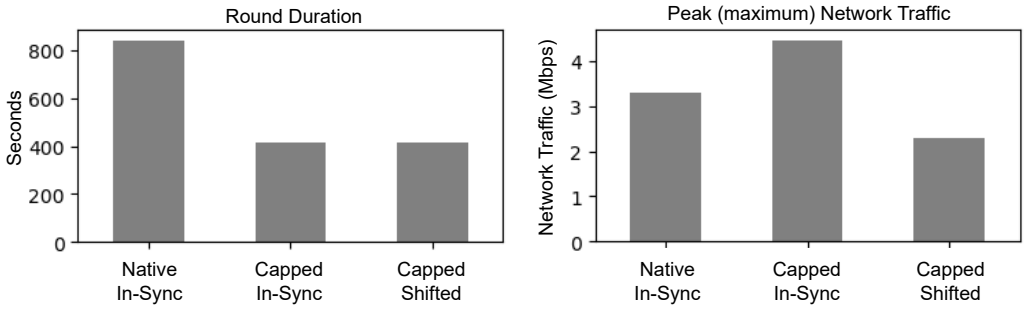


Figure 16. Duration per round and Peak network traffic of FL Servers. Capped and Shifted workloads have smaller round duration, with shifted workloads needing less bandwidth to be optimal.

7. Related Work

Due to its large-scale distributed nature, Federated learning (FL) presents unique challenges in creating FL orchestration systems including the minimization of communication cost, node heterogeneity, and unreliable network links [2,9]. Evaluating FL system performance requires stress testing the infrastructure with representative workloads and analyzing a wide range of quantitative and qualitative performance metrics [44].

Several FL frameworks exist in the ML community. LEAF offers FL training workloads and datasets [45], while FedML [46] and Flower [47] provide interoperability and extensibility across ML models. However, these frameworks often overlook the implications related to computing systems. Simulation-based studies like FedScale [17] examine the system-based performance of FL at scale by simulating realistic data partitioning approaches, network delays, and resource constraints. This approach, however, is only effective for theoretical estimations rather than benchmarking actual implementations. Oppositely, studies like EdgeTB [16] combine distributed learning techniques with virtualized testbed but do not explore the full range of implications related to FL and ML frameworks, such as resource fitting or parallel FL workflow execution. As a result, no evaluation framework currently examines resource fitting and co-location of FL workloads in cloud and edge environments while providing configurable workloads and datasets.

Performance profiling on cloud-based environments with virtualized resources has been explored from various perspectives, such as resource allocation, workload co-location, and interference analysis in cloud data centers [48–50]. Existing efforts often extract traces from entire data centers without considering the performance of specific applications. PANIC focuses on extracting insights from complex applications and creating their resource profiles [51]. BenchPilot [44] automates the deployment and evaluation of big data streaming analytics frameworks on edge data centers, aiming for repeatability and reproducibility. Unfortunately, these frameworks do not provide specialized functionalities for FL workloads and primarily focus on general-purpose applications running on virtualized cloud or edge resources.

Another aspect of research involves extracting performance insights from co-located FL workloads and analyzing how native cloud-based orchestrators place FL services in virtualized environments. Chen et al. [52] analyze interference impact on hardware and application-level behaviors in co-located online and offline workloads on the cloud. Researchers also try to exploit this information to create more efficient workload schedulers. Frameworks like Perph [53] and Perphon [54] employ ML-based methods for performance prediction and resource inference, while CTCL scheduler [55] aims to improve resource utilization and minimize task evictions in containerized cloud computing environments. While these papers contribute to resource allocation, workload co-location, and interference analysis in cloud data centers, they lack a specific focus on FL pipelines.

8. Key Takeaways and Future Work

The article presented a comprehensive and reproducible experimentation process that aimed to understand the relationship between deployment configurations and the

performance of Federated Learning workflows. The study integrated Kubernetes for cluster orchestration and Flower for distributed training, both of which are widely used open-source frameworks. The benchmarking analysis focused on assessing the performance of the default scheduler with and without user hints regarding the specific requirements of a Federated Learning deployment. We experimented using two popular ML/FL datasets (MNIST, CIFAR10) and two ML backends (PyTorch, TensorFlow).

Our experiments provided many takeaways that we divide into three classes:

1. Native Performance of ML Frameworks:
    - TensorFlow requires more memory and network resources per FL training round, but demands less computational processing power compared to PyTorch.
    - The performance and accuracy of these frameworks can vary depending on the model architecture and dataset.
2. Performance over Constrained Resources:
    - Training time increases exponentially as processing power decreases.
    - Implementing resource-capping policies can improve workflow stability, but CPU capping may impact the utilization metrics of other uncapped resources like network traffic.
    - Resource homogeneity leads to faster training times and more predictable resource utilization.
3. Scheduling across Multiple Nodes:
    - Using the default Kubernetes scheduler without user hints can result in imbalanced deployment and introduce stragglers in a Federated Learning deployment.
    - Deadlocks can occur when multiple concurrent training workflows have resource-capping constraints.
    - Introducing time skew among concurrent workflows can significantly reduce network bandwidth requirements.
    - Co-locating clients and servers does not affect performance and resource utilization.

Based on the aforementioned findings, it is clear that local optimization alone, whether at the FL framework or orchestrator level, is insufficient. To achieve globally optimal decision-making, information exchange across layers is necessary. For example, FL framework client-selection algorithms should consider Kubernetes information, while Kubernetes should incorporate FL-level metrics like entropy loss and completion time. Furthermore, since FL involves geo-distributed deployments with unpredictable runtime conditions, continuous updating of information is crucial. Fortunately, FL exhibits highly predictable performance patterns that can reduce the need for continuous profiling and enhance deployment performance.

Building upon these insights, our future work focuses on two main objectives. Firstly, we aim to develop a recommendation service that provides (near-) optimal scheduling policies and configurations for federated learning. This service will be particularly valuable for novice users, offering a solid starting point to minimize resource wastage and prevent resource starvation and contention in the presence of multiple workloads. Additionally, our second goal is to create a runtime module specifically designed for profiling the current state of a dedicated FL cluster. This module will dynamically adjust configurations based on current demand and user-defined policies. By leveraging data from ongoing runs, we can fine-tune the selected parameterization to ensure efficient resource allocation and meet the evolving requirements of the FL environment. We aspire that such an adaptive approach will ultimately lower the adoption barrier for FL by reducing the time and monetary efforts involved in the design, testing, and runtime management.

Author Contributions: All three authors contributed equally towards the Conceptualization and Methodology. The Experimentation and Validation was lead by F.N., Software design was lead equally by F.N. and M.S., while the Project Management was conducted by D.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: In order to foster reproducibility and enhance flexibility for upcoming experiments, we have openly shared the testing patterns we devised for this paper on the GitHub repository of Frisbee [37]. This ensures that researchers and practitioners can access and utilize these patterns to replicate our results and explore further variations in their own work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. McMahan, B.; Moore, E.; Ramage, D.; Hampson, S.; y Arcas, B.A. Communication-efficient learning of deep networks from decentralized data. In Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS) 2017, Fort Lauderdale, FL, USA, 20–22 April 2017; pp. 1273–1282.
2. Zhang, C.; Xie, Y.; Bai, H.; Yu, B.; Li, W.; Gao, Y. A survey on federated learning. *Knowl. Based Syst.* **2021**, *216*, 106775. [CrossRef]
3. Li, L.; Fan, Y.; Tse, M.; Lin, K.Y. A review of applications in federated learning. *Comput. Ind. Eng.* **2020**, *149*, 106854. [CrossRef]
4. The European General Data Protection Regulation (EU 2016/67). *Off. J. Eur. Union* **2016**, *L 119*, 1–88.
5. Truong, N.; Sun, K.; Wang, S.; Guitton, F.; Guo, Y. Privacy preservation in federated learning: An insightful survey from the GDPR perspective. *Comput. Secur.* **2021**, *110*, 102402. [CrossRef]
6. Flower: A Friendly Federated Learning Framework. Available online: https://flower.dev/ (accessed on 24 July 2023).
7. Bonawitz, K.A.; Eichner, H.; Grieskamp, W.; Huba, D.; Ingerman, A.; Ivanov, V.; Kiddon, C.; Konečný, J.; Mazzocchi, S.; McMahan, H.B.; et al. Towards Federated Learning at Scale: System Design. *arXiv* **2019**, arXiv:1902.01046.
8. Liu, R.; Wu, F.; Wu, C.; Wang, Y.; Lyu, L.; Chen, H.; Xie, X. No one left behind: Inclusive federated learning over heterogeneous devices. In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, 14–18 August 2022; pp. 3398–3406.
9. Xia, Q.; Ye, W.; Tao, Z.; Wu, J.; Li, Q. A survey of federated learning for edge computing: Research problems and solutions. *High Confid. Comput.* **2021**, *1*, 100008. [CrossRef]
10. Symeonides, M.; Trihinas, D.; Georgiou, Z.; Pallis, G.; Dikaiakos, M. Query-driven descriptive analytics for IoT and edge computing. In Proceedings of the 2019 IEEE International Conference on Cloud Engineering (IC2E), Prague, Czech Republic, 24–27 June 2019; pp. 1–11.
11. Production-Grade Container Orchestration. Available online: https://kubernetes.io/ (accessed on 24 July 2023).
12. Nikolaidis, F.; Marazakis, M.; Bilas, A. IOTier: A Virtual Testbed to evaluate systems for IoT environments. In Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Melbourne, Australia, 10–13 May 2021; pp. 676–683.
13. Brewer, E.A. Kubernetes and the path to cloud native. In Proceedings of the Sixth ACM Symposium on Cloud Computing, Kohala Coast, HI, USA, 27–29 August 2015; p. 167.
14. Al-Dhuraibi, Y.; Paraiso, F.; Djarallah, N.; Merle, P. Autonomic vertical elasticity of docker containers with elasticdocker. In Proceedings of the 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, 25–30 June 2017; pp. 472–479.
15. Arunarani, A.; Manjula, D.; Sugumaran, V. Task scheduling techniques in cloud computing: A literature survey. *Future Gener. Comput. Syst.* **2019**, *91*, 407–415. [CrossRef]
16. Yang, L.; Wen, F.; Cao, J.; Wang, Z. EdgeTB: A Hybrid Testbed for Distributed Machine Learning at the Edge with High Fidelity. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 2540–2553. [CrossRef]
17. Lai, F.; Dai, Y.; Singapuram, S.; Liu, J.; Zhu, X.; Madhyastha, H.; Chowdhury, M. FedScale: Benchmarking Model and System Performance of Federated Learning at Scale. In Proceedings of the 39th International Conference on Machine Learning, Baltimore, MD, USA, 17–23 July 2022; Volume 162, pp. 11814–11827.
18. THE MNIST DATABASE of Handwritten Digits. Available online: http://yann.lecun.com/exdb/mnist/ (accessed on 24 July 2023).
19. The CIFAR-10 Dataset. Available online: https://www.cs.toronto.edu/~kriz/cifar.html (accessed on 24 July 2023).
20. PyTorch. Available online: https://pytorch.org/ (accessed on 24 July 2023).
21. TensorFlow: An End-to-End Machine Learning Platform. Available online: https://www.tensorflow.org/ (accessed on 24 July 2023).
22. Konečnỳ, J.; McMahan, B.; Ramage, D. Federated optimization: Distributed optimization beyond the datacenter. *arXiv* **2015**, arXiv:1511.03575.
23. Konečnỳ, J.; McMahan, H.B.; Ramage, D.; Richtárik, P. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv* **2016**, arXiv:1610.02527.
24. McMahan, B.; Ramage, D. Federated Learning: Collaborative Machine Learning without Centralized Training Data. 2017. Available online: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html (accessed on 24 July 2023).
25. Gadekallu, T.R.; Pham, Q.V.; Huynh-The, T.; Bhattacharya, S.; Maddikunta, P.K.R.; Liyanage, M. Federated Learning for Big Data: A Survey on Opportunities, Applications, and Future Directions. *arXiv* **2021**, arXiv:2110.04160.
26. Li, T.; Sahu, A.K.; Talwalkar, A.; Smith, V. Federated Learning: Challenges, Methods, and Future Directions. *IEEE Signal Process. Mag.* **2020**, *37*, 50–60. [CrossRef]

27. Karimireddy, S.P.; Kale, S.; Mohri, M.; Reddi, S.J.; Stich, S.U.; Suresh, A.T. SCAFFOLD: Stochastic Controlled Averaging for Federated Learning. *arXiv* **2021**, arXiv:1910.06378.
28. Acar, D.A.E.; Zhao, Y.; Navarro, R.M.; Mattina, M.; Whatmough, P.N.; Saligrama, V. Federated Learning Based on Dynamic Regularization. *arXiv* **2021**, arXiv:2111.04263.
29. Cho, Y.J.; Wang, J.; Joshi, G. Client Selection in Federated Learning: Convergence Analysis and Power-of-Choice Selection Strategies. *arXiv* **2020**, arXiv:2010.01243.
30. Nishio, T.; Yonetani, R. Client Selection for Federated Learning with Heterogeneous Resources in Mobile Edge. In Proceedings of the ICC 2019—2019 IEEE International Conference on Communications (ICC), Shanghai, China, 20–24 May 2019.
31. Li, T.; Sanjabi, M.; Beirami, A.; Smith, V. Fair resource allocation in federated learning. *arXiv* **2019**, arXiv:1905.10497.
32. Symeonides, M.; Georgiou, Z.; Trihinas, D.; Pallis, G.; Dikaiakos, M.D. Fogify: A fog computing emulation framework. In Proceedings of the 2020 IEEE/ACM Symposium on Edge Computing (SEC), San Jose, CA, USA, 12–14 November 2020; pp. 42–54.
33. Dockerfile Reference. Available online: https://docs.docker.com/engine/reference/builder/ (accessed on 24 July 2023).
34. Kubernetes Components. Available online: https://kubernetes.io/docs/concepts/overview/components/ (accessed on 24 July 2023).
35. Nikolaidis, F.; Chazapis, A.; Marazakis, M.; Bilas, A. Frisbee: Automated testing of Cloud-native applications in Kubernetes. *arXiv* **2021**, arXiv:2109.10727.
36. Nikolaidis, F.; Chazapis, A.; Marazakis, M.; Bilas, A. Event-Driven Testing For Edge Applications. *arXiv* **2022**, arXiv:2212.12370.
37. Testing Patterns for Federated Learning Deployments. Available online: https://github.com/CARV-ICS-FORTH/frisbee/tree/main/examples/patterns/federated-learning (accessed on 24 July 2023).
38. Trihinas, D.; Pallis, G.; Dikaiakos, M.D. Monitoring Elastically Adaptive Multi-Cloud Services. *IEEE Trans. Cloud Comput.* **2018**, *6*, 800–814. [CrossRef]
39. cadvisor: Analyzes Resource Usage and Performance Characteristics of Running Containers. Available online: https://github.com/google/cadvisor (accessed on 24 July 2023).
40. Prometheus: From Metrics to Insight. Available online: https://prometheus.io (accessed on 24 July 2023).
41. Grafana: Compose and Scale Observability with One or All Pieces of the Stack. Available online: https://grafana.com/ (accessed on 24 July 2023).
42. Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *arXiv* **2019**, arXiv:1801.04381.
43. Ananthanarayanan, G.; Ghodsi, A.; Shenker, S.; Stoica, I. Effective Straggler Mitigation: Attack of the Clones. In Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), Lombard, IL, USA, 2–5 April 2013; pp. 185–198.
44. Georgiou, J.; Symeonides, M.; Kasioulis, M.; Trihinas, D.; Pallis, G.; Dikaiakos, M.D. BenchPilot: Repeatable & Reproducible Benchmarking for Edge Micro-DCs. In Proceedings of the 2022 IEEE Symposium on Computers and Communications (ISCC), Rhodes, Greece, 30 June–3 July 2022; pp. 1–6. [CrossRef]
45. Caldas, S.; Duddu, S.M.K.; Wu, P.; Li, T.; Konečný, J.; McMahan, H.B.; Smith, V.; Talwalkar, A. LEAF: A Benchmark for Federated Settings. *arXiv* **2018**, arXiv:1812.01097. [CrossRef].
46. He, C.; Li, S.; So, J.; Zeng, X.; Zhang, M.; Wang, H.; Wang, X.; Vepakomma, P.; Singh, A.; Qiu, H.; et al. FedML: A Research Library and Benchmark for Federated Machine Learning. *arXiv* **2020**, arXiv:2007.13518. [CrossRef].
47. Beutel, D.J.; Topal, T.; Mathur, A.; Qiu, X.; Parcollet, T.; Lane, N.D. Flower: A Friendly Federated Learning Research Framework. *arXiv* **2020**, arXiv:2007.14390.
48. Sharma, P.; Chaufournier, L.; Shenoy, P.; Tay, Y.C. Containers and Virtual Machines at Scale: A Comparative Study. In Proceedings of the 17th International Middleware Conference, New York, NY, USA, 12–16 December 2016. [CrossRef]
49. Jiang, C.; Qiu, Y.; Shi, W.; Ge, Z.; Wang, J.; Chen, S.; Cérin, C.; Ren, Z.; Xu, G.; Lin, J. Characterizing Co-Located Workloads in Alibaba Cloud Datacenters. *IEEE Trans. Cloud Comput.* **2022**, *10*, 2381–2397. [CrossRef]
50. Gao, J.; Wang, H.; Shen, H. Machine Learning Based Workload Prediction in Cloud Computing. In Proceedings of the 2020 29th International Conference on Computer Communications and Networks (ICCCN), Honolulu, HI, USA, 3–6 August 2020; pp. 1–9. [CrossRef]
51. Giannakopoulos, I.; Tsoumakos, D.; Papailiou, N.; Koziris, N. PANIC: Modeling Application Performance over Virtualized Resources. In Proceedings of the 2015 IEEE International Conference on Cloud Engineering, Tempe, AZ, USA, 9–13 March 2015, pp. 213–218. [CrossRef]
52. Chen, W.; Ye, K.; Xu, C.Z. Co-Locating Online Workload and Offline Workload in the Cloud: An Interference Analysis. In Proceedings of the 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), Zhangjiajie, China, 10–12 August 2019; pp. 2278–2283. [CrossRef].
53. Zhu, J.; Yang, R.; Hu, C.; Wo, T.; Xue, S.; Ouyang, J.; Xu, J. Perph: A Workload Co-location Agent with Online Performance Prediction and Resource Inference. In Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Melbourne, Australia, 10–13 May 2021; pp. 176–185. [CrossRef]

54. Zhu, J.; Yang, R.; Hu, C.; Wo, T.; Xue, S.; Ouyang, J.; Xu, J. Perphon: A ML-Based Agent for Workload Co-Location via Performance Prediction and Resource Inference. In Proceedings of the SoCC '19 ACM Symposium on Cloud Computing, New York, NY, USA, 20–23 November 2019; p. 478. [CrossRef]
55. Zhong, Z.; He, J.; Rodriguez, M.A.; Erfani, S.; Kotagiri, R.; Buyya, R. Heterogeneous Task Co-location in Containerized Cloud Computing Environments. In Proceedings of the 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC), Nashville, TN, USA, 19–21 May 2020; pp. 79–88. [CrossRef]