FedBed: Benchmarking Federated Learning over Virtualized Edge Testbeds

Moysis Symeonides
University of Cyprus
Cyprus
msymeo03@ucy.ac.cy

Fotis Nikolaidis
FORTH-ICS
Greece
fnikol@ics.forth.gr

Demetris Trihinas
University of Nicosia
Cyprus
trihinas.d@unic.ac.cy

George Pallis
University of Cyprus
Cyprus
pallis@ucy.ac.cy

Marios D. Dikaiakos
University of Cyprus
Cyprus
mdd@ucy.ac.cy

Angelos Bilas
FORTH-ICS
Greece
bilas@ics.forth.gr

Abstract

Federated Learning has become the de facto paradigm for training AI models under a distributed modality where the computational effort is spread across several clients without sharing local data. Despite its distributed nature, enabling FL in an Edge-Cloud continuum is challenging with resource and network heterogeneity, different AI models and libraries, and non-uniform data distributions, all hampering QoS and limiting innovation potential. This work introduces FedBed, a testing framework that enables the rapid and reproducible benchmarking of FL deployments on virtualized testbeds. FedBed aids users in assessing the numerous trade-offs that result from combining a variety of FL software and infrastructure configurations in Edge-Cloud settings. This reduces the time-consuming process that includes the setup of either a virtual physical or emulation testbed, experiment configurations, and the monitoring of the resulting FL testbed.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → *Distributed artificial intelligence.*

Keywords: Federated Learning, Edge Computing

ACM Reference Format:
Moysis Symeonides, Fotis Nikolaidis, Demetris Trihinas, George Pallis, Marios D. Dikaiakos, and Angelos Bilas. 2023. FedBed: Benchmarking Federated Learning over Virtualized Edge Testbeds. In *Proceedings of IEEE/ACM UCC 2023 (UCC'23).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

UCC'23, December 2023, Taormina, Italy

© 2023 Association for Computing Machinery.

ACM ISBN xxx-xxx-xx-xxx/YY/MM. . . $15.00
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Federated Learning (FL) is transforming the realm of Artificial Intelligence (AI) by enabling collaborative model training among multiple clients in a distributed manner [15]. With FL, geo-dispersed and sensitive data such as personal activity, bio-signals and financial records remain localized and unexposed to the other collaborators during training [11]. FL embraces the benefits of Edge Computing, by training models at the data origins, reducing data movement and leveraging a central entity, the FL server, solely for the training coordination and the interim aggregation of model weights [7].

Tools such as Flower [2] and FATE [14] contribute to the democratization of FL reducing the entry barrier by handling the coordination of the distributed training. Still, migrating FL to the Edge presents significant challenges that FL tools cannot address alone. In edge settings, resource heterogeneity is normal as FL clients may run on diverse hardware with varying capabilities. In addition, network connectivity may fluctuate and significantly impair QoS. These infrastructure challenges can lead to performance bottlenecks not originally envisioned in the FL process. However, FL features several knobs for optimization, which range from parameters of the distributed training (i.e., termination clause, aggregation algorithm) to the ML backend and model architecture as well as dataset partitioning. For example, PyTorch as an ML backend is more compute-hungry than TensorFlow, while the latter has a higher memory footprint and consumes more network bandwidth [17]. In turn, opting for a deep neural network can reduce model loss but requires excessive compute resources in contrast to less complex but more loss-prone models [22]. Hence, thoroughly testing multiple FL settings is of paramount importance to improve not only QoS but also to reduce resource waste and monetary costs.

Recent works propose a diverse solution set for FL optimization. Yet, they fall short when it comes to experimental evaluation. For example, many focus on providing realistic federated datasets or models with reference benchmark metrics, albeit experimentation is usually limited to a single physical node [3]. Likewise, works for FL in IoT settings,

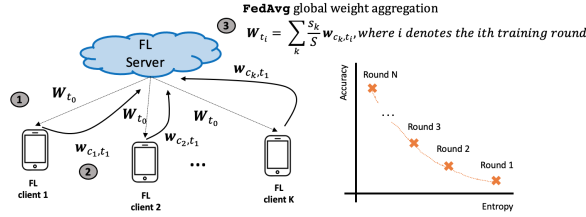


Figure 1. High-Level Overview of Federated Learning

are usually limited to a handful of Raspberry Pis [7]. Finally, tools, like FL simulators [5] or emulators [24], are dependent on specific use-cases or provide only FL-level evaluation (i.e., model loss, statistical metrics) without considering the implications of the actual underlying resources to the FL process.

This leads us to the focal point of our work: rapidly designing FL testbeds and conducting realistic experiments. Several challenges manifest in this vision, including the high cost involved in the design of an appropriate testbed, the time-consuming process of selecting realistic data distributions for FL and configuring resource profiles for compute nodes and networking. In turn, the choice of the ML backend can impact performance metrics such as accuracy and training duration. As a result, FL practitioners invest time in developing and assessing multiple ML models. To achieve intelligence at the Edge Continuum, it is necessary to carry out comprehensive evaluations for FL workflows.

In this paper, we present FedBed [6], an open-source framework for the systematic evaluation of FL workflows. FedBed abstracts FL deployments into tunable virtualized testbeds so that users can assess the performance and optimize the QoS of their FL workflows. Users can investigate several trade-offs by changing FL and ML configurations along with the distribution of the underlying computational resources and datasets. We integrate FedBed with two state-of-the-art evaluation frameworks [16, 21] that utilize container orchestrators (e.g., Kubernetes), allowing FedBed to operate across multiple nodes. Lastly, we conduct an empirical study that compares performance and utilization metrics by accounting for the underlying AI models, data distributions, scalability, connectivity and computation limits, offering insights into the strengths and weaknesses of FL deployments.

The rest of the paper is structured as follows: Section 2 elaborates on FL over the Edge Continuum. Sections 3 and 4 introduce FedBed and its creation details. Sections 5 provides a comprehensive evaluation. Section 6 presents the related work, and Section 7 concludes this paper.

2 Background

2.1 The FL Process and Algorithms

Figure 1 depicts a typical FL flow, where a central server obtains a set of available clients (c_1, ..., c_k) that meet certain criteria (i.e., resource availability) and subsequently broadcasts to the clients a training program and an initial model $\mathbf{W}_{t_0}$ with t_0 denoting the initial training round ①. Next, clients update the model locally $\mathbf{w}_{c_i,t_1}$, based on local knowledge without exchanging data among themselves ②. The amount of samples used during local training can differ per client. When finished, the Server collects an aggregate of the client updates creating a new global model $\mathbf{W}_{t_1}$ ③. This is repeated for several rounds until a termination criterion is met that can include a max number of rounds or the convergence to a certain model loss for early termination.

Hence, the central Server only facilitates the training coordination that implies the client selection process and model aggregation. For the aggregation, the only requirement is that the process yields a weight vector for the model under training. FedAvg [15] is the most well-known FL algorithm and is often considered the baseline for FL. For local training, FedAvg embraces in parallel, E epochs of Stochastic Gradient Descent (SGD) where local model weights are updated to optimize the model loss based only on client samples. At the end of the round, the derived model weights are collected per client by the Server. Aggregation is then performed using a weighted average where s_k is the number of samples used by each client during local training and $S = \sum_k s_k$:

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

So, clients that have used more samples during the training process have a larger influence on the new state of the model. Other than FedAvg, there is a plethora of FL aggregation algorithms. For instance, FedProx [12] is 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 an FL algorithm that attempts to optimize the training process for non independent and identically distributed (Non-IID) data by providing a "correction" mechanism for the client-drift problem during local training [9].

In terms of client selection, these algorithms employ a common strategy where they opt for a random selection from the pool of available clients via a uniform distribution. However, studies show that performing a biased selection of clients can yield faster global model convergence by selecting clients with higher local loss [4] and reduce the training time by estimating the time required to complete a round [18].

2.2 FL Challenges over the Edge Continuum

Realising FL in Edge-Cloud settings presents challenges for evaluating performance, considering FL parameters, infrastructure heterogeneity, and data distribution volatility. Collaboration between data scientists and system engineers is crucial to find the best-fit model that meets performance requirements and infrastructure constraints. However, excessive focus on infrastructure configuration may divert attention from the primary FL optimization objective. Below

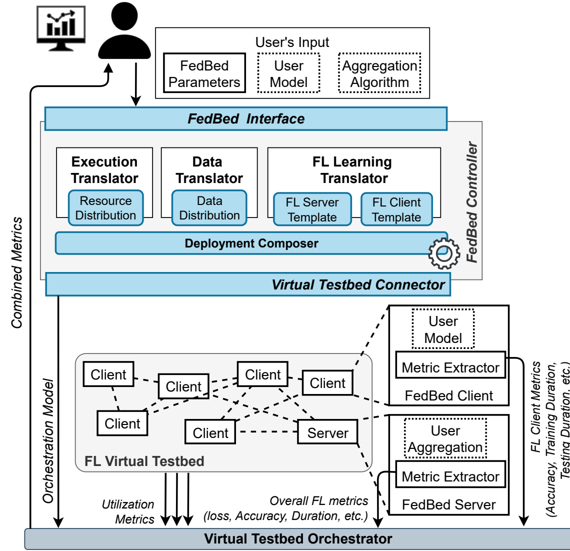


Figure 2. FedBed Testing Framework High-Level Overview

are the key challenges that FL practitioners must tackle when evaluating FL workflows in Edge Continuum settings.

Challenge #1: ML & FL configuration. FL workflows are heavily influenced by the ML backend, model architecture and the FL coordination. Fine-tuning these aspects is essential to balance model loss and training time. ML backends such as TensorFlow and PyTorch follow similar prototyping aspects by operating on tensors and viewing models as computational graphs with both offering many pre-compiled SOTA model architectures. Still, the implementation of these backends can impact FL performance [17]. In turn, certain models like linear regression enable fast training and inference but reduced accuracy, while neural networks (NNs) prioritize accuracy at the expense of training time and resources [23]. Lastly, the parameters of FL aggregation and client selection may alter the model's loss and training time.

Challenge #2: Resource heterogeneity. Moving FL to the Edge means training models over devices that come in many shapes and sizes with a plethora of hardware and software configurations. One client may feature powerful processors, GPUs or specialized accelerators, while others may participate in the training process under limited resources. Straggling clients in the training process delay the completion of a training round with a round considered finished only when all clients report updated model weights to the Server to produce the new state and synchronize the execution of the next round [4]. While some may advocate that potential stragglers should not be part of the training process, this may not be possible as a diverse set of clients and their data are often anticipated to reduce model bias [3].

Challenge #3: Network fluctuations and timeouts. Traditional networks require more bandwidth at central points (i.e., data center), whereas deploying ML tasks on Edge may require more bandwidth across individual edge nodes. Edge devices often connect to wireless mobile networks with constrained bandwidth and varying signal strength. FL can reduce the communication overhead by only exchanging model weights, but NNs can be complex with millions of parameters in the weight vector. Timeouts are more frequent in edge links making training round duration unpredictable leading to idle nodes waiting till a disconnected clients resumes work and the round can be marked as finished.

Challenge #4: Unbalanced and non-IID data. A key challenge for FL are non-IID data across clients that causes statistical variability and sub-optimal model training. This increases the training effort and can drive the local training towards a local optima that is far from the global optima – hence degrading the effectiveness of FL [9, 13]. Moreover, FL clients with skewed datasets can introduce bias in the learning process, while some of them could be stragglers becoming bottlenecks in the training process.

3 The FedBed Framework

3.1 Overview

Users face increased challenges mentioned above when executing FL workloads on Edge devices, and seek solutions to automatically evaluate various FL aspects such as ML models, aggregation algorithms, infrastructure properties, and more. To address these challenges, the FedBed testing framework allows users to choose their desired combination of built-in ML models, datasets, and aggregation algorithms, making it easy for them to evaluate the FL performance and infrastructure implications with minimal effort. The only precondition for users is to have an already installed virtual testbed orchestrator compatible with the FedBed framework.

Fig. 2 provides a high-level overview of the framework and its functionalities. Users start the evaluation by designing the FedBed's composable FL model in a YAML file and submitting it to the *FedBed Interface*, which is a Python light-weight library. FedBed Interface can be utilized from interactive data analysis tools, e.g., Jupyter Notebooks, and users can re-use the model's YAML file and the notebooks to easily reproduce their experimental analysis. With FL model submitted, FedBed validates model's parameters and the available resources, and propagates the model to FedBed Controller.

Then, *FedBed Controller* coordinates the experimentation by, firstly, dividing the parameters into execution, data, and FL learning sub-parameters and invoking the respective subcomponents, namely, Execution, Data, and FL Learning Translator. *Execution Translator* takes care of infrastructure-related parameters, generating resource limits by utilizing its resource distributions (e.g., Homogeneous, Gaussian, Pareto, etc.). Similarly, *Data Translator* creates the data partitions based on the submitted configurations. Lastly, *FL Learning Translator* populates the templates for the FL server and clients, customizing them with the selected ML and aggregation parameters. If users would like to introduce custom ML models or aggregation algorithms, they need to materialize the respective interfaces of the FedBed framework.

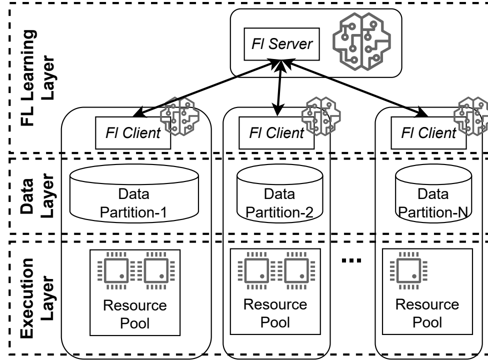


Figure 3. FL as a Multilayered System

Specifically, FedBed provides an interface-oriented design that allows users to introduce custom ML models or aggregation strategies in FL services with minimal effort. With custom artifacts introduced, the FL Learning Translator is responsible to include them in the FL services templates, and, at the runtime, the system invokes these artifacts without needing users to update and rebuild the whole framework.

At the next step, the *Deployment Composer* combines the results from the translators, creating a set of FL deployment objects with all the necessary information and generated parameters. To deploy the generated FL workload, FedBed uses a *Virtual Testbed Connector*, which (i) translates the deployment objects into low-level primitives for the underlying Virtual Testbed Orchestrator; (ii) facilitates deployment and testbed configurations; and, at runtime, (iii) retrieves monitoring metrics from the underlying virtualized testbed. With the respective primitives on hand, *Virtual Testbed Orchestrator* deploys the FedBed FL services in separate containerized environments, connects them over a virtualized network, and injects the network and computing resource limits. FedBed integrates two Virtual Testbed Orchestrators, namely, Fogify [21] and Frisbee [16]. These testbed orchestrators are built upon the foundation of multi-host docker orchestrators, such as Swarm and Kubernetes, enabling FedBed to effortlessly scale across an extensive array of nodes.

During experimentation, FedBed gathers various infrastructure and FL service metrics. It retrieves utilization metrics like CPU and memory usage from testbed orchestrators, and, also extracts fine-grained FL metrics from FL Client and Server, including loss, accuracy, and round duration. These metrics empower users to evaluate trade-offs between model's performance and the infrastructure's efficiency.

3.2 Composable FL Pipelines Modeling

FedBed considers an FL deployment as a multilayered system, as depicted in Fig. 3, with: (i) *FL Training Layer* being deployed on top, responsible for the FL training; (ii) *Data Layer* that is characterized by its dataset distribution; and (iii) *Execution Layer*, which illustrates the underlying compute and network resources. FedBed modeling allows users to configure fine-grained parameters for each layer, namely:

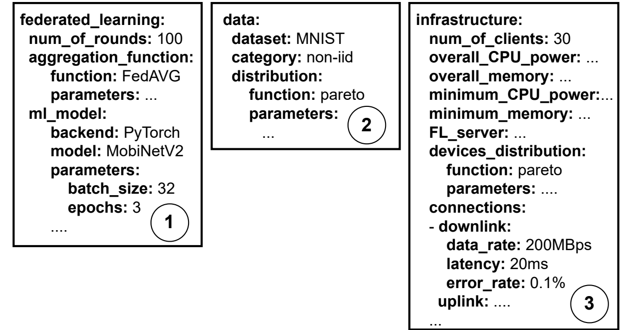


Figure 4. FedBed Abstractions

FL Learning Layer: Fig. 4 ① shows an FL object with user-selected parameters (`federated_learning`), such as hundred rounds for FL training (`num_of_rounds`) and the `FedAVG` aggregation function. Users can also customize the aggregation function's parameters using the `parameters` property. Moreover, users choose the ML model (`ml_model`) and its `parameters`, as seen in Fig. 4 ①, where `PyTorch` is selected as `backend` with `MobiNetV2` as the model and its tuning `parameters`, like, `batch_size`, `epochs`, etc.

Data Layer: The data layer is responsible for partitioning real-world datasets into smaller subsets. A set of exposed parameters enable users to control and quantify the imbalance properties of an FL deployment, thus addressing the data distribution challenge, which is not easily achievable with real federated datasets [3]. The current implementation is focused on horizontal FL, where each party shares the same feature space but owns different samples. By adopting partitioning strategies, researchers can configure the size of local data on each FL client and realistically correlate this imbalance with the computation resources of the Edge nodes hosting the FL clients, e.g., low-capacity nodes are not expected to handle large datasets. To achieve this correlation, users select a statistical distribution, such as `flat`, `normal`, `pareto`, etc. Fig. 4 ② shows the MNIST dataset (details in Sec. 4.2) as non-IID, following the Pareto distribution.

Execution Layer: configures the available resources on the underlying deployment. For instance, in Fig. 4③, users select the number of clients (`num_of_clients`) and define the overall CPUs (`overall_CPU_power`) and memory size (`overall_memory`). They can set the minimum CPU power (`min_CPU_power`) and memory (`min_memory`) assigned to a single FL client. Users also describe high-level resource distribution (`devices_distribution`) along with its properties, and the FL server's CPUs and memory (`FL_server`). Moreover, users should introduce a network connection that determines the network QoS among the FL server and clients. A network connection provides `uplink` & `downlink` QoS, including `data_rate`, `latency`, and `error_rate`.

It should be noted that the `parameters` for models, aggregation functions, data and resource distributions, are tailored to the underlying implementation. The parameters from already provided methods are described in FedBed's site [6].

4 FedBed Layered Implementation

FedBed simplifies the integration of AI/ML models, libraries, and datasets by employing an abstract class with two loosely-coupled abstract handlers: Model Handler and Dataset Handler, representing FL Learning Layer and Data Layer per node. New AI/ML models or datasets can be introduced by extending the latter handlers. The Execution Layer, on top of which the FL Learning and Data Layers operate, forms a scalable testbed managed by a Virtual Testbed Orchestrator.

4.1 FL Learning Layer

FL training requires services that handle the low-level implementation aspects, such as FL server-client communication, health checks, orchestration, etc. To alleviate the difficulties of a new FL framework creation, FedBed extends the well-known Flower FL framework [2]. We rely on the frameworks' aggregation algorithms and services, and we notably extend and automate the submission of FL training pipelines.

An FL client follows similar sequential execution steps in each FL round, specifically: (i) the client introduces the data partition to the model; (ii) trains the model; (iii) sends the trained parameters to the server; (iv) updates the model's parameters with the newly received aggregated parameters; and (v) may evaluate the model. Moreover, the FL server also uses the same methods, e.g., for the evaluation of a separate sample, or to send the newly created parameters. For that reason, we created a class, namely `Model Handler`, that provides a set of abstract methods, such as `train`, `test`, `get_parameters`, etc. These methods can be applied to AI/ML models, which are the handler's main input and provide the structure of the deployed model. Unfortunately, different backends provide different implementations of the latter methods, so FedBed introduces materialized classes for the most well-known libraries. In particular, FedBed supports two deep learning backends, PyTorch and TensorFlow, along with SKlearn as a lightweight AI library. Table 1 includes the available combinations of models, backends, and datasets.

Moreover, FedBed currently offers AI/ML models for two well-known datasets MNIST and CIFAR10/100, which are described in detail in Section 4.2. For the MNIST dataset, a Convolutional Neural Network (CNN) is implemented in both PyTorch and TensorFlow. The CNN architecture consists of six trainable layers, including two 2D convolutional layers, two 2D dropout layers, and two linear layers. Additionally, non-trainable layers such as ReLU activation 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 are included. Furthermore, linear and logistic regression models using the SKlearn library are available for the MNIST dataset. For CIFAR datasets, FedBed offers the well-known MobileNetV2 model in both TensorFlow and PyTorch. MobileNetV2 is a CNN model commonly used for image classification, it consists of 53 layers, and its details can be found in [20].

Models	Backends	Datasets
CNN	`TensorFlow`	`MNIST`
CNN	`PyTorch`	`MNIST`
Linear & Logistic Regression	`SKlearn`	`MNIST`
MobileNetV2	`TensorFlow`	`CIFAR10/100`
MobileNetV2	`PyTorch`	`CIFAR10/100`

Table 1. Framework's Integrated Models and Datasets

4.2 Data Layer

Each FL client is equipped with a `Dataset Handler` responsible for retrieving a partitioned dataset. The handler interacts with the IO system, fetches the data from storage, and deserializes it into a format suitable for the AI process. To meet the AI model's requirements, data may undergo further transformations, such as normalization and resizing. The formating and transformation of the data are driven by the backend and the respective dataset, so FedBed implements a Dataset Handler for each Backend-Dataset pairing (Table 1).

The transformed data is then allocated in the memory space of the FL node ensuring faster access. Once in memory, the data is fed to the AI model's input layers for processing through the forward pass. Currently, FedBed allows datasets that fit within a client's memory capacity, with larger datasets extending beyond this limit not being factored in. Our future roadmap includes the integration of a batch processing methodology, wherein substantial datasets can be effectively processed in smaller, manageable batches.

Additionally, FedBed provides an automated data partitioning mechanism to emulate different data distributions across clients. Users define the shared dataset location, the number of clients, and the desired data distribution strategy (e.g., Flat, Pareto, Gaussian). During deployment, FedBed generates statistics for the distribution and uses them to determine partition parameters (offset, size). Each client is then provided with these parameters, ensuring that they read distinct ranges from the shared dataset.

FedBed uses two well-known datasets for image classification: (i) MNIST that contains 60k grayscale images of handwritten digits (28x28 pixels) grouped into 10 classes (0 to 9); and (ii) CIFAR10 and CIFAR100 contain colored images (32x32 pixels) of various objects like airplanes, cars, and animals, with 10 and 100 classes, respectively. Both CIFAR10 and CIFAR100 have 50k training images and 10k test images.

4.3 Execution Layer

To ensure portability and extensibility across different environments, FedBed introduces its components into Docker images. Moreover, the FedBed codebase provides various environmental variables, through which one changes the FL services' flow without needing to rebuild the FedBed Image. Thus, a running FedBed container executes a specific FedBed FL service, server or client, and invokes particular parts of code for aggregation algorithms, ML models, datasets, etc.

The manual execution of FedBed containers enforces users to define every environmental variable on each container.

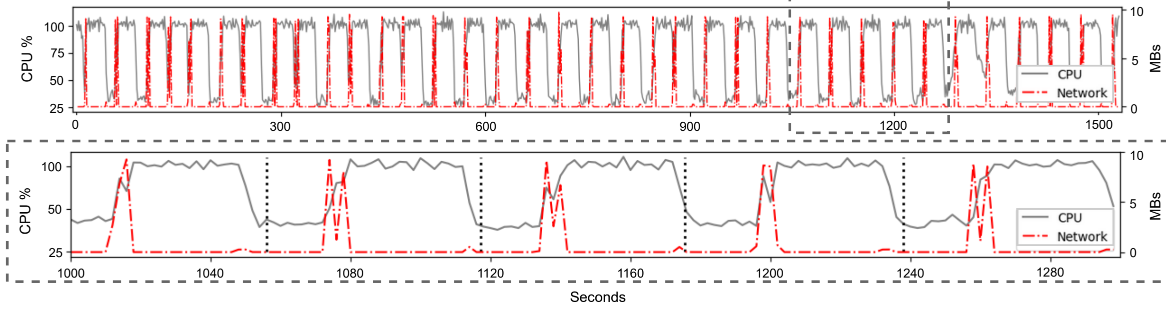


Figure 5. FL clients' mean CPU and network usage

To tackle this issue, FedBed provides an automated translation process for the framework's modeling abstractions into deployment objects, each of which represents an FL service (server or client), keeping the respective environmental parameters along with infrastructure properties. Next, the system invokes the respective Virtual Testbed Connector, which is capable of translating the deployment objects into primitives readable from the underlying Virtual Testbed Orchestrator. FedBed currently integrates Fogify [21] and Frisbee [16] frameworks as Virtual Testbed Orchestrators. The process of testbed instantiation is consistent for both orchestrators, so next we describe the process of the Fogify framework.

When Fogify receives the translated model, it instantiates the FedBed containers, constrains their compute resources, and establishes their network connectivity and QoS. To do that, Fogify utilizes a multi-host Docker-based cluster orchestrator, namely Docker Swarm, whose responsibility is to start the containerized services and limit their execution capabilities via Linux Cgroups, based on the deployed description. In parallel, Fogify disseminates a set of instructions to Fogify's Agents, which are located on every cluster node and apply the controller's commands to the running instances. These commands mainly refer to network shaping at the testbed bootstrapping. Finally, Fogify's Agent encapsulates a monitoring enabler that captures the testbed's utilization metrics.

4.4 Cross-Layer Monitoring

At the runtime, FedBed logs a comprehensive set of monitoring metrics. Since the data layer in our approach is static during the execution, the system keeps only the number of data points and the initial size of the respective partition.

For Learning layer, we extract two types of metrics, namely *Model Performance Metrics* and *Time Metrics*. Model Performance Metrics provide ML-level KPIs per round, including overall and per-client accuracy and loss. Moreover, Time Metrics are related to the overall latency and duration of the FL process, such as the training duration and round duration per client. The framework also monitors the duration of every learning stage, including data fetching, training, and evaluation timings for each client. To extract these metrics, FedBed introduces custom methods that intersect the Model Handler execution at runtime and generate per-client duration-related FL metrics. Moreover, FedBed extends the FL server code to log the overall model metrics, like global model accuracy and loss, FL round duration, etc.

Finally, FedBed also extracts *Utilization Metrics* from the Execution Layer such as CPU and memory usage, network traffic, etc, which are provided by the Virtual Testbed Orchestrator. Since all metrics are timestamped, users can compare utilization metrics with the ML performance and time metrics to gain insights for system's reactions and resource consumption, identifying potential bottlenecks.

5 Experimentation Study

We employ a bare metal cluster of 48 CPUs, all clocked at 2.45GHz, and 176GB RAM. We deploy Fogify on top to manage the resource pool and provision virtual appliances through the FedBed testbed connector. Each FedBed trial requires only a template declaring the experiment resources, network connectivity, FL parameters, ML backend and dataset. The study's templates are accessible at FedBed repo [6].

5.1 FL Client CPU and Network Patterns

This experiment showcases the repeatable behavior of FL from a systemic viewpoint. We request through FedBed 10 homogeneous FL clients (2cores@1.6GHz, 4GB RAM) and set TensorFlow as the ML backend and the CNN model for the MNIST dataset with the data equally partitioned among the clients. Fig. 5 (top) portrays the clients' CPU (left axis) and network (right axis) during distributed training and with the x-axis depicting the experiment duration till completion (26mins). We observe a distinguishable periodic pattern where a high network spike is always succeeded by a considerable rise in CPU utilization (~100%) and then followed by a lower network spike, after which the CPU falls below 50%.

To further scrutinize FL behavior, we zoom-in on a random period of 5 training rounds (Fig. 5 bottom). In this plot, the vertical dotted lines signify the end of the local training as reported by the FL clients. One can observe that after local training the clients are underutilized with CPU usage to be slightly below 50%. At this time, the central Server begins the aggregation of the clients' weight vectors. Once a new state is produced, the Server disseminates to clients the new model. This is noticeable by the large network spike. Afterwards, the clients commence local training, with the CPU jumping to 100% for the training duration. This concludes the periodic

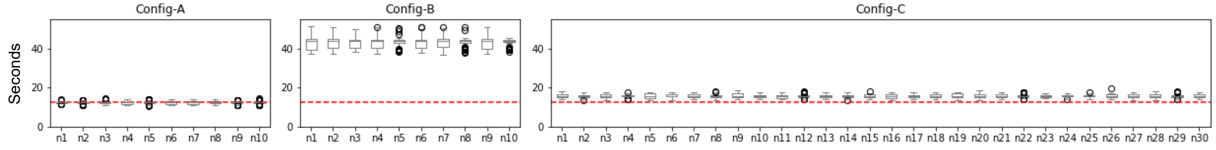


Figure 6. Training round duration for the 3 under examined configurations

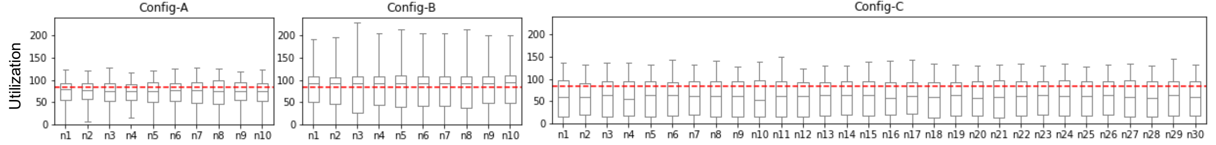


Figure 7. Client CPU utilization for the 3 under examined configurations

Name	Rational	Clients	CPUs/Freq. per Client	Samples per Client
Config-A	Baseline	10	4cores@2.4GHz	6000
Config-B	Scale-down	10	2cores@1.6GHz	6000
Config-C	Scale-out	30	2cores@1.6GHz	2000

Table 2. Experimental Configurations

nature of FL. From the zoom-in, we note that some rounds feature not 1 but 2 network spikes. This extra overhead is not obvious from a theoretical perspective and is attributed to the FL coordination where weights are disseminated twice - one time for the FL assessment (i.e., loss computation) and again for the next FL round. In conclusion, *during FL, one can observe specific usage patterns in computing and network resources with different utilization requirements per FL phase.*

5.2 Vertical & Horizontal Scaling

Next, we examine the impact of resource shaping on FL performance by introducing 3 configurations (see Table 2). As a baseline (named `Config-A`), we have 10 clients with 4cores@2.4GHz. Then, we introduce `Config-B`, where we scale-down the computing power of the FL clients to 1/3 of the baseline (same configuration as Section 5.1) and `Config-C` where we scale-out `Config-B` to reach 30 clients.

Fig. 6 and 7 depict the results. The left plots refer to `Config-A`, where the mean training time per round is approx. 15s and the CPU 75-80%. For comparison, these insights are highlighted with red in the other plots. Since the baseline clients are slightly under-utilized, next we try the scale-downed `Config-B`. By doing so, we observe that the clients are now almost fully utilized but a performance penalty is introduced with the training time increasing to 45-50s. Acknowledging the unfortunate effect of the scale-down, `Config-C` is considered next. Scaling out the client set, naturally, reduces the client load about ~55% but we observe that while training time lowers it remains more elevated than the baseline at 17-19s. This strikes for more investigation, where more clients result in more network traffic, and coordination effort for both model updating and synchronization. Inspecting the network traffic of all configurations, the setups with the same number of clients, `Config-A` and `Config-B`, have almost identical traffic (9646 & 9407 MB). On the contrary, `Config-C` imposes a penalty in training time despite the same resource pool with `Config-A` attributed to the $x2.9$ jump in traffic (28590 MB). To this end, *finding the sweet-spot for the number of clients and amount of resources each should harbor requires the examination of multiple trade-offs.*

5.3 Impact of ML Libraries and Models

Let us consider the `Config-A` and `Config-B` settings of the previous experiment. We compare the same CNN model from 2 libraries (PyTorch, TensorFlow) and a linear regression from SKlearn as a light alternative. For all tests, FedBed uses the MNIST dataset and the monitoring is set to report round duration, model loss and accuracy, and system-level metrics.

Fig. 8 and 9 depict the experiment results. Initially we concentrate on the top plot of Fig. 8 and compare the 2 settings where `Config-A` exhibits a smaller training round than `Config-B`. This is expected with the latter featuring $3x$ less computational resources. Next, we compare the ML backends. We start with SKlearn where we observe that it features the better training time and consumes less CPU resources for both configurations compared with the CNN models. Looking at the network plot, one immediately observes that SKlearn has a significantly low network footprint compared to the other backends. These savings, though, come at a cost where the less complex model takes an accuracy hit of more than 12% and with a model loss of almost $x2.5$ more than the other backends. Still, in some extremely constrained cases, this trade-off may be worth pursuing. Moving on, we compare PyTorch and TensorFlow. First, we observe that accuracy and model loss are almost identical. This is expected as they deploy the same CNN model architecture. Examining the `Config-A`, PyTorch and TensorFlow have almost the same FL round duration, but TensorFlow consumes more CPU resources than PyTorch. Interestingly, we observe that PyTorch outperforms about 10% TensorFlow in terms of training round duration with `Config-B`. This performance boost can be interpreted as the better utilization of available compute resources when looking at the CPU insights for `Config-B`. We will investigate the effects of neural network libraries deeper in the experimentation of Section 5.6 where TensorFlow can perform better in extreme network conditions. In conclusion, *even before FL knob optimization or even*

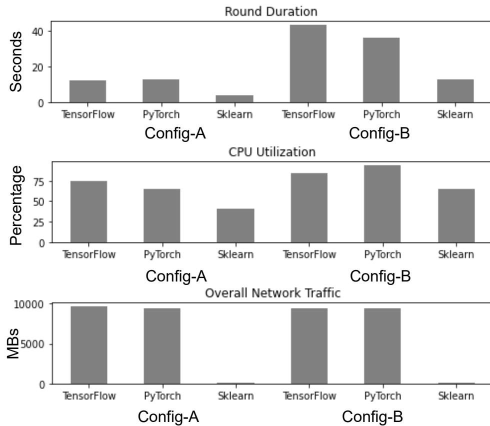


Figure 8. Training round duration, CPU and network traffic

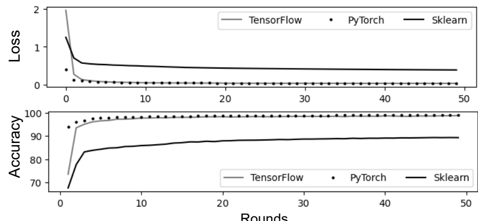


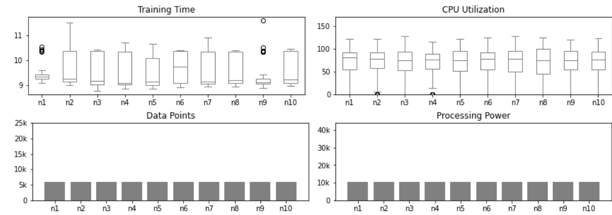
Figure 9. Model loss and accuracy for each ML backend

infrastructure resource profiles, the choice of ML backend as well as model can impact both the FL performance metrics and the system utilization footprint of the deployment.

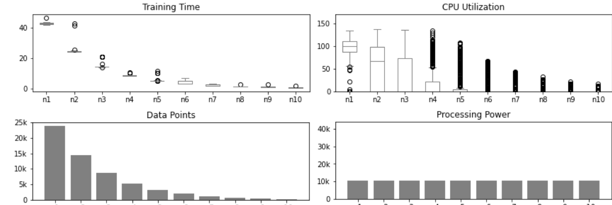
5.4 Data and Compute Distribution

This experiment examines the impact of different data distributions and computational resources to FL performance. We begin by considering the MNIST dataset and the TensorFlow backend employing a CNN model. Next, we configure FedBed with 2 partitioning strategies in the form of a homogeneous (denoted as Flat) and Pareto distribution. These will be considered for both the dataset and the computational resources. As metrics of interest, we consider the per round training time, CPU, the number of data points (sample size), and processing power for each FL client. In brief, processing power refers to a client's cumulative clock rate that is the number of cores multiplied by the CPU frequency. Fig. 10 depicts the results of the experiment runs. Specifically, Fig. 10a focuses on the FL deployment where a Flat distribution is imposed to both the dataset and computational resources. This is an ideal scenario where no heterogeneity is evident; hence, the load is equally distributed with the clients finishing local training simultaneously. Moving to Fig. 10b, we observe that adding data skewness to the data partitioning (via a Pareto distribution) while maintaining fairness in the processing capabilities, introduces a notable impact to training performance. Here, the clients with more data are now bottlenecks with the others finished and remaining idle till all mark the round as completed and the aggregation process can start.

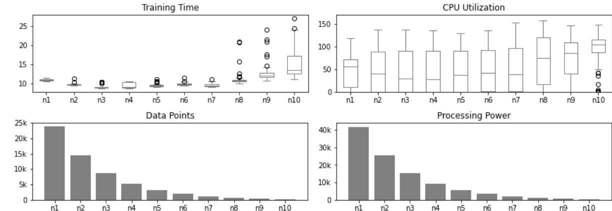
At this point one may ask "*can we distribute the processing power to to mitigate the effect of data skewness?*'. The answer



(a) Flat Distribution of both Compute Resources and Dataset



(b) Flat Compute Resources and Pareto Dataset Distribution



(c) Pareto Distribution of both Compute Resources and Dataset

Figure 10. Impact of Data and Computational Skewness

to this lies in the findings of Fig. 10c where we observe that when clients feature more processing power, they can accept a larger portion of the data load without imposing delays in the training performance. In conclusion, *(i) the ideal case for training time and predictable utilization in FL workflows is to distribute data and processing homogeneously; (ii) with homogeneous compute resources, the data distribution dictates the training time;* and *(iii) if the processing capabilities follows the data distribution alleviates the effects of unbalanced data.*

5.5 FL Stragglers and FL Clients Selection

Assessing the overall duration of Flat-Pareto deployment (Fig. 10b), we observed the straggler node phenomenon, where the slowest worker determines the overall round duration of the FL process. In this case, n1 is the main straggler with about 40s median training duration, followed by n2 (22s) and n3 (17s). Next, we investigate how the proportion of chosen nodes affects the performance of the FL execution. In each iteration, the FL server picks a subset of available nodes. We introduce different selection percentages of 20%, 50%, and 100% of the total nodes. The left graph in Fig. 11 illustrates the distribution of the overall round durations for 50 rounds. When all nodes are selected, node n1 heavily influences the round duration, surpassing 41s. In the 50% selection, the median duration is about 32s, while the 20% selection results in a slightly lower than 20s median. The right graph in Fig. 11 shows the loss for each configuration, displaying a similar

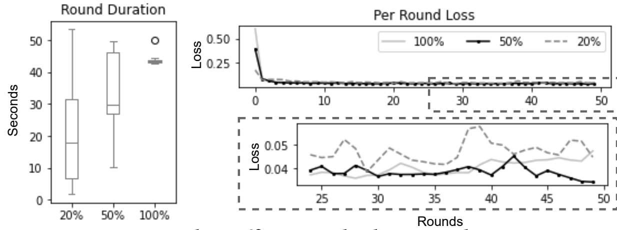


Figure 11. Straggler Effects and Clients Selection Fraction

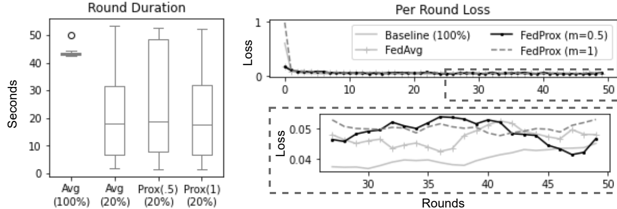


Figure 12. Straggler Effects and Aggregation Algorithms

consistent trend. The last 25 rounds notably show that the loss for the 20% setup exceeds those of the other trails, with the 50% and 100% setups yielding comparable outcomes.

Considering that the 20% node selection yields the optimal round duration but results in the least favorable loss when using our default aggregation algorithm (FedAvg), we investigate the FedProx [12] aggregation algorithm as an alternative. So, we replicate the same experiment involving imbalanced data distribution and configure FedProx with two different values for its proximal term, m=0.5 and m=1. Fig. 12 (left) presents that the performance of the 20% node selection maintains a consistent distribution in both FedAvg and FedProx. Additionally, Fig. 12 (right) juxtaposes the loss across all configurations. With the 20% client selection, it is observed that FedProx, under both proximal term configurations, slightly underperforms compared to FedAvg (20%). Meanwhile, FedAvg with the complete dataset demonstrates the most favorable outcomes. Thus, *the selection of FL clients influences FL duration and the ML metrics, with more sophisticated methods not always providing better results.*

5.6 Network Heterogeneity

Unlike high-speed network configurations found in datacenters, the traffic of Edge computing is bound to irregularly dense multi-tier networks to satisfy the exponential growth of wireless data [16]. Hence, this experiment examines the impact of network QoS to FL performance. For this, we capitalize on the topology of the baseline setup (`Config-A`) with the FedBed experiment description extended to consider 2 ML Backends, PyTorch and TensorFlow. We employ the (same) MobileNetV2 model to avoid accuracy comparison among the 2 backends. The scenario is the following: FL training for 100 rounds over the CIFAR-10 dataset and during each experiment a different RTT is employed as the main network knob with RTT configured to «10/25/50/100/200/400ms».

Fig. 13 compares the 2 backends with the left axis showing training duration in absolute numbers and the right-axis

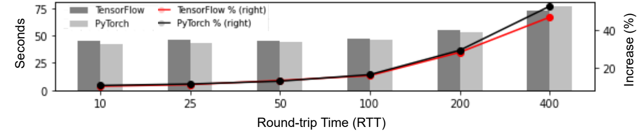


Figure 13. Network delay effect on FL training

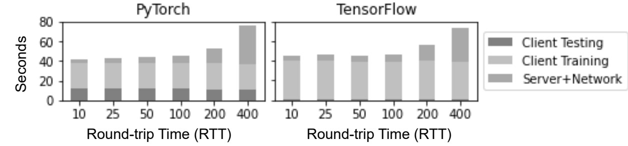


Figure 14. Fine-grain timings for CIFAR10

ratio increments. From this, we observe that the network RTT clearly impacts FL. PyTorch exhibits slightly better performance but as the network delay increases the gap closes, and TensorFlow becomes the better backend for the extreme case with an RTT of 400ms. Next, we employ FedBed monitoring to explore where the backends consume effort and the FL stage is influenced by RTT increments (Fig. 14). For this FedBed captures the duration of: (i) the *local training* per client; (ii) the *local testing* time to derive model loss over local data; and (iii) the *server-side* time for the receipt of the client weight vectors and the aggregation. First, it is evident that local training is what dominates an FL round. Second, the TensorFlow testing stage is faster than PyTorch, whereas PyTorch performs slightly better in terms of *local* training and overall duration. Third, as expected, *local* training and testing are not affected by the RTT increment. Fourth, when the RTT surpasses 100ms, the server-side effort increases exponentially, and this significantly impacts the overall FL round duration. Hence, improving the clients' computational capacity lowers the local training effort, but the server-side effort will remain and attribute a larger percentage of the overall round duration. In terms of network i/o, PyTorch traffic was elevated by 6% in comparison to TensorFlow signifying why at 400ms RTT the latter performed better (Fig. 13). In conclusion, *server-side aggregation and network delay impact FL attributing to at least 10% of the training effort and exponentially increasing as the network delay surpasses 100ms.*

6 Related Work

There are numerous testing tools that support Edge Computing emulation in virtual environments (e.g., cloud clusters) with resource and network shaping. For instance, systems, like MockFog [8], Fogify [21] and Frisbee [16], take advantage of cloud resources to create large-scale multi-host testbeds offering realistic compute and network resource emulation, allowing ad-hoc network changes, mobility, faults introduction, etc. Targeting the trials reproducibility, E2Clab [19] offers integration with various systems, including kubernetes, and allows users to auto-deploy workloads on testbeds emulating user-defined network QoS. *Even if these tools provide realistic execution, offer reproducible experiments, and alleviate difficulties of application performance*

evaluation, they do not automate the deployment of FL workloads and do not provide FL-specific performance metrics.

Moreover, there are many evaluation and execution FL frameworks from the ML community. For instance, LEAF [3] introduces a range of FL training workloads and datasets, while FATE [14] and Flower [2] provide interoperability for ML models and extensibility via well-defined interfaces. Commercial Cloud-MLOps frameworks, like SageMaker [1], offer various testing features, including FL evaluation, but they lack customizable network properties and resource distribution, limiting evaluation to fixed compute nodes. Moreover, FedScale [10] offers a wide range of data partitioning methods and introduces network and resource heterogeneity at scale. However, its resource shaping is not implemented via realistic emulation but via simulated artificially injected delays. EdgeTB [24] is the only effort that combines distributed learning in virtualized testbeds, but it leaves unexplored implications for ML backends and models, without focusing solely on FL. So, except for FedBed, *there is no testing framework that examines the Edge Computing implications on FL workflows, providing configurable models and datasets.*

7 Conclusion

This paper introduces FedBed, a testing framework for FL workloads on virtualized Edge testbeds, addressing resource heterogeneity, network availability, AI/ML backends, and data distribution challenges. It facilitates the reproducible evaluation of both software and infrastructure configurations, encompassing the setup of emulation testbeds, parameterization of experiments, and real-time monitoring of FL processes and edge infrastructures. During our experimentation, we examined how well models and libraries perform in terms of scalability, data distribution, and network aspects. In the future, we plan to use FedBed on real-world setups with different devices, like sensors, mobile devices, and GPUs, and test more generic FL tasks, like natural language processing.

Acknowledgments

This work is supported by the Cyprus Research and Innovation Foundation (CODEVELOP-ICT-HEALTH/0322/0047) and the Horizon Europe Framework (dAIEDGE/101120726).

References

[1] Amazon. 2023. https://aws.amazon.com/sagemaker/.

[2] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Kwing Hei Li, Titouan Parcollet, Pedro Porto Buarque de Gusmão, and Nicholas D. Lane. 2022. Flower: A Friendly Federated Learning Research Framework. arXiv:2007.14390 [cs.LG]

[3] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečný, H. Brendan McMahan, Virginia Smith, and Ameet Talwalkar. 2019. LEAF: A Benchmark for Federated Settings. arXiv:1812.01097 [cs.LG]

[4] Yae Jee Cho, Jianyu Wang, and Gauri Joshi. 2020. Client Selection in Federated Learning: Convergence Analysis and Power-of-Choice Selection Strategies. arXiv:2010.01243 [cs.LG]

[5] Dimitrios Dimitriadis, Mirian Hipolito Garcia, Daniel Madrigal, Andre Manoel, and Robert Sim. 2022. FLUTE: A Scalable, Extensible Framework for High-Performance Federated Learning Simulations.

[6] FedBed. 2023. https://github.com/UCY-LINC-LAB/FedBed.

[7] Yansong Gao, Minki Kim, Sharif Abuadbba, Yeonjae Kim, Chandra Thapa, Kyuyeon Kim, Seyit A. Camtep, Hyoungshick Kim, and Surya Nepal. 2020. End-to-End Evaluation of Federated Learning and Split Learning for Internet of Things. In *SRDS*. IEEE, China, 91–100.

[8] J. Hasenburg, M. Grambow, E. Grünewald, S. Huk, and D. Bermbach. 2019. MockFog: Emulating Fog Computing Infrastructure in the Cloud. In *IEEE ICFC*. IEEE, Prague, Czech Republic, 144–152.

[9] Sai P. Karimireddy, Satyen Kale, Mehryar Mohri, Sashank J. Reddi, Sebastian U. Stich, and Ananda T. Suresh. 2021. SCAFFOLD: Stochastic Controlled Averaging for Federated Learning. arXiv:1910.06378 [cs.LG]

[10] Fan Lai, Yinwei Dai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. 2021. FedScale: Benchmarking Model and System Performance of Federated Learning. In *Proc. of the First Workshop on Systems Challenges in Reliable and Secure Federated Learning*. ACM, 1–3.

[11] Li Li, Yuxi Fan, Mike Tse, and Kuo-Yi Lin. 2020. A review of applications in federated learning. *Computers & Industrial Engineering* 149 (2020).

[12] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. 2020. Federated Optimization in Heterogeneous Networks. arXiv:1812.06127 [cs.LG]

[13] Xiang Li, Kaixuan Huang, Wenhao Yang, Shusen Wang, and Zhihua Zhang. 2020. On the Convergence of FedAvg on Non-IID Data. arXiv:1907.02189 [stat.ML]

[14] Yang Liu, Tao Fan, Tianjian Chen, Qian Xu, and Qiang Yang. 2021. FATE: An Industrial Grade Platform for Collaborative Learning with Data Protection. *J. Mach. Learn. Res.* 22, 1, Article 226 (jan 2021).

[15] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th AISTATS*, Vol. 54. PMLR, 1273–1282.

[16] Fotis Nikolaidis, Antony Chazapis, Manolis Marazakis, and Angelos Bilas. 2021. Frisbee: A Suite for Benchmarking Systems Recovery. In *HAOC*. ACM, New York, NY, USA, 18–24.

[17] Fotis Nikolaidis, Moysis Symeonides, and Demetris Trihinas. 2023. Towards Efficient Resource Allocation for Federated Learning in Virtualized Managed Environments. *Future Internet* 15, 8 (2023), 25 pages.

[18] Takayuki Nishio and Ryo Yonetani. 2019. Client Selection for Federated Learning with Heterogeneous Resources in Mobile Edge. In *ICC*. IEEE, Shanghai, China, 1–7.

[19] Daniel Rosendo, Pedro Silva, Matthieu Simonin, Alexandru Costan, and Gabriel Antoniu. 2020. E2Clab: Exploring the Computing Continuum through Repeatable, Replicable and Reproducible Edge-to-Cloud Experiments. In *IEEE CLUSTER 2020*. 176–186.

[20] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *IEEE/CVF CVPR*. IEEE, Los Alamitos, CA, USA, 4510–4520.

[21] Moysis Symeonides, Zacharias Georgiou, Demetris Trihinas, George Pallis, and Marios D. Dikaiakos. 2020. Fogify: A Fog Computing Emulation Framework. In *IEEE/ACM SEC*. IEEE, San Jose, CA, USA, 42–54.

[22] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. 2018. Adaptive Deep Learning Model Selection on Embedded Systems. In *Proceedings of the 19th ACM SIGPLAN/SIGBED LCTES* (Philadelphia, PA, USA). ACM, New York, NY, USA, 31–43.

[23] Demetris Trihinas, Michalis Agathocleous, Karlen Avogian, and Ioannis Katakis. 2021. FlockAI: A Testing Suite for ML-Driven Drone Applications. *Future Internet* 13, 12 (2021), 24 pages.

[24] Lei Yang, Fulin Wen, Jiannong Cao, and Zhenyu Wang. 2022. EdgeTB: A Hybrid Testbed for Distributed Machine Learning at the Edge With High Fidelity. *IEEE TPDS* 33, 10 (2022), 2540–2553.