FLoX: Federated Learning with FaaS at the Edge

Nikita Kotsehub*, Matt Baughman†, Ryan Chard‡, Nathaniel Hudson†‡,
Panos Patros§, Omer Rana¶, Ian Foster†‡, Kyle Chard†‡
*Minerva University, San Francisco, California
†Department of Computer Science, University of Chicago, Chicago, Illinois
‡Data Science and Learning Division, Argonne National Laboratory, Lemont, Illinois
§Department of Software Engineering, University of Waikato, Hamilton, Aotearoa New Zealand
¶School of Computer Science and Informatics, Cardiff University, Cardiff, UK

***Abstract*—Federated learning (FL) is a technique for distributed machine learning that enables the use of siloed and distributed data. With FL, individual machine learning models are trained separately and then only model parameters (e.g., weights in a neural network) are shared and aggregated to create a global model, allowing data to remain in its original environment. While many applications can benefit from FL, existing frameworks are incomplete, cumbersome, and environment-dependent. To address these issues, we present *FLoX*, an FL framework built on the funcX federated serverless computing platform. *FLoX* decouples FL model training/inference from infrastructure management and thus enables users to easily deploy FL models on one or more remote computers with a single line of Python code. We evaluate *FLoX* using three benchmark datasets deployed on ten heterogeneous and distributed compute endpoints. We show that *FLoX* incurs minimal overhead, especially with respect to the large communication overheads between endpoints for data transfer. We show how balancing the number of samples and epochs with respect to the capacities of participating endpoints can significantly reduce training time with minimal reduction in accuracy. Finally, we show that global models consistently outperform any single model on average by 8%.**

***Index Terms*—Federated learning, serverless, edge computing, heterogeneous computing, computing continuum.**

1. Introduction

FL is a decentralized machine learning (ML) technique that was proposed to address privacy concerns and reduce data communication when training ML models over distributed data. The FL paradigm aims to train a ML model without requiring that all data be moved to a central location. Thus, raw data owned by user devices need not be shared or communicated [1]. This provides an immediate layer of privacy when compared to centralized techniques where all data are sent to a centralized location for training and reduces communication costs as model parameters can be significantly smaller than raw data [2]–[4].

Despite the obvious need for and benefits of FL, there are not yet sufficiently generalizable and flexible frameworks that can be readily adopted and used to deploy FL in practice. As a result, most FL systems are built from scratch, requiring developers to coordinate all aspects of FL, such as: network communication, training on remote heterogeneous devices, and model aggregation. Existing frameworks aim to partially address this gap; however, they are not without limitations. TensorFlow Federated [5] does not include a public implementation of their remote execution API; PySyft [6] supports multi-node execution but only on cloud-hosted resources; Flower [7] enables FL experimentation with heterogeneous devices but bases communication on a user-managed client-server architecture.

We present a new FL framework called *FLoX* (**F**ederated **L**earning **o**n func**X**) that builds upon the funcX [8] federated *Function-as-a-Service* (FaaS) platform. *FLoX* is designed on a *serverless computing* framework to better support diverse and distributed deployment environments. Serverless computing abstracts device heterogeneity and provides a high-level interface via which computation (e.g., model training and model aggregation) can be performed irrespective of the specific location in which it is executed.

We evaluate *FLoX* using three benchmark ML datasets, with two neural network architectures, deployed over ten heterogeneous and distributed compute resources (including Raspberry Pi, Jetson Nano, and desktop PCs). We show that FL training can produce models that outperform those trained entirely at the edge while exhibiting comparable accuracy to those trained centrally. We also demonstrate that the overhead of *FLoX* is minimal. Finally, we explore methods for balancing training workloads according to resource capacity and show that such balancing can reduce training time while maintaining model accuracy.

This paper is organized as follows: Section 2 presents background information about FL and serverless computing. Section 3 presents several use cases that motivate our work. Section 4 introduces the *FLoX* framework and outlines the use of *FLoX* for a typical FL workflow. Section 5 demonstrates the performance of *FLoX* on three different datasets evaluating accuracy, performance, and overheads. Finally, Section 6 summarizes and concludes our work.

2. Background and Motivation

To contextualize our work, we describe relevant background in both FL and serverless computing. We also differentiate our approach from other FL frameworks.

2.1. Federated Learning

An FL workflow generally consists of the following steps: *(i)* the central server distributes a copy of the global model (initially with random parameters) to client devices; *(ii)* the client devices independently train their copy of the global model on their local data; *(iii)* the client devices send back the parameters from local training to the central server; *(iv)* the central server aggregates the received model parameters to update the global model (fusing what has been learned); and *(v)* the loop restarts [1]. Because FL does not rely on transferring raw data to the central server, it is typically used in settings where locally stored data cannot be transferred due to privacy concerns or data transfer costs [2]. FL deployments can largely be classified into two classes [9]: *cross-device* and *cross-silo*.

Cross-device. Cross-device FL refers to settings where there exist a large number of low-power heterogeneous devices (e.g., smart phones, smart wearables, edge devices, sensors). One cross-device application of FL is training mobile keyboards (e.g., Google Gboard [10], [11]) across user devices in a federated fashion. In this way, user predicting typing patterns is learned across all users without sharing raw user typing data. Cross-device FL is commonly used in edge and other IoT applications in domains, such as industrial IoT, smart cities, smart healthcare, smart transportation, and mobile crowdsensing [12]. In these settings, privacy is sometimes a challenge but use cases also rely on FL due to sporadic availability of devices (i.e., some devices may be offline due to no power or poor connectivity) and diverse hardware resources (i.e., low powered edge devices).

Cross-silo. Cross-silo FL refers to settings where there exist several large-scale datacenters or sites (e.g., hospitals, banks) wanting to collectively train a model on decentralized data without revealing their own sensitive data. Concerns regarding security and privacy are heightened in this context; however, there is less concern regarding resource constraints and availability. Applications of cross-silo FL are often found in biomedical (e.g., to create brain tumor segmentation models [13], mine electronic health records [14], and develop pediatric risk models [15]) and commercial (e.g., to train marketing and risk management models using data stored across multiple companies [16]) scenarios.

2.2. Federated Learning Challenges

We are motivated by the need to easily deploy FL in heterogeneous environments. Li et al. [17] identified four key challenges associated with cross-device FL in such environments. Below we describe these challenges and add a fifth, dynamic infrastructure, that is equally important.

Expensive communication. FL algorithms send only model updates to the server, potentially reducing the data transfer cost when compared to sending entire datasets; however, model parameters can be large. Therefore, proper analysis of the tradeoffs is necessary to ensure moving raw data or model parameters off-device is not prohibitively expensive in terms of time, energy, or bandwidth.

Systems heterogeneity. Participating devices can vary in architecture, storage capacity, computational power, network connectivity and more. Furthermore, participating devices can drop out during the training process because of connectivity or energy constraints. Hence, FL frameworks must be flexible to accommodate heterogeneous devices and fault-tolerant to accommodate failures.

Statistical heterogeneity. Devices may observe different volumes and distributions of data depending on usage and location. For example, a device that observes only one class of images in its location might make non-generalizable updates to the model. Including its updates in the global model could harm the accuracy of devices in other locations that observe more classes. This challenge is defined as the problem of non-IID (independent and identically distributed) data due to non-identical client distribution [9]. Some work has been done in multi-task [18] and meta-learning [19], which enable personalized and device-specific modeling to help solve the challenge.

Privacy concerns. While FL sends only model updates instead of the training data, malicious attackers may still use individual model parameters to reconstruct the data [20]. A lot of work has been done in applying cryptographic protocols (e.g., homomorphic encryption) and differential privacy to protect the individual model updates [21]–[23].

Dynamic infrastructure. Beyond the four challenges identified by Li et al. [17] there are many infrastructural challenges that must be addressed. For example, often only a subset of available devices participates in an FL training round. The other devices are idle. If clients are deployed on dedicated resources (e.g., if using cloud Infrastructure-as-a-Service), these resources are wasted and may incur significant costs [24]. There is a need to efficiently handle intensive and sporadic FL workloads (e.g., when clients are training their models and when clients return the local model updates [5]). Finally, there are significant challenges in deploying and configuring the infrastructure for FL, such as deploying servers for a network of edge devices, configuring network connectivity, and managing security—all of which can be cumbersome when working with many heterogeneous and distributed devices.

In this paper, we address these infrastructural challenges by applying a federated serverless model, which decouples the communication and task execution from the FL workflow. Our approach simplifies device setup, supports rapid scalability based on computational necessity, and enables sharing between users and FL models using the underlying federated FaaS framework.

TABLE 1: Comparison of federated learning frameworks, extended from Beutel et al. [7].

	TFF	PySysft	FedScale	LEAF	Flower	FedLess	$\lambda - FL$	*FLoX*
Single-node simulation	✓	✓	✓	✓	✓	✓	✓	✓
Multi-node execution	*	✓	✓		✓	✓	✓	✓
Scalability	*		**		✓	✓	✓	✓
Heterogeneous clients		✓	**		✓	✓		✓
Language-agnostic					✓			
ML framework-agnostic		***	***		✓	***	***	***
Baselines			✓	✓	*			
Lightweight endpoints		✓	✓	✓	✓			✓
Controller-driven					✓			✓
Serverless						✓	✓	✓

*: planned; **: simulated only; ***: PyTorch or TF/Keras only

2.3. Serverless Computing

Serverless computing is a modern cloud computing paradigm that enables execution of user workloads without regard for the underlying physical and virtual infrastructure [25]. Function-as-a-Service (FaaS), the most common example of serverless computing, allows scalable execution of programming functions via a cloud-hosted platform. Users register functions, and optionally any system or language dependencies needed to execute each function, with a cloud provider. Authorized individuals may subsequently execute that function one or many times by passing the function ID and input arguments to the cloud provider. Users pay only for the compute resources used (often measured in execution time) and need not provision, configure, and manage dedicated servers that otherwise would continuously run and consume resources.

There are many examples of both commercial and open-source FaaS providers, including AWS Lambda [26], Google Cloud Functions [27], Azure Functions [28], and OpenWhisk [29], among many others [30]. However, these services are either operated centrally, as is the case with commercial providers, or must be set up and deployed locally, typically on a Kubernetes cluster.

funcX [8] is a federated FaaS platform that decouples the cloud-hosted function registration and management from function execution. Thus, in this model, users register, share, and execute functions via the cloud service in much the same way as a centralized FaaS platform; however, they may choose on which external *endpoint* those functions are executed. Endpoints are lightweight and require only installation of a Python agent on a remote computer or device. After requesting that a function is executed on a remote endpoint, funcX stages function code and input arguments to the endpoint. The endpoint deploys an execution environment, starts and monitors execution, and returns the output to the user via the funcX service. funcX also provides a robust security model using Globus Auth [31], via which endpoints may be shared with one or more users.

We implement *FLoX* using funcX because it supports high-performance function execution across heterogeneous, distributed compute resources; decouples deployment of FL software on edge devices from the FL framework; provides reliability and robustness to overcome intermittent connectivity of edge devices; and multiplexes execution of functions on endpoints associated with different users or FL models.

2.4. Related FL Frameworks

There are several general-purpose and open-source FL libraries and frameworks, including TensorFlow Federated (TFF) [5], LEAF [32], and PySyft [6]. These frameworks are primarily used in local deployments and provide only building blocks for distributed deployments. For example, TFF defines an API for remote execution; however, they do not provide a public implementation of this API.

Flower [7] enables experiments on heterogeneous devices; however, its communication model is based on a client-server architecture in which users must install and configure client software, run a persistent server, and manage network connectivity. Thus, deploying a FL application on heterogeneous hardware in diverse physical locations requires significant effort to configure networking permissions, security certificates, and unique clients for each device. This makes it difficult both for rapid prototyping and to manage production deployments at large scale.

Two frameworks closely related to *FLoX* are λ-FL [33] and FedLess [24]. λ-FL is a system for serverless FL aggregation that addresses the issue of idle aggregators wasting resources while waiting for model updates. It deploys FL models on a Kubernetes cluster and manages the training and aggregation of models. It does not support execution on remote edge devices. FedLess implements a distributed model in which multiple FaaS providers can be used concurrently, such as AWS Lambda, OpenWhisk, and OpenFaaS. However, FedLess requires significant control infrastructure and is targeted at production deployments across cloud providers. It assumes use of pre-deployed FaaS systems, such as OpenWhisk, which are not designed to be deployed on low-power edge devices. Each of these FaaS systems is itself an entire system with its own identity and access management, communication, and execution models.

To the best of our knowledge, *FLoX* is the first library to support FL over a federated serverless framework.The goal of *FLoX* is to simplify the FL setup and workflow using

serverless deployment and enable flexibility for models, clients, datasets, and FL algorithms. To summarize how *FLoX* compares in general features and characteristics to these other frameworks, we adapt a table from Beutel et al. [7] comparing different FL frameworks in Table 1. We augment the table with new dimensions "serverless" (leverages a flexible serverless model), "lightweight endpoints" (requires minimal configuration and management of software on remote devices), and "controller-driven" (the FL application can be driven the FL process for centralized experimentation rather than by the participating clients).

3. Use Cases

We describe example use cases in science applications and FL experiments that motivate our design decisions.

3.1. Science Applications

Scientific applications increasingly rely on ML methods deployed in distributed environments. We describe two example use cases in rural settings and sensor networks.

Rural AI. In this application, the goal is to perform FL across physically separated rural data sources, providing technologically-enabled insights to otherwise unserved applications and communities [34]. One example application, robotic weed spraying, requires that field robots first identify weeds using powerful, multi-spectrum cameras. Sharing models trained in different areas can improve performance, however imagery collected by these robots is large and network connections are unreliable. Further, there is reluctance to share images between farms.

Sensor networks. The Sage project [35] aims to bring AI to the edge using smart sensor networks for real-time monitoring and fine-grained predictive applications. Motivated by a diverse set of ML-driven tasks (e.g., endangered animal observation, wildfire prediction, flood tracking), Sage requires infrastructure that allows computation to be performed across sensors, edge devices, and the cloud. Some of these tasks benefit from FL techniques to train models across many edge devices, including animal tracking, wildfire identification, and social distancing analysis. FL is crucial as the overhead of sending large datasets from low-powered edge devices can be significant.

3.2. FL Experiments

There is an important, yet unmet, need to provide systems on which users can develop and evaluate FL methods in real, rather than simulated, environments.

Comparing FL workflow on different devices. There is a need to evaluate FL models on real-world testbeds comprised of distributed and heterogeneous devices. For example, many real-world environments contain edge devices (e.g., Raspberry Pi, Jetson Nano), personal computers, cloud instances, clusters, and other resources.

Comparing different FL algorithms. An important need when developing FL algorithms is to be able to implement and compare possible algorithms easily. For example, one may choose to compare the performance of Federated Averaging (FedAvg) [1] with FedProx which supports heterogeneous network conditions [36] on various datasets and on different testbeds.

Comparing FL performance with different settings. It is valuable to be able to change the training parameters of the FL workflow. For example, users may wish to train a model on edge devices and compare how the accuracy and training time change in response to the number of training samples and epochs sent to each device. Other parameters may include changing the dataset, model size, participating devices, optimizers, and more.

4. Implementation

FLoX is an open-source [37], federated-FaaS-based FL framework that allows users to deploy FL workflows across heterogeneous devices with custom datasets, algorithms, and settings. The FL workflow can be managed from *any* computing device (e.g., laptop, PC, or cloud instance); funcX manages reliable execution of the training and inference process as well as communication with edge devices. We describe our design goals, architecture, and interface.

4.1. Design Goals

Based on the use cases presented above, we identify the following design goals for *FLoX*.

- **Device-agnostic.** There are myriad types of edge devices with different configurations. Our goal is to support them regardless of their hardware or operating system.
- **Scalable.** To enable the use of FL in realistic settings, *FLoX* must scale to support many connected devices, large training data, and complex models.
- **Flexible & usable.** It must be easy to set up an FL environment and experiment with different types of ML workflows, datasets, and settings.
- **Controlled access.** Access to remote devices must be secure but also easily shareable.
- **Fault-tolerant.** Device dropouts is a primary challenge in edge computing, it is important that *FLoX* is resilient to intermittent and permanent failures.
- **Centrally managed.** For ease of experimentation and deployment, FL workflows must be able to be launched from a single, remote computer without requiring manual device configuration.
- **Distribution-first design.** A robust FL framework must enable workflow distribution and modification as easily as modifying local code, which can be accomplished via a self-adaptive design using serverless.

4.2. *FLoX* Workflow

FLoX is implemented as a Python library capable of training and deploying Tensorflow-based ML models. The

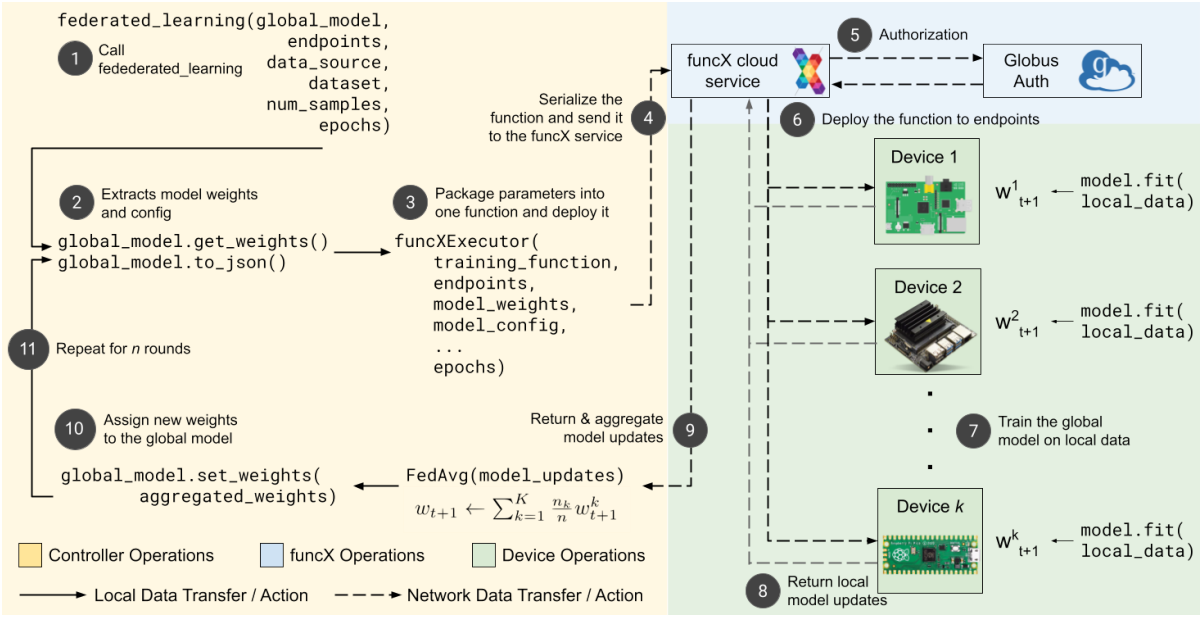


Figure 1: Sample learning round in a *FLoX* experiment flow. On the left (yellow) are the actions performed by the *FLoX* client library. Here a user invokes the *FLoX* API, *FLoX* configures a global model, and packages that model as a function, and calls funcX to execute the FL process. The top right (blue) shows the funcX authorization process and deployment of training functions on each of the participating devices. Lower right (green) shows the participating devices training models on local data) before returning results back to the client library via funcX.

library exposes an interface (see Section 4.3) for users to define their FL workflow. The *FLoX* library is then responsible for managing the deployment of that workflow on participating devices.

funcX manages the remote execution of FL training processes. To do so, we register the FL workflow as a set of funcX functions. The functions are then deployed to the participating devices and invoked along with the specific input arguments to configure relevant training process. funcX manages communication to the device and serializes input and output data (e.g., configuration parameters, model weights, metrics). The funcX endpoints manage execution of their respective functions, optionally deploying functions inside preconfigured containers.

The typical FL workflow includes two types of actor: one *controller* (perhaps implemented in a Jupyter notebook or deployed as a cloud service) and several participating *devices*. The main activities performed by each are:

Controller.

1) Select participating and eligible devices for the training round from a pool of available devices.
2) Send the training function along with the global model to the participating devices.
3) Retrieve the trained model (often weights) once the devices finish local training.
4) Aggregate the received model weights using an FL aggregation algorithm.
5) Repeat steps 1-4.

Devices.

1) Retrieve local data (e.g., from a sensor stream or a file).
2) Process the data if needed.
3) Train the model on the local data.
4) Send back the updated model weights.

Sample Flow. Figure 1 shows the specific steps involved in executing the FL workflow with *FLoX*. 1) the user calls `federated_learning`. 2) the function extracts the weights and architecture of the input ML model. 3) the input parameters and data are packaged into a single training function; 4) `FuncXExecutor`, part of funcX's Python SDK, serializes the function and sends it as a task to funcX. 5) Upon receiving the task, funcX first authorizes the user via Globus Auth and checks whether the user is permitted to access the endpoints. Shared access to endpoints can be configured using Globus groups. 6) funcX forwards the function and data to the selected devices. Each device's funcX endpoint is responsible for managing compute resources and containers, if configured it may reuse an existing container or start a new one for executing the training function. 7) The training function retrieves data from the specified source, processes it, compiles the model using the global model's configuration and weights, and performs training on the local data. 8) The model updates are returned to the funcX service that registers task completion and makes the results available for fetching. 9) `federated_learning` fetches the results at the completion of the round and aggregates the model updates using the weighted FedAvg [1] algorithm. 10) the centralized model's weights are updated. 11) the entire process is repeated for n rounds.

```
from flox import federated_learning

federated_learning(
    global_model=tf_model,
    endpoint_ids=[ep1, ep2, ep3],
    loops=10,
    epochs=10,
    data_source="keras",
    keras_dataset="mnist",
    num_samples=1000
)
```

Listing 1: Example invocation of *FLoX*

Aggregation. For aggregation in *FLoX*, we implement two aggregation techniques: *standard FedAvg* and *weighted FedAvg* [1]. Before defining the aggregation technique, we first note that we denote the parameters of an ML model trained by device k at time-step t by ω_t^k (the aggregated, global model denoted by ω_t). Standard FedAvg is defined as $\omega_{t+1} = (1/K) \cdot \sum_{k=1}^{K} \omega_{t+1}^k$ and is suitable for cases where data are distributed in a balanced way among client devices. Weighted FedAvg is defined as $\omega_{t+1} = \sum_{k=1}^{K} (n_k/n) \cdot \omega_{t+1}^k$ where n_k is the number of data samples at client device k and $n = \sum_{k=1}^{K} n_k$. This technique is meant for cases when data distribution among client devices is imbalanced.

The ability to apply a weighted average is important in environments in which devices have differing capacities or unbalanced training data. *FLoX* allows users to configure the number of training epochs or the number of samples used in training each endpoint. It may then apply an average with weighting based on the number of samples or epochs so as to avoid overvaluing weights from data-sparse endpoints (addressing the statistical heterogeneity challenge [9]).

FLoX is designed to be extensible such that users can configure the FedAvg algorithms or implement their own aggregation algorithm. For example, users may want to weight model updates based on their deviation from the global model (i.e., diff-aware weighting) or apply a hyperparameter optimization function to enable weighting of each endpoint based on the observed effect its contributions have on final model accuracy. To do so, users may simply replace the function ID in *FLoX*'s Python interface to point to their own aggregation algorithm.

Device configuration. Prior to running the FL workflow, users must first configure devices to execute funcX functions. This is a one-time cost and requires simply downloading (e.g., via pip) and configuring `funcx-endpoint`, a lightweight Python agent that enables serverless execution.

4.3. *FLoX* Interface

The largest hurdle to effective adoption of FL is the ease of development, configuration, and deployment. Overcoming these initial steps may pose too large of a barrier to entry for some even before beginning training of the FL model itself. To this end, we designed the interface to *FLoX* such that users with minimal systems or machine learning expertise can begin training FL models in minutes.

The core of the library's interface is a single-line invocation function: `federated_learning`. Using this function, the user can specify as little as the model architecture they wish to use and a list of funcX endpoints to be used for training. *FLoX* will otherwise use default arguments and deploy the FL experiment on those endpoints.

Users can optionally specify the number of FL *rounds* (or *loops*) they wish to train their model, the number of epochs and samples, as well as the data source (either globally or by endpoint), and any specifications for data processing. Users may also specify custom training, aggregation, and data retrieval & processing functions for advanced use cases.

Listing 1 shows an example of a call that facilitates 10 rounds of FL training on each endpoint with 10 epochs and 1000 MNIST images. The complete set of configurable parameters are:

- **ML model.** The ML model architecture. *FLoX* currently only supports Tensorflow/Keras models; however, other frameworks will be added in the future.
- **Data source**. The source of data for training. *FLoX* currently supports built-in Keras datasets (e.g., MNIST), locally stored files, or retrieved sensor data.
- **Data processing.** Custom prepossessing functions used to prepare edge data for training.
- **Training parameters.** Standard ML parameters such as the optimizer, loss function, metrics, etc.
- **Aggregation algorithm.** The algorithm used to aggregate models. *FLoX* offers a simple average and a weighted average of model updates. Users can provide custom aggregation algorithms.
- **Epochs.** The number of epochs to be used in training (set globally or per endpoint).
- **Samples.** The number of samples to be required for training (set globally or per endpoint).
- **Loops.** The number of FL training rounds.
- **Time interval.** The interval between FL rounds. For example, the devices can perform a FL round every hour.
- **Evaluation.** An optional validation dataset and evaluation function. *FLoX* evaluates the aggregated and individual models if the validation set is provided. Users can provide custom evaluation datasets and functions.

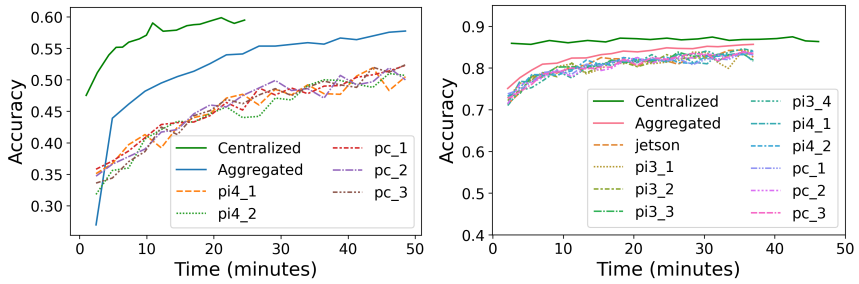
5. Evaluation

We evaluate *FLoX* by deploying FL models using three common ML datasets (fashion-MNIST [38], CIFAR-10 [39], and Animals-10 [40]) on a heterogeneous and geographically distributed testbed comprised of ten devices with six different configurations (see Table 2). We configure each device with a funcX endpoint with a fork-based executor.

For this work, our testbed includes small edge devices (e.g., Raspberry Pis) to represent resource-constrained devices that are common in edge computing scenarios (e.g., sensors for retrieving real-time data). In prior work we

TABLE 2: The heterogeneous testbed used for experiments.

Description	Hardware	Name	Count
Edge device (Raspberry Pi 3B)	ARM Cortex-A53, 4-core, 1GB	pi3	4
Edge device (Raspberry Pi 4B)	ARM Cortex-A72, 4-core, 4/8GB	pi4	2
Edge device (Jetson Nano 2GB)	ARM Cortex-A57, 4-core, Maxwell GPU, 128-core, 2GB	jetson	1
Fast desktop (Department Workstation)	Intel Core i7-8700, 12-thread, 16GB	pc_1	1
Average desktop (Department Workstation)	Intel Core i7-6700, 8-thread, 8GB	pc_2	1
Slow desktop (Department Workstation)	Intel Core i7-3770, 8-thread, 8GB	pc_3	1



(a) CIFAR-10 (b) Fashion-MNIST

Figure 2: Validation accuracy of local, aggregated, and centrally trained models on different datasets. (a) uses 400 samples and 20 epochs per device per training round; (b) uses 200 samples and 20 epochs per device per training round; (c) uses varying numbers of samples and epochs per training round.

showed that funcX scales well to much more powerful computing systems, including supercomputers [8].

We summarize the specific experiments and results in Table 4 and the configurations employed for each dataset in Table 3. We manage all experiments from a cloud-hosted Google Colab notebook. For model architectures, we use a CNN with two convolutional layers (32×3×3 and 64×3×3), pooling layers after each one, and a final dense classifier for the fashion-MNIST dataset; we use a CNN with four convolutional layers (two 32×3×3 and two 64×3×3), pooling layers after the second and fourth convolutional layers, a dense layer with 128 units, and a final dense classifier for CIFAR-10 and Animals-10. All models are trained with the ADAM optimizer [41].

5.1. Runtime and Overhead

First, we measure the runtime and overhead of using *FLoX* by logging the time spent in each *FLoX* workflow stage, per device per round. We run a regular FL workflow with 100 samples and 10 epochs on the Fashion-MNIST dataset for 10 rounds. We use timestamps, timers, and funcX endpoint logs to measure these timing values.

Figure 4 shows that the model training and communication time are most significant, with less powerful devices (e.g., Jetson Nano) taking more time than more powerful devices (e.g., PCs). On the other hand, the aggregation of model weights takes only 0.3 seconds, and the funcX overhead is only 0.1 seconds, on average. Thus, the most optimal way to reduce the round trip time is to deploy smaller models. They take less time to train, and their data sizes are smaller, decreasing the communication time.

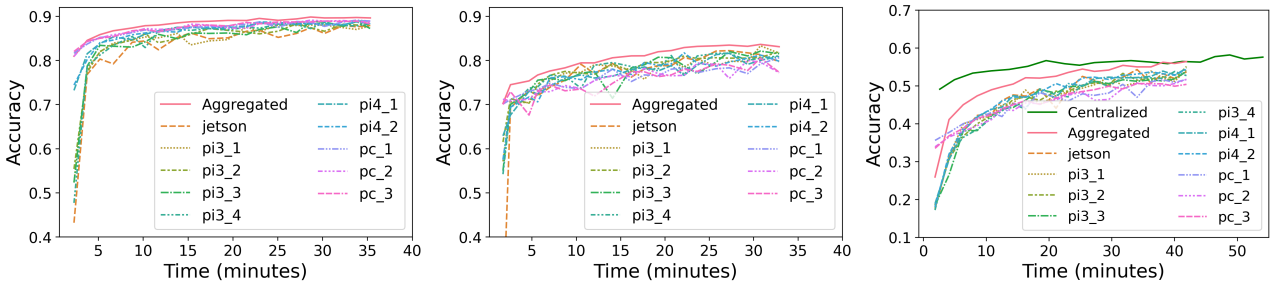
This experiment demonstrates the small overhead of *FLoX* when deployed in a distributed environment. It also shows that *FLoX* is able to orchestrate FL over a diverse set of 10 remote devices, thus addressing the system heterogeneity challenge of being device-agnostic (see Sec. 2). Finally, we also see the impact of individual devices on the overall performance of the FL workflow—demonstrating the possible bottlenecks imposed by low-power devices.

5.2. Model Accuracy

As a functional verification of *FLoX*'s functionality, we now consider the accuracy of models trained with *FLoX*. Specifically, we train a model across our testbed and compare the accuracy of the aggregated FL model, individual models (trained per device), and a centrally trained model.

We run 20 rounds of learning on 10 devices with a fixed number of samples and epochs per training round for the CIFAR-10 and Fashion-MNIST datasets. The data are sampled randomly on each device, creating the non-IID data setting. As per the FL workflow, a global model is deployed to each device, and that model is trained using only local data. Then, the updated weights are returned to the server, aggregated using the weighted FedAvg algorithm, and redeployed to the devices for the next round. The evaluation of individual and aggregated models is done on the controller using a validation set.

We performed centralized training as a baseline for comparison. We fetch the same number of samples from each device (using a function that simply returns a data subset), combine the data together to get a total of 2,000 samples for Fashion-MNIST (200 samples from 10 devices)



(a) Balancing samples (b) Balancing epochs (c) Animals-10

Figure 3: Accuracy vs. time when training on all endpoints for the same duration by balancing the number of samples and the number of epochs. (a) 100–1000 samples per endpoint; (b) 10–60 epochs per endpoint.

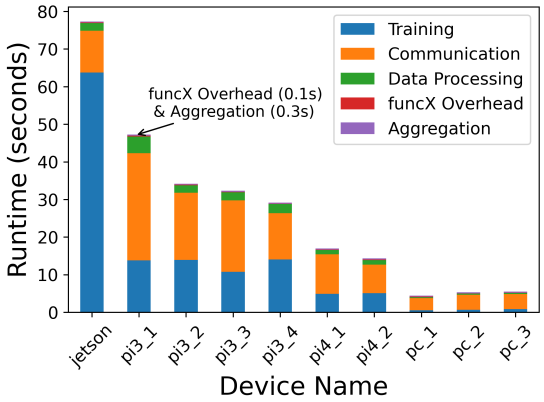


Figure 4: Runtime for each stage of the FL workflow for each device, averaged over 10 rounds. Training uses 100 samples and 10 epochs on the Fashion MNIST dataset.

and CIFAR-10 (400 samples from 5 devices). We then train the model on the controller for 200 epochs (20 epochs from 10 devices) for Fashion-MNIST and 100 for CIFAR-10. We repeat this sequence for 20 rounds. We use the free Google Colab notebook as the controller, enabling the use of a GPU to accelerate centralized training.

CIFAR-10. Figure 2a shows the validation accuracy vs. training time for the centrally trained model, aggregated model, and individual model weights. On average, the aggregated model's accuracy is 7 percentage points (pp) higher than the individual models. The centrally trained model is 4 pp higher than the aggregated model, on average per round, and it also completed training 24 minutes faster. However, training the model centrally required more data to be transferred to the server, which might not be desirable or feasible due to privacy or dataset sizes in some scenarios. Furthermore, the aggregated FL model (represented by the blue line in Figure 2a) converged to almost the same accuracy after 20 rounds, although taking longer.

Fashion-MNIST. Figure 2b shows that the aggregated model's accuracy is 2.5 pp higher than the individual models. The centralized model is 4 pp higher than the aggregated model, on average per round, although in this case the training took 10 minutes longer than for FL. This can be explained by long communication overheads that make aggregating data and training centrally untenable. When the participating devices have suboptimal network capabilities (often present in edge environments [34]), FL can enable a faster training regime.

TABLE 3: # of samples/epochs per device group by dataset for performance-aware balancing experiments.

Dataset	Feature	Jetson	Pi 3	Pi 4	PC
Animals-10	Samples	30	100	200	1000
	Epochs	3	5	10	30
Fashion-MNIST (a)	Samples	50	100	400	2000
	Epochs	10	10	10	10
Fashion-MNIST (b)	Samples	200	200	200	200
	Epochs	5	10	15	60

Given that training the model with FL was faster than centrally on fashion-MNIST, we see *FLoX*'s ability to outperform centralized training when training data are large or network bandwidth is low. Specifically, this demonstration shows that *FLoX* is able to succeed in these network-constrained environments, while enabling device-agnostic and fault-tolerant execution using serverless.

These experiments demonstrate that *FLoX* effectively facilitates FL across heterogeneous devices and datasets.

5.3. Performance-Aware Parameterization

We designed *FLoX* for managing FL in diverse environments comprised of devices with varied capacities. To address this use case, we provide support to vary the number of samples and epochs considered by each device. In the following experiments, we use these options to balance the number of samples and epochs per device based on its capabilities [42]. We use the fashion-MNIST and Animals-10 datasets for this experiment.

Animals-10. The Animals-10 dataset contains color images of 10 different animals, thus presenting a more life-like

TABLE 4: Summary of centralized and federated learning for Fashion-MNIST, CIFAR-10, Animals-10. Parentheticals indicate which parameter was used to balance on for endpoint heterogeneity, if any. AFL = Aggregated Federated Learning.

Dataset	Training Mode	Final Accuracy	Total Time (minutes)	Total Data Transferred per Round per Device (MB)	Total Number of Samples per Round	Total Number of Epochs per Round
Fashion-MNIST	AFL	86%	37	0.55	2000	200
	Centralized	86%	46	0.63	2000	200
	AFL (Samples)	90%	35	0.55	7250	100
	AFL (Epochs)	83%	33	0.55	1000	255
CIFAR-10	AFL	58%	49	4.35	2000	100
	Centralized	60%	25	4.9	2000	100
Animals-10	AFL (Samples & Epochs)	56%	42	4.35	3830	133
	Centralized	58%	54	5.05	3830	133

scenario. To demonstrate *FLoX*'s flexibility for data sources, we retrieve the data from local files. We adapt the number of samples and epochs to each device's capabilities, as per Table 3. Figure 3c shows that the aggregated model outperforms individual models by an average of 5 pp in validation accuracy, reaching 56% on the last training round. While centralized training reached the same accuracy 15 minutes faster than FL approach,, the complete training process took 12 minutes longer for the centralized model due to initial data transfer overheads.

Fashion-MNIST. We now compare varying the number of samples and epochs while keeping the other constant for the Fashion-MNIST dataset. The exact balances used in this experiment are shown in Table 3. To aggregate the model weights, we set the weight for the local updates by the number of samples they were trained on, thus giving more credibility to updates from devices that have seen more data. This functionality is built into *FLoX*, though other aggregation methods can be used by replacing the serverless aggregation function in the workflow.

Figure 3a shows the validation accuracy for aggregated and individual models with balanced number of samples. Figure 3b shows the same with a balanced number of epochs. Both aggregated models have higher accuracy than individual models. However, the balanced samples model achieves higher accuracy in the same time as the unbalanced model from Figure 2b, while the balanced epoch model achieves the same accuracy but in less time.

In future work, we plan to investigate self-adaptation so that the number of samples and epochs could change dynamically based on training performance after each round.

5.4. Discussion

We present consolidated results, training times, transferred data sizes, and other metrics for all experiments in Table 4. This table demonstrates that *FLoX* is consistently able to achieve comparable or superior accuracy compared to centralized training. Additionally, we can see that this is achieved while reducing training time by up to 24% or increasing throughput by more than 80%.

6. Summary

FLoX is a serverless federated learning framework designed to support the deployment of FL models on heterogeneous and distributed devices. We described the current state of federated learning—including the relative sparsity of other serverless-based FL frameworks—and motivated our use of serverless with real-world use cases. After describing our design goals, we illustrated how *FLoX* enables simple and seamless FL by using preexisting funcX endpoints to train and deploy FL models with as little as a single line of code. We showed that *FLoX* can represent real-world FL workflows by using three benchmark datasets and can make effective use of highly heterogeneous compute resources to accomplish superior accuracy and training time when compared with local-only training. We also showed that *FLoX* can aggregate compute resources from 10 relatively low-power devices to achieve model accuracies and training times comparable to centralized training on a server-class GPU. Finally, we showed that the overheads of using *FLoX* are low, enabling deployment on a broad range of edge devices. Overall, *FLoX* demonstrates that simplicity and usability in federated learning need not come at the cost of performance or accuracy.

Acknowledgments

This research was supported in part by DOE contract DE-AC02-06CH11357 and by NSF grants 1816611, 2004894, and 1550588.

References

[1] B. McMahan and D. Ramage, "Federated learning: Collaborative machine learning without centralized training data," *Google Research Blog*, vol. 3, 2017.

[2] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial Intelligence and Statistics*. PMLR, 2017, pp. 1273–1282.

[3] N. Hudson, P. Oza, H. Khamfroush, and T. Chantem, "Smart edge-enabled traffic light control: Improving reward-communication tradeoffs with federated reinforcement learning," in *IEEE International Conference on Smart Computing*, 2022, pp. 40–47.

[4] N. Hudson, M. J. Hossain, M. Hosseinzadeh, H. Khamfroush, M. Rahnamay-Naeini, and N. Ghani, "A framework for edge intelligent smart distribution grids via federated learning," in *International Conference on Computer Communications and Networks (ICCCN)*, 2021, pp. 1–9.

[5] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečnỳ, S. Mazzocchi, B. McMahan *et al.*, "Towards federated learning at scale: System design," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 374–388, 2019.

[6] A. Ziller, A. Trask, A. Lopardo, B. Szymkow, B. Wagner, E. Bluemke, J.-M. Nounahon, J. Passerat-Palmbach, K. Prakash, N. Rose *et al.*, "PySyft: A library for easy federated learning," in *Federated Learning Systems*. Springer, 2021, pp. 111–139.

[7] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, T. Parcollet, P. P. de Gusmão, and N. D. Lane, "Flower: A friendly federated learning research framework," *arXiv preprint arXiv:2007.14390*, 2020.

[8] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "FuncX: A federated function serving fabric for science," in *29th International symposium on high-performance parallel and distributed computing*, 2020, pp. 65–76.

[9] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings *et al.*, "Advances and open problems in federated learning," *Foundations and Trends® in Machine Learning*, vol. 14, no. 1–2, pp. 1–210, 2021.

[10] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage, "Federated learning for mobile keyboard prediction," *arXiv preprint arXiv:1811.03604*, 2018.

[11] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays, "Applied federated learning: Improving Google keyboard query suggestions," *arXiv preprint arXiv:1812.02903*, 2018.

[12] D. C. Nguyen, M. Ding, Q.-V. Pham, P. N. Pathirana, L. B. Le, A. Seneviratne, J. Li, D. Niyato, and H. V. Poor, "Federated learning meets blockchain in edge computing: Opportunities and challenges," *IEEE Internet of Things Journal*, 2021.

[13] W. Li, F. Milletarì, D. Xu, N. Rieke, J. Hancox, W. Zhu, M. Baust, Y. Cheng, S. Ourselin, M. J. Cardoso *et al.*, "Privacy-preserving federated brain tumour segmentation," in *International workshop on machine learning in medical imaging*. Springer, 2019, pp. 133–141.

[14] J. Matschinske, J. Späth, R. Nasirigerdeh, R. Torkzadehmahani, A. Hartebrodt, B. Orbán, S. Fejér, O. Zolotareva, M. Bakhtiari, B. Bihari *et al.*, "The FeatureCloud AI store for federated learning in biomedicine and beyond," *arXiv preprint arXiv:2105.05734*, 2021.

[15] A. Mayampurath, P. Jani, Y. Dai, R. Gibbons, D. Edelson, and M. M. Churpek, "A vital sign-based model to predict clinical deterioration in hospitalized children," *Pediatric Critical Care Medicine*, vol. 21, no. 9, p. 820, 2020.

[16] Y. Liu, T. Fan, T. Chen, Q. Xu, and Q. Yang, "FATE: An industrial grade platform for collaborative learning with data protection," *Journal of Machine Learning Research*, vol. 22, no. 226, pp. 1–6, 2021.

[17] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[18] H. Peng, S. Thomson, and N. A. Smith, "Deep multitask learning for semantic dependency parsing," *arXiv preprint arXiv:1704.06855*, 2017.

[19] Y. Li and J. Yang, "Meta-learning baselines and database for few-shot classification in agriculture," *Computers and Electronics in Agriculture*, vol. 182, p. 106055, 2021.

[20] A. Bhowmick, J. Duchi, J. Freudiger, G. Kapoor, and R. Rogers, "Protection against reconstruction and its applications in private federated learning," *arXiv preprint arXiv:1812.00984*, 2018.

[21] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1175–1191.

[22] R. C. Geyer, T. Klein, and M. Nabi, "Differentially private federated learning: A client level perspective," *arXiv preprint arXiv:1712.07557*, 2017.

[23] B. Ghazi, R. Pagh, and A. Velingker, "Scalable and differentially private distributed aggregation in the shuffled model," *arXiv preprint arXiv:1906.08320*, 2019.

[24] A. Grafberger, M. Chadha, A. Jindal, J. Gu, and M. Gerndt, "Fedless: Secure and scalable federated learning using serverless computing," in *IEEE International Conference on Big Data (Big Data)*, 2021, pp. 164–173.

[25] S. Kounev, C. Abad, I. Foster, N. Herbst, A. Iosup, S. Al-Kiswany, A. A.-E. Hassan, B. Balis, A. Bauer, A. Bondi *et al.*, "Toward a definition for serverless computing," *Dagstuhl Reports*, vol. 11, no. 4, 2021.

[26] Amazon Web Services, "AWS Lambda: Run code without thinking about servers or clusters," https://aws.amazon.com/lambda/. Accessed September, 2022.

[27] Google, "Cloud functions," https://cloud.google.com/functions. Accessed September, 2022.

[28] Microsoft Azure, "Azure functions," https://azure.microsoft.com/en-us/services/functions/. Accessed September, 2022.

[29] Apache, "Apache OpenWhisk: Open source serverless cloud platform," https://openwhisk.apache.org/. Accessed September, 2022.

[30] S. K. Mohanty, G. Premsankar, and M. di Francesco, "An evaluation of open source serverless computing frameworks," in *IEEE International Conference on Cloud Computing Technology and Science*, 2018, pp. 115–120.

[31] S. Tuecke, R. Ananthakrishnan, K. Chard, M. Lidman, B. McCollam, S. Rosen, and I. Foster, "Globus Auth: A research identity and access management platform," in *12th International Conference on e-Science (e-Science)*, 2016, pp. 203–212.

[32] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečnỳ, H. B. McMahan, V. Smith, and A. Talwalkar, "LEAF: A benchmark for federated settings," *arXiv preprint arXiv:1812.01097*, 2018.

[33] K. R. Jayaram, V. Muthusamy, G. Thomas, A. Verma, and M. Purcell, "Lambda FL: Serverless aggregation for federated learning," in *International Workshop on Trustable, Verifiable and Auditable Federated Learning*, Feb. 2022, p. 9. [Online]. Available: https://federated-learning.org/fl-aaai-2022/Papers/FL-AAAI-22_paper_44.pdf

[34] P. Patros, M. Ooi, V. Huang, M. Mayo, C. Anderson, S. Burroughs, M. Baughman, O. Almurshed, O. Rana, R. Chard, K. Chard, and I. Foster, "Rural ai: Serverless-powered federated learning for remote applications," *IEEE Internet Computing*, 2022.

[35] "Sage: Cyberinfrastructure for AI at the edge," https://sagecontinuum.org/. Accessed September, 2022.

[36] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 429–450, 2020.

[37] N. Kotsehub, "FLoX," https://github.com/globus-labs/FLoX, 2022.

[38] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[39] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep. TR-2009, 2009.

[40] Kaggle, "Animals-10," https://www.kaggle.com/alessiocorrado99/animals10. Accessed September, 2022.

[41] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[42] M. Baughman, N. Chakubaji, H.-L. Truong, K. Kreics, K. Chard, and I. Foster, "Measuring, quantifying, and predicting the cost-accuracy tradeoff," in *IEEE International Conference on Big Data*, 2019, pp. 3616–3622.