# FedLess: Secure and Scalable Federated Learning Using Serverless Computing

Andreas Grafberger\*, Mohak Chadha\*, Anshul Jindal\*, Jianfeng Gu\*, Michael Gerndt\*
\*Chair of Computer Architecture and Parallel Systems, Technische Universität München
Garching (near Munich), Germany

Email: {andreas.grafberger, mohak.chadha, anshul.jindal, jianfeng.gu}@tum.de, gerndt@in.tum.de

Abstract—The traditional cloud-centric approach for Deep Learning (DL) requires training data to be collected and processed at a central server which is often challenging in privacysensitive domains like healthcare. Towards this, a new learning paradigm called Federated Learning (FL) has been proposed that brings the potential of DL to these domains while addressing privacy and data ownership issues. FL enables remote clients to learn a shared ML model while keeping the data local. However, conventional FL systems face several challenges such as scalability, complex infrastructure management, and wasted compute and incurred costs due to idle clients. These challenges of FL systems closely align with the core problems that serverless computing and Function-as-a-Service (FaaS) platforms aim to solve. These include rapid scalability, no infrastructure management, automatic scaling to zero for idle clients, and a payper-use billing model. To this end, we present a novel system and framework for serverless FL, called FedLess. Our system supports multiple commercial and self-hosted FaaS providers and can be deployed in the cloud, on-premise in institutional data centers, and on edge devices. To the best of our knowledge, we are the first to enable FL across a large fabric of heterogeneous FaaS providers while providing important features like security and Differential Privacy. We demonstrate with comprehensive experiments that the successful training of DNNs for different tasks across up to 200 client functions and more is easily possible using our system. Furthermore, we demonstrate the practical viability of our methodology by comparing it against a traditional FL system and show that it can be cheaper and more resourceefficient.

Index Terms—Function-as-a-service (FaaS), serverless computing, federated learning, deep learning

### I. INTRODUCTION

Heterogeneous remote edge devices or siloed data centers such as mobile phones or hospitals generate manifolds of data each day [1]. Using deep learning, the data generated by these devices and institutions can be used to enable smart applications [2]. In the conventional deep learning [3] approach, all the raw training data is collected and stored on a central server. However, increasing privacy concerns of data holders and recent legislation on data protection and privacy such as the European General Data Protection Regulation (GDPR) [4] prevent the transmission of data to a centralized location, thus making it impossible to train DNNs on data stored and processed in different locations. Towards this, a new paradigm for distributed ML training called Federated Learning (FL) [5] has been introduced. FL enables the collaborative training of ML models and addresses the fundamental problems of privacy and ownership of data. In FL, remote clients learn a shared model by optimizing its parameters on their local data and sending back the updated parameters. These local model updates are then aggregated to form the new, updated shared model. Clients in FL can be mobile, edge devices, institutions operating their own data centers, or Virtual Machines managed by Infrastructure-as-a-Service (IaaS) providers [6]. In this paper, we argue that both components in a traditional FL system, i.e., clients and server, can immensely benefit from a new computing paradigm called serverless computing.

In serverless computing, developers do not have to manage infrastructure themselves but completely hand over this responsibility to a Function-as-a-Service Platform [7]. Several open-source and commercial FaaS platforms such as Open-Whisk [8], OpenFaaS [9], AWS Lambda [10], and Google Cloud Functions (GCF) [11] are currently available. Applications are developed as small units of code, called functions that are independently packaged and uploaded to a FaaS platform and executed on event triggers such as HTTP requests. In the context of serverless, FL clients are functions deployed on a FaaS platform that are capable of calculating the shared model updates. A connected group of different FaaS platforms is referred to as a FaaS fabric [12].

On the client-side, using FaaS technologies can lead to improvements in resource-efficiency and cost. For instance, a common practice in FL is to only select a small fraction of available clients for each training round. Increasing client parallelism is only beneficial up to a certain point with diminishing returns when more clients are included in each training round [5], [13]. As a result, most clients wait until they are selected in a later round and stay idle, leading to wastage of resources and incurred costs if the clients use an IaaS provider. The idle times are further enhanced due to the potential heterogeneity in clients' computing capabilities or local dataset size (§II-B). On the FL server-side, one common challenge is to efficiently handle sudden bursts of compute-intensive workloads whenever clients report their local updates for aggregation [13]. In addition to wasted compute or scalability challenges, requiring all data holders to manage the complex infrastructure for their clients can be an enormous burden. These challenges of current FL systems closely align with the core problems that serverless technologies and FaaS platforms aim to solve. These include rapid scalability during request bursts, automatic scaling to zero when resources are unused, and an attractive pricing and development model [7].

Towards efficient, scalable, and enabling ease-of-use for FL,

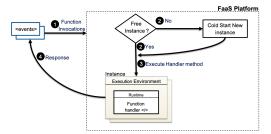


Fig. 1: Workflow demonstrating functioning of the FaaS paradigm.

our key contributions are:

- We present a novel system and framework called *Fed-Less*<sup>1</sup> to perform FL on a heterogeneous fabric of FaaS platforms with support for arbitrary DNN models using Tensorflow [14], proper authentication and authorization of clients, and a privacy-protecting training mechanism.
- FedLess supports four major commercial FaaS platforms, i.e., AWS Lambda, GCF, Azure Functions [15] and IBM Cloud Functions [16], and two major open-source platforms, i.e., OpenWhisk [8] and OpenFaaS [9]. We provide examples and helper functions to run FL client functions on all these platforms.
- We demonstrate with extensive experiments the features and scalability of *FedLess* on three popular FL benchmark datasets using different DNN models.
- We compare *FedLess* with an optimized reimplementation of an existing FaaS based FL system [12] and demonstrate that our system is significantly faster.
- We practically demonstrate the viability of FedLess by comparing it with a traditional IaaS based FL system using Flower [17] with respect to performance and cost.

The rest of this paper is structured as follows. §II provides a background on FaaS and FL. In §III, the previous work on serverless ML, FL, and other FL frameworks are described. §IV describes our system and framework *FedLess*. In §V, our experimental setup is described, while §VI presents our experimental results. Finally, §VII concludes the paper and presents an outlook.

## II. BACKGROUND

## A. Function-as-a-Service (FaaS)

Due to its simplicity, client-friendly cost model, and automatic scaling, FaaS is emerging as a preferred cloud-based programming paradigm. It has gained popularity and widespread adoption in different application domains such as linear algebra [18] and heterogeneous computing [19], [20], [21]. An overview of the functioning of the FaaS paradigm is shown in Figure 1. In FaaS, the user implements fine-grained functions that are executed in response to event triggers or HTTP requests (1). On function invocation, the FaaS platform is responsible for providing resources to the function and its isolation in ephemeral, stateless containers. These containers are commonly referred to as function instances [22], [23]

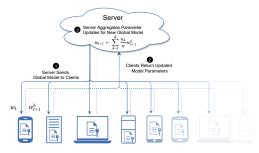


Fig. 2: General synchronous training workflow in FL.

and contain the function's language-specific runtime environment. For commercial FaaS platforms, the function instances are launched on the FaaS platforms' traditional Infrastructureas-a-Service Virtual Machines as described in [24], [25]. The runtime is responsible for relaying the invocation events, context information, and responses between the FaaS platform and the function. On the first invocation of a function, the FaaS platform creates a new function instance, i.e., cold start [26] and runs its handler method to process the event (2-3). When the handler returns a response or exits (4), the function instance remains active for a specific duration to handle successive events. The FaaS platform can create concurrent function instances to handle multiple events or requests ondemand. When the number of requests decreases, the FaaS platform automatically scales down the number of active function instances. For commercial FaaS platforms, users are billed based on the execution time of the functions measured in 100ms or 1ms intervals, i.e., pay-per-use. Moreover, most commercial FaaS platforms limit the maximum execution time and memory a function can have. For instance, with AWS Lambda, these limits are 15 minutes and 10GB, respectively. However, due to the rapid development in FaaS, these limits might disappear or be relaxed soon [27].

# B. Federated Learning (FL)

FL describes a new learning paradigm for data holders to collaboratively train an ML model while keeping their data private. A typical FL system has two main entities, i.e., clients and a central server. Depending on the FL category, i.e., crossdevice or cross-silo, the clients in FL can range from mobile, edge devices to machines running in cloud data centers [6]. In cross-device setting, the number of clients can be significantly large with limited network and compute capabilities, while in cross-silo setting, there are few clients, all with abundant computing resources. Figure 2 describes the general synchronous training workflow for both FL categories. At the start of each round (1), the server sends the current (at the start randomly initialized) global model to all the clients. Following this, each client improves the shared model by optimizing it on its local dataset and sends back only the updated model parameters (2). Finally, the local model updates from all participating clients are collected and aggregated (3). The server usually does not possess its own dataset and is primarily responsible for organizing the training and deciding which clients contribute in a new round. The default implementation

<sup>&</sup>lt;sup>1</sup>https://github.com/andreas-grafberger/fedless

in *FedLess* uses the FedAvg [5] algorithm for aggregating the model parameters. In contrast to traditional distributed ML, the training data present on clients in FL is *non-IID*, i.e., one client's dataset is not representative of the full data distribution across all clients, and *unbalanced*, i.e., a small number of clients can have large local datasets while other clients can contain only a few records.

#### III. RELATED WORK

**Serverless Machine Learning.** The majority of the previous works [28], [29], [30] in this domain have focused on distributed ML training using FaaS functions. Siren [28], allows users to train ML models in the cloud using finegrained functions, thereby removing the burden of non-trivial cluster provisioning and management from the developers. Furthermore, it enables asynchronous distributed training and features an algorithm based on Deep Reinforcement Learning that continuously tunes the number of functions and their memory to optimize performance and cost. In contrast to Siren, Cirrus [29] supports complete end-to-end ML workflows including data preprocessing and hyperparameter tuning. It provides a lightweight worker runtime for the cloud functions that support various ML models. Jiang et al. [30] analyze the cost-performance trade-offs between Infrastructure-as-a-Service (IaaS) and FaaS for distributed ML. Towards this, they developed LambdaML, which supports different distributed ML variants such as synchronous vs. asynchronous training, purely FaaS-based, or a hybrid (FaaS/IaaS) training approach. In contrast to previous works, *FedLess* differs in several ways. First, although FL and distributed ML share some similarities, they have key fundamental differences. For instance, data in FL is private to the client and not accessible by the central server. Furthermore, in distributed ML, workers are rarely idle, whereas, in FL, usually not all workers participate in a round of training. To this end, we designed FedLess from the ground up, keeping FL in mind. Second, none of the tools mentioned above are vendor-agnostic and support only one FaaS platform, i.e., AWS Lambda. In contrast, FedLess supports clients distributed across all major commercial and open-source FaaS platforms. Third, none of the tools consider security aspects for function invocations and model training. On the other hand, FedLess supports cloud-agnostic function authentication/authorization. Finally, FedLess is built on top of Tensorflow [31], which allows easy integration and usage for developers. In contrast, Siren features a runtime based on a pruned version of MXNet [32], while Cirrus does not support any DL framework.

**Federated Learning Frameworks**. There exists several open-source frameworks for FL such as *PySyft* [33], *Tensorflow Federated* (TFF) [34], *Flower* [17], *FedML* [35], *PaddleFL* [36], and *Fate* [37]. Although many of these frameworks provide implementations for different FL algorithms and privacy mechanisms by default, none of them have been developed to be used in a serverless context. Moreover, they require all participating clients to rely on specific communication protocols such as gRPC. As a result, for maximum flexibility and to optimize our implementation for the serverless use-case, we do not use them.

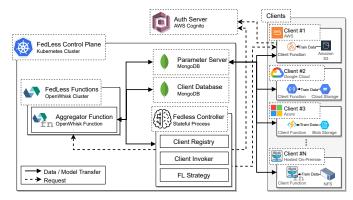


Fig. 3: FedLess System Architecture.

Serverless Federated Learning. To the best of our knowledge, we are the only ones to have tried FaaS for FL. In our previous work [12], we presented FedKeeper, a tool to facilitate FL of ML models across a fabric of heterogeneous FaaS platforms. However, FedKeeper lacks crucial features that are required by FL systems in practice. For instance, it only supports relatively small models, does not provide support for security or a privacy protection mechanism, and is not optimized for performance. In this paper, we present a new system, FedLess that addresses the drawbacks present in FedKeeper. Moreover, FedLess is designed to be modular and can be easily extended to support additional data sources, models, and optimization algorithms.

#### IV. FEDLESS

FedLess is a system and framework to perform FL on a heterogeneous fabric of FaaS platforms. In this section, we first describe the overall architecture of FedLess. Following this, we present an overview of its implementation, describe its security features, the mechanism for privacy-preserving training of clients, and performance optimizations that improve scalability and reliability in detail. Finally, we describe the workflow for training clients using FedLess.

#### A. System Design

Figure 3 gives an overview of the different components of *FedLess*. In the serverless context, we refer to an FL *client* as the associated *client function*, i.e., the function deployed on a FaaS platform that calculates the local model updates. A *client administrator* is the person who is responsible for managing and deploying the function and is usually also the data holder. The FL training process and the registration of clients are managed by the FL *administrator* (FL admin).

FedLess supports both cross-silo and cross-device FL usecases (§II-B). In the cross-silo setting, client functions would belong to a few institutions and either run in on-premise data centers with self-hosted FaaS platforms or on the cloud. In the latter, client functions would run on edge devices with lightweight FaaS platforms such as faasd (OpenFaaS) [38]. All clients are externally deployed functions that are callable through an HTTP interface. By default, FedLess supports four commercial and two open-source FaaS platforms. We provide scripts for deploying client functions on these FaaS platforms. Apart from setting up optional parameters, such as request limits, no further action is required from the user.

The central component in *FedLess* is the control plane which comprises of the Parameter Server, the Client Database, an OpenWhisk cluster to run the Aggregator Function, and the FedLess Controller. The controller comprises of a Client Registry, Client Invoker, and the FL Strategy. The default FL strategy implemented in our system is the FedAvg algorithm [5]. Due to its ease-of-use, reliability, replication support, and security features, we chose MongoDB [39] as our parameter server. Moreover, we make use of MongoDB's GridFS specification to support models larger than the document size limit, i.e., 16MB. The global model architecture and its hyperparameters are also stored in the parameter server. The Client Database is also a MongoDB instance that serves as a persistent registry for the client functions, their hyperparameters, and dataset locations, along with any additional information supplied by the client administrator during sign-up. The Controller is a lightweight process that manages and monitors the entire training cycle. It is responsible for selecting the clients that participate in each round, invoking them and the aggregator. The client registry component in the controller interacts with the client database for managing the clients. The aggregator is a FaaS function that is invoked by the controller after all clients that participated in a training round have finished. It aggregates the updated client model parameters according to the FL strategy.

New FL strategies other than FedAvg can easily be implemented by modifying how the controller samples clients for each round and writing a custom aggregator function, taking the one provided as a blueprint. Having the parameter server physically close to the aggregation process is a crucial advantage of our approach due to minimized networking overhead, as is also presented in [30]. Although we deploy the aggregator function using OpenWhisk as shown in Fig 3, it can be easily deployed using another FaaS platform. The *Auth Server* is an external entity responsible for authentication and is described in detail in §IV-C2.

#### B. Implementation Overview

The *FedLess* framework was implemented using Python3 and also includes a command line tool to orchestrate the entire FL training process. It relies on Tensorflow [14] and Keras [40] to support arbitrary DNN models. Our framework provides:

- Low-level, easily exchangeable implementations for serialization methods, helper functions for tasks related to security and general optimizations, as well as classes for easy database access and interaction with the parameter server.
- Functions and decorators to abstract away differences of supported FaaS platforms, in turn, unifying the implementation of client functions.
- Extensive example implementations of client and aggregation functions that serve as easily modifiable blueprints.
- Custom data types and models using the Pydantic [41] library, which ensures that all clients and the FL server rely on the same data formats and schemata.

#### C. Security

Since the clients in FL belong to separate institutions and networks and have to expose their functionality to the public internet to be accessible by the FL server, it is significantly important that only authenticated and authorized entities can invoke client functions. The security features in *FedLess* are described in the following four aspects.

- 1) Function Ownership: In FedLess, all participating institutions and users that manage one or multiple clients are responsible for deploying the client functions themselves. This is because data holders have complete control over what happens with their data. Moreover, this approach provides greater flexibility for institutions to adjust their complete endto-end local training workflows. In addition, full oversight on security and access control is essential, especially for domains such as healthcare [42] with special legal regulations for the data. Clients deployed on different FaaS platforms can directly use the authentication system provided by FedLess (§IV-C2). However, since data holders are free to use different FaaS platforms, they can require additional, platform-specific authentication secrets. A data holder could, for example, deploy a client function with OpenWhisk and specify an API token that has to be supplied by FedLess with every request. Our system supports additional API tokens in client invocation requests. We provide several examples for writing client functions using our framework for multiple FaaS platforms.
- 2) Authentication and Authorization: In a centralized FL system, like FedLess, multiple security interests come together. For instance, data holders require that only the central FL server they integrate with can call their client functions. This means that the central server has to authenticate itself and possess proper authorization. At the same time, before being added to the list of valid client functions for an FL training session, data holders have to identify themselves to the FL server and provide appropriate proof. As a result, we need to support authentication, authorization, and integration with external identity providers through protocols like OAuth 2.0 [43] and SAML 2.0 [44]. Integration with external identity providers enables easy and direct support for identity providers already used by academic institutions and enterprises. Towards this, we use AWS Cognito [45]. It is a service that provides all the required features, has been used in large-scale production systems, and has a convenient pricing structure that scales with the size of the system. Furthermore, it does not restrict FedLess to a specific cloud provider's ecosystem since the interaction with Cognito is the same for all the clients.

The authentication and authorization system is based on *JSON Web Tokens* [46] (JWTs) attached to all HTTP requests between the FL server and the clients. Client administrators manually register through a web interface with a Cognito User Pool, offering sign-up via external identity providers. Once they submit their registration request, a central FL server administrator checks the validity of the registration information and confirms it. After the registration process, all necessary information for the client function to check the authenticity and permissions of requests by the FL server is then communicated to the client administrator via a web

interface. This includes the public key of the key-pair used to sign the authentication headers, the FL server's client-id, and required authorization scopes. At the start of each training round, the FL server uses its credentials to fetch a token from Cognito with sufficient privileges to invoke all client functions. For each new request, all clients first check if the token supplied with the request by the FL server was signed by the trusted Cognito User Pool and has the proper authorization.

- 3) Parameter Server Access: The different client functions need to access the parameter server without compromising security and privacy. For instance, one client function should not be able to read the result of another client or modify the global parameters. To this end, the FL server creates temporary users in the parameter server for each client function that can only read the shared global model and write back results without access to other clients. In every request to a client, the FL server attaches the credentials with which the client can retrieve the global model and write back its results to the parameter server. Furthermore, because the credentials given to the client function belong to a custom user, the access to the parameter server could be easily revoked for individual client functions. Our parameter server, i.e., MongoDB (§IV-A) allows the definition of such custom users and role policies.
- 4) General Security Features: The Open Web Application Security Project [47] represents the top ten issues that are widely accepted to be most common and critical in software applications. While most security concerns depend on the system and function implementation details and lie in the hands of the data holders, we counter some of the most common vulnerabilities with our system design and implementation in FedLess. First, since all function requests use HTTP, we can easily enforce all communication to use Transport Layer Security (TLS) to encrypt the exchanged data. Second, to protect against insecure serialization and injection attacks, we advise data holders to use the model schemata provided in our framework implemented using the Pydantic [41] library. Third, we provide examples and demonstrate how to isolate all authentication and authorization by using separate functions as suggested by [48].

## D. Privacy

The two main concerns in an FL system in which not all participants can be trusted are poisoning and inference attacks [49], [50]. As a result, several strategies to protect the privacy of clients for FL have been proposed. These include Secure Multiparty Computation (SMC) [51], Homomorphic Encryption (HE) [52], and Differential Privacy (DP) [53], [54]. While SMC and HE involve encrypting the client updates, DP involves adding Gaussian noise to them. Due to its reliable privacy guarantees at the cost of degraded prediction accuracy, it is a common practice to combine DP with SMC or HE methods [6], [51]. This is because due to encryption, less noise would have to be added to the client updates. However, a drawback of these encryption-based approaches is that they require a large amount of memory and have significant computational and networking costs [52], [55], [56]. This makes them unsuitable for most FL applications [57]. For FaaS, using these methods leads to function timeouts due to fixed limits (§II-A). As a result, we do not employ secure aggregation in our system. By default, *FedLess* supports *Local Differential Privacy* [58], [59], [60] (LDP). LDP is a form of *record-level* privacy [61] in which the client functions add noise to their parameters before uploading them to the parameter server.

We use the Tensorflow Privacy [62], [63] library that provides DP versions of popular optimization algorithms that are also compatible with Keras. A minor caveat of using LDP with ephemeral serverless functions is keeping track of past invocations to a client function through an external storage system. Subsequent invocations of the client function degrade the privacy guarantees [59]. As a result, data holders would want to prohibit requests by the FL server after a specific privacy budget has been exhausted. For this, they would need to store their state between requests. All of this, combined with our reference implementation of using DP optimizers with client functions, makes it easy for data holders to slightly modify their client function code to benefit from DP and provide proven privacy guarantees for their user data. Investigating the use of other secure aggregation methods such as Additive Masking [57] which require complex interaction between clients, is of our interest in the future.

# E. Performance Optimizations

- 1) Global namespace caching: Due to the ephemeral, stateless nature of FaaS functions, they need to load and preprocess the dataset, compile and instantiate the model on every invocation. This overhead becomes significantly large for large local datasets and models. To mitigate this, we exploit that all cloud providers and FaaS frameworks retain global variables in the function instances between invocations. Towards this, we implement a custom Python caching decorator that, after the first function invocation, stores the output of the Python function in an in-memory LRU cache. All subsequent calls to the function return the cached result. Note that the cached result is only available until the duration of the function instance (§II-A).
- 2) Running average model aggregation: In a naive implementation of FedAvg, the aggregator function has to load all model updates in memory for the aggregation. However, due to the limitations on the amount of memory configurable for FaaS functions (§II-A) on most commercial FaaS providers, this approach is not feasible. To this end, we implement a running average mode in the aggregator function. This is done by loading the client updates from the parameter server only in small batches. After a batch is loaded, we use it to update the current average global model parameters and free up the memory occupied by the processed batch.
- 3) Federated Evaluation: The aggregator function can evaluate the updated global model after each round if a global test exists. However, in FL, since this is rare, we also support client-side evaluation as in [64]. Each client can be invoked independently to run the global model on a local test set and return the evaluation metrics. The metrics can then be aggregated by the controller. Moreover, independent selection of clients for training and evaluation between rounds minimizes function cold starts.

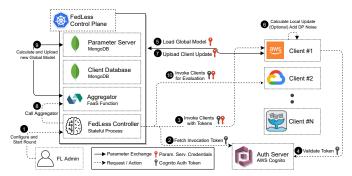


Fig. 4: FedLess workflow for a complete FL round.

# F. Putting It All Together

The workflow for training multiple clients using FedLess in a single FL round is presented in Figure 4. First, the FL admin (§IV-A) selects the model to be trained, the registered client functions that will participate in the training, and hyperparameters such as the number of clients per round (1). Following this, the training is started by the FL admin. The FedLess controller first requests a new invocation token from the Auth Server using the credentials configured by the FL admin (2). Using this token along with the credentials to access the parameter server, the controller invokes the clients it selected for this round (3). The clients involved in an FL training round are selected randomly by the controller. To ensure that the invocation is valid, the clients validate the signature and authorization of the token using the public key from the Auth Server (4). The clients then use the supplied parameter server credentials to load the latest global model (5). Following this, the clients load their local datasets and perform the local training, optionally using LDP (6). Once the clients have finished training, they again use their credentials to upload their parameters to the parameter server (1). The FedLess controller waits until all clients are either finished, reached a configured timeout, or failed. It then starts the model aggregation by invoking the aggregator function (8). The aggregator loads the client results from the parameter server, aggregates the parameters, and stores the new global model (9). Finally, the controller starts the evaluation (10). If a global test set exists, then the evaluation was already done in the aggregator. If not, a new selection of clients is invoked for evaluation in which they load the updated model and evaluate on their test set (§IV-E3). The controller aggregates the returned metrics and resumes the training process from (2) if a configured accuracy threshold is not reached.

#### V. EXPERIMENTAL SETUP

In this section, we describe the datasets and DNN model architectures we used for evaluating *FedLess*. Furthermore, we describe the distribution and configuration of client functions across the different FaaS platforms.

#### A. Datasets

**MNIST**: The MNIST Handwritten Image Database [65] comprises 70,000 images of handwritten digits with ten classes

corresponding to the respective digits. We use all 10,000 test images of the dataset to evaluate the global model after each training round centrally. To simulate a non-IID setting as in [5], [12], we sort the 60,000 images in the training set by label, split them into 200 shards of 300 images each, and distribute the shards to the clients. This dataset represents an image classification task.

We also perform experiments on more realistic datasets from the *LEAF* benchmarking framework [66]. The datasets chosen are described below and are by nature non-IID. For both datasets, we use the same preprocessing steps as in [66].

**Shakespeare**: This dataset contains sentences from the *The Complete Works of William Shakespeare* [67], partitioned by the speaker and play. In total, it contains 4,226,158 utterances of length 80 by 1,129 different *users*, with on-average  $\sim 3,700$  for each. The task of this dataset is to predict the next character in a sentence given the previous 80.

**FEMNIST**: It is a modification of the EMNIST [68] dataset and contains over 800,000 images of digits and characters partitioned by the writer. In total, it contains contributions from 3,550 writers with  $\sim$  226 images from each on average. Similar to MNIST, the task of this dataset is to classify images.

In our experiments, we serve all datasets using an *nginx* [69] store with each dataset partition being available at a separate URL and assign it to one client.

#### B. Model Architectures and Hyperparameters

MNIST: We use the same CNN as in [12], consisting of 2 convolutional layers with kernels of size 5x5, a fully connected layer with 512 neurons, and the output layer with ten neurons. The network uses ReLU as the sole activation function and the categorical-cross-entropy loss function. In total, the model has 582,026 trainable parameters, which amount to 2.3 MB of memory when serialized. We use a batch size of 10 as in [12] with five local epochs for each client. Because we experienced faster convergence with momentum-based optimization methods, we used Adam as the optimizer.

**Shakespeare**: Similar to [66], we use an *LSTM* [70]. First, the characters in sequences of length 80 are each embedded in an eight-dimensional space, followed by two LSTM layers, each comprising 256 units. In the end, the output layer has the same size as the vocabulary and a softmax activation function. Like [66], we optimize the model using standard Stochastic Gradient Descent with a learning rate of 0.8 and use the categorical cross-entropy as the loss function. The LSTM has 818,402 trainable parameters, which take  $\sim$  3MB in serialized form. Due to long training times of *LSTMS*, and limitations with function timeouts in current FaaS providers (§II-A), we use a batch size of 32 with one local epoch.

**FEMNIST**: As in [66], we use a large CNN model with two convolutional layers, each followed by a max-pooling layer, followed by a fully-connected layer of size 2048 before resulting in the output layer of size 62. The final model size comes down to 6,603,710 trainable parameters, ~ 26.4MB when serialized. Similar to MNIST, we use a batch size of 10 with five local epochs and Adam as the optimizer. For both MNIST and FEMNIST, we use the default values for the learning rate.

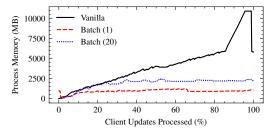


Fig. 5: Decreased memory footprint of our aggregator implementation. Results are generated in a simulation with 200 client updates of the FEMNIST model.

To calculate a global test accuracy and loss for FEMNIST and Shakespeare, we use the reported test set evaluation metrics from the clients and calculate a weighted average based on the test set cardinalities from the clients (§IV-E3,§IV-F).

#### C. Serverless client functions configuration and distribution

To host our parameter server, we used a virtual machine (VM) with 40vCPUs and 177GB RAM on the LRZ compute cloud [71]. On the same VM, we ran an OpenWhisk cluster on top of Kubernetes. All client functions are configured with a memory limit of 2048MB, and functions using public cloud providers are deployed in the same region, i.e., Frankfurt, Germany. Only the memory limit for functions on Azure could not be directly configured since it uses dynamic memory allocation with a maximum memory value of 1.5GB per function. We set the memory limit for the aggregation function of *FedLess* to 4096MB (§IV-A).

For all the three datasets (§V-A) we ran experiments with 200 clients in total and varied the number of clients sampled in each training round. For MNIST and FEMNIST, we deployed 170 Google Cloud Functions (GCF), 10 AWS Lambda functions, 10 IBM Cloud Functions, 5 OpenWhisk functions on an on-premise cluster, and 5 Azure Functions. Training LSTMs using a large number of simultaneous client functions up to convergence is out of our cloud budget. As a result, for the Shakespeare experiments, we only use up to 25 simultaneous clients.

#### VI. EXPERIMENTAL RESULTS

We repeat all our experiments five times and follow best practices while presenting our results [72].

# A. Analyzing FedLess Performance

As described in §IV-E2, due to the limitations on the amount of memory configurable for a FaaS function, using a standard FaaS function for the aggregation would not be possible for large-scale scenarios. Figure 5 demonstrates the behaviour of our modified FedAvg implementation in a simulated setting. Aggregating the parameters of 200 clients for the FEMNIST CNN (§V-B) would require more than 6GB of memory for the vanilla FedAvg implementation. In contrast, using batches of 20 updates, our batched running average calculation requires slightly more than 2GB of memory. Note that the overhead created by our implemented security features (§IV-C4) is

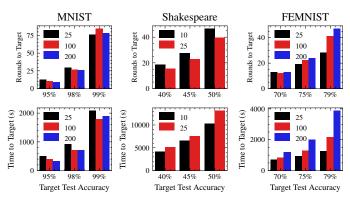


Fig. 6: Convergence speed of FedLess for different datasets and numbers of clients per round. Each dataset is shown in a separate column.

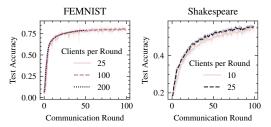


Fig. 7: FedLess test accuracies over time for FEMNIST and Shakespeare.

minimal. The actual token validation and checking takes only a few milliseconds in our client functions. Moreover, after a client fetches the token from the Cognito server, we cache it in the function instance's memory, requiring no further requests (§II-A).

The number of training rounds and the overall time taken by FedLess to reach a specific target accuracy for the three datasets is shown in Figure 6. Results are shown for different numbers of clients involved in each round. For MNIST, we observe that increasing the number of clients involved in each round did not significantly influence the overall number of required rounds. Using a higher number of clients per round did speed up training in terms of required rounds and time for reaching accuracies less than 99%. However, for higher accuracies, the difference is not that evident. Similarly, for Shakespeare, we observe that more clients per round lead to faster convergence only with respect to communication rounds. This behavior is also visible from Figure 7 and can be attributed to the remarkable variance in the execution time of each client in this dataset due to the non-IID distribution (§II-B). As a result, using more clients per round increases the chance of waiting longer for stragglers, increasing the overall time until a certain accuracy is reached. For the FEMNIST dataset, using more clients per round leads to slower convergence in both dimensions, i.e., communication rounds and time to reach the target accuracy. This can be attributed to the large variance in the average reported test set accuracy as shown in Figure 7. Using fewer clients in each round for evaluation leads to a larger variance in the average evaluation metrics. The overall training behavior for different

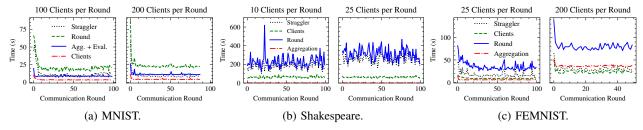


Fig. 8: Mean required time for different steps in FedLess for the different datasets. *Straggler* shows the runtime of the slowest client. If present, *Agg.* + *Eval.* shows the runtime of the aggregator function that also performs evaluation (§IV-F).

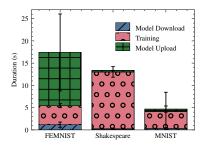


Fig. 9: Time distributions for FedLess client functions. Numbers are based on one client function and multiple training rounds for all three datasets.

numbers of clients is approximately the same, showing no significant difference in using more clients per round for our chosen hyperparameters.

Figure 8 shows how long the different actors in our system take over the course of the training for the different datasets. Across all experiments, the total round time is mainly determined by the slowest client of each round, the straggler. In each training round, vanilla FedAvg waits for each client to finish or reach a timeout. As a result, only one straggler can drastically increase the total round time. For MNIST, increasing the number of clients from 100 to 200 per round does not impact their execution time, as shown in Figure 8a. The aggregator function becomes slightly slower since it has to load and aggregate results of more clients, slightly increasing the overall round time. Based on these findings, we see that for small local datasets and small model sizes, FedLess scales relatively easily. A crucial benefit of our system is that longer round times due to stragglers have almost zero influence on the overall costs since client functions are only billed for their actual runtime and not the time they wait for a new round. Similarly, for the Shakespeare dataset, increasing the number of clients from 10 to 25 did not impact the execution time of the clients, as shown in Figure 8b. However, sampling more clients increases the chance of including clients that take a long time to finish. Thus, the overall round time increases. Note that the time required for aggregation or client-side evaluation is negligible for this dataset compared to the time it requires to wait for stragglers. Using an FL strategy that accounts for stragglers [73] is of our interest in the future but out of scope for this work.

For the FEMNIST dataset with the large CNN model (§V-B), increasing the number of clients per round from 100

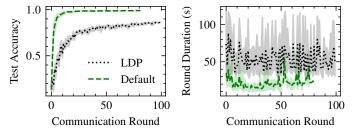


Fig. 10: Test accuracies and increased round durations for MNIST when using FedLess with LDP across 200 client functions with 25 clients per round.

to 200 almost doubles the total time per round. Unlike the other two datasets, we observe an increase in the execution time of both the aggregator and the clients. The increase in the aggregator's runtime can be attributed to the networking overhead of loading 200 × 26MB of parameters and calculating the aggregated model using a running average. The reason for the increase in the mean execution time of the clients can be seen from Figure 9. While the clients training on Shakespeare and MNIST spend most of their time on actual model training and very little on loading or writing parameters from and to the parameter server, FEMNIST clients spend a significant portion of their execution time writing back their results to the parameter server. Since the parameter download time is much less than the upload time and shows little variability, we can confidently say that the overall networking capabilities of the clients are not the problem. Therefore, the ability of MongoDB's GridFS specification to handle multiple simultaneous uploads of large models is the root cause for the increase in the mean execution time of clients. Investigating the use of another high-performance parameter server with role-based access control rules (§IV-C3) is part of future work. For all datasets, the time for initial rounds is high due to cold starts associated with FaaS functions (§II-A).

#### B. Local Differential Privacy

To demonstrate that it is feasible to use LDP with our system, we train a model on MNIST with 200 client functions (§V-B, §V-C). We use 25 clients per round, five local epochs, and a batch size of five. After a grid search for the hyperparameters specific to the Adam-DP optimizer in the TF-Privacy library, we use a noise multiplier and L2-clip norm of 1.0, and ten microbatches. The results with LDP as compared to a default MNIST run are shown in Figure 10.

	FL Round Duration (s)	
Clients	FedKeeper	FedLess
25	$18.4 \pm 9.6$	$12.3 \pm 4.6$
50	$17.9 \pm 11.5$	$12.8\pm4.2$
75	$19.0\pm12.2$	$14.0\pm3.8$

TABLE I: FL Round durations of FedKeeper and FedLess for varying numbers of clients. FedKeeper clients in these experiments already made use of our caching mechanism.

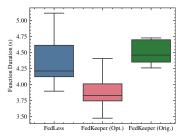


Fig. 11: Client execution times for FedKeeper and FedLess on MNIST without cold-starts.

Due to budget constraints, we do not train the LDP model until convergence but demonstrate that it is possible to use LDP with our system and different supported cloud providers. The significant decrease in test set accuracy reached by our model using LDP in 100 rounds as compared to without can be attributed to the addition of Gaussian noise to the model parameters. However, the same approach might not work for all model sizes directly due to the slower training using DP optimizers. Due to the function execution time limits (§II-A) on most commercial FaaS providers, training larger models such as LSTMs with LDP will often lead to a function timeout. One could mitigate this problem by only training on a certain number of batches instead of whole epochs or using a smaller microbatch size.

## C. Comparison with Fedkeeper

To see how well *FedLess* performs compared to *Fed-Keeper* [12], we re-implemented *FedKeeper* with our *FedLess* components. Our implementation works almost identical to the original implementation. However, it uses the modular components we implemented for *FedLess*, and we introduced performance improvements like caching.

We deploy 100 client functions with 2048MB memory for both systems on GCF. We only use one cloud provider to minimize the impact of performance variations between FaaS platforms. We use an on-premise OpenWhisk cluster for the aggregation for both systems and invoker functions for *Fed-Keeper* (§V-C). While in *FedLess* the FL client functions are directly invoked by the client invoker in the *FedLess* controller (§IV-A), *Fedkeeper* uses a separate component called invoker functions which are responsible for invoking the FL clients via an HTTP request. The invoker functions are FaaS functions configured with a memory limit of 256MB. We compare both systems using the MNIST dataset and model (§V-A, §V-B) for 25, 50, and 75 clients per round.

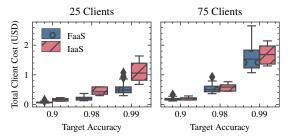


Fig. 12: Total client costs of FaaS and IaaS to reach a target accuracy on MNIST. Boxplots visualize rough upper and lower bounds depending on the IaaS and FaaS provider's compute capabilities.

Table I shows the round duration for both the systems. We observe that FedLess is always significantly faster than FedKeeper with less variability in the round times. The large variance in the round times of FedKeeper can be attributed to the cold starts of the invoker functions and their varying runtimes. We re-ran these experiments and exchanged one client function of FedKeeper with the original, unoptimized version without caching. The different client execution times are shown in Figure 11. Caching the model architecture and local dataset in-memory of the function instance reduces the median client execution time from 4.46 seconds to 3.83 seconds, a speed-up of roughly 14%. Only the optimized client functions of FedKeeper are slightly faster than the FedLess clients, whose median execution time is 4.21 seconds. The additional time in the FedLess clients is spent on communicating with the parameter server directly instead of through the invoker functions, which is slightly faster. These results clearly show that FedLess is faster and imposes much fewer hardware requirements due to eliminating invoker functions.

## D. Comparing IaaS and FaaS for FL

To compare how our system performs compared to a traditional, non-FaaS based FL system, we implemented a baseline system using the open-source FL framework *Flower* [17]. We chose Flower since it was easy to modify, extend, and integrate into our existing experimentation pipeline. Furthermore, it provides baseline implementations for various FL strategies. For our experiments, we use the default FedAvg implementation. For the communication between the central server and the clients Flower uses gRPC. To support invoking a large number of clients in parallel, we had to modify the current Flower implementation. For our IaaS based experiments, we use cgroups to simulate compute instances similar to generalpurpose instances of public cloud providers (e.g. m4.large on AWS EC2 [74], n2-standard-2 on Google Cloud Compute Engine [75]) on the LRZ compute cloud [76]. This is also similar to the setup used in [29] (mostly 2 CPUs, 8GB RAM), and similar to some workloads in [30]. We deploy all Flower clients in separate Docker containers that can each use two vCPUs and 8GB memory. Like in [30] our central server for Flower has CPUs and RAM similar to a c5.4xlarge instance on AWS EC2, i.e., 16 vCPU cores and 32 GB RAM [74]. For a fair comparison, we deploy the client functions for FedLess

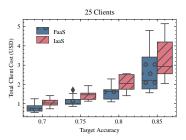


Fig. 13: Total client costs of FaaS and IaaS to reach a target accuracy on FEMNIST.

on the same hardware using OpenFaaS. OpenFaaS allows us to limit each client function to 2 vCPU cores and 2GB of memory. As a result, the *FedLess* client functions have the same processing power as the Flower clients. We compare both systems using the MNIST and FEMNIST datasets with 100 clients using the same models and hyperparameters as described in §V-B.

Since both systems use FedAvg and the same hyperparameters, we are primarily interested in timing and pricing differences. Due to the ephemeral, stateless nature of FaaS based systems, we observe that Flower is faster than FedLess in all experiments. The relative difference depends on the dataset used. To assess this difference, we use the geometric mean to calculate the mean training round times for FedLess and Flower, as proposed in [77]. The mean training round times for FedLess are on average  $1.77 \times$  as long for FEMNIST compared to Flower, whereas they take 1.6× as long for MNIST. However, a key advantage of our system is that client functions that are already finished in a round do not cost anything while waiting and can be scaled to zero. In traditional FL systems, like the one we implemented with Flower, clients simply turn idle while waiting for further requests, accumulating cost, or blocking hardware resources.

To understand the differences between FaaS/IaaS for FL in terms of cost, we created pricing estimates for MNIST and FEMNIST, based on the results of our previous experiments. We base these calculations on the assumption that all clients run on Google Cloud, both for IaaS and FaaS-based training. The compute resources we used for the Flower clients are similar to n2-standard-2 VM instances in both vCPUs and memory [78]. For the *FedLess* clients, we use the execution time presented in experiments from §VI-C (MNIST), §VI-A (FEMNIST). Since the Fedless clients use the same parameter server, hyperparameters, and local datasets, the comparison is fair. For *FedLess* and Flower, we calculate the mean estimated cost of the training runs for both datasets. To not get results skewed by implementation differences, we assume FedLess and Flower take an equal number of rounds to reach a particular target accuracy. In addition, for both FaaS functions and IaaS VMs, we ignore additional costs like disk space, costs for cloud buckets, costs to set up and deploy clients, and free-tiers. All prices are calculated for the us-central1 region. For our experiments, we only calculate direct costs. For Google Cloud Functions, these are GB-seconds and GHzseconds [79], invocations and networking costs, and for VMs,

they are instance runtime and networking costs.

Since we can only provide estimates, we do not calculate point estimates for the pricing but compute broader bounds. Towards this, we do not only take individual client functions execution time and total training time from the FedLess/Flower experiments but additionally include in our calculations how costs would change if the *FedLess* client functions took  $2\times$  and  $3\times$  as long, and the Flower clients only took  $0.5\times$  as long. For the MNIST dataset, the total training costs for both systems until certain target accuracies are reached for different numbers of clients per round is shown in Figure 12. We see that overall, the clients' costs are smaller with our FaaS-based approach for the shown settings. Especially further into the training, pricing differences are amplified. However, the cost benefits of using FedLess are larger for a smaller number of clients sampled per round. This is intuitive since the more clients are used per round, the less compute time is wasted in the IaaS based approach. Comparing the results for 25 clients per round for MNIST and FEMNIST in Figures 12 and 13, we observe that although FedLess is still less expensive, the relative difference becomes smaller for larger model sizes. The networking cost due to uploading the updated model is the same for FaaS and IaaS and is responsible for a significant portion of the overall cost. Based on our experiments, we gather first evidence that a FaaS-based FL approach is more cost-efficient for settings where only a fraction of active clients participate in a round. Investigating these trade-offs in detail is part of our future work.

# VII. CONCLUSION AND FUTURE WORK

In this paper, we presented a novel system and framework for serverless FL called *FedLess*. *FedLess* enables FL across a large fabric of heterogeneous FaaS providers while providing important features such as authentication, authorization, and differential privacy. We demonstrated with comprehensive experiments the features and scalability of our system. In comparison with a traditional IaaS based FL system, we demonstrated the practical viability of our system and showed that, albeit being slower, it can be cheaper and more resource-efficient.

In the future, we plan to investigate the benefits of switching from our current hybrid system design to a fully event-driven system using message queues and publish-subscribe systems such as Rabbitmq. It also remains to be seen how well our FaaS-based approach works for even larger datasets and models. Although the conventional client heterogeneity in FL is less of a problem in FaaS as compared to traditional IaaS based approaches since idle clients do not cost anything and can scale to zero, we plan to investigate efficient FL using FaaS, which accounts for clients' heterogeneous compute requirements or dataset sizes.

### VIII. ACKNOWLEDGEMENT AND REPRODUCIBILITY

This work was supported by the funding of the German Federal Ministry of Education and Research (BMBF) in the scope of the Software Campus program. Google Cloud credits in this

work were provided by the *Google Cloud Research Credits* program with the award number NH93G06K20KDXH9U.

All code artifacts related to this work are available at<sup>2</sup>.

#### REFERENCES

- M. Chiang and T. Zhang, "Fog and iot: An overview of research opportunities," *IEEE Internet of things journal*, vol. 3, no. 6, pp. 854– 864, 2016.
- [2] Nvidia Clara, "NVIDIA Clara NVIDIA Developer," 2020. [Online]. Available: https://developer.nvidia.com/blog/federated-learning-clara/
- [3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [4] 2018 reform of EU data protection rules. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2016/679/oj
- [5] 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.
- [6] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konečný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, M. Raykova, H. Qi, D. Ramage, R. Raskar, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao, "Advances and Open Problems in Federated Learning," p. 16, dec 2019. [Online]. Available: http://arxiv.org/abs/1912.04977
- [7] S. Allen, B. Browning, L. Calcote, A. Chaudhry, D. Davis, L. Fourie, A. Gulli, Y. Haviv, D. Krook, O. Nissan-Messing, C. Munns, K. Owens, M. Peek, C. Zhang, and C. A., "CNCF WG-Serverless Whitepaper v1.0," CNCF, Tech. Rep., 2018. [Online]. Available: https://github.com/cncf/wg-serverless/blob/master/ whitepapers/serverless-overview/cncf\_serverless\_whitepaper\_v1.0.pdf
- [8] The Apache Foundation, "Apache OpenWhisk is a serverless, open source cloud platform," 2018. [Online]. Available: https://openwhisk.apache.org/
- [9] OpenFaaS, "OpenFaaS Serverless Functions Made Simple," 2019. [Online]. Available: https://www.openfaas.com/https://docs.openfaas.com/
- [10] Aws, "AWS Lambda Serverless Compute Amazon Web Services," 2020. [Online]. Available: https://aws.amazon.com/lambda/
- [11] Google Cloud, "Cloud Functions Google Cloud," 2020. [Online]. Available: https://cloud.google.com/functions/
- [12] M. Chadha, A. Jindal, and M. Gerndt, "Towards Federated Learning Using FaaS Fabric," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, ser. WoSC'20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 49–54. [Online]. Available: https://doi.org/10.1145/3429880.3430100
- [13] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, H. B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander, "Towards federated learning at scale: System design," 2019.
- [14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, vol. 10, no. July, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-liu.pdf
- [15] M. Azure, "Azure Functions Serverless Compute Microsoft Azure," 2021. [Online]. Available: https://azure.microsoft.com/en-us/services/ functions/
- [16] "IBM Cloud Functions IBM." [Online]. Available: https://www.ibm. com/cloud/functions
- [17] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, T. Parcollet, N. D. Lane, P. P. B. de Gusmão, and N. D. Lane, "Flower: A Friendly Federated Learning Research Framework," arXiv, pp. 1–22, jul 2020. [Online]. Available: http://arxiv.org/abs/2007.14390
  - <sup>2</sup>https://github.com/andreas-grafberger/fedless

- [18] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless linear algebra," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 281–295. [Online]. Available: https://doi-org.eaccess.ub.tum.de/10.1145/3419111.3421287
- [19] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen, "Function delivery network: Extending serverless computing for heterogeneous platforms," *Software: Practice and Experience*, vol. 51, no. 9, pp. 1936–1963, 2021. [Online]. Available: https://onlinelibrary.wiley.com/ doi/abs/10.1002/spe.2966
- [20] A. Jindal, M. Chadha, M. Gerndt, J. Frielinghaus, V. Podolskiy, and P. Chen, "Poster: Function delivery network: Extending serverless to heterogeneous computing," in 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), 2021, pp. 1128–1129.
- [21] A. Jindal, J. Frielinghaus, M. Chadha, and M. Gerndt, "Courier: Delivering serverless functions within heterogeneous faas deployments," in 2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC'21), ser. UCC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3468737.3494097
- [22] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18), 2018, pp. 133–146.
- [23] A. Jindal, M. Chadha, S. Benedict, and M. Gerndt, "Estimating the capacities of function-as-a-service functions," in *Proceedings of* the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion, ser. UCC '21 Companion. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3492323.3495628
- [24] M. Chadha, A. Jindal, and M. Gerndt, "Architecture-specific performance optimization of compute-intensive faas functions," arXiv preprint arXiv:2107.10008, 2021.
- [25] M. Kiener, M. Chadha, and M. Gerndt, "Towards demystifying intra-function parallelism in serverless computing," arXiv preprint arXiv:2110.12090, 2021.
- [26] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19). Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotcloud19/presentation/mohan
- [27] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research," aug 2017. [Online]. Available: http://arxiv.org/abs/1708.08028
- [28] H. Wang, D. Niu, and B. Li, "Distributed Machine Learning with a Serverless Architecture," in *Proceedings - IEEE INFOCOM*, vol. 2019-April. Institute of Electrical and Electronics Engineers Inc., apr 2019, pp. 1288–1296. [Online]. Available: https://doi.org/10.1109/ INFOCOM.2019.8737391
- [29] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A Serverless Framework for End-To-end ML Workflows," in SoCC 2019 -Proceedings of the ACM Symposium on Cloud Computing. Association for Computing Machinery, nov 2019, pp. 13–24.
- [30] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards Demystifying Serverless Machine Learning Training," vol. 15, no. 21, may 2021. [Online]. Available: http://dx.doi.org/10.1145/3448016.3459240
- [31] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "Tensorflow: A system for largescale machine learning," in 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), 2016, pp. 265–283.
- [32] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015.
- [33] "OpenMined/PySyft: A library for answering questions using data you cannot see." [Online]. Available: https://github.com/OpenMined/PySyft
- [34] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. K. Konečny, S. Mazzocchi, H. B. Mcmahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander, "TensorFlow Federated," 2020. [Online]. Available: https://www.tensorflow.org/federated
- [35] C. He, S. Li, J. So, X. Zeng, M. Zhang, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, X. Zhu, J. Wang, L. Shen, P. Zhao, Y. Kang, Y. Liu, R. Raskar, Q. Yang, M. Annavaram, and S. Avestimehr, "FedML: A Research Library and Benchmark for Federated Machine Learning," 2020. [Online]. Available: https://fedml.ai.

- [36] "PaddlePaddle/PaddleFL: Federated Deep Learning in PaddlePaddle." [Online]. Available: https://github.com/PaddlePaddle/PaddleFL
- [37] "Fate." [Online]. Available: https://fate.fedai.org/
- [38] "faasd OpenFaaS." [Online]. Available: https://docs.openfaas.com/ deployment/faasd/
- [39] MongoDB Inc., "The most popular database for modern apps MongoDB," 2021. [Online]. Available: https://www.mongodb.com/
- [40] F. Chollet and Others, "Keras: the Python deep learning API," 2020. [Online]. Available: https://keras.io/
- [41] "pydantic." [Online]. Available: https://pydantic-docs.helpmanual.io/
- [42] Z. Xu, F. Yu, J. Xiong, and X. Chen, "Helios: Heterogeneity-Aware Federated Learning with Dynamically Balanced Collaboration," arXiv, dec 2019. [Online]. Available: http://arxiv.org/abs/1912.01684
- [43] D. Hardt, "The OAuth 2.0 Authorization Framework," Internet Requests for Comments, RFC Editor, RFC 6749, oct 2012. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6749.txt
- [44] S. Cantor, J. Kemp, R. Philpott, E. Maler, and E. Goodman, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML)," OASIS Open, Tech. Rep. November, 2005. [Online]. Available: http://www.oasis-open.org/committees/documents. php?wg\_abbrev=security
- [45] AWS, "Amazon Cognito Simple and Secure User Sign Up & Sign In — Amazon Web Services (AWS)," 2020. [Online]. Available: https://aws.amazon.com/cognito/
- [46] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," Internet Requests for Comments, RFC Editor, RFC 7519, may 2015. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7519.txt
- [47] "OWASP Top Ten Web Application Security Risks OWASP."
  [Online]. Available: https://owasp.org/www-project-top-ten/
- [48] J. Michener, "Security issues with functions as a service," IT Professional, vol. 22, no. 5, pp. 24–31, sep 2020.
- [49] L. Lyu, H. Yu, J. Zhao, and Q. Yang, "Threats to Federated Learning: A Survey," in *Federated Learning Privacy and Incentive*, ser. Lecture Notes in Computer Science, Q. Yang, L. Fan, and H. Yu, Eds. Springer, Cham, 2020, vol. 12500, pp. 3–16. [Online]. Available: https://doi.org/10.1007/978-3-030-63076-8\_1
- [50] D. Enthoven and Z. Al-Ars, An Overview of Federated Deep Learning Privacy Attacks and Defensive Strategies. Cham: Springer International Publishing, 2021, pp. 173–196. [Online]. Available: https://doi.org/10.1007/978-3-030-70604-3\_8
- [51] S. Truex, T. Steinke, N. Baracaldo, H. Ludwig, Y. Zhou, A. Anwar, and R. Zhang, "A hybrid approach to privacy-preserving federated learning," in *Proceedings of the ACM Conference on Computer and Communications Security*. Association for Computing Machinery, dec 2019, pp. 1–11. [Online]. Available: https://arxiv.org/abs/1812.03224v2
- [52] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai, "Privacy-Preserving Deep Learning via Additively Homomorphic Encryption," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1333–1345, may 2018.
- [53] C. Dwork, "Differential Privacy: A Survey of Results," in *Theory and Applications of Models of Computation*. Springer Berlin Heidelberg, apr 2008, pp. 1–19. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-79228-4\_1
- [54] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," Foundations and Trends in Theoretical Computer Science, vol. 9, no. 3-4, pp. 211–407, 2014.
- [55] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "BatchCrypt: Efficient homomorphic encryption for cross-silo federated learning," in *Proceedings of the 2020 USENIX Annual Technical Conference*, ATC 2020, 2020, pp. 493–506. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/zhang-chengliang
- [56] H. Zhu, R. Wang, Y. Jin, K. Liang, and J. Ning, "Distributed Additive Encryption and Quantization for Privacy Preserving Federated Deep Learning," nov 2020. [Online]. Available: http://arxiv.org/abs/2011. 12623
- [57] 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 *Proceedings of the ACM Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, oct 2017, pp. 1175–1191. [Online]. Available: https://dl.acm.org/doi/10.1145/3133956.3133982
- [58] V. Mothukuri, R. M. Parizi, S. Pouriyeh, Y. Huang, A. Dehghantanha, and G. Srivastava, "A survey on security and privacy of federated learning," *Future Generation Computer Systems*, vol. 115, pp. 619–640, feb 2021.

- [59] M. Kim, O. Günlü, and R. F. Schaefer, "Federated Learning with Local Differential Privacy: Trade-offs between Privacy, Utility, and Communication," feb 2021. [Online]. Available: http://arxiv.org/abs/ 2102.04737
- [60] K. Wei, J. Li, M. Ding, C. Ma, H. H. Yang, F. Farokhi, S. Jin, T. Q. Quek, and H. Vincent Poor, "Federated Learning with Differential Privacy: Algorithms and Performance Analysis," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3454–3469, 2020.
- [61] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov, "Exploiting unintended feature leakage in collaborative learning," in *Proceedings IEEE Symposium on Security and Privacy*, vol. 2019-May. Institute of Electrical and Electronics Engineers Inc., may 2019, pp. 691–706. [Online]. Available: http://arxiv.org/abs/1805.04049
- [62] H. B. McMahan, G. Andrew, U. Erlingsson, S. Chien, I. Mironov, N. Papernot, and P. Kairouz, "A General Approach to Adding Differential Privacy to Iterative Training Procedures," arXiv, dec 2018. [Online]. Available: http://arxiv.org/abs/1812.06210
- [63] T. community, "tensorflow/privacy: Library for training machine learning models with privacy for training data," 2020. [Online]. Available: https://github.com/tensorflow/privacy
- [64] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar, "LEAF: A Benchmark for Federated Settings," in Workshop on Federated Learning for Data Privacy and Confidentiality, NeurIPS, 2018, pp. 1–9. [Online]. Available: http://arxiv.org/abs/1812.01097
- [65] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/
- [66] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar, "LEAF: A Benchmark for Federated Settings," in Workshop on Federated Learning for Data Privacy and Confidentiality, NeurIPS, 2018, pp. 1–9. [Online]. Available: http://arxiv.org/abs/1812.01097
- [67] "The Complete Works of William Shakespeare by William Shakespeare -Free Ebook." [Online]. Available: http://www.gutenberg.org/ebooks/100
- [68] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "EMNIST: an extension of MNIST to handwritten letters," feb 2017. [Online]. Available: http://arxiv.org/abs/1702.05373
- [69] "NGINX High Performance Load Balancer, Web Server, & Reverse Proxy." [Online]. Available: https://www.nginx.com/
- [70] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [71] LRZ Compute Cloud, https://doku.lrz.de/display/PUBLIC/Compute+ Cloud, accessed on 09/24/2020.
- [72] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. v. Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tůma, and A. Iosup, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1528–1543, 2021.
- [73] Y. Chen, Y. Ning, M. Slawski, and H. Rangwala, "Asynchronous online federated learning for edge devices with non-iid data," in 2020 IEEE International Conference on Big Data (Big Data). IEEE, 2020, pp. 15–24.
- [74] "Amazon EC2 Amazon Web Services." [Online]. Available: http://aws.amazon.com/ec2/
- [75] Google, "Machine types Compute Engine Documentation Google Cloud," 2019. [Online]. Available: https://cloud.google.com/compute/ docs/machine-types
- [76] "Compute Cloud Leibniz-Rechenzentrum (LRZ) Dokumentation."
  [Online]. Available: https://doku.lrz.de/display/PUBLIC/Compute+Cloud
- [77] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807644
- [78] "VM instances pricing Compute Engine: Virtual Machines (VMs)." [Online]. Available: https://cloud.google.com/compute/ vm-instance-pricing
- [79] Google Cloud Functions Pricing, https://cloud.google.com/functions/ pricing, accessed 09/24/2020.