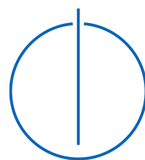
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Mitigation of Stragglers in Serverless Federated Learning

Mohamed Elzohairy



DEPARTMENT OF INFORMATICS

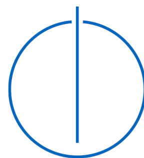
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Mitigation of Stragglers in Serverless Federated Learning

Eindämmung von Nachzüglern beim Serverless Federated Learning

Author:	Mohamed Elzohairy
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	M.Sc. Mohak Chadha
Submission Date:	15.05.2022



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.05.2022 Mohamed Elzohairy

Acknowledgments

First, I would like to thank my supervisor Prof. Dr. Michael Gerndt, for offering me an opportunity to work on such an exciting project and providing a suitable work environment.

It is my genuine pleasure to express my special appreciation and gratitude to my advisor, Mohak Chadha, for his guidance and patience throughout the thesis. Without his help and support, I could not have reached this stage. I also want to thank Andreas Grafberger for his previous work and help in the early phase of the project.

In memory of my dad, I will always be grateful for everything you did for me. I am eternally grateful to my beloved family, my mother, and sisters for their tremendous support throughout the project, especially in the final phase.

Abstract

In recent years, Machine Learning (ML) has made significant progress in being incorporated into many aspects of our life. While centralized ML paradigms provide good performance, they typically require access to the entire dataset, which may be impractical for specific tasks and may raise privacy concerns. Federated Learning (FL) was proposed to address this issue and migrate from using a centralized training approach to a distributed one. FL Clients preserve their data for local training and send the locally optimized model updates for the aggregation to the central server. FL, like any distributed system, has issues such as security, scalability, fault tolerance, and synchronization. Previous work proposed using Function-as-a-Service (FaaS) platforms to address some of these issues and facilitate an efficient training process among heterogeneous clients. FaaS is an efficient computation model to run stateless functions without worrying about infrastructure management. Moreover, FaaS environments only charge for consumed resources that dynamically scale according to demand. Although FaaS addresses problems such as scalability and infrastructure management, it still leaves some issues to be handled by the developers. Stragglers in an FL system are one of these issues. The effect of stragglers can severely hinder system performance, slow down convergence, or increase training time. Consequently, FL algorithms must be resilient against stragglers for the system to function efficiently. Our work focuses mainly on the problem of stragglers, particularly in FaaS-based federated learning. We propose a new clustering-based strategy, FedLesScan, designed to mitigate the effect of stragglers in serverless-based federated learning. We provide an extensive evaluation of our strategy and compare it to novel FL approaches on four different datasets with different ratios of stragglers. Compared to novel approaches, FedLesScan reduces training time by an average of 8% and retains a cost advantage of 20%, while improving test accuracy by 2%.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
1.1 Problem Definition 2
1.2 Research Objectives 3
1.3 Thesis Overview 4

2 Background **5**
2.1 Federated Learning 5
2.1.1 FederatedAveraging 6
2.1.2 Federated Learning Challenges 6
2.2 Clustering 7
2.2.1 Hierarchical Algorithms 7
2.2.2 Partition-based Clustering 8
2.2.3 Density-based Clustering 9
2.2.4 Choosing Clustering Algorithm 10

3 Related Work **11**
3.1 Serverless Machine Learning 11
3.2 Serverless Federated Learning 13
3.3 Stragglers in Federated Learning 16
3.4 Stragglers in Serverless Federated Learning 20

4 System and Strategy Design **22**
4.1 FedLess System Enhancements 22
4.1.1 System Architecture 22
4.1.2 System Debugging Capabilities 23
4.1.3 Training Workflow 24
4.2 Intelligent Client Selection 25
4.2.1 Gathering Behavioral Data 25
4.2.2 FedLesScan Selection Algorithm 28
4.2.3 Selecting Clustering Clients 30
4.3 Staleness-Aware Aggregation 31

5 Experiments and Evaluation **32**
5.1 Metrics and Experiments Configuration 32
5.1.1 Evaluation Metrics 32
5.1.2 Benchmarks and Datasets 33
5.1.3 Models Architecture and Parameters 34
5.1.4 Experiment Setup 35
5.2 Utilization and Strategy Performance 36
5.3 Accuracy and Model Performance 45
5.4 Time and Cost Analysis 52
5.5 IaaS vs FaaS Federated Learning 55
5.6 Discussion 56

6 Conclusion and Future Work **59**

List of Figures **60**

List of Tables **62**

Acronyms **63**

Bibliography **64**

1 Introduction

In recent days, Machine Learning (ML) has made a noticeable leap forward in solving problems and building complex systems that do not have an algorithmic solution. ML systems are now able to identify patterns and provide accurate predictions with a high certainty rivaling human predictions in some cases [1]. Centralized learning has been used for a long time. Nowadays, ML models require a huge amount of data to reach acceptable results [2]. Data processing outpaced the computation power of centralized systems. Another limitation is privacy constraints since the training data is collected and shared with the centralized model. This limitation is increasing as countries try to enforce data protection, such as General Data Protection Regulation (GDPR) [3] enforced by European Union. There emerges Federated Learning (FL) to tackle both issues. FL is a privacy-preserving alternative to the centralized approaches. The objective of FL is to address the problem of data governance and privacy by eliminating the need for data transfer to a centralized server. It distributes the training load over many mobile learners (clients). Unlike centralized learning, clients retain their data and only share their local model parameters. The shared parameters are combined to form a global model that can be used again to train the clients. There are no constraints on the type of devices participating in training; they can range from powerful machines to edge devices such as mobile devices. Since FL is a distributed machine learning approach, it carries some of the challenges that a distributed system might have, such as synchronization, scalability, and fault tolerance [4, 5]. FL has been slowly making its way into different industries such as the health industry, recommendation systems, autonomous vehicles, and mobile devices [6, 7, 8]. Each of these applications still encounters challenges that come with using a distributed learning process, such as statistical heterogeneity, security, data imbalance, resource allocation, and privacy [8, 9, 10].

Serverless Computing is a computing model that abstracts the process of server management such as provisioning, maintenance, and scaling away from the consumer [11]. Consequently, giving the developers more room to focus on the application business logic without spending time or resources on management and provisioning. In a serverless environment, functions represent the bundled applications requested or scaled according to the current demand. Serverless functions run on demand. They do not consume resources when not needed; as a result, serverless applications tend to have a lower cost [11]. The ease of use of serverless applications relies on their adoption in most of Function-as-a-Service (FaaS) platforms such as AWS Lambda [12], Google Cloud Functions [13], Microsoft Azure Cloud Functions [14], and IBM Cloud functions [15]. Although serverless has been adopted by multiple cloud providers and has many viable use cases, it suffers from a few limitations that affect the performance and type of application that can utilize this kind of infrastructure. One of the main challenges that serverless face is the limitation on function resources. Currently, the

duration of a function execution is relatively short(minutes) [16, 17, 11]. Furthermore, the latency of cold starts for functions might affect latency-critical applications [17]. Functions tend to have limited resources in terms of allocated CPU time and memory. Moreover, the lack of GPU support makes functions slower for some parallel execution tasks [17].

We argue that the idea of a serverless environment can benefit FL in terms of resource efficiency and cost in many scenarios. In a traditional FL, a subset of the clients is selected to participate in the training round. This means that the rest of the clients are waiting to be selected while wasting hardware resources. A FaaS environment can address this issue with clients utilizing hardware resources only if they participate in the current training round. As a result, serverless-based FL should have lower resource utilization and lower cost. Furthermore, the problem of rapidly scaling or managing infrastructure for the clients is one of the core benefits that FaaS provides. This means that setting up and updating a serverless-based FL platform is much easier.

As a first effort to use an FL system over a combination of connected FaaS platforms (FaaS fabric), Chadha, Jindal, and Gerndt [18] proposed *fedkeeper* as a tool to manage FL over FaaS fabric. The system models clients as separate functions each of which has its own independent training data. The client functions supported various FaaS providers such as Google Cloud Functions [13], OpenFaaS [19] and OpenWhisk [20].

Grafberger et. al [21] proposed FedLess as an evolution of *fedkeeper*. The platform supports more commercial and self-hosted providers such as Azure functions [14] and AWS Lambda [12]. Furthermore, many features and enhancements were added, such as client authentication and *Local Differential Privacy* [22]. Their evaluation showed the efficiency and cost advantage of the FaaS-based FL system compared to traditional IaaS-based FL systems.

1.1 Problem Definition

The problem of stragglers threatens the stability of FL systems. The presence of stragglers can affect training causing a reduction in accuracy and wasting resources. In a FaaS environment, stragglers might run, wasting money while their contributions are wasted due to failures, network issues, low computation capacity, or just slow starts. Furthermore, The scale-to-zero model used by FaaS platforms means there might be request latency for the idle functions to start.

To demonstrate the effect of stragglers, we performed an experiment using Google Speech Commands dataset [23], a real-world speech recognition dataset. Figure 1.1 depicts the effect of stragglers on FedLess, a FaaS-based FL system. Stragglers do not only affect the accuracy but can also hinder system performance by wasting resources as well as increasing both training time and cost. Notice that we show the total round duration in this experiment, including the aggregation of client updates. We show that despite the shorter aggregation time in the presence of stragglers, the wasted time due to waiting for clients is more significant.

Moreover, calling unavailable clients can cause under-utilization of system resources while other clients are waiting to participate in the training.

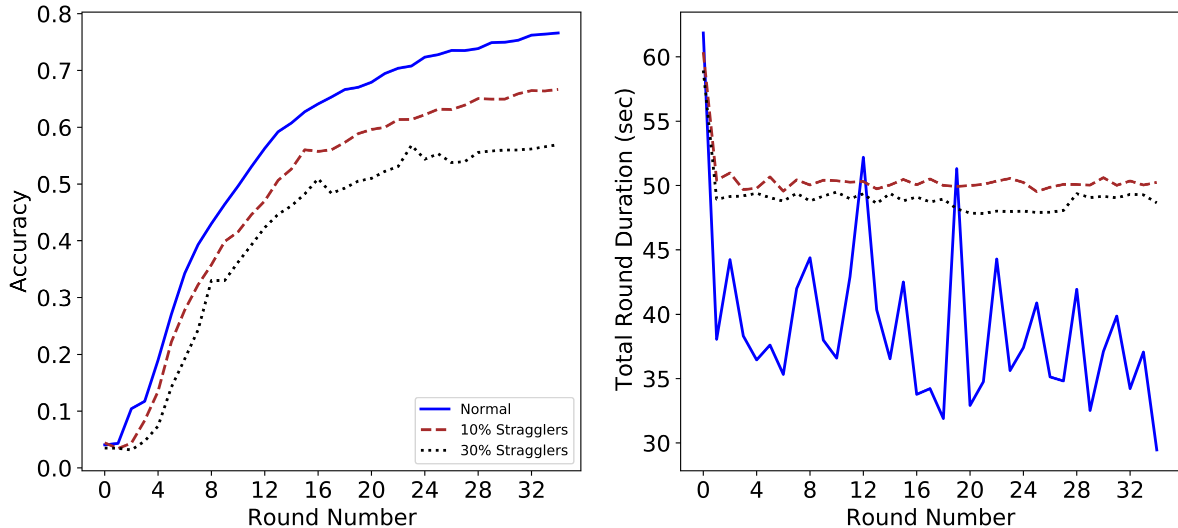


Figure 1.1: Model accuracy (Left) and average FL training round duration (Right) for different ratios of stragglers in the serverless FL system [21] for the Speech Commands dataset [23] using the `FedAvg` algorithm [24].

1.2 Research Objectives

Our objective is to explore methods that can mitigate the effect of stragglers in serverless federated learning. We rely on FedLess Platform [21] as a representative for FaaS-based FL systems. Our work focuses on two aspects of the platform. First, we propose a few enhancements to the FedLess platform implementation that incorporates the following:

- Implementation of Client mocks to make developing and using FedLess easier and more cost-efficient.
- Support for multiple FL training strategies.
- Prototype implementation for Speech Commands dataset [23] from FedScale benchmark [25].

Second, we propose *FedLesScan*, a clustering-based semi-asynchronous training strategy, specifically tailored for serverless FL. The strategy consists of two main components: The first is a clustering-based client selection algorithm that selects a subset of clients for training based on their previous behavior. The second is a staleness-aware aggregation scheme to mitigate slow updates and avoid wasted contribution.

We demonstrate the effectiveness of the FedLesScan by providing an extensive evaluation and comparison to two FL training strategies, FedAvg [24], and FedProx [26]. We evaluate the strategies on four different audio, image, and text datasets from two different benchmarks to ensure the results are conclusive. Our experiments show that FedLesScan achieves better accuracy in most scenarios in fewer rounds. Furthermore, FedLesScan can minimize the effect of stragglers, providing better resource utilization, lower cost, and faster convergence time.

1.3 Thesis Overview

In the second chapter, we discuss the core concepts of FL. We also provide an overview of various clustering mechanisms, their differences, and usecases. In the third chapter, we explore solutions for serverless ML. Then we discuss the origin of serverless FL and the core architecture and limitations of FedLess. We then examine work done to mitigate stragglers in FL and their limitations. In the fourth chapter, We demonstrate the enhancements made to the platform to facilitate the integration of our new strategy. Afterward, we discuss the operation and design of the FedLesScan algorithm. In the fifth chapter, we demonstrate metrics we use to evaluate FedLesScan. Furthermore, we provide detailed information about the experiment setup. We thoroughly evaluate our strategy on four different datasets in various synthesized and real-world scenarios. Moreover, we compare the performance of our strategy to two popular training schemes FedAvg and FedProx. We also perform experiments to demonstrate the resource efficiency of FaaS compared to IaaS platforms in FL. The last chapter includes a summary of our work and the limitations of our approach. We also include further recommendations for the future of the FedLess platform.

2 Background

2.1 Federated Learning

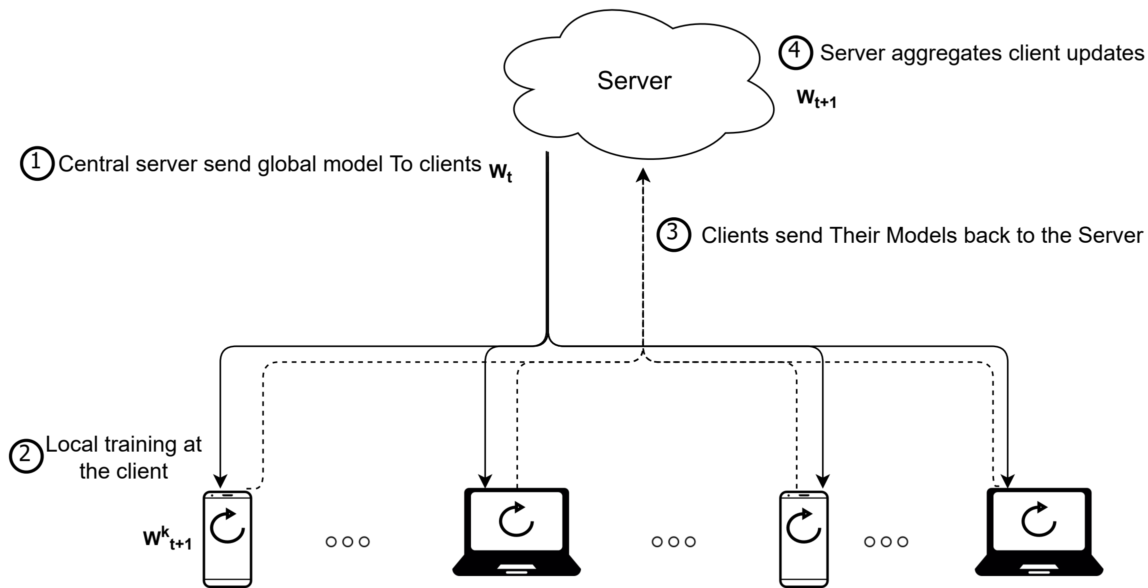


Figure 2.1: FL system architecture and training flow

Originally proposed by McMahan et al. [24], Federated Learning (FL) is a distributed ML scheme that allows collaborative training among multiple data holders. Unlike traditional ML, FL does not require data to leave its source. The data privacy guarantees provided by FL make it suitable for many privacy-sensitive domains such as healthcare systems [27, 28]. FL has gained a lot of traction in various fields such as autonomous vehicles [29, 30], text and emoji prediction for mobile keyboards [31, 32], and brain tumor diagnosis [33, 34].

FL is designed to solve federated optimization problems. Their paper highlighted characteristics distinguishing federated optimization problems from typical distributed optimization problems by the following assumptions:

- Data available on a client is *Non Independent Identically Distributed (IID)*: local clients' dataset is not representative of the population distribution [24].
- Client's local data in *unbalanced*: the amount of local training data is not the same for all clients.

- The client might become offline or slow; therefore, the system operates under the assumption of *limited communication*.
- Data is *massively distributed*.

2.1.1 FederatedAveraging

To solve federated optimization problems, FL relies on a synchronous update scheme called *FederateAveraging* or FedAvg for short. Figure 2.1 shows the training flow of a typical FL system. The algorithm starts by having a set of K clients participating in the training. A training session consists of multiple rounds. At the start of each round, C clients are randomly sampled from all the clients. The central server sends the current global model to the participating clients. Each client performs the training using its local dataset. After that, the clients send their updated models back to the server. The central server combines the models from all clients to form a new global model. The cycle repeats with the new model until we reach the desired number of rounds. For a system with K clients where P_k is the data items assigned to client k and n_k is the number of data items in P_k. Equation 2.1 shows the loss computation at the client where $f_i(w)$ is the loss for sample i. Equation 2.2 indicates the aggregation formula used back at the central server where w_t is the model weights at round t.

$$F_k(w) = \frac{1}{n_k} \sum_{\forall i \in P_k} f_i(w) \quad \text{and} \quad w \longleftarrow w - \mu \nabla F_k(w) \tag{2.1}$$

$$w_{t+1} \longleftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^{k} \tag{2.2}$$

Their evaluation demonstrated that this method is robust to unbalanced and non-IID data distribution results with experiments considering four different datasets with different models. Although the original work was based on Stochastic Gradient Descend (SGD) [35], this method can be used with other optimizers [24, 36].

2.1.2 Federated Learning Challenges

Since FL is a distributed system, it faces various challenges [9, 10]. For example, sharing models between clients and the central server can cause communication issues, especially with large models and clients with limited network bandwidth [9]. Some work tries to improve communication efficiency by using model compression [37], while others opted for limiting communication between the client and the central server for only useful or relevant updates [38].

Another challenge facing FL is the system and data heterogeneity [10]. Clients' hardware and network capabilities are not necessarily the same. There might be drastic variations in clients' computational power or communication capabilities [9, 10]. A robust FL system should be able to mitigate slow or offline clients and tolerate the differences in clients' capabilities.

One of the main objectives of FL is preserving privacy. Although FL does not require the training data to leave the client, there are several ways to deduce information about the client's data from the model updates [9]. One of the mechanisms that can address this issue is Differential Privacy (DP) [39, 22]. DP prioritizes privacy at the expense of accuracy. In this technique, clients add random noise to their updates before sending them back to the server. The noise distorts the information about the dataset that the client's local updates might carry.

Being a system with multiple parties raises some security concerns. In FL, Malicious clients can perform attacks such as data and model poisoning attacks [9, 40]. The target of these attacks is to influence the global model prediction by either injecting poisonous data or changing local model weights on the client-side. These attacks can be partially mitigated by filtering client updates based on the model's loss before aggregation [9].

2.2 Clustering

Clustering is a type of unsupervised learning. The objective is to divide and partition unlabeled data into groups or categories [41]. Although there are many definitions of clustering[42], most of them identify clusters by the intra-cluster connectivity and inter-cluster separation [42]. This section provides an overview on the most popular types of clustering algorithms and the rationale behind choosing DBSCAN [43] in our strategy. The discussion in this section is based on the survey done by Halkidi, Batistakis, and Vazirgiannis [44] unless otherwise stated.

2.2.1 Hierarchical Algorithms

Hierarchical clustering (HC) is a type of clustering where the data is categorized in an iterative approach [42]. The outcome is typically a tree-like structure (binary tree or a dendrogram). The root represents a single cluster with all the data points, while each leaf represents a single data element. Intermediate nodes represent clusters where the descendent leaves are the members. Cutting the structure at any level represents a way of clustering the dataset.

There are two types of HC. The difference between both is the method of building the tree itself. Divisive clustering is When using a top-down approach [42]. Starting from a single cluster that contains all data, we keep splitting the data until we reach all clusters are singletons. Because the number of ways to split the data is exponential, this strategy is computationally inefficient.

The second method is Agglomerative clustering. It is based on a bottom-up approach [42]. The algorithm starts with N clusters, where N is the number of data elements. Successive merge operations are made until we reach a single cluster containing all data elements. The merge operation depends on the proximity among clusters using a specific similarity measure. Before each merge operation, the algorithm computes a proximity matrix P. The entry $P(i,j)$ is the distance between cluster i and cluster j. At each merge step, the algorithm computes P and merges the clusters with the least $P(i,j)$. The process is repeated until all the clusters are merged. The entries in the proximity matrix are computed based on different definitions. The

most popular methods are:

- Single Linkage [45]: determine distance using the closest two objects from each cluster
- Complete Linkage [46]: determine distance using the furthest two objects from each cluster

There are other definitions for computing distances between clusters: average linkage, median linkage, and centroid linkage [42].

Basic HC approaches suffer from a few limitations. One of the limitations of HC is the sensitivity to noise and outliers [42, 44]. Furthermore, data elements assigned to clusters can not be adjusted or reassigned to other clusters. Moreover, computing the proximity matrix P is computationally intensive. Consequently, the complexity of the algorithm is $O(N^2)$ which is not practical for large datasets.

2.2.2 Partition-based Clustering

Partitional clustering uses a heuristic-based algorithm to partition the data into a set of clusters. Unlike HC, the set of clusters is not based on any hierarchical structure. The heuristic algorithm uses a criterion function, typically sum squared error, to organize the data points into a set of clusters.

One of the most popular partition-based algorithms is the *K-Means* algorithm [47]. The idea is to move clusters around iteratively until reaching a reasonable partitioning. The algorithm can be summarized in the following steps.

1. Select K random points as cluster centroids.
2. Assign all the points to clusters based on their distance to the cluster centroid.
3. Recalculate the clusters' centroids based on the new assignment.
4. Go back to step 2 until clusters do not change or a maximum number of iterations is reached.

The time complexity of the K-means algorithm is $O(NKd)$ for N(d-dimensional) data points and K Clusters.

There are a few drawbacks to K-means. For instance, there is no well-defined initialization method for the initial selection of the clusters' centroids. As a result, the end cluster centroids can vary depending on the initialization. One way to mitigate this effect is to run the algorithm multiple times with random initialization [42]. Furthermore, K-means is not robust against outliers and noise. Since every data point must be assigned to a cluster, outliers can affect the cluster centroids and disrupt the cluster shape. There are methods to reduce the effect of noise by discarding clusters with few data points [48]. K-means variations, such as PAM(Partitioning Around Medoids) [49] address the algorithm robustness by using medoids instead of mean. However, PAM still has a higher time complexity ($O(K(N-K)^2)$) than normal K-means.

2.2.3 Density-based Clustering

In density-based algorithms, clusters are represented as regions of high density separated by low-density regions. One of the most popular density-based clustering algorithms is *Density Based Spatial Clustering of Applications with Noise (DBSCAN)*. The algorithm relies on point density in the space to construct the clusters. DBSCAN defines *core points* to be the points in high-density regions.

The approach relies on one main parameter ϵ, representing the maximum distance between two samples to be considered in the neighborhood of each other [43]. In other words, it dictates the density for a neighborhood to be considered a cluster. There is another noise control parameter, *minPts* which represents the minimum number of points in the neighborhood of a certain point to be considered a core point.

With both parameters, we define two more types of data points. *Border points* are defined as points that are in the neighborhood of core points. *Outliers* are the points that are not reachable by a distance ϵ from any core points.

the algorithm does the following steps:

- For each data point p, define the ϵ-neighborhood as all points within a distance ϵ from p.
- For each point p, if the size of its ϵ-neighborhood is greater than *minPts*, define p as a core point.
- Define a cluster as a group of core points in the same ϵ-neighborhood
- Attach each border point (points reachable from core points) to a nearby cluster
- Points not reachable from any core are defined as outliers

In DBSCAN, both ϵ and *minPts* can affect the outcome of the clustering [50]. A large ϵ will increase the cluster size. If ϵ is too large, the whole data will be included in a single cluster. A small ϵ can result in a high number of clusters or no clustering at all [43]. On the other hand, *minPts* is a noise parameter. It can be tuned to reduce the effect of outliers by managing the number of core points in the algorithm. Small *minPts* will result in more core points. A large *minPts* can deal with noise better because it will only consider points with a higher density as core points.

One of the main advantages of DBSCAN is that it does not require the number of clusters to be specified. Furthermore, DBSCAN is robust against outliers [50, 42]. Moreover, the algorithm can find clusters of arbitrary shapes. DBSCAN is computationally efficient. For N data points the algorithm has a time complexity of $O(Nlog(N))$ [42], making it suitable for large-scale datasets.

Similar to other clustering algorithms, DBSCAN has limitations. One of the limitations of DBSCAN is dealing with high dimensional data [42]. When the data dimensionality increases beyond a certain limit, distance measures become less effective [51] in representing the differences among data points. Consequently, clustering based on neighborhood distance becomes less effective [42].

2.2.4 Choosing Clustering Algorithm

The increase in the amount of raw data is one of the challenges that face clustering algorithms. Some algorithms are not well suited to deal with large-scale data, such as HC algorithms, due to their high complexity $O(N^2)$. On the other hand, K-means has a near-linear time and space complexity, making it more suited to large-scale datasets. However, K-means still requires the number of clusters to be stated beforehand. There are numerous algorithms developed to target the problem of clustering large datasets. The rationale behind choosing DBSCAN in our strategy is its robustness against outliers. Furthermore, explicitly setting the number of clusters means that we have pre-acquired information about the underlying system behavior, which is not the case in a large-scale FL. DBSCAN does not require the number of clusters to be explicitly set. As a result, the system can adjust to changing behavior of the clients. Moreover, the algorithm's computation efficiency makes it suitable for large-scale FL systems with large number of clients. Although DBSCAN suffers in the case of high-dimensional data, we disregard this limitation because we only rely on a few features to distinguish clients' behavior; therefore, we argue that this limitation will not exist in our system.

3 Related Work

Although using serverless architecture for distributed ML problems is a rising area of research, numerous platforms emerged trying to leverage the simplicity and cost advantage of the serverless execution model. This chapter first explores previous work integrating serverless architecture in distributed ML systems. Then we provide an overview of previous efforts in using FaaS in FL, particularly the architecture and training workflow of the FedLess platform. We then move to the problem of stragglers in FL and explore previous approaches to tackle the issue. In the end, we explore the differences between using Infrastructure-as-a-Service (IaaS) and FaaS in FL in terms of reliability and performance to help us well understand how stragglers behave in a serverless environment.

3.1 Serverless Machine Learning

The typical workflow for a distributed ML system includes, first, data preprocessing, where each worker fetches the assigned dataset shard and performs its preprocessing script. This process is followed by training the model where workers perform the training and synchronize with the central parameter server. The last step involves running multiple instances with different setups to search for the best-performing hyperparameters.

One of the main Problems of ML is resource management [52]. Not only ML developers have to develop a working model, but they also perform hyperparameter tuning for variables such as learning rate and the number of epochs. In a distributed ML environment, the workload increases by the additional management of many system-level parameters such as the number of workers (virtual machines), their memory, and the number of CPUs. In a study made by Carreira et al. [52], they discussed the benefits of using serverless architecture for ML distributed workloads. Serverless Computing addresses two of the main problems in traditional distributed ML [52]. First, low-level resource management puts more work on the shoulder of ML developers. The second problem is that the heterogeneity of ML tasks can lead to *over-provisioning* due to an imbalance of resources. Although serverless computing represents a suitable solution to those problems, it still suffers from a the following limitations [52].

- Short running time and low memory
- Low network bandwidth compared to virtual machines
- No communication among functions
- Limited support for GPU workloads

With the developments in serverless computing, these issues will gradually fade away [53]. In that regard, many serverless ML platforms were developed. For instance, *PyWren* developed by Jonas et al. to run on AWS by utilizing S3 [54] and Lambda services [12]. The system contains a central server that invokes a specific function on a specific dataset shard. The function code and the sharded data are serialized and put in an S3 bucket. A single lambda function fetches the serialized function code on the specific data. The serialized function code is dynamically injected into the common function and executed on the obtained data. After the end of the execution, the result is uploaded back to the S3 bucket. Their evaluation included testing multiple distributed computing models such as the bulk-synchronous processing (BSP) scheme. They also suggested that PyWren can be used to implement distributed ML applications that utilize the parameter server style if a fast storage service with high throughput and low latency is used, such as Redis [55]. They discussed some limitations of their prototype, such as Debugging, which poses a significant challenge due to many system components. Furthermore, the existing limits on functions' bandwidth and storage might hurt performance in high-performance data processing workloads.

SIREN [56] is a distributed ML frameworks that is based on a serverless architecture. Their work promotes one of the main features of serverless architecture: ease of use and eliminating complex infrastructure management. The platform contains a server that manages parallel workers running as stateless functions. The main server has full control and assigns a batch of data to each function. Furthermore, the paper proposed a scheduler based on *reinforcement learning* [57] to adjust the number of functions and the allocated memory during training. The scheduler learns the best way to provide the resources throughout the training to yield the minimum training time given a specific cost cap. The proposed prototype was based on AWS Lambda. Their evaluation provided experiments for mainly logistic regression models. Their results demonstrate that using Lambda functions with the proposed scheduler yields a 44% decrease in training time in comparison to traditional distributed ML benchmarks.

MLLESS is another distributed ML framework that utilizes FaaS. The platform is built based on IBM Cloud Functions [15]. Their work shows the advantages of using FaaS-based solutions to the traditional IaaS reservation model. The key difference in their approach is the implementation of two optimizations: a significance filter and a scale-in auto-tuner. The significance filter restricts workers from sending insignificant updates, reducing the communication bandwidth required to share the model updates. The scale-in auto-tuner reduces the number of workers participating in the training as the training progresses. Their idea is to drop the workers with low local progress towards the end of the training, resulting in lower training costs. The platform contains two main components. First, the *driver* which is the main script that runs on the owner's personal computer. The driver manages and calls participating serverless workers. The second component is the *supervisor*, which is a function that runs during training to monitor and synchronize workers. Furthermore, the supervisor can communicate with worker functions using *RabbitMQ* [58] to end training or limit model divergence between functions. The workers use Redis for sharing local gradients. Each worker performs the aggregation locally to update their local model by pulling the intermediate results from the external storage (Redis). An object store hosted on *IBM COS*

stores the dataset used by the workers in training. Their work examined basic ML tasks such as logistic regression and matrix factorization. Their approach was 15X times faster than traditional IaaS ML systems.

3.2 Serverless Federated Learning

Although FL is a distributed machine learning approach. The main difference between FL and traditional distributed ML is the additional privacy and security guarantees. Exploring serverless capabilities in FL is a new area of research. This section provides an overview of previous efforts to integrate serverless architectures in FL. We also discuss the current implementation of the FedLess platform, including its architecture and limitations.

Jayaram et al. [59] proposed *λ-FL*, a new aggregation architecture for large-scale FL systems based on serverless functions over multiple steps. The aggregation process must be associative to carry the aggregation on multiple functions [59]. They discussed horizontally scaling functions based on aggregation demand, where functions are only active when clients push their model updates. The system contains two types of functions: First, leaf aggregation functions which generate intermediate models by aggregating parameters from a group of clients. Second, intermediate aggregation functions which produce the final global model by combining intermediate models. The system contains a messaging queue based on Kafka to hold the client updates and store intermediate models. Furthermore, the messaging queue triggers aggregation functions based on the number of received updates. Each FL-training session is assigned a unique *SessionID*). Before the start of the session, the system creates two queues, one for aggregation updates *(SessionID-Agg)* and the other for client updates *(SessionID-Parties)*. All clients subscribe to the SessionID-Agg queue to get the latest model updates. At the start of the training, the initial model is published by the aggregator to the SessionID-Agg queue. Next, clients fetch the initial model and start training. Afterward, clients publish their model updates to SessionID-Parties. For every K updates in SessionID-Parties, Kafka marks the updates and triggers a leaf aggregation function. These functions perform the aggregation and push their results back to Kafka with a cascading effect, triggering other intermediate aggregation functions. Although the system is not fully serverless, they demonstrate the advantages of using serverless architecture in the aggregation by achieving a $> 90\%$ reduction in resource utilization compared to traditional tree-based parallelization schemes at a small aggregation latency.

Chadha, Jindal, and Gerndt were first to propose *FedKeeper* as a client-based python tool for orchestrating FL clients distributed over different FaaS platforms. Their objective was to build a system that can efficiently train a shared model for heterogeneous devices over FaaS fabric. The platform should also facilitate the scaling of FL process over different FaaS platforms. Furthermore, it leverages one of the key advantages of FaaS by abstracting the clients' infrastructure management away from the developer.

FedKeeper relies on Openwhisk for running the main functions of the Central server, while the participants can be deployed on any FaaS platform. The Openwhisk cluster contains several components. The central server manages the training round and stores the initial

model in the local object store. *Client Register* which manages all information about the participating clients in a local MongoDB instance. *Client-Invoker* manages clients' invocation by creating invoker functions that are one-to-one mapped to clients participating in this round. An invoker function reads the current global model from a global object store and forwards it to the client. After the client finishes training, it sends the updated model back to the invoker function, which pushes the update back to the global object-store and notifies the *weight updater* function to start the aggregation. Upon successful aggregation, the *weight updater* stores the new global model in the object store and notifies the central server to start the next round. Their work demonstrated the ability to perform FL tasks on multi-platform FaaS-based systems.

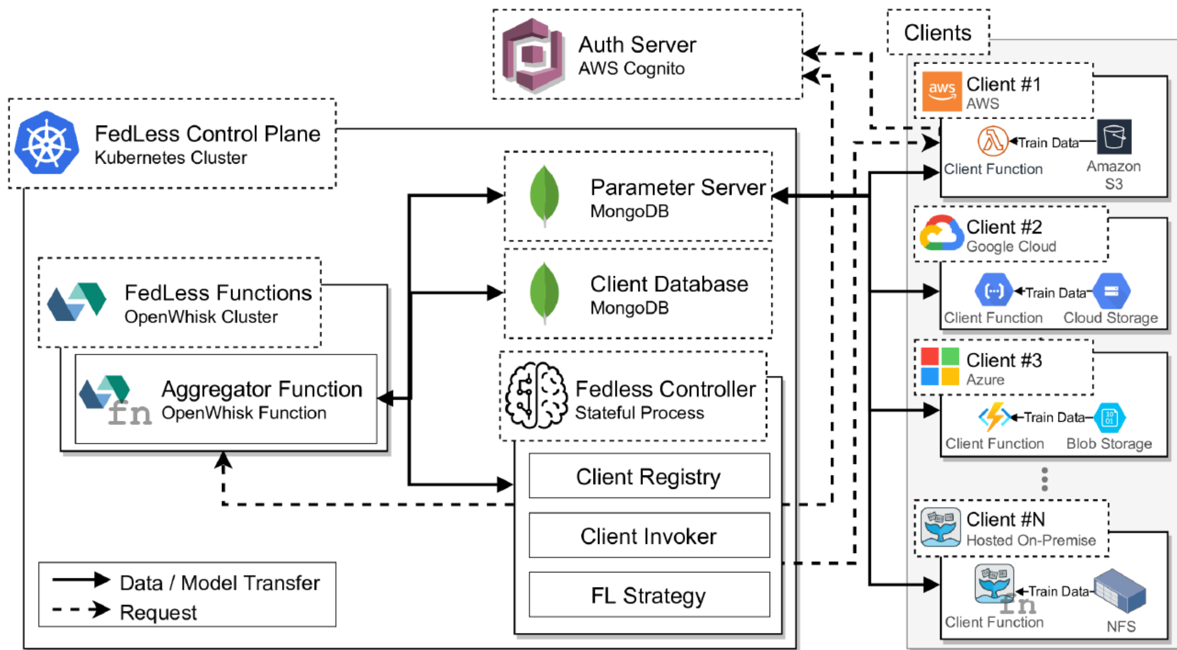


Figure 3.1: FedLess System Architecture [21].

FedLess [21] was designed as the evolution of FedKeeper [18]. It provided multiple enhancements in terms of security and performance over FedKeeper. Figure 3.1 show the platform architecture and interaction among different platform components. Unlike FedKeeper, FedLess does not rely on invoker functions; therefore, it has less overhead because fewer functions are running during training. The Fedless controller contains a few sub-components. Client registry stores information about the clients in the client database. Client invoker manages function invocation for clients and aggregation functions. The parameter server is a MongoDB [60] instance that stores the global model and client updates. The aggregator is a local Openwhisk function that manages the aggregation of clients' results at the end of the round. The key addition is the integration of a separate authentication entity running in *AWS Cognito* [61]. This entity ensures that only the FedLess Controller is allowed to call the client functions and that functions are verified before participating in the training.

Figure 3.2 shows the interactions among FedLess components during a training round. Currently, FedLess uses FedAvg as the only strategy available for training the client functions. Clients are required to submit a registration request to the authentication server to be able to participate in the training. The request is then approved by the FL admin who runs the FedLess Controller. At the start of the training, the FL admin configures the model, dataset, and hyperparameters. Afterward, the controller fetches invocation tokens from the authentication server. At the start of the round, The controller invokes the client function using the invocation tokens. The client function contacts the authentication server to validate the invocation token. Upon successful validation, the clients fetch the latest global model from the parameter server and perform the training locally. When the training finishes, the clients upload the new local model to the parameter server and notify the controller. Afterward, the controller invokes the local aggregation function, which combines the clients' results into a new global model. At the end of the round, the controller invokes a subset of the clients for evaluation. The process then repeats for subsequent rounds.

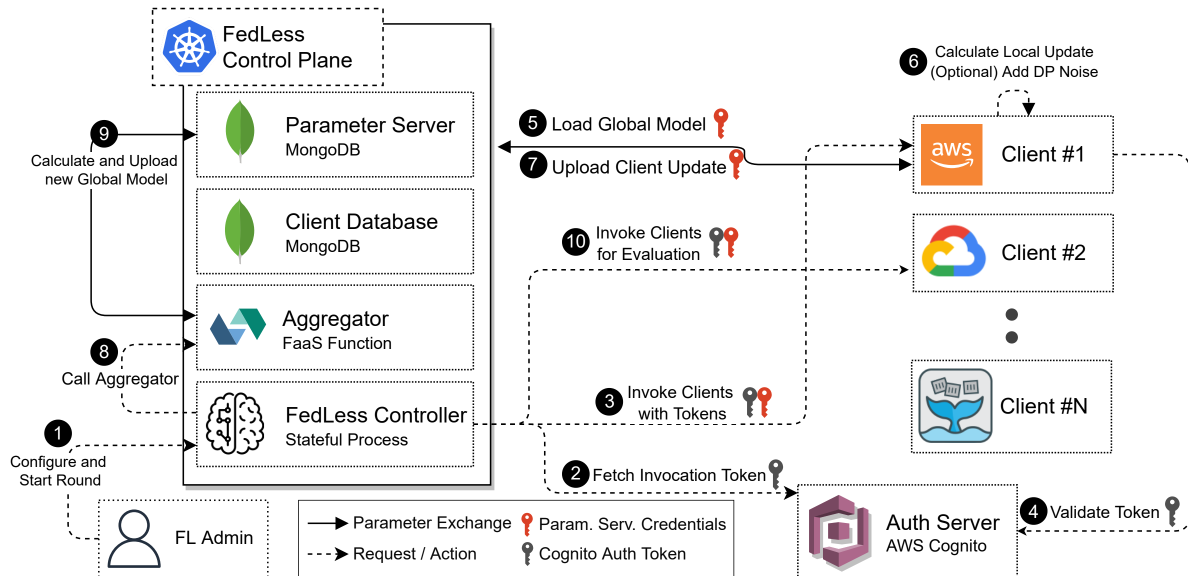


Figure 3.2: FedLesstraining workflow [21].

As we mentioned earlier, we use FedLess to develop and evaluate our strategy. Although the end goal is to mitigate the effect of stragglers in serverless FL, we want to overcome limitations that affect the platform usage or the experience of using FedLess itself to demonstrate the capabilities of FaaS-based FL. We notice the following architectural limitations in FedLess:

- FedLess Control Plane is only beneficial for the aggregation function to run inside the Openwhisk Cluster. Therefore, the current implementation requires more effort to run the platform by deploying a Kubernetes Cluster and then deploying Openwhisk inside.
- The aggregator function is only available in Openwhisk: this causes an issue of vendor lock-in while we try to make the platform as versatile as possible.

- Client Functions must be deployed even for development or debugging purposes, increasing development costs.
- Support for only a single strategy: Stragglers are a common issue in FL. Although FedAvg is relatively robust against stragglers to some extent [36], stragglers can significantly impact training time, which causes underutilization of the system's resources. Additionally, random selection ignores the client's performance in the selection criteria, which means that a single straggler can be triggered repeatedly, decreasing round efficiency and potentially increasing cost.

In addition to our novel training strategy, this work provides architectural changes to the core platform design to address the limitations mentioned above.

3.3 Stragglers in Federated Learning

Resource and data heterogeneity restricts the collective learning process in large-scale FL systems. It is not practical to expect consistent performance and reliable communication throughout the training. In real-world scenarios, clients can be offline due to network or resource constraints. Furthermore, clients' training speed depends on the amount of data and computation power. *Synchronous training mechanisms* such as FedAvg can be significantly affected by stragglers where the slowest client dictates training pace. In addition, offline clients which do not respond can cause significant training delays. Previous efforts proposed *asynchronous FL schemes* [62, 63] to mitigate the effect of stragglers and avoid wasted contributions. The system allows all clients to communicate with the central server asynchronously. As a result, these systems suffer from high communication costs. Furthermore, stall gradients might affect the system's performance [63, 64]. Other approaches adapt semi-asynchronous strategies as a tradeoff between the latency of synchronous approaches and communication and staleness issues of asynchronous approaches. In this section, we explore various synchronous, asynchronous, and semi-asynchronous techniques used to mitigate the effect of stragglers in FL and discuss their limitations within the context of serverless FL.

Li et al. [26] proposed a protocol called *FedProx* to tackle heterogeneity in federated learning. The algorithm is based on FedAvg with two minor differences. The first is a custom loss function at the client, which contains a proximal term to limit the fluctuating effect of local updates. This proximal term helps control the local model deviation from the global model by factoring in the square difference in model weights in the loss computation. Equation 3.1 shows the formula used for the client loss function. Notice that w^t represents the weights of the global model, and w represents the local model during training. Setting μ to zero defaults the algorithm back to FedAvg. The second difference is the idea of tolerating partial work. The clients can perform a variable amount of work to accommodate constraints in terms of hardware, network, and battery levels. To achieve this, clients can run a different number of local epochs. Their experiments were done in an environment to simulate clients with different local computations. The results have shown about 22% improvement of test accuracy

over FedAvg in high heterogeneous settings (90% stragglers in the system) while performing similarly in scenarios with less amount of stragglers.

$$h_k(w; wt) = F_k(w) + \frac{\mu}{2}\|w - w^t\| \tag{3.1}$$

From the perspective of a FaaS FL, FedProx has a few downsides. First, the proximity term depends on the difference in weights between the global model and the current local model. Consequently, training large models may suffer performance degradation, as clients might take longer to train compared to strategies like FedAvg. This issue is particularly critical in a serverless environment because current implementations of functions allow for a limited duration. We also confirmed this behavior during our experiments where FedProx clients take slightly longer to train compared to FedAvg. Second, incorporating partial work requires tailoring the number of local epochs for each client individually, which might be infeasible for a significantly large number of clients. The work did not specify an automatic way to incorporate partial work from clients at a reasonable communication cost. Finally, using random client selection means that the algorithm is exposed to stragglers similar to FedAvg.

Xie, Koyejo, and Gupta proposed an asynchronous federated optimization algorithm *FedAsync* [63]. They rely on the parameter server architecture to invoke and synchronize clients. The server contains two parallel working threads. The first is a scheduler thread that periodically triggers clients to perform the training using the latest global model. The second thread (updater) receives client updates and directly aggregates them to the global model. To mitigate staleness in client results, they use an adaptive weighted average at the updater, which dynamically changes the weights based on the staleness of the updates. Their experiments show that FedAsync can provide similar performance to FedAvg in scenarios with a low number of stragglers in the system while outperforming it in straggler-heavy scenarios. The robustness of this system against slow clients and its adaptability to stall updates are two of its main advantages [63]. Nevertheless, the system suffers from a few limitations. First, the aggregation computations for a large-scale system can overwhelm the central server. They discussed implementing multiple updater threads to mitigate this issue. Second, in the context of a serverless FL, this system has a high communication cost. It is not efficient in terms of cost or resources to have a function that does the aggregation after receiving each client update.

Chai et al. proposed *FedAT* as a semi-synchronous tier-based FL system. Their idea is to partition the clients into tiers based on their performance. The system consists of two components. Firstly, a tiering module that partitions clients into M tiers where $tier1$ contains the fastest clients and $tierM$ contains the slowest clients. Secondly, a central server maintains a local model for each tier, $\{w^t_{tier1}, w^t_{tier2}, ..., w^t_{tierM}\}$, where w^t_{tierX} is the most updated model for clients in tier X at round t. Furthermore, the central server maintains the global model updated asynchronously from the M tiers.

Each tier performs its updates synchronously, where S clients are randomly selected to participate in the training. Once a tier finishes training, its clients send their updates to the server. Next, the server aggregates the updates to compute the tiers' new model w^t_{tierX}. Then, the server aggregates the updates from all tiers to form a new global model w^{t+1}. The server

uses a weighted aggregation strategy where slow tiers have higher weights to balance the difference in update frequency between tiers and counter the bias towards faster clients. They also included miscellaneous improvements to enhance the quality of their training strategy. Firstly, to keep the client's local model from deviating from the global model, they utilized FedProx's custom loss function 3.1. Secondly, they implemented a weights compression scheme based on polyline encoding to improve communication efficiency. Clients flatten the weights of each layer and compress the output weights before pushing them to the server. On the other side, the server reconstructs the weights after decompression to start the aggregation. Their experiments showed improvements by up to 21.09% accuracy on five selected datasets from the LEAF benchmark [65]. Furthermore, they showed an 8.5X reduction in communication cost compared to FedAsync. Although the system was effective, it suffered from a few limitations. Firstly, a constant number of tiers prevent the strategy from dynamically adjusting to system changes. Furthermore, the system does not specify how to mitigate client failures. For instance, clients which do not respond to the central server can cause tiers to take significantly longer time and affect the efficiency of training.

Zang et al. proposed CSAFL, a clustered semi-asynchronous FL scheme [66]. Their approach was to divide clients into groups based on similarities in computation and communication latencies. At first, the central server computes an affinity matrix with mutual similarity values between each pair of clients. Then the server performs a spectral clustering algorithm to divide the clients into M groups. The central server maintains M global models $\{W^{g^0}, W^{g^1}, ..., W^{g^m}\}$, each representing a specific group. Furthermore, each group is trained independently of the other groups, so there is no global model that combines the M models. At the start of the round, K candidates are randomly selected from each group. Next, the server starts training each subset in parallel. The initial model is broadcast to the participating clients within a group with a version number $(W^{g^x}, 0)$. Each participant updates their local model's version number to maintain the difference between the old and the new model (V_{pre}, V_{new}). If the difference between V_{new} and V_{pre} is larger than a certain tolerance H, the client is forced to update the model synchronously. If the difference is less than H, the central server invokes the client and updates the group model once the client replies with the update. After each synchronization, the new group model is broadcasted to all group clients. The same process repeats until the end of the round.

They experimented with over four datasets from the LEAF benchmark and showed a 5% accuracy gain over FedAvg. Despite the accuracy gain, their approach suffers from a few limitations. Similar to FedAT, the number of groups can not be changed during training. As a result, the system can not always produce the best partitions for grouping the clients. Moreover, calculating and storing the affinity matrix is computationally intensive, which might cause performance issues in large-scale systems. Furthermore, The system does not use the notion of a global model that can perform predictions on different data. In their evaluation, they used a weighted test accuracy to evaluate each group model on the test data and combine the results.

Wu et al. [67] proposed a semi-asynchronous FL protocol (*SAFA*). The algorithm focused on low round efficiency and slow convergence in scenarios where clients drop frequently. To

mitigate the impact of stragglers, they suggested new designs in client selection and global aggregation. They rely on caching client updates to avoid wasted contributions. Therefore, a mapping between clients and their updates is saved in a cache maintained by the central server.

In the client selection, a client can have one of the following states:

- *Up-to-date*: Clients who completed the previous round and have the latest model at the start of the current round
- *Tolerable clients*: Clients that do not use the latest global model as a basis for training but whose model is not too old.
- *Deprecated clients*: Clients that still use a stale global model as the basis for their local training.

The semi-asynchronous nature of the algorithm allows tolerable clients to stay asynchronous with the server, while the other two types are required to synchronize with the server. The algorithm allows all clients to participate in the current round if they are willing. The central server signals the start of the round to all clients. Afterward, it waits until C updates are received, where C is the minimum number of updates required to start the aggregation phase.

After C clients are finished, the server tags clients with one of the following labels.

- *Crashed*: clients who did not complete the training or decided not to complete it.
- *Undrafted*: clients whose results are not selected but cached by the server for future use.
- *Picked*: clients picked to be used in the aggregation.

The cache is then updated once before aggregation and once after aggregation.

- Pre-aggregation update:

$$w_k^*(t) = \begin{cases} w_k^{'}(t) & : \text{if k} \in \text{Picked Clients} \\ w(t-1) & : \text{if client k has model older than t-}\tau \\ w_k^*(t-1) & : \mathit{otherwise} \end{cases} \tag{3.2}$$

  In the pre-aggregation phase, picked clients save their latest local model while tolerable clients maintain their cache entry. On the other hand, deprecated clients are forced to synchronize their model with the latest global model and abandon their stall progress. Equation 3.2 shows the cache update conditions before aggregation where $w_k^{'}(t)$ is the trained local model at client k for round t and $w_k^*(t)$ is the weights saved in cache for client k. τ is a tunable parameter that represents the algorithm tolerance to stragglers.

- Post-aggregation update:

$$w_k^*(t+1) = \begin{cases} w_k^{'}(t) : \text{if k} \in \text{Undrafted Clients} \\ w_k^*(t) : \mathit{otherwise} \end{cases} \tag{3.3}$$

In the post-aggregation step, *undrafted* clients keep their updates in the cache to be used in the subsequent round, while the rest of the clients use the most recent global model after the aggregation, as shown in Equation 3.3.

On the client-side, the algorithm follows the same procedure as FedAvg. They performed experiments with multiple classifications and regression tasks. Their experiments demonstrated accuracy and efficiency improvements at a slightly higher communication cost. Although we believe a custom client selection algorithm is beneficial in overcoming the stragglers' problem, this approach still has some issues if used in a FaaS environment. One of the downsides is overutilizing the clients, which increases the experiment's cost. The deprecated clients might stay behind for the whole training session, wasting their contributions and consuming resources. Furthermore, their selection strategy does not benefit from the scale-to-zero capabilities of a serverless infrastructure by involving all clients each round. Clients will always be running in some scenarios since they are called every round, increasing cost and resource utilization.

Other approaches proposed ideas to benefit from the contributions of slow clients, which are normally wasted. Damaskinos et al. [64] proposed *FLEET*, an online FL framework for Android devices. The framework focuses on integrating stale updates from clients. Although delayed updates can result in noise that might slow down or even prevent convergence [68, 69, 70], they might contain important information about unseen data. The algorithm proposed an Adaptive Stochastic Gradient Descent (ADASGD) learning paradigm that starts the aggregation process when receiving a certain number of updates from the participating clients. The algorithm uses weighted aggregation to combine client updates. They multiply each client update by a certain factor composed of 2 elements. First is the damping element, an exponentially decreasing function based on the system's expected percentage of reliable clients. The second element is a boosting factor that boosts the gradient of unique data. Their results accuracy improvements compared to standard FL schemes. Furthermore, ADSGD showed 18.8% improvements in convergence speed compared to other staleness-aware approaches. This approach is designed to work in scenarios with slow and unreliable clients. However, it still suffers from similar limitations to FedAvg in terms of dealing with client failures. For instance, if a client fails repeatedly, it might still be called multiple times, affecting the round time.

3.4 Stragglers in Serverless Federated Learning

Although Serverless computing models show advantages in terms of resource efficiency and cost, the reliability guarantees of such systems are weak [71, 72]. Node failures can cause requests to be dropped or even executed multiple times. Furthermore, the scale-to-zero model implies that starting function instances from scratch *(cold starts)* might cause higher request latency. Therefore, the reliability assurances in serverless infrastructures are left to the developers [72]. There are two types of clients that can participate in serverless FL. The first type is functions running on edge devices. These functions inherit the reliability issues of edge devices. The devices themselves can drop or suffer communication or network issues.

Furthermore, edge devices tend to have slower performance due to hardware limitations. The second type is the functions running in a FaaS platform, either public or self-hosted. Although they tend to be more reliable than edge devices due to their Service Level Agreements (SLAs) providing better guarantees [73, 74], the serverless computing model itself provides best-effort infrastructure [72]. As a result, using functions might be less reliable than using virtual machines in some scenarios. For instance, the Service Level Objective (SLO) for Google Cloud functions is an uptime percentage of 99.95% [73] compared to 99.99% for multi-zone compute instances [75]. In the context of serverless FL with heterogeneous clients, the system encounters failures or variations in performance. Furthermore, cold starts might cause unexpected delays to function executions. As a result, even a few stragglers can affect system efficiency. Although we cannot eliminate stragglers from the system, we address the issue by adapting to clients' behavior and minimizing the effect of stragglers on the overall system performance.

4 System and Strategy Design

4.1 FedLess System Enhancements

In this section, we explain the enhancements added to the FedLess platform. These improvements aim to make the platform easier to use and provide the prerequisite architectural changes to include FedLesScan.

4.1.1 System Architecture

In section 3.2, we discussed the architecture of FedLess and its current limitations. The objective of improving the platform is to make it easier for users to develop and train FL models. Furthermore, the changes facilitate incorporating various FL strategies and techniques in the future. Figure 4.1 provides an overview of the modified system architecture and the core system components. The first significant enhancement is in the FedLess controller. Previously, the FedLess controller ran in an Openwhisk cluster on top of a Kubernetes deployment. Therefore, even for debugging, getting started with the platform required a complex deployment configuration. One of the main objectives of a FaaS-based FL platform is its abstraction of the infrastructure management layer away from the developer. We addressed this limitation by removing the need for Openwhisk and Kubernetes. In the new architecture, the FedLess controller is a light weight process that runs on any machine that includes its dependencies without any infrastructure management. As a result, the cost and time needed for development and maintenance should decrease compared to the old architecture.

We also added multiple sub-components to the FedLess controller. The strategy manager controls the behavior of the selected strategy. Each strategy contains the following:

- Client selection Scheme: Function responsible for clients selected for this training round.
- Aggregation Scheme: The aggregation scheme used by the aggregation function.

Developers can switch between different strategies with a single command line parameter.

Furthermore, FedLess used to support aggregation only on functions deployed in the local Openwhisk cluster. We abstracted the aggregation function to be usable on any self-hosted or public cloud. The effect of these changes lies in the ability to set up the platform quickly. The system no longer requires setting up the FedLess controller. Moreover, the system is no longer locked to using a specific serverless platform. On the database side, we added client history collection to store information about clients' behavior, such as failures, training duration, and the identifiers of missed rounds. We rely on the obtained behavioral data to implement our strategy and measure clients' performance. We also include components to facilitate debugging the platform. We explain their details in the next section.

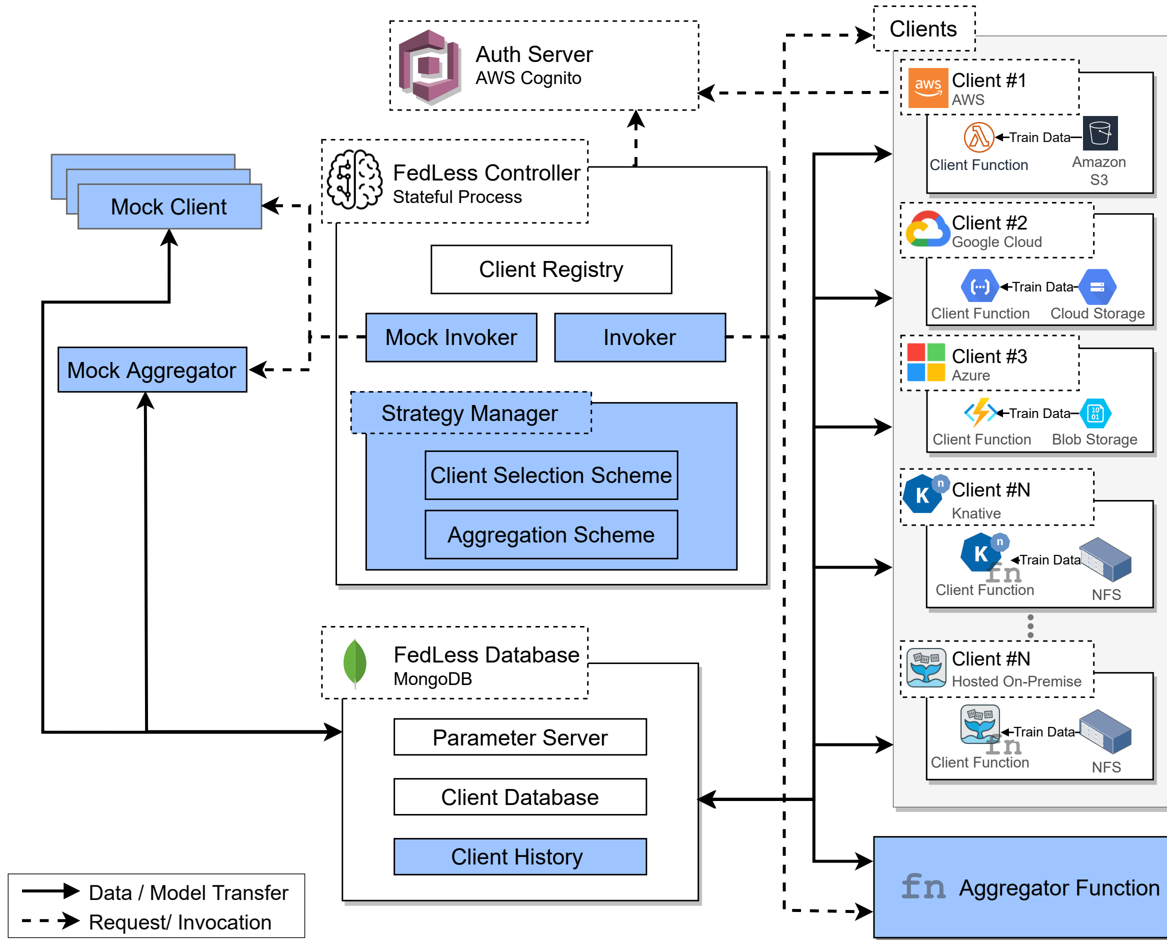


Figure 4.1: Modified Architecture of the FedLess platform. The highlighted components shows the modified components and additions to the system.

4.1.2 System Debugging Capabilities

One of the main challenges in a distributed system is developing and debugging the whole system. Critically developers spend a considerable amount of time tuning the model's hyper-parameters. Furthermore, debugging client functions is even more challenging, especially when the functions are deployed using cloud providers. Therefore, we introduced a mocking system in FedLess to enable developers to run and debug the whole platform on a single machine. The FedLess controller contains mock components *Mock Invoker*, *Mock Client*, and *Mock Aggregator* that simulates the behavior of the actual components. The Mock Client is a Class that runs the same code present in the client functions. The Mock Aggregator is a Class that runs the aggregation process locally. When running the FedLess controller in debug mode, the Mock Invoker is activated and passed as a parameter to the Strategy Manager. After clients are selected to participate in the current round, the Strategy Manager delegates the clients' invocation to the Mock Invoker, which invokes multiple instances of the Mock

Client class. When the clients finish training, the Mock Invoker receives a signal from the Strategy Manager to invoke the Mock Aggregator. The process repeats until the end of the training session. The developer can activate the mocking feature by specifying the *–mock* parameter when running the FedLess controller.

4.1.3 Training Workflow

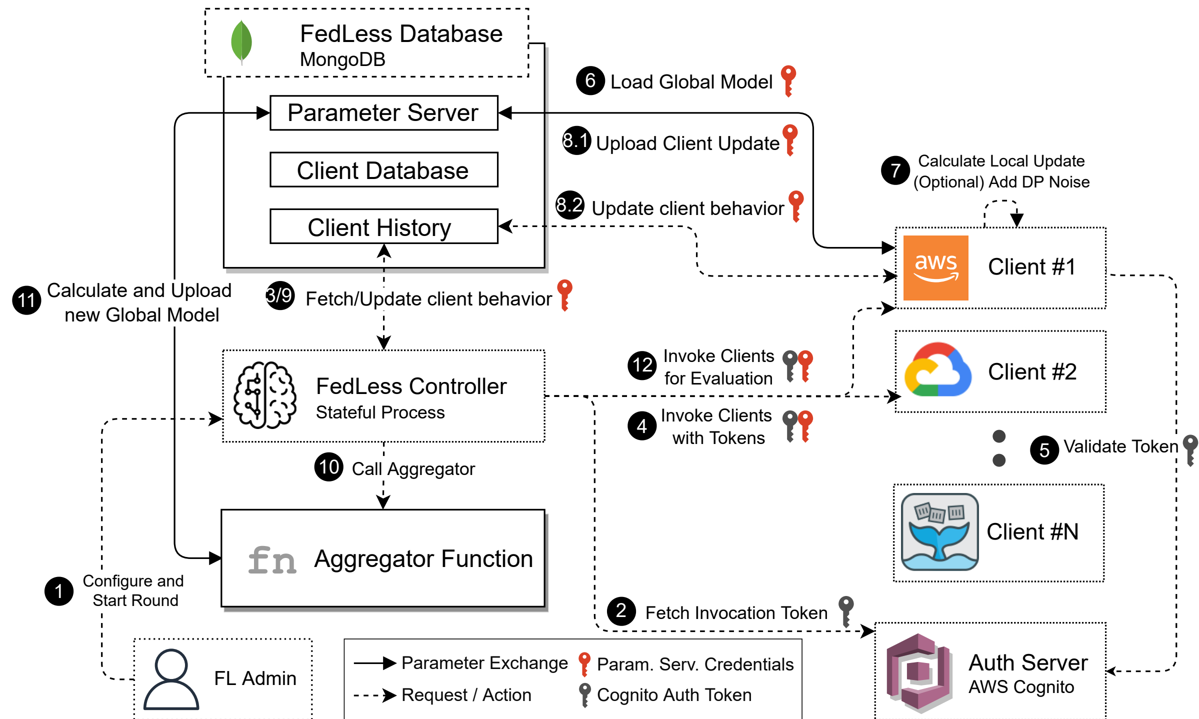


Figure 4.2: The training workflow of a typical training round in FedLess

Although the core idea of FedLess did not change, the training workflow changed slightly to accommodate the new architectural changes. Furthermore, the new flow should facilitate the implementation of other strategies, particularly FedLesScan, with minimal communication costs. The protocols required to invoke and authenticate clients remain the same.

Figure 4.2 demonstrates the communication among FedLess components in a typical training round. As discussed in Section 3.2, the FL admin configures the model, dataset, and hyperparameters before starting the training. In the beginning, the controller fetches invocation tokens from the authentication server. Next, the controller fetches the clients' behavioral data from the FedLess database. This step is done based on the selected training strategy. For instance, FedAvg uses random selection; therefore, it does not need to perform this step. On the other hand, other strategies can update the behavioral data of the clients if needed. Based on the selected strategy, a subset of the clients is chosen to participate in the current round. The controller invokes the client function using the invocation tokens. In

the next step, the client function contacts the authentication server to validate the invocation token. Upon successful validation, the clients fetch the latest global model from the parameter server and perform the training locally while collecting information about the current training progress. When the training is complete, the clients upload the new local model to the parameter server and push the collected information to the client History collection. Next, the client function responds to the controller signaling the end of the training. At the end of the training round, the controller adjusts the participant's behavioral attributes based on the strategy and pushes them back to FedLess database. Afterward, the controller invokes the aggregation function, which combines the clients' results into a new global model. At the end of the round, the controller invokes a subset of the clients for evaluation. The process then repeats for subsequent rounds.

4.2 Intelligent Client Selection

In this section, we explain our client selection algorithm. First, we explain the data we collect about the clients' behavior and how we distinguish between failures and slow updates. Second, we explain the details of the FedLesScan client selection algorithm. Lastly, we explain the details of some auxiliary functions used inside the algorithm.

4.2.1 Gathering Behavioral Data

The client selection scheme relies on data collected on clients' behavior in previous rounds. Our selection scheme considers separating reliable clients that do not miss their training round from stragglers. Furthermore, we use clustering to partition the reliable clients into subsets with similar behavior. In other words, clients with similar performance are called in the same round to decrease waiting time at the controller and increase the system's efficiency. That said, we divide clients into three tiers. The first tier is for clients which never participated in a training round. We use this tier to make sure that each client has a chance to participate at least once to collect data about their behavior. The second tier includes reliable clients which participated in training at least once before. The clustering is limited to the second-tier clients. Clustering these clients aims to group clients who behave similarly to improve system efficiency and reduce round time. We explain the details of clustering the second-tier clients in the next section. The last tier contains inconsistent clients which miss one or more successive rounds. The last-tier clients are not permitted to participate in the clustering. Clients are chosen from the last tier only if all clients from the first two tiers have been chosen and the required number of participants is not met.

The selection strategy relies on the following attributes to use in the clustering:

- *training time*: time taken by the client to finish training.
- *missed rounds*: a list that contains the round number of all missed rounds. In the next section, we discuss the usage of missed rounds to evaluate the client's penalty.

Moreover, each client has a *cooldown* variable, representing the number of rounds a client has to stay in the last tier. We evaluate the cooldown period from the client's last missed round. For instance, if client A missed round 2, the cooldown is set to 1. In this case, client A must stay in the last tier for one more round (round 3) before returning to the second tier in round 4. If client A missed round 4 (assuming client A is selected), the cooldown is multiplied by two. Consequently, client A has to stay in the last tier for two rounds. Equation 4.1 demonstrates how the cooldown variable is updated. Using this variable prevents wasting time and resources on calling clients which fail repeatedly. It can also reduce the impact of temporarily slow or unavailable clients by lowering their priority for a certain number of rounds.

$$cooldown = \begin{cases} 0 & : \text{if client completed round successfully} \\ 1 & : \text{if the client failed and cooldown = zero} \\ cooldown \times 2 & : \text{otherwise} \end{cases} \tag{4.1}$$

Algorithm 1 shows the steps taken by the FedLess controller and client function to update the clients' attributes and how we distinguish between crashed and slow clients. FedLess controller runs the *Train_Global_Model* function to train the global model for a single round. FedLess Clients run *Train* function to train their local model. We define *nClientsPerRound* as the number of clients that must be selected every round and *maxRounds* as the maximum number of training rounds allowed.

At the FedLess controller, we select a subset of the clients and wait until they finish, or a timeout occurs (lines 2-3). We explain the details of the *Select_Clients* function in the next section. Then we iterate over each successful response for which we reset the cooldown variable (lines 5-8). Because FedLess controller does not know if a client is slow or ultimately crashed, it assumes that the remainder of the invoked clients failed to finish. Accordingly, we update their attributes in which the current round is recorded in the missed rounds list (line 10), and the cooldown is updated according to Equation 4.1 (line 11-15).

The *Client_Update* function shows the training flow at the client-side. We start by loading the client's behavioral history in the previous rounds (line 22). Afterward, we start to record the start time before loading the model and dataset (lines 23-25). Next, the client function trains the model on the local data (line 26). We record the training time at the end of training (line 27). The local model updates are sent to the parameter server (line 28). Clients are then required to update their training time for the current round (line 29). Furthermore, slower clients have a chance to correct information about their missed rounds if they finish the round later. As previously stated, the controller considers clients that did not finish the round in time as crashed. Therefore, distinguishing between slow clients and crashes is done on the client-side. We do that by allowing clients to correct assumptions made by the controller and erase the current round from their missed rounds list (lines 30-32).

As per our three-tier division, we can define clients in terms of their assigned tier:

1. *Rookies* (first-tier): which were not called before and yet to participate in training.
2. *Participants* (second-tier): which are allowed to participate in the clustering.

Algorithm 1: Client History Updates at FedLess Controller and Clients.

```
1  FedLess Controller:
2  Function Train_Global_Model(Clients, round):
3      selectedClients = Select_Clients(clients, round, maxRounds, nClientsPerRound)
4      success, failures= Invoke_Clients(selectedClients)
5      for each client in success do
6          client.cooldown = 0
7          updateClientHistory(client)
8      end
9      for each client in failures do
10         client.missedRounds.append(round)
11         if client.cooldown ≤ 0 then
12             client.cooldown = 1
13         else
14             client.cooldown = client.cooldown × 2
15         end
16         updateClientHistory(client)
17     end
18 return
19
20 FedLess Client:
21 Function Client_Update(hyperParameters, round, database):
22     clientHistory = loadClientHistory()
23     startTime = timer.startTime()
24     model = loadModel()
25     dataset = loadDataset(database)
26     model.train(hyperParameters, dataset)
27     endTime = timer.endTime()
28     saveModel(database, model)
29     clientHistory.trainingTimes.append(endTime - startTime)
30     if round in clientHistory.missedRounds then
31         clientHistory.missedRounds.remove(round)
32     end
33     updateClientHistory(database, clientHistory)
34 return
```

3. *Stragglers* (third-tier): which missed one or more successive rounds (*cooldown* ≥ 0).

We ensure that all clients have an opportunity to participate at least once at the start of the training by giving *Rookies* the highest priority in our selection scheme. This method also allows us to collect behavioral data on all participants, which can be used for clustering in subsequent rounds. As the training progresses, the number of rookies decreases until it reaches zero, when all clients have been invoked before. *Participants* represent the clients ready to participate in the training round. During training, reliable clients are labeled as participants and are clustered based on their previous behavior. *Stragglers* have the least priority.

4.2.2 FedLesScan Selection Algorithm

Algorithm 2: Client selection

```
1  Function Select_Clients(clients, round, maxRounds, nClientsPerRound):
2      rookies ⟵ list of clients not called before
3      pariticipants ⟵ list of clients available for clustering
4      stragglers ⟵ list of clients where missedRounds.last + cooldown > round
5      startClusteringRound ⟵ −1
6      if len(rookies) ≥ nClientsPerRound then
7          return random.sample(rookies, nClientsPerRound)
8      end
9      nClientsFromClustering = min(nClientsPerRound - len(rookies), len(pariticpants))
10     if nClientsFromClustering ≥ 0 and startClusteringRound == -1 then
11         startClusteringRound = round
12     end
13     nStragglers = nClientsPerRound - nClientsFromClustering - len(rookies)
14     roundStragglers = chooseRandomly(stragglers, nStragglers)
15     clusteringData = []
16     for each client in participants do
17         trainingEma = getEma(client.trainingTimes)
18         missedRoundRatios = divide(client.missedRounds, round)
19         missedRoundEma = getEma(missedRoundRatios)
20         clusteringData.append((trainingEma, missedRoundEma))
21     end
22     labels = DBScanClustering(clusteringData)
23     sortedClusters = sortClusters(participants, labels, round)
24     clusteringResults = Sample(sortedClusters, round, maxRounds,
            startClusteringRound, nClientsFromClustering)
25 return [rookies + clusteringResults + roundStragglers]
```

Algorithm 2 shows the pseudo-code for our selection scheme. The algorithm promotes fair

selection for reliable clients while involving stragglers less in the training process. We start by selecting from the pool of rookie clients. The selection is made randomly if the pool is large enough until the remaining rookies are less than the required number of clients per round *(nClientsPerRound)* (lines 6-8). We determine the number of clients selected from the second tier by subtracting the number of remaining rookies from nClientsPerRound (line 9). Furthermore, we record the round in which we begin sampling tier 2 clients to use it later to choose which cluster to train (lines 10-12). If the clients selected from the first and second tiers are not enough, we randomly select the remaining number from the third tier (lines 13-14).

For clients which will participate in clustering, we obtain two attributes. First, *trainingEma* which represents an exponential moving average (EMA) [76] on the previously recorded training time (line 17). Using a weighted average better represents the current client behavior by giving higher weight to the recent recorded times. Second, *missedRoundEma* which is a penalty factor based on previous missed rounds. This factor should satisfy two objectives. First, recent failures should have higher penalties. For instance, A client which failed in the second round would have a higher penalty than a client which failed in the first round. Second, the penalty should fade as we progress in training if the client becomes more reliable. In other words, if a client only missed the first round, the calculated client's penalty in round five must be less than its penalty in round two. To achieve this, we divide the numbers in the missed rounds list by the current round number to get a list of ratios (line 18). As the training progresses, the effect of a specific missed round decreases because the current round number increases. We then compute the missedRoundEma as an EMA on the computed list of the missed rounds (line 19).

After collecting the data for the participating clients, we use it to cluster the clients. The *DBScanClustering* function partitions the clients' data into separate clusters, each with a specific label (line 22). As explained in Section 2.2.3, DBSCAN requires ϵ parameter to be set. For simplicity, we treat outliers as a single cluster. Although developers can tune the clustering parameters manually, we provide a simple method to pick the best-performing parameters. *DBScanClustering* performs a grid search for the best parameter values that yield the highest *Calinski-Harabasz index* [77]. This score measures the ratio between intra-cluster and inter-cluster dispersion to evaluate the quality of the clusters. We chose this index because it is fast to compute; therefore, it will not affect our running time. Moreover, the low time complexity of DBSCAN means that the time needed to run the clustering multiple times takes a few seconds, which is insignificant compared to the overall round time. Instead of developing the clustering algorithm from scratch, we rely on the DBSCAN implementation provided by *scikit- learn* library [78].

The next step involves sorting the clusters according to the average total time of their members, where the total time of a client is the sum of its training time and its penalty time, as shown in Equation 4.2 (line 23).

$$totalEma = trainingEma + missedRoundEma \times maxTrainingTime \tag{4.2}$$

We then select clients according to the *Sample* function explained in section 4.2.3 (line 24).

The returned list of clients contains rookie clients, if any, appended to the selected clients from clustering and the list of stragglers that are selected to complete the required number of clients (line 25).

4.2.3 Selecting Clustering Clients

Algorithm 3: Sampling Function for clustering participants

```
1  Function Sample(sortedClusters, round, maxRounds, startClusteringRound,
     nClientsFromClustering):
2      perc = (round - startClusteringRound)/ max(maxRounds - startClusteringRound,
        1)
3      startIdx = percentile(sortedClusters, perc)
4      samples = []
5      while nClientsFromClustering > 0 do
6          cluster = sortedClusters[startIdx]
7          if len(cluster.clients) ≥ nClientsFromClustering then
8              sort cluster.clients according to the number of successful contributions.
9              samples.append(cluster.clients[0: nClientsFromClustering])
10             nClientsFromClustering = 0
11         else
12             samples.append(cluster.clients)
13             nClientsFromClustering -= len(cluster.clients)
14         end
15         startIdx = (startIdx + 1) % len(sortedClusters)
16     end
17 return samples
```

The *Sample* function exists to select clients from the list of sorted clusters. Instead of randomly selecting clusters to run, we start by running the fastest clusters earlier in the training process. Then, we gradually move to slower clusters further in the sorted list. That way, faster clients serve as a hot start for the slower clients. Algorithm 3 shows the process used to pick clients from the clusters. We choose the cluster corresponding to our current training progress (lines 2-3). The current progress is determined using the ratio between the current round and the maximum number of rounds. Initially, we start with an empty list and add clients as we move along (line 4). Afterward, we check whether the current cluster size is more than the number of clients required (line 7). If that is the case, we sample clients from the selected cluster prioritizing the ones involved less in training (lines 8-9); otherwise, we select the whole cluster (line 12). We update the required number of clients in both cases by subtracting the number of selected clients (lines 10, 13). As long as we do not reach the required number of clients, we move to the next cluster and repeat the same process (line 15). If we reach the end of the final cluster, we return to the first cluster in the list.

4.3 Staleness-Aware Aggregation

In the previous section, we discussed our intelligent selection strategy. Although an informed selection mechanism should improve the system's efficiency, slow clients will still be in the system. Therefore, we use a staleness-aware aggregation scheme to convert FedLesScan to a semi-asynchronous strategy. In FedLess, slow clients might push their updates to the parameter server after the round ends. Usually, these updates are considered wasted contributions. Additionally, client updates might still hold valuable information to improve the model's performance. Our approach to tackling this problem is to aggregate delayed updates with a dampening effect asynchronously. In other words, delayed updates are considered the next time the aggregation function is called. For a system with K clients, the global model w_{t+1} after aggregation at round t is computed as a weighted average of clients' contributions that exist in the FedLess database. Equation 4.3 shows the updated aggregation function used to include delayed updates. Notice that $w_{t_k}^k$ is the local model of client k at round t_k and n_k represents the cardinality of the dataset at client k while n is the total cardinality of the aggregated clients. If the updates arrive at the same round ($t_k = t$), the equation becomes the same as Equation 2.1. On the other hand, older updates ($t_k < t$), are dampened by $\frac{t_k}{t}$. To avoid obsolete updates from affecting the training, the aggregator uses a parameter τ to dictate the maximum age of updates included in the aggregation. Updates with $t - t_k \geq \tau$ are discarded by the aggregator.

$$w_{t+1} \longleftarrow \sum_{k=1}^{K} \frac{t_k}{t} \times \frac{n_k}{n} w_{t_k}^k \tag{4.3}$$

5 Experiments and Evaluation

The system improvements added to FedLess enabled us to implement other strategies to compare their performance to FedLesScan. We compare FedLesScan to two novel training strategies, FedAvg [24] and FedProx [26]. The evaluation should provide insights into FedLesScan's performance expectations and limitations.

5.1 Metrics and Experiments Configuration

5.1.1 Evaluation Metrics

In this section, we provide an overview of the metrics and evaluation methodologies used to evaluate FedLesScan. We compare our results to both FedAvg and FedProx. Our evaluation covers three aspects:

- Round utilization and system performance.
- Model performance, including model loss and test accuracy.
- Strategy efficiency by analyzing experiment time and cost.

To properly evaluate round utilization and strategy performance, we use Effective Update Ratio (EUR) [67], demonstrated by Equation 5.1. EUR per round is defined as the ratio between successful clients (S) and the subset of selected clients (C). EUR shows the effect of stragglers on round utilization. Higher EUR means less wasted resources since clients requested to participate in a certain round end up contributing to the global model.

$$EUR = \frac{S \cap C}{C} \tag{5.1}$$

Furthermore, we provide insights into the bias of the client selection scheme using variance plots. This works by showing the frequency of selection for each client across the training session. We define bias as the difference between the frequency of the least called client and the most client [67]. Ideally, we target low bias in light-straggler situations. On the other hand, the bias should increase in straggler-heavy scenarios due to prioritizing reliable clients in training.

To evaluate the model performance, we measure the end accuracy reached by the model in a specific number of rounds. Furthermore, we show the progress of the model's accuracy throughout the training. The use of centralized evaluation does not accurately represent what happens in a highly scalable distributed FL system. Therefore, we chose to evaluate the system in a distributed manner by randomly choosing clients to evaluate the model on

their test dataset. The number of clients participating in the testing phase (C_t) is the same as the number of clients called per round. Each client computes the accuracy according to Equation 5.2. Then the average accuracy is weighted by the ratio between client dataset cardinalities (n_c) and the total cardinality of the test dataset (N). Equation 5.3 shows the formula used to compute the average accuracy per round. Notice that for MNIST, the dataset is small enough to perform a central evaluation.

$$acc = \frac{TruePredictions}{TotalNumberOfPredictions} \tag{5.2}$$

$$average_acc = \sum_{\forall c \in C_t} \frac{n_c}{N} \times acc \tag{5.3}$$

We analyzed the experiment's time and cost to evaluate the system's efficiency. We utilize Google Cloud's next-generation FaaS offering [79] to deploy all client functions for our experiment. The rationale behind using 2nd gen cloud functions is the major enhancements in infrastructure, which address a few of the previously mentioned limitations of FaaS infrastructures. These enhancements include enhanced runtime for up to 60 mins, larger instances with up to 16 GB of RAM and four vCPUs, concurrency, a minimum number of pre-warmed instances to minimize cold starts, and enhanced traffic management. We use their cost computation model [80] to estimate the cost for each client function based on the number of invocations, memory, and duration of execution.

5.1.2 Benchmarks and Datasets

For our experiments, we rely on four datasets from 4 different benchmarks. We pick different domains to make the evaluation conclusive to test the new strategy. These domains include image classification (MNIST, FEMNIST), speech recognition (Speech Commands), and language modeling (Shakespeare).

The first dataset is MNIST Handwritten Image Database. It contains 60000 images available for training and 10000 for central evaluation. The images are sorted and randomly distributed to 300 clients.

We chose two datasets from *LEAF* [65] which is a benchmarking framework for FL. It contains a variety of datasets that cover different domains. As per the original setup covered in [21], we use both the FEMNIST dataset for image classification and the Shakespeare dataset for character prediction. FEMNIST is short for Federated Extended MNIST, which is based on modifying the Extended MNIST dataset [81]. The dataset contains more than eight hundred thousand images, with an average of 226 images per partition. Shakespeare dataset contains sentences from *The Complete Works of William Shakespeare* [82]. The dataset is partitioned such that each role in each play is mapped to a specific partition. With more than four million samples, each of length 80 characters, a device is responsible for an average of 3743 samples.

The last benchmark is *FedScale* [25], which contains a group of large-scale realistic benchmark datasets for tasks such as object detection, word prediction, and speech recognition. From FedScale, we choose real-world speech recognition represented by the Google Speech

Commands dataset [23]. The dataset was designed to build basic and helpful voice interfaces for applications with common words such as "Yes", "No," and directions [23]. The dataset contains 105,000 samples of 1-second audio files distributed across 2618 clients. One of the challenges we faced with the Fedscale benchmark is that their implementation uses Pytorch [83] while FedLess is based on Tensorflow [84]. Furthermore, the dataset is split using client-data mapping, which maps file ids to a particular client. The dataset was designed to be split into 2618 clients, 2168 of which are for training and the rest for validation and testing. Accordingly, we reimplemented the preprocessing scripts for the Speech Commands dataset. Moreover, we implemented a custom mapping algorithm to scale the number of clients down according to a specific ratio. For example, a ratio of 4 means that one FedLess client is responsible for the training data of 4 FedScale clients. In our experiments, we used a ratio of 4 to map 542 FedLess clients to 2168 Fedscale clients.

5.1.3 Models Architecture and Parameters

Each experiment trains a different model suited for the task. For MNIST, FEMNIST, and Shakespeare, we use the same model architecture mentioned by [21] and used in the original LEAF benchmark paper [65]. For MNIST, we use a 2-layer CNN with a 5x5 kernel. A 2x2 max-pooling layer follows each Convolutional layer. The model ends with a fully-connected layer with 512 neurons and a ten-neuron output layer with softmax activation. The model had a total of 582,026 trainable parameters.

We picked two datasets from the LEAF benchmark. For FEMNIST, we use a 2-layer CNN with a 5x5 kernel. A 2x2 max-pooling layer follows each Convolutional layer. The model ends with a fully-connected layer with 2048 neurons followed by an output layer of 62 neurons with softmax activation. The final model contains 6,603,710 trainable parameters.

We use a Long Short Term Memory (LSTM) recurrent neural network [85] for the Shakespeare dataset. The model consists of an embedding layer of size eight followed by two LSTM layers with 256 units and an output layer of size 82 with softmax activation. The number of trainable parameters in the final model is 818,402.

For the FedScale benchmark, we opted for designing a simple CNN instead of replicating models from the FedScale benchmark itself. Since we are only interested in comparing different strategies, the designed model should show any differences if they exist. Nevertheless, the designed model was on-par with the results from the original FedScale paper [25] in terms of accuracy. The model's architecture consists of two identical blocks followed by an average pooling layer and an output layer with 35 neurons, and a softmax activation. A block contains two convolutional layers with a 3x3 kernel followed by a max-pooling layer. A dropout layer follows the max-pooling layer with a rate of 0.25 to avoid overfitting. The number of trainable parameters in the model is 67,267.

Table 5.1 shows the different hyperparameters used for each dataset. Due to the divergence issues [65] that faces FedAvg, we use a high learning rate and a small number of epochs for the Shakespeare dataset.

Dataset	Hyper Parameters			
	Epochs	Batch Size	Optimizer	Learning Rate
MNIST	5	10	Adam [86]	0.001
FEMNIST	5	10	Adam	0.001
Shakespeare	1	32	SGD [35]	0.8
Speech Command	5	5	Adam	0.001

Table 5.1: Model hyperparameters used for each dataset.

5.1.4 Experiment Setup

To properly scale our experiments and eliminate database bottlenecks, we deployed the FedLess database on a single machine with 40vCPUs and 177GB of RAM. Furthermore, we use the same machine to run the FedLess controller. We deployed the aggregator function on a self-hosted single node OpenFaas cluster with 45GB of RAM and 10 vCPUs. The aggregation function had a memory limit of 7GB. Furthermore, we used a file server with 45GB of RAM, 10 vCPUs, and 200GB of storage to host the four datasets.

The client functions are deployed in the *europe-west4* region due to the limited availability of 2nd gen functions at the time of writing. Each client function had a memory limit of 2048MB and a maximum timeout of 540 sec. Table 5.2 show the configuration we used for each dataset. Because each dataset has a different number of clients, a single client runs on one function instance to ensure all clients can perform the training independently.

Benchmark	Dataset Name	No. of Clients	Clients Per Round
-	MNIST	300	200
LEAF	FEMNIST	300	175
LEAF	Shakespeare	100	50
FedScale	Speech Command	542	200

Table 5.2: Total number of clients and clients per round used for each dataset.

We aim to evaluate the performance of FedLesScan against delays and function dropouts. Although the deployed functions will show delays, failures, and cold starts, it does not indicate how our strategy behaves in extreme situations [21]. Therefore, we decided to test two separate scenarios.

In the standard scenario, we perform the experiments on the client function without any modifications. Furthermore, the round time is adjusted to ensure clients can finish their local training before the round ends. This scenario shows the performance in a more real-world situation when using FaaS providers as an infrastructure base for the experiment. In subsequent sections, we refer to this experiment as the standard scenario.

The second scenario includes synthesized experiments where we simulate multiple stragglers ratios in the system. Although there might be different reasons for client failure, such as memory limit, function timeout, or communication issues, simulating different failure types

will significantly increase the experiment cost. These failures can only have one of two effects on the clients. Clients can completely crash (not push their updates) or push their updates after the training round is finished (slow updates).

In order to simulate slow updates, we opted to strict the round time such that clients with cold starts will not finish before the end of the round. We considered simulating slow updates by introducing delays in the functions, but we found a significant increase in the experiment cost. To simulate failures, we randomly select a specific ratio of clients to fail their training at the beginning of each experiment. We perform four different experiments for each dataset with 10%, 30%, 50%, and 70% stragglers. We refer to the experiment by its stragglers' ratio in subsequent sections.

With about 60 experiments, we analyze the results on each dataset separately and provide further insights on the behavior of FedLesScan compared to FedAvg and FedProx.

5.2 Utilization and Strategy Performance

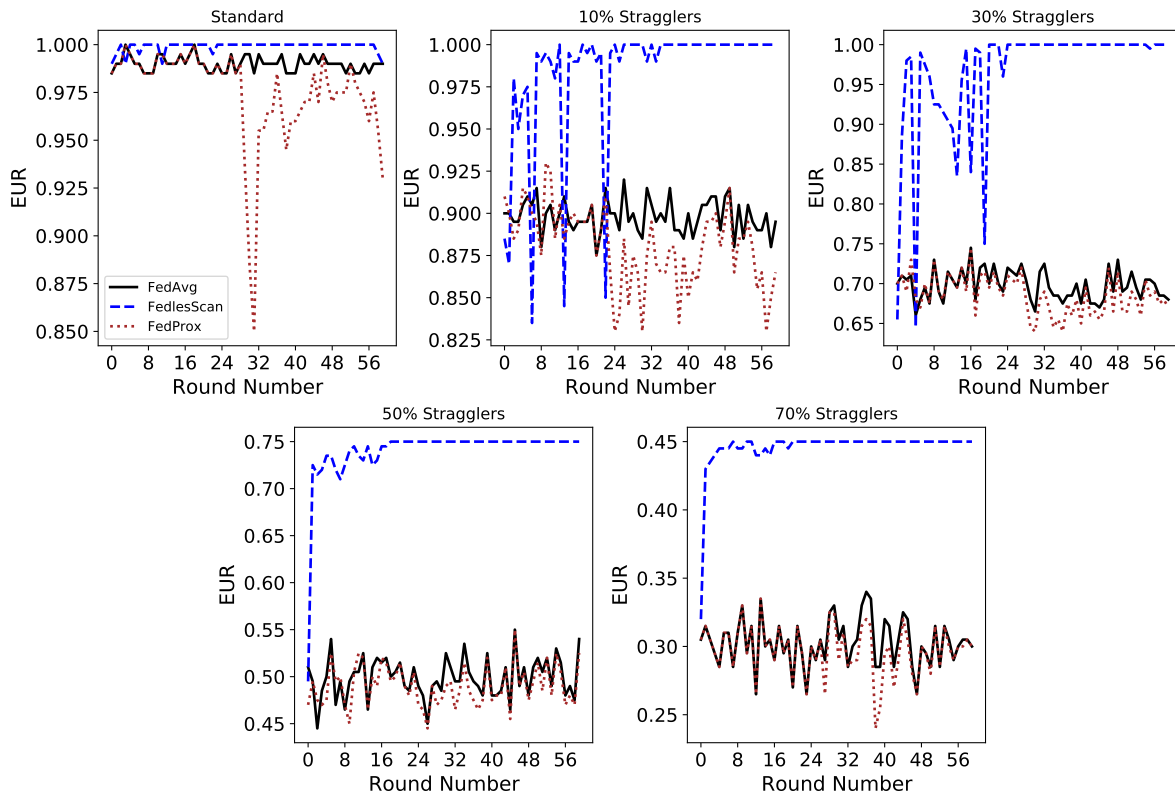


Figure 5.1: EUR comparison between FedAvg [24], FedProx [26] and FedLesScan on the MNIST dataset.

We start our evaluation by comparing the EUR of the three strategies. The EUR shows the utilization of the rounds, which demonstrates how FedLesScan's selection strategy chooses the

clients to increase the overall round utilization. Figure 5.1 depicts the EUR comparison among the three strategies. First, in the scenario with no straggler simulation, we see FedProx suffers from slight drops compared to the other algorithms. This happens because the algorithm uses a custom loss function at the client-side, as discussed in section 3.3. This function includes a factor that relies on the difference between the local and global models to reduce divergence from the global model. As a result, computing the difference between both models every epoch might affect the overall performance of the client function. Therefore, some clients do not finish their training in time. With large models FedProx clients tend to take a longer time or exceed their memory limit compared to the other two strategies.

On the other hand, FedLesScan consistently achieves higher EUR compared to the other strategies; we also see the gap increase as the percentage of stragglers increases. Furthermore, we notice occasional drops in EUR plots for FedLesScan. These drops show the effect of clustering the clients. Distributing stragglers across the training rounds will affect the efficiency of more rounds. For instance, consider a system that contains ten stragglers. If the invocation of those ten stragglers is distributed among all rounds, there will be at least ten rounds affected by stragglers. These effects can lead to clients waiting for the straggler to finish, decreasing the system efficiency. Alternatively, FedLesScan combines stragglers with similar behavior together, decreasing their effect on training the other clients. This leads to occasional drops in the EUR, as seen in the plots, but it shows that the system can maintain a higher ratio in subsequent rounds.

Although the EUR shows the efficiency of the system, it does not show the bias of the strategy. A system that utilizes a specific subset of clients and discards the rest of the clients will have high EUR but will underutilize the rest of the clients. The violin plots in Figure 5.2 provides us with further insights into the bias encountered by our strategy. The graph shows a distribution based on the number of invocations for each client (y-axis). We demonstrate bias by the difference between the highest and lowest point in the distribution. When the difference between the two points increases (the height increase), the algorithm is biased towards a specific subset of clients. On the contrary, when the height decreases, the difference between the most and least invoked clients is low. Furthermore, the width of the plot at a particular point X provides insights into the ratio of clients called X times. If the width at point X is larger than at an arbitrary point Y, then the number of clients called X times is more than the number called Y times. We rely on this explanation to understand the behavior of our strategy on the four datasets.

As per the information discussed above, the graphs in Figure 5.2 show the bias observed by our selection strategy. FedAvg and FedProx use random client selection; therefore, they show similar behavior without distinguishing between stragglers and reliable clients. On the other hand, FedLesScan dynamically adapts to stragglers. We see that the strategy promotes fair client selection in the standard scenario. We demonstrate this by comparing the height difference in each plot. We notice that the FedLesScan plot is more concentrated in the middle, which means clients get an equal share of training.

In the synthesized scenarios, we see FedLesScan prioritizing reliable clients while relying on stragglers less during training. We see a clear separation between the number of invocations

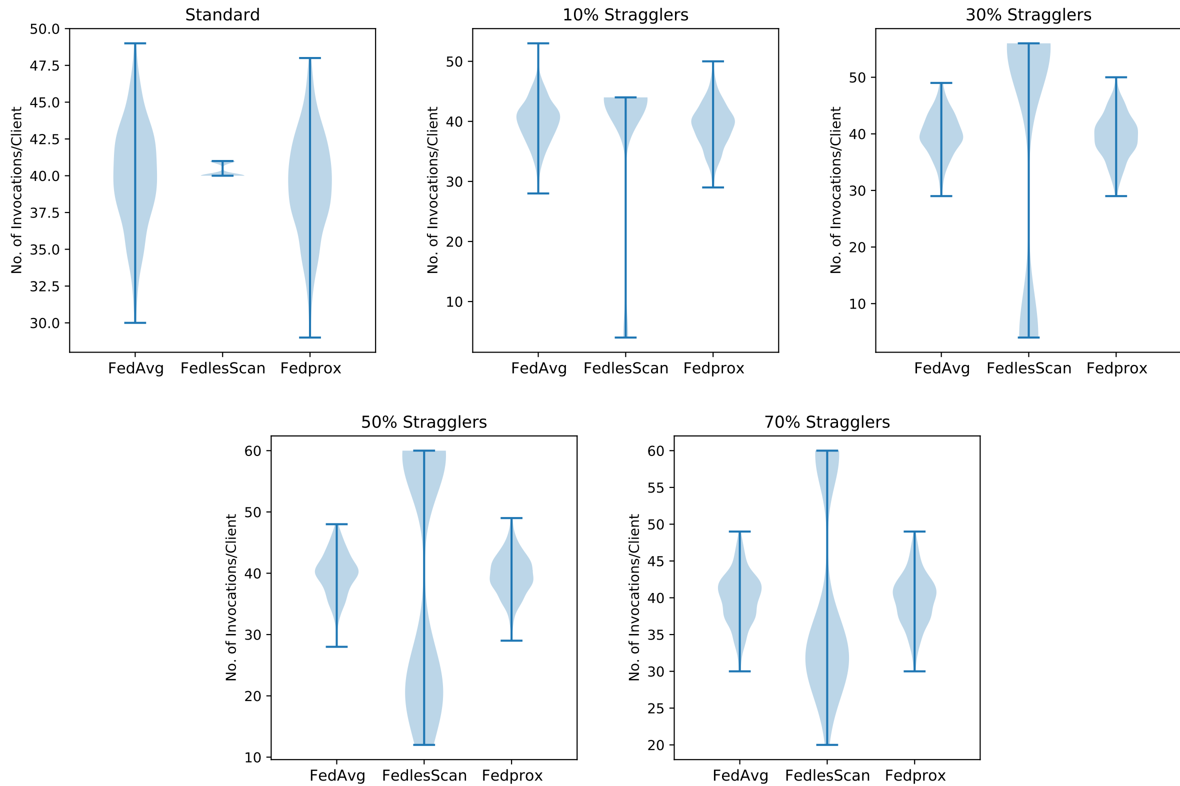


Figure 5.2: Distribution of client's invocation frequency on the MNIST dataset.

of reliable clients compared to stragglers.

For the FEMNIST dataset, Figure 5.3 illustrates the difference between the three strategies in EUR during training. In the standard scenario, we see FedLesScan and FedAvg perform similarly while producing better round utilization than FedProx. We see the effect of FedProx's custom loss function, which increases the training time at the client leading to more stragglers. This effect is particularly noticeable in FEMNIST because the number of trainable model parameters is large, making the loss function more computationally intensive. To further quantify the difference, we computed the average EUR during training. FedProx had an average EUR of 0.962 compared to 0.995 and 0.996 to FedAvg and FedLesScan respectively.

The rest of the plots in Figure 5.3 depicts the performance of the three strategies in our synthesized scenarios. We notice that the round utilization gap increase as the percentage of stragglers increases. Similar to the behavior on MNIST, FedLesScan shows occasional drops in EUR as an effect of clustering clients with similar behavior together. Although invoking stragglers in the same round affects the round's utilization, it improves the long-term system efficiency by mitigating their impact on other clients.

Figure 5.4 shows the client distribution based on the number of invocations per client. In the standard scenario, FedAvg and FedProx show more bias than FedLesScan. The observed behavior shows that our strategy utilizes all participants more fairly in the standard scenario

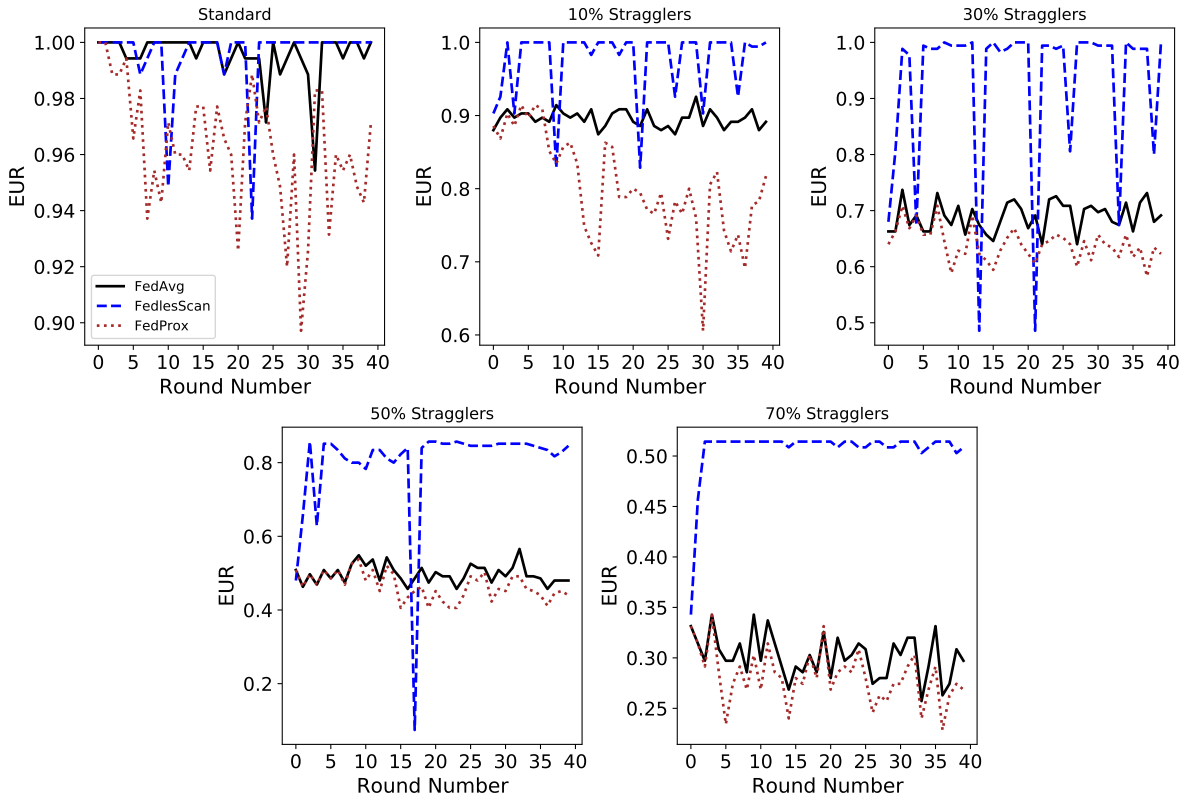


Figure 5.3: EUR comparison between FedAvg [24], FedProx [26] and FedLesScan on the FEMNIST dataset.

than using random client selection. In the synthesized scenarios, We notice a similar behavior to MNIST. We see a distinction between reliable clients and stragglers in the number of invocations. The effect becomes more evident in the system that suffers from more stragglers.

For the language modeling task on the Shakespeare dataset, we observe a similar pattern in EUR comparison, as shown in Figure 5.5. We calculated the average EUR for the three strategies. In the standard scenario, FedLesScan had an EUR of 0.94 compared to 0.87 and 0.86 for FedAvg and FedProx respectively. Furthermore, we noticed a few clients failed because they exceeded the memory limit of 2GB set for each function. As a result, we see lower round utilization in the first plot when no synthesized stragglers are present in the system. We refer to this note when we demonstrate the violin graph for the Shakespeare dataset.

The rest of the plots in Figure 5.5 show that the difference in utilization increases as the ratio of stragglers increases. In the scenario with 70% stragglers, FedLesScan had an average EUR of 0.7, which is 30% percent more than FedAvg and FedProx. We choose to run a few numbers of rounds for the Shakespeare dataset because LSTMs take a long training time, especially on CPU-only functions. Most clients in the Shakespeare dataset take more than 9 mins/round to finish their local training compared to 2 minutes for the other datasets.

The distribution of invocation per client shows similar behavior to other datasets, as shown

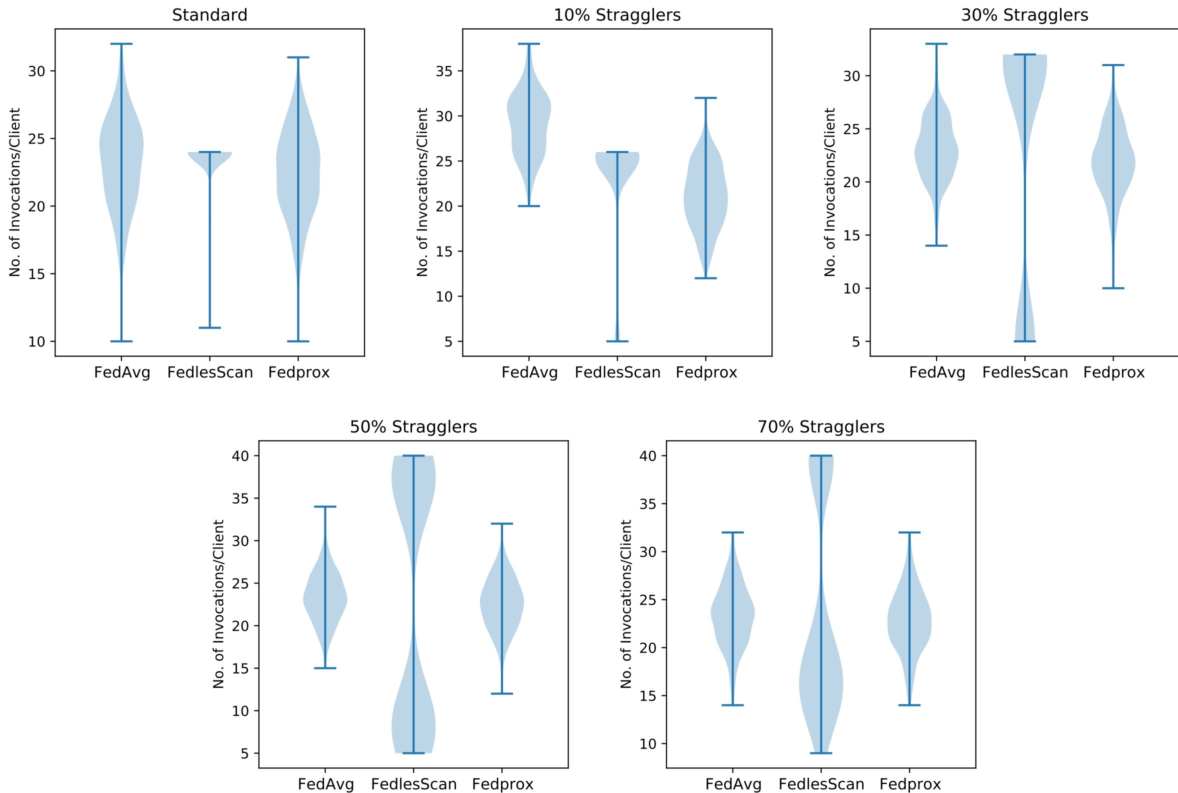


Figure 5.4: Distribution of client's invocation frequency for the FEMNIST dataset.

in Figure 5.6. Unlike the previous two datasets, we notice FedLesScan had a high bias in the standard scenario. We argue that the cause of the high bias is because some clients failed due to exceeding their memory limit, as we previously noted. The synthesized scenarios show similar behavior to the previous two datasets, with FedLesScan having slightly higher variance than the other two approaches. This behavior becomes more evident as the ratio of stragglers increases.

To further inspect our approach, we evaluated the training statistics on FedScale, a benchmark that contains real-world datasets. We choose the Speech Commands dataset to evaluate the performance on a real-world speech recognition task. We ran the experiment for 35 rounds in the standard scenario, while synthesized Scenarios ran for 60 rounds to show the system behavior for prolonged training sessions. Figure 5.7 depicts the difference in EUR during training between the three strategies.

The standard scenario in the first plot shows similar behavior to our evaluation on the FEMNIST dataset where FedProx struggles to match the EUR of the other two approaches. In all approaches, few of the clients did not meet the memory limit requirements for the functions. As a result, we still see slight drops in EUR for all strategies. Because of the large number of clients invoked per round, this effect is not as noticeable as it was with the Shakespeare dataset.

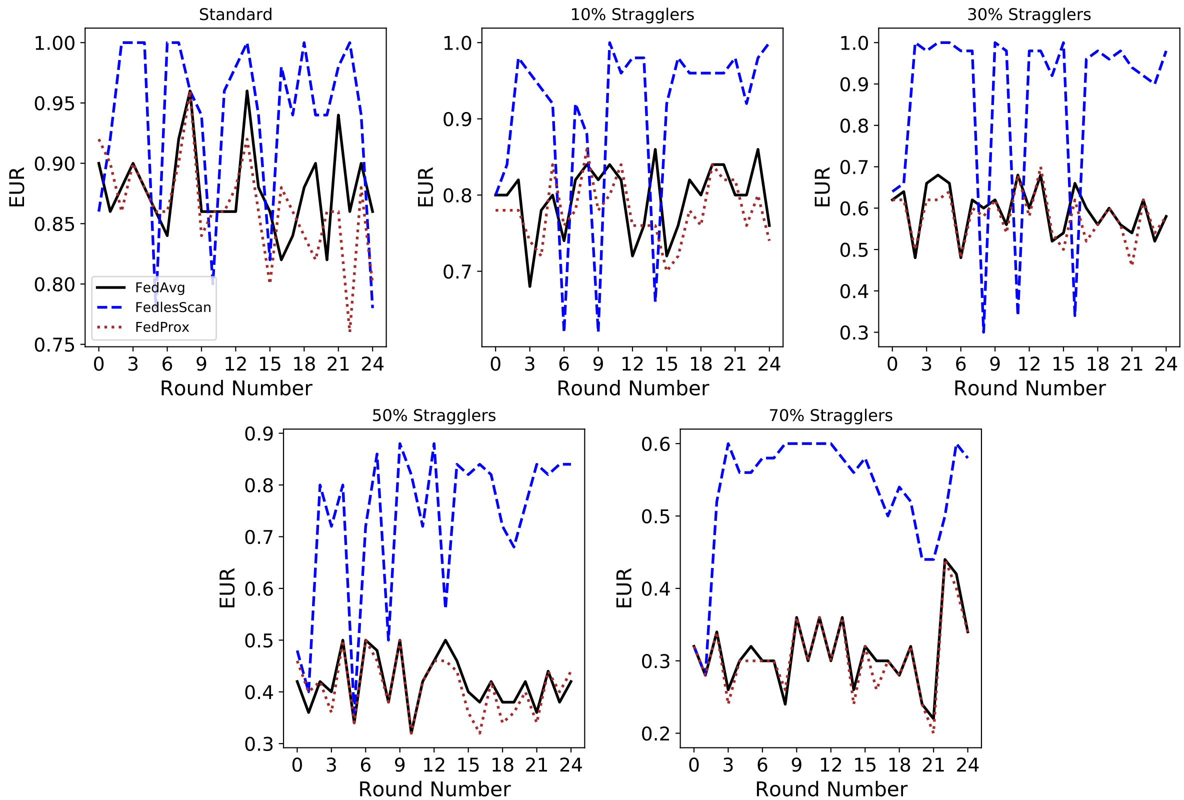


Figure 5.5: EUR comparison between FedAvg [24], FedProx [26] and FedLesScan on Shakespeare dataset for language modeling task.

In the synthesized scenarios, FedLesScan achieved a higher average EUR across four different ratios, with the difference further increasing as the ratio of stragglers increases. Furthermore, we see the effect of using a clustering-based algorithm in the steep drops in EUR that occasionally happens during training. We noticed this behavior before with MNIST and FEMNIST datasets where stragglers are grouped in the same round to reduce their effect on training the other fast clients and avoid long waiting times at the controller.

Figure 5.8 shows the difference in behavior across the different scenarios. In the standard scenario, we notice that the width of the distribution is higher in the range of 15 invocations per client. This means that number of invocations is similar for most of the clients. Furthermore, we observe that the FedLesScan distribution shows few clients with a low number of invocations. This is because clients that have repeatedly failed due to memory constraints are not used as often as the rest of the clients. This effect is similar to what we noticed in the standard scenario in the Shakespeare dataset. The rest of the plots show a higher bias than random client selection, while the values are spread more than in previous experiments.

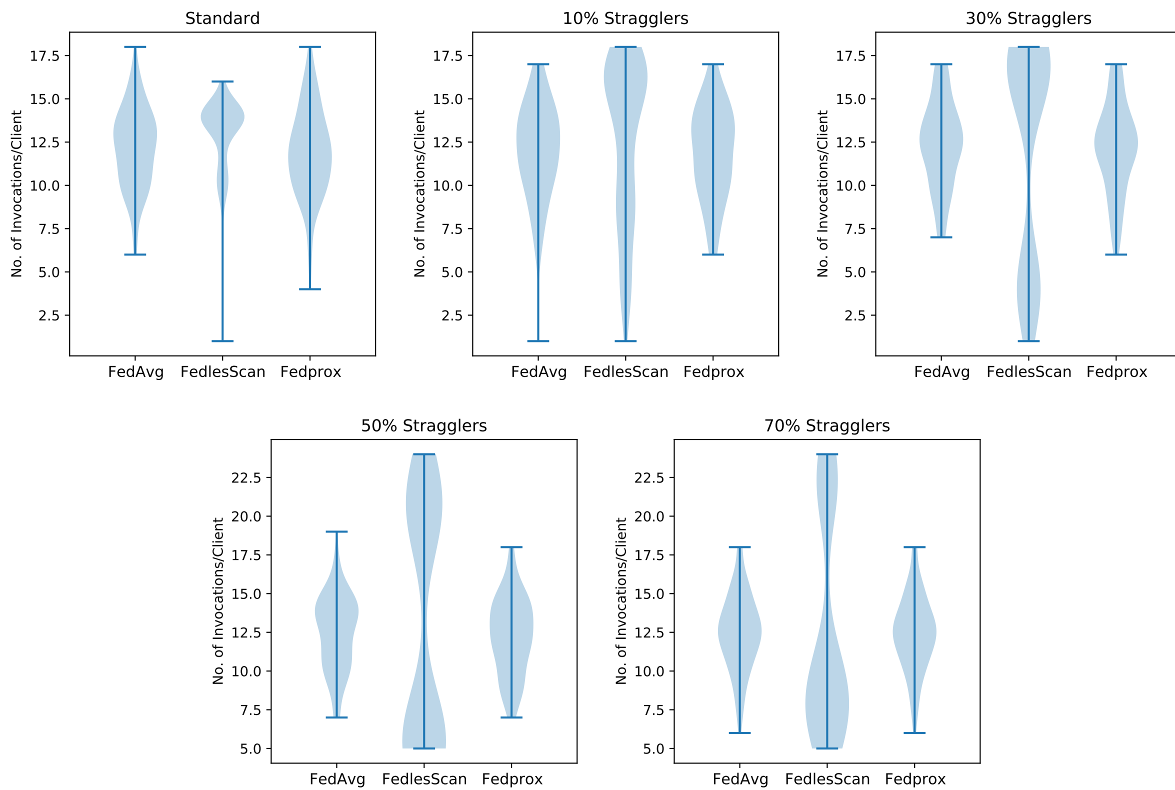


Figure 5.6: Distribution of client's invocation frequency on Shakespeare Dataset.

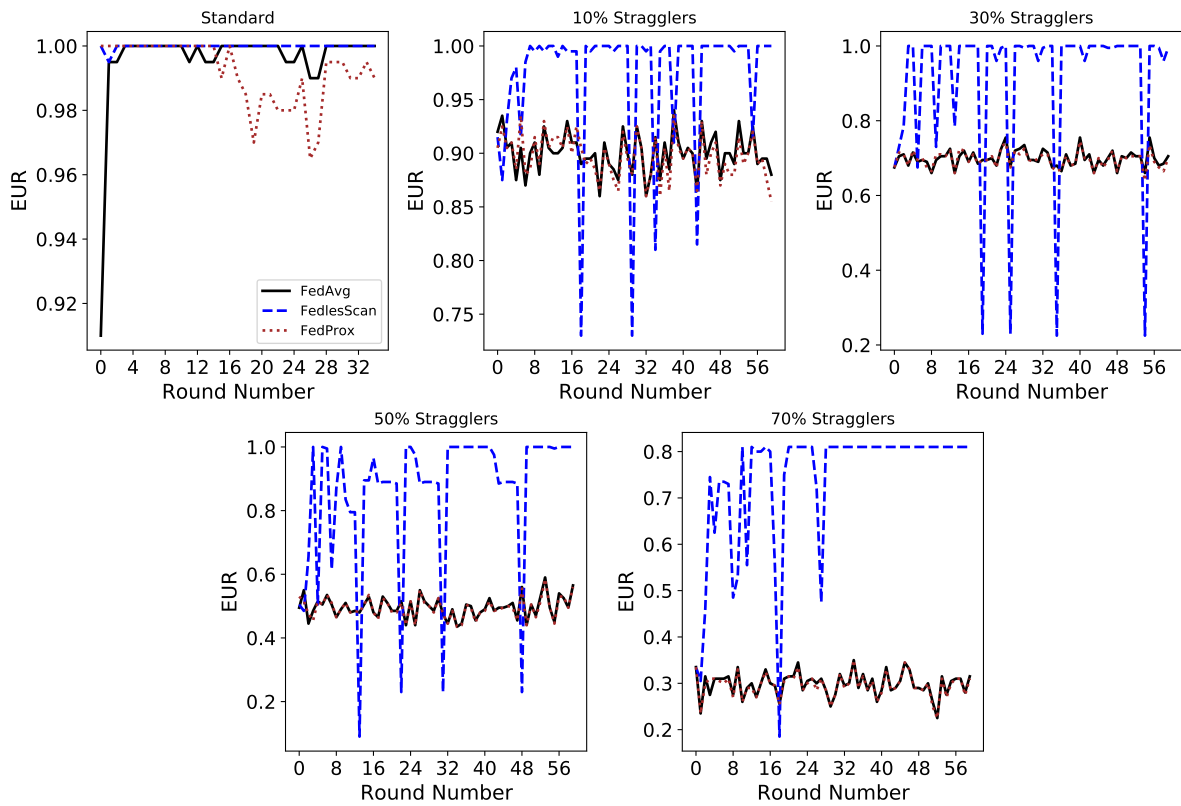


Figure 5.7: EUR comparison between FedAvg [24], FedProx [26] and FedLesScan on the Speech Commands dataset.

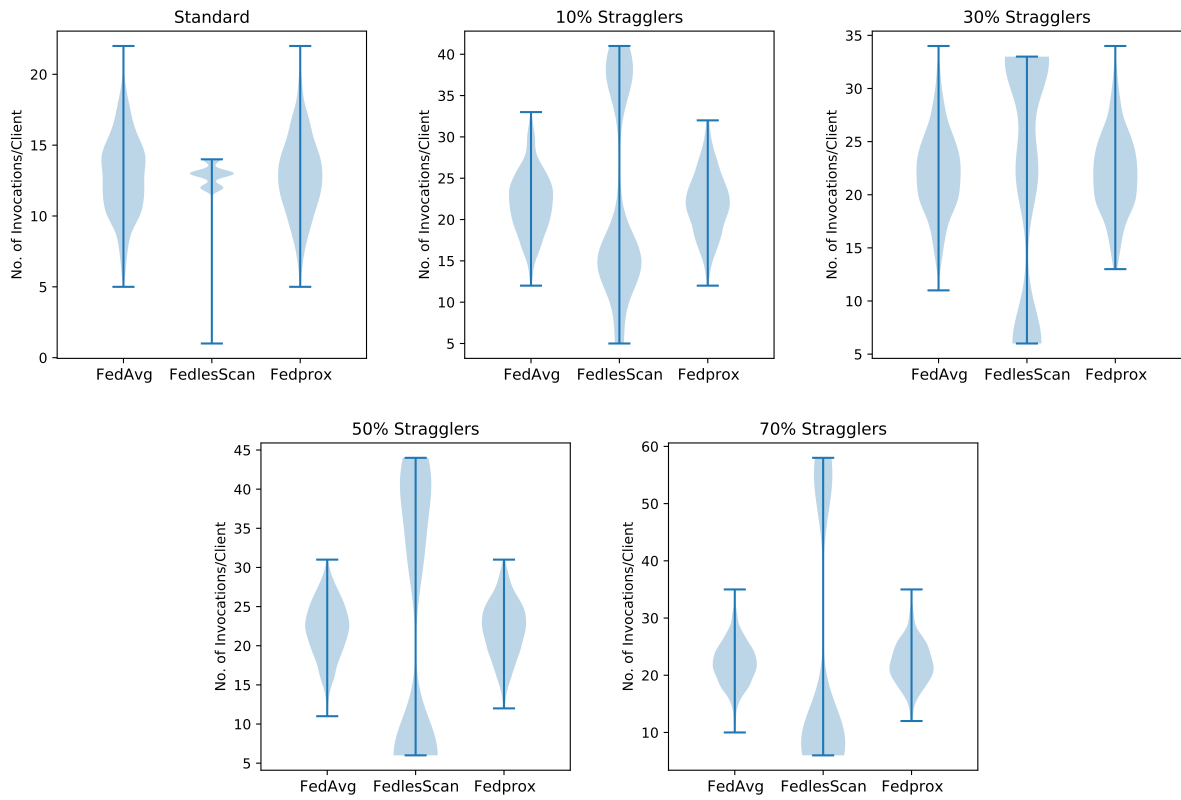


Figure 5.8: Distribution of client's invocation frequency on the Speech Commands dataset.

5.3 Accuracy and Model Performance

In the previous section, we compared the three strategies in terms of round efficiency and utilization. Although these results show the advantages of using our intelligent selection algorithm, they do not show a concrete evaluation of the system performance. This section evaluates the model performance of the three strategies in terms of loss and test accuracy.

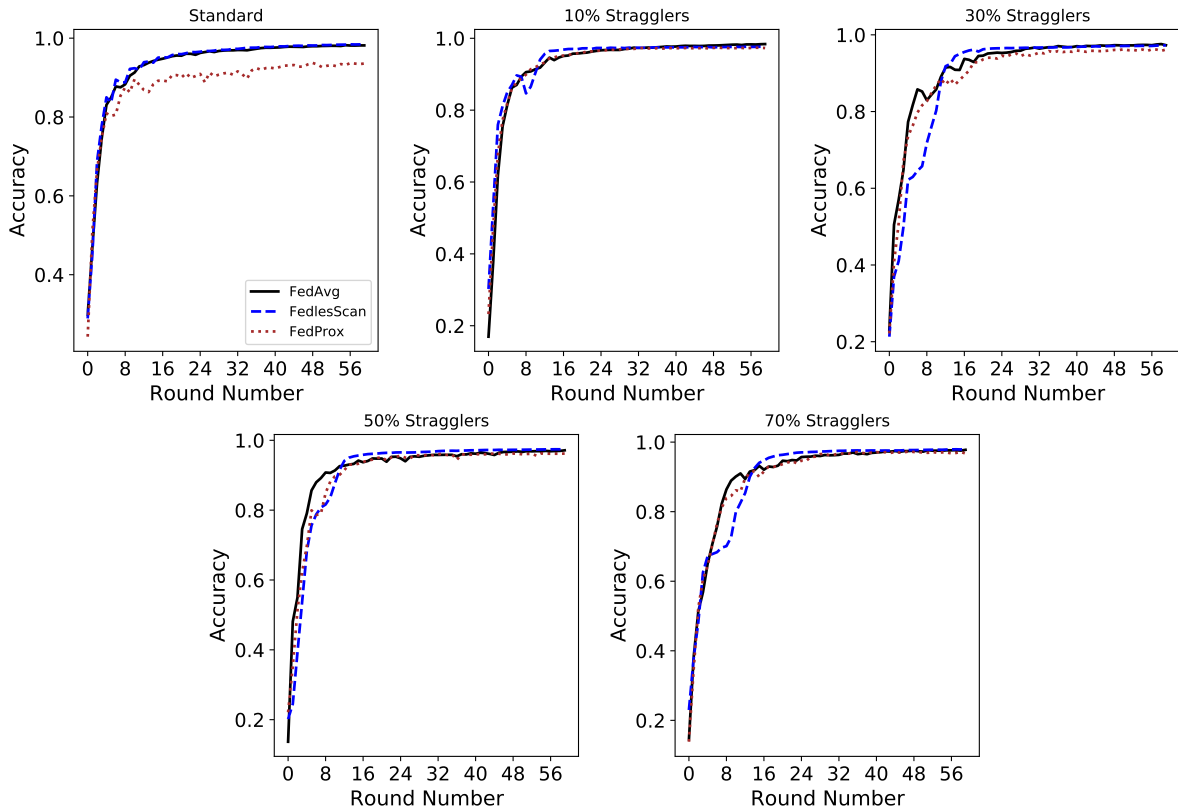


Figure 5.9: Comparison between test accuracy for FedAvg [24], FedProx [26], and FedLesScan on MNIST dataset with different straggler ratios.

For the MNIST dataset, we ran the experiments for 60 rounds. As shown in Figure 5.9, in the standard setting, FedLesScan reached about 0.4% better accuracy compared to FedAvg. FedProx had slightly lower performance because clients took slightly longer to train, which resulted in more clients missing their round, as discussed in the previous section. In the rest of the synthesized experiments, we see a pattern where FedAvg and FedProx reach higher accuracy at the beginning of the training, but FedLesScan catches up at the end, even reaching higher accuracies faster. In our experiments, the end accuracy achieved by FedLesScan is constantly higher than FedAvg and FedProx by an average of 0.2% and 0.3% respectively.

Similarly, Figure 5.10 depicts the loss across the five experiments. We notice a behavior similar to the accuracy graphs mentioned above. In the standard scenario, FedLesScan and FedAvg perform similarly during training, while FedProx performed worse. In the

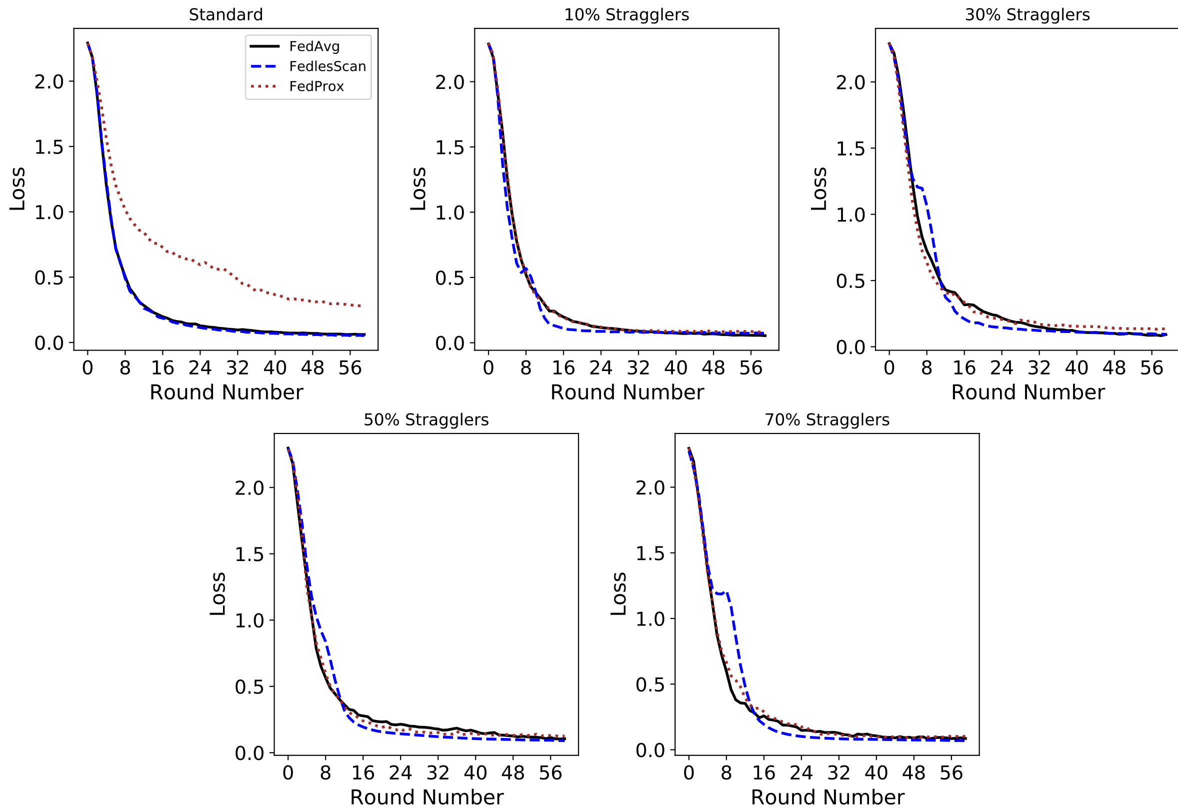


Figure 5.10: Comparison between loss for FedAvg [24], FedProx [26], and FedLesScan on MNIST dataset with different straggler ratios.

synthesized scenarios, FedLesScan can consistently achieve lower loss compared to FedAvg and FedProx.

On the Image Classification task of the FEMNIST dataset, we performed the training for 40 rounds. Figure 5.11 depicts the progress of accuracy over time comparison among the three different strategies. In the standard setting, we see the three approaches perform similarly with a slight advantage to FedLesScan in terms of convergence speed. Furthermore, FedLesScan outperformed FedAvg and FedProx in terms of the end accuracy by 2% and 0.5% respectively. In our synthesized settings, the overall training behavior for different numbers of stragglers shows the same trend with a slight advantage to FedLesScan in terms of the convergence speed. Although FedProx showed slightly faster convergence than FedLesScan in the 10% straggler setting, both reach accuracies higher than FedAvg by about 1% at the end of training. In the 30% and 50% straggler settings, FedLesScan achieved consistently higher accuracy during training, while the end accuracy was similar among the three strategies. In the 70% settings, FedLesScan had a higher convergence rate and achieved an end accuracy of 75% compared to 73% and 74% compared to FedProx and FedAvg respectively.

Investigation of the loss behavior on FEMNIST in Figure 5.12 shows a similar trend to the accuracy plots. In the standard settings, FedLesScan had faster convergence compared to

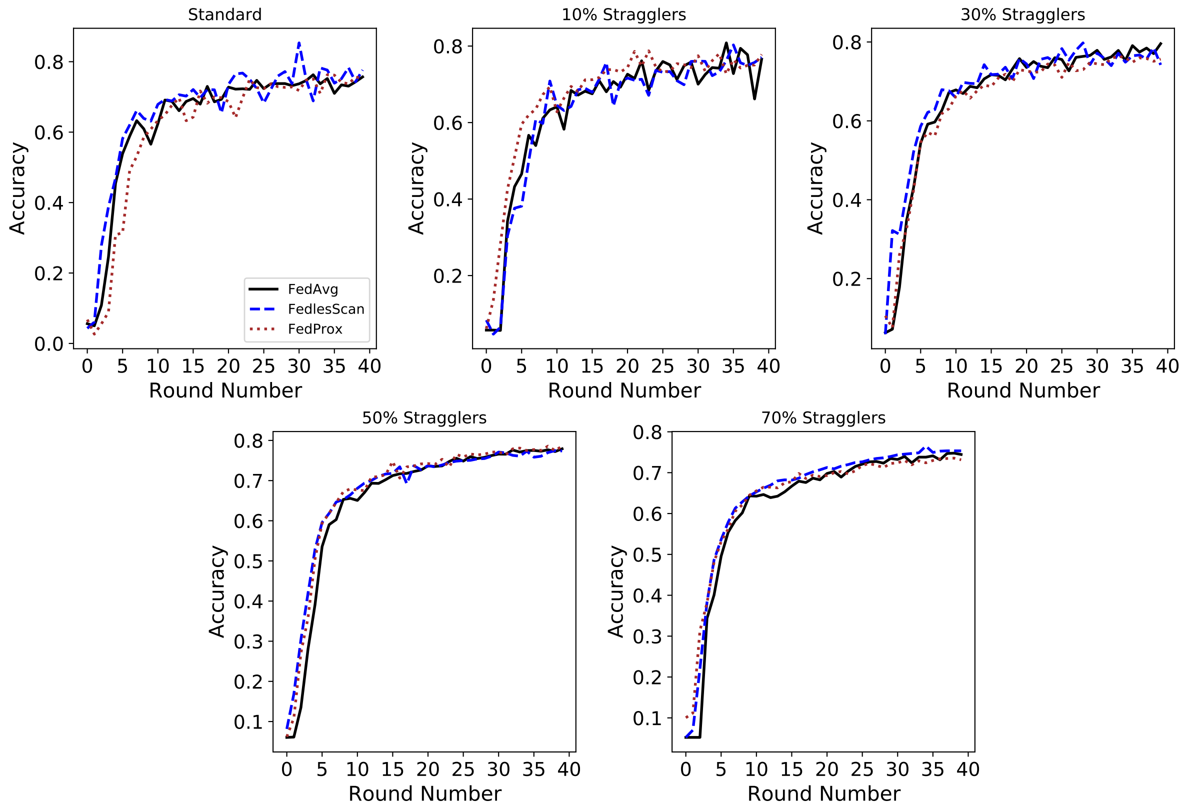


Figure 5.11: Comparison between test accuracy for FedAvg [24], FedProx [26], and FedLesScan on FEMNIST dataset with different straggler ratios.

the other two approaches while reaching a lower loss at the end. The synthesized settings show FedLesScan achieves lower loss during training compared to FedAvg and FedProx. Furthermore, we notice that the loss fluctuations between rounds decrease as the number of stragglers increases. We believe this effect happens because as the number of stragglers increases, the number of clients contributing to the global model decreases. As a result, the learning process is easier as the model represents fewer clients' contributions. We can see that FedLesScan has higher fluctuations in the first three experiments because our selection scheme involves more clients in training.

We investigate the performance of the language modeling task on the Shakespeare dataset. Compared to other experiments, the Shakespeare dataset clients take longer time to train. Therefore, we opted for fewer training rounds due to budget limitations. Although training for fewer rounds will not reach the best possible accuracy, it was enough to give us insights into the behavior of our strategy. While the reached accuracy documented in the original paper [65] was about 50% in 40 rounds, In our experiments, the three approaches reached accuracies in the range of 40% within 25 rounds. For all experiments we performed on the Shakespeare dataset, the round time was 520 seconds.

Figure 5.13 shows the performance of the three strategies in standard and synthesized

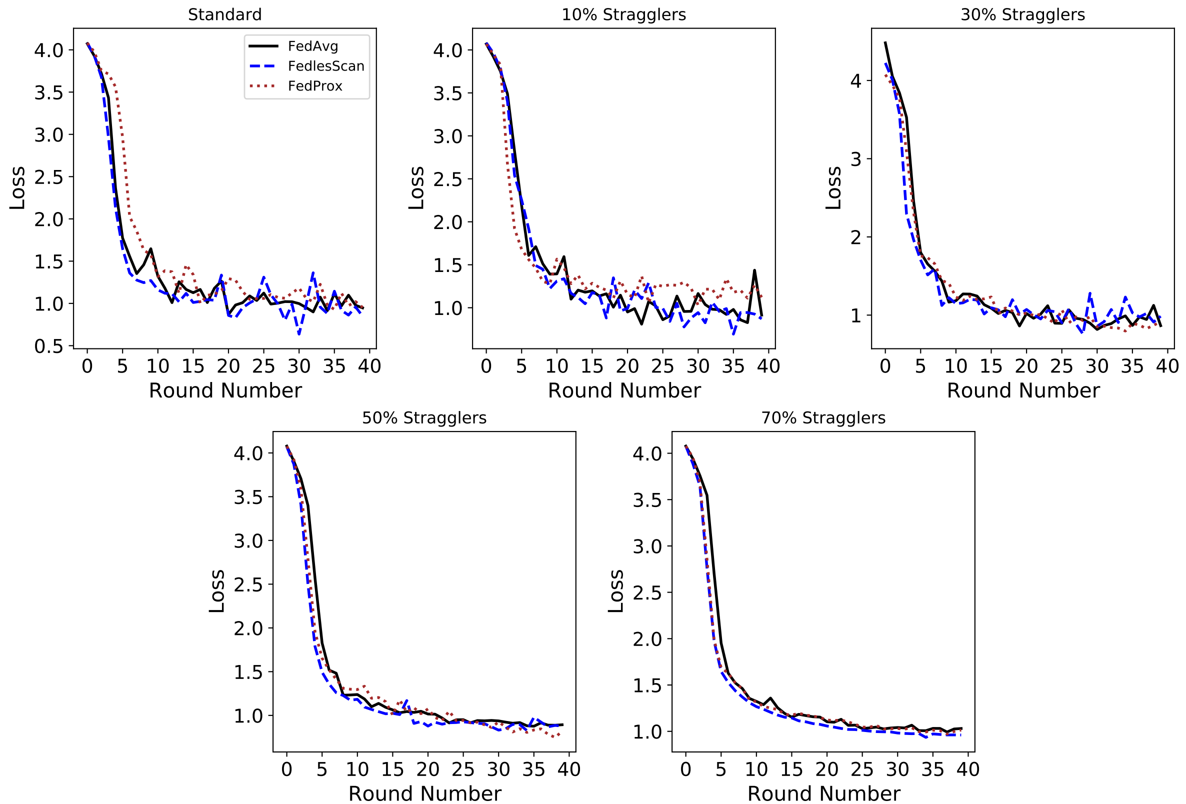


Figure 5.12: Comparison between loss for FedAvg [24], FedProx [26], and FedLesScan on FEMNIST dataset with different straggler ratios.

scenarios. In the first plot, FedAvg outperformed both FedProx and FedLesScan. Furthermore, FedLesScan had the worst performance of the three approaches, reaching about 39% accuracy compared to 40% for FedProx and 43% for FedAvg. An explanation for the performance difference is that the way the selection works for FedLess does not provide the best results on the Shakespeare dataset. In our experiment setup, the functions have similar computation power. Therefore, data heterogeneity among clients plays an essential role in dictating the running time of each client. In other words, faster clients tend to have less training data. As discussed in Section 4.2.2, FedLesScan prioritizes fast clients at the beginning of the training process by starting with faster clusters first. Selecting faster clients first means that the algorithm trains on a smaller part of the dataset at the beginning of the training than the other strategies. As a result, FedLesScan typically shows slower convergence at the beginning. As the training progresses, the strategy gradually involves the rest of the clients. Therefore we see the gap between FedAvg and FedLesScan decrease as the number of rounds increases. We further analyzed the round duration in this particular experiment to verify our hypothesis. We noticed that FedLesScan takes a shorter time to finish a training round. Table 5.3 shows a comparison between the round duration for the three strategies. We notice that the average round duration for FedLesScan is lower by almost 2 minutes compared to the

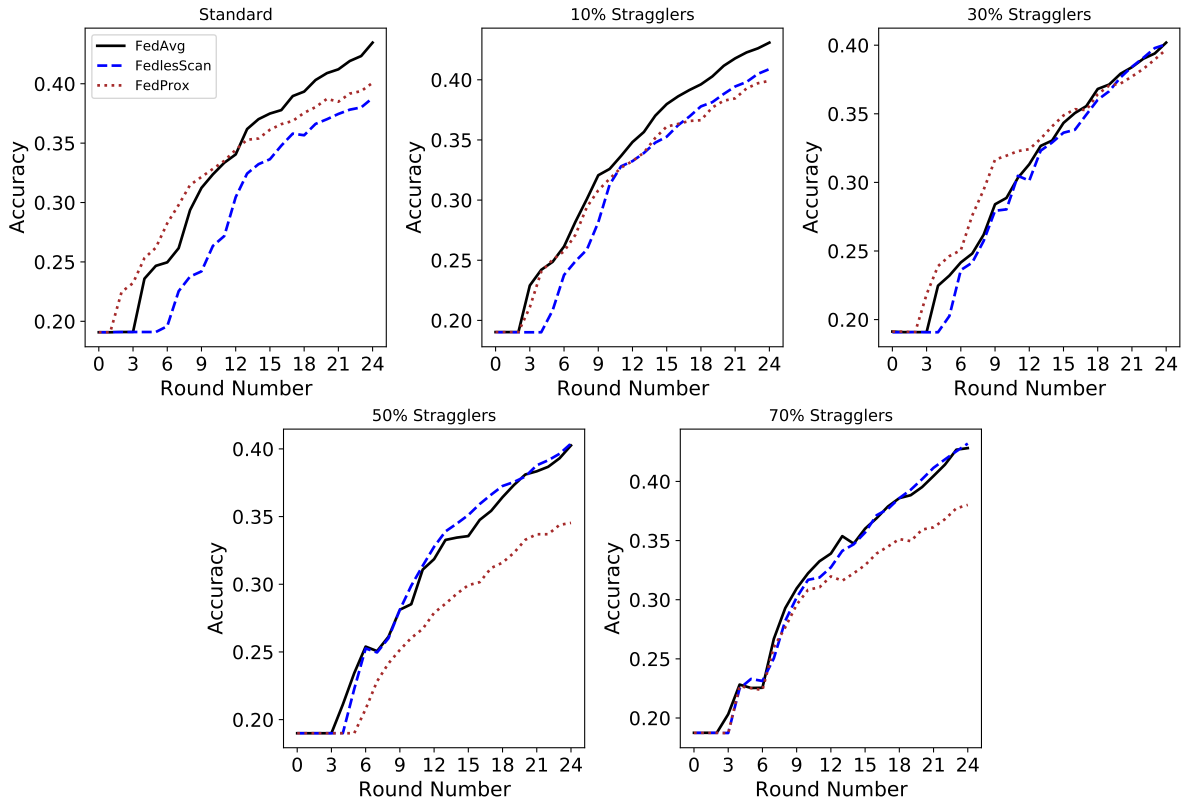


Figure 5.13: Comparison between test accuracy for FedAvg [24], FedProx [26], and FedLesScan on Shakespeare dataset with different straggler ratios.

other two approaches. This result shows the effect of grouping clients with similar training time together on the round duration. Based on these results, we argue that FedLesScan can afford to run more rounds and finish in a comparable time.

The rest of the plots in Figure 5.13 show the performance in our synthesized scenarios. We notice that the performance deficiency to FedAvg and FedProx decreases as the number of stragglers increases in the system. In scenarios with 30% and 50% stragglers, FedLesScan matched FedAvg in terms of convergence rate and end accuracy by reaching 40%. In the scenario with 70% stragglers, FedLesScan outperformed FedAvg and FedProx in terms of the end accuracy by 1% and 5% respectively. FedProx struggles to match the same performance level of the other approaches in the situations with 50% and 70% of the system clients.

In Figure 5.14, we examine the loss during training. The first plot confirms our accuracy analysis, with FedLesScan being the slowest to converge. Furthermore, FedAvg had the lowest loss during training. In the rest of the synthesized scenarios, we see the performance of FedLesScan catch up to the other approaches, even surpassing them in the straggler-heavy situations. Although the behavior of FedProx in scenarios with 50% and 70% stragglers was worse than the other two approaches, the algorithms can be adjusted to behave similarly to FedAvg [26].

Strategy	Round Time (sec)			Time to Reach 39% Accuracy (Hr)
	Min	Max	Mean	
FedLesScan	143	550	447 ± 146	3.1
FedAvg	550	562	558.5687 ± 2.344	2.8
FedProx	549	560	554.8362 ± 3.97	3.2

Table 5.3: Analysis of round duration on the Shakespeare dataset for normal behavior with no straggler simulation. Furthermore, we show the time taken by the three strategies to reach the same accuracy.

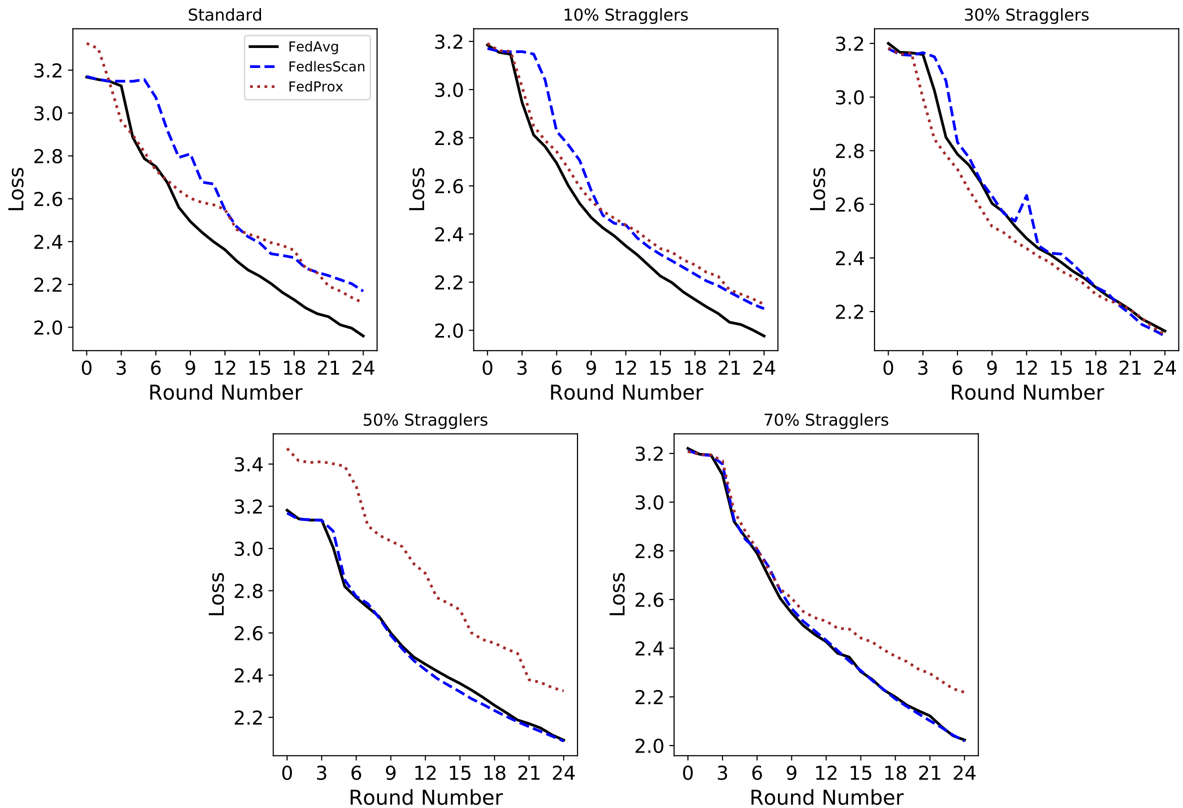


Figure 5.14: Comparison between loss for FedAvg, FedProx, and FedLesScan on the Shakespeare dataset with different straggler ratios.

We further investigate the speech recognition task's performance of the Speech Commands dataset from the FedScale benchmark. Figure 5.15 depicts the accuracy over time comparison for the three strategies in both standard and synthesized scenarios. In the standard scenario, we ran the three strategies for 35 rounds. We see FedLesScan outperform both FedAvg and FedProx in terms of convergence rate and accuracy at the end of training. FedLesScan reached an accuracy of 79.4% compared to 76.6% for FedAvg and 77.4% for FedProx. Furthermore, FedLesScan showed faster convergence by reaching an accuracy of 70% in 19 rounds compared

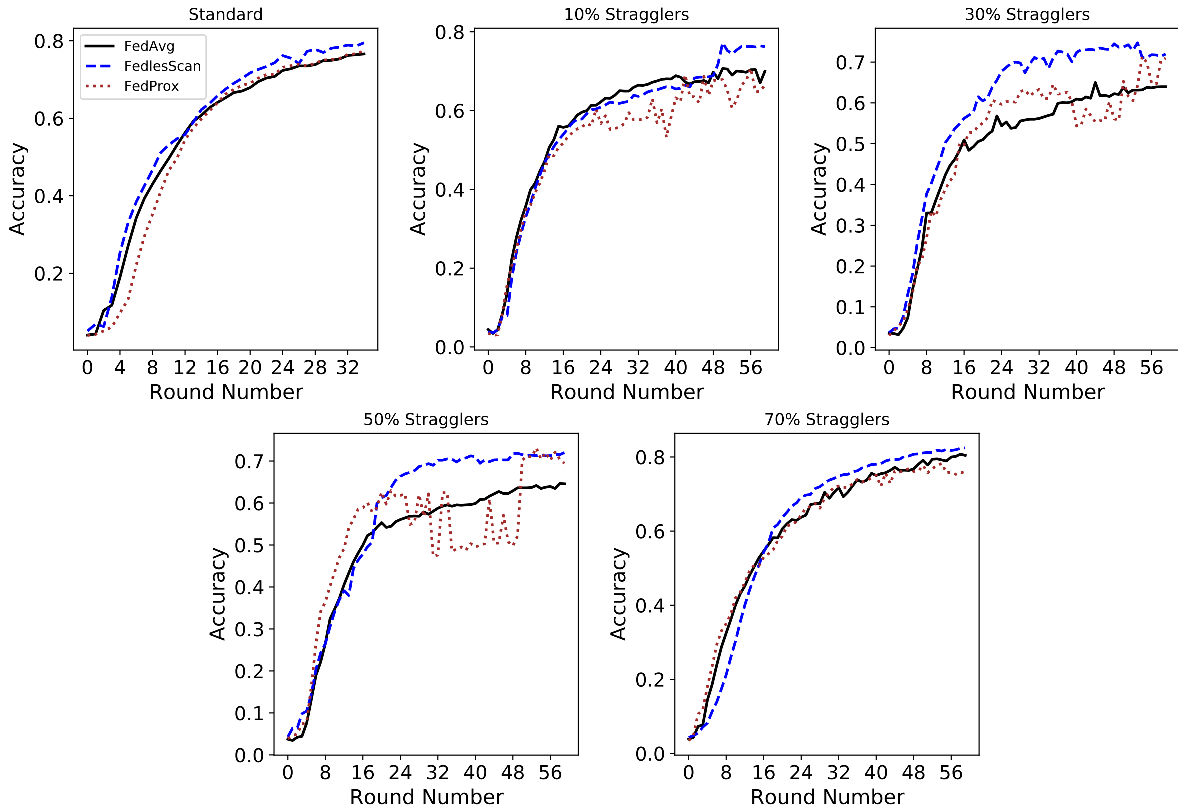


Figure 5.15: Comparison between test accuracy for FedAvg, FedProx, and FedLesScan on the Speech Commands dataset with different straggler ratios.

to 21 and 22 for FedProx and FedAvg respectively.

For the synthesized experiments, we ran the experiments for 60 rounds to see the long-term behavior of the three approaches on a real world-dataset. For the second plot, with 10% stragglers, FedLesScan and FedAvg had a similar convergence rate while FedProx was slightly behind. Additionally, FedLesScan reached an accuracy of 76%, which is a 6% and 10% increase over FedAvg and FedProx respectively. The third plot with the 30% stragglers shows FedLesScan consistently besting FedAvg by about 8% towards the end of the training while outperforming FedProx by a smaller margin of 1%. Although FedProx suffered a slight decline in accuracy towards the end of the training, it managed to outperform FedAvg at the end. The same trend continues for the rest of the synthesized experiments. I the 50% stragglers scenario, FedLesScan outperformed FedAvg and FedProx by 7% and 3% respectively. In the last experiment, with 70% stragglers, FedAvg performed better compared to its previous results outperforming FedProx by 1%. Nevertheless, the end accuracy of FedLesScan was higher than FedAvg by 3%.

We further examine the loss for each experiment. Figure 5.16 demonstrates the performance gains when using FedLesScan. We also notice that FedProx suffers more fluctuations during training compared to the other two approaches. Notice that the three approaches reach lower

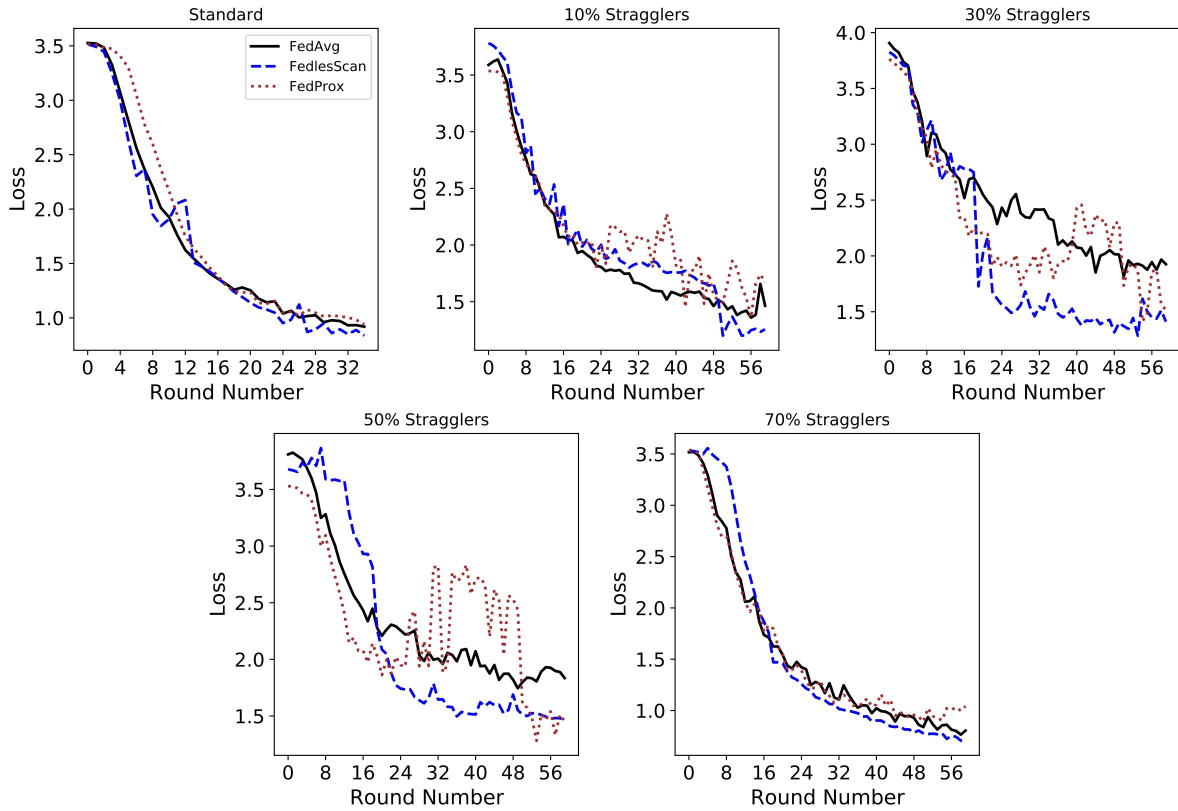


Figure 5.16: Comparison between loss during training for FedAvg, FedProx, and FedLesScan on the Speech Commands dataset with different straggler ratios.

loss and better accuracy with 70% stragglers in the system compared to 50%. We believe that this phenomenon shows that clients have a different effect on model performance. For example, some clients might hold more information in their training data; therefore, they have a higher impact on accuracy when they face issues during training compared to other clients.

5.4 Time and Cost Analysis

Although accuracy is an essential factor, it does not inform us about the efficiency of the system or the strategy used. Furthermore, the accuracy results in the previous section were expressed in terms of the number of rounds. Fast convergence in terms of the number of rounds does not provide a complete picture of the system efficiency. This section provides a collective analysis of all experiments in terms of duration and cost. Time computation was done based on the maximum round timeout. In the three strategies, the round time is dictated by the slowest client invoked. Therefore, the round time is determined by either the response of the slowest client or a predetermined timeout. Our calculations compute the

total experiment time by aggregating the round time during training. We further simulate real-world behavior by assuming that stragglers in the synthesized scenarios do not respond, requiring the controller to wait until the round timeout expires.

Dataset	Strategy	Experiment Time (mins)				
		Standard	**10%**	**30%**	**50%**	**70%**
MNIST	FedAvg	39.6	40.2	40	40	40
	FedProx	40	40.2	40.0	40	40
	FedLesScan	23.7	28.6	27.2	40	40
FEMNIST	FedAvg	75.5	86.6	86.6	86.6	86.6
	FedProx	112	88	88	87	87
	FedLesScan	70.9	75.6	82.8	86.8	86.6
Shakespeare	FedAvg	216.9	216.9	216.9	216.9	216.9
	FedProx	216.9	216.9	216.9	216.9	216.9
	FedLesScan	185.5	215	205	216	216
Speech Command	FedAvg	20	40	40	40	40
	FedProx	21.5	40	40	40	40
	FedLesScan	15	31	28	33	40

Table 5.4: Experiment duration comparison between FedAvg, FedProx, and FedLesScan across all datasets.

Table 5.4 shows the total aggregated time per experiment. In the standard scenario, the duration of the FedLesScan is significantly shorter than FedAvg and FedProx across all datasets. On the MNIST dataset, FedLesScan had a 40% lower duration than both strategies. On the FEMNIST dataset, our approach reached higher-end accuracy in 7% less time compared to FedAvg and 40% less time compared to FedProx. On the Shakespeare dataset, FedAvg and FedProx had 15% longer duration than FedLesScan. On the Speech Commands dataset, FedLesScan had a 25% and 30% improvement in experiment duration compared to FedAvg and FedProx respectively.

In the synthesized scenarios, we see the effect of stragglers on increasing round duration. In the scenarios with 10% and 30% straggler ratios, we notice that FedLesScan maintains a lower duration across all experiments. When the percentage of stragglers surpasses a certain value, they must be invoked in almost all training rounds to meet the minimum number of clients per round. For instance, if a system has 100 clients and 50 are called every round, at least one straggler must be called every round if the straggler ratio exceeds 50%; because the system contains less than 50 reliable clients, it has to choose from stragglers to complete the required number of clients per round. We notice this effect on FedLesScan as the ratio of stragglers goes beyond a certain level. In the 50% scenarios, we see a similar training time for all strategies except the Speech Commands dataset, where FedLesScan had an 18% lower experiment duration. We justify this advantage by estimating the number of stragglers in the Speech experiment. The experiment was performed using 542 clients. At a 50% straggler ratio, the number of reliable clients is about 272. Our approach is not bound to choose from

stragglers each round because the number of participants per round is 200 (<272). On the other hand, at a 70% straggler ratio, the system contains 162 reliable clients. Consequently, Stragglers must be selected every round to reach the minimum number of clients (200). We see this in the 70% scenarios, where all approaches have similar experiment times across all datasets.

To analyze the cost of the experiments, we had to estimate the cost of stragglers since stragglers can either miss their round or crash. Their running cost still factors in the experiment cost. The three approaches determine the round time by the slowest client participating in the current round or a specific timeout. In the worst-case scenario, stragglers can increase costs by wasting resources doing computations on wasted contributions. Therefore, we estimate the cost of stragglers as the cost of running the functions for the round duration. As discussed in section 5.1.1, we use the computation model [80] provided by Google Cloud to calculate the expected client cost.

Dataset	**Strategy**	**Experiment Cost ($)**				
		Baseline	**10%**	**30%**	**50%**	**70%**
MNIST	FedAvg	2.9	3.9	6	8	10.4
	FedProx	5.5	6.4	7.6	9.21	11.03
	FedLesScan	2.7	3.8	4	6	9.2
FEMNIST	FedAvg	13.5	16.1	17.87	20.54	24.7
	FedProx	16.7	17.3	19.4	22.42	25.8
	FedLesScan	13.2	14.58	14.4	14.81	20.6
Shakespeare	FedAvg	5.4	6.6	9.2	12.5	15.4
	FedProx	5.12	6.72	9	12.2	15.4
	FedLesScan	5.33	5.5	6.75	8.46	12
Speech Command	FedAvg	1.98	3.9	6.4	8.3	10.5
	FedProx	2.39	4.6	6.77	8.7	10.8
	FedLesScan	1.73	2.7	3.68	4.2	5.5

Table 5.5: Cost analysis of the three approaches in real-world and synthesized scenarios across all datasets.

Table 5.5 shows the cost of each experiment grouped by strategy and dataset. Investigating experiment cost for MNIST dataset shows that FedLesScan has a 9% lower cost than FedAvg and 40% lower cost compared to FedProx in the standard scenario. We notice that the FedLesScan efficiency becomes more visible as the number of stragglers in the system increases. In the synthesized scenarios, FedLesScan maintains the cost advantage over FedAvg while FedProx had the highest cost.

For the FEMNIST dataset, FedLesScan had the lowest overall experiment cost in all scenarios. Furthermore, FedProx had the highest overall cost. In the standard scenario, FedLesScan was more cost-efficient compared to FedAvg and FedProx by 2.2% and 20% respectively. In the synthesized scenarios, the average cost across all scenarios for our strategy was 18% and 23% lower compared to FedAvg and FedProx respectively.

For the Shakespeare dataset, the same trend continues where FedLesScan had a lower cost across all experiments. The cost gap among the three strategies was not significant in the standard scenario. Nevertheless, FedProx performed better than FedLesScan and FedAvg by about 4% and 6% respectively. In the synthesized scenarios, FedLesScan achieve an average experiment cost which was 20% lower compared to FedProx and FedAvg. Notice that we ran the experiment for a few rounds with fewer clients each round. As a result, the small scale of the experiment might underestimate the cost gains for FedLesScan.

We further investigate the cost performance of our large-scale experiment using the Speech Commands dataset. The Cost analysis results follow a trend similar to MNIST and FEMNIST datasets. FedLesScan had the lowest cost compared to the other approaches across the five experiments. In standard settings, FedLesScan was 12% more cost-efficient compared to FedAvg. Moreover, FedLesScan had a 27% lower cost than FedProx. In the synthesized settings, The average cost gains across experiments increase to about 30% when compared to FedAvg and 40% when compared to FedProx.

5.5 IaaS vs FaaS Federated Learning

In the original [21] paper, they compared FedLess to the IaaS platform *Flower* [87] in terms of training time and cost. Although their results showed that FedLess could reach had overall lower cost because of the scale to zero advantages of FaaS platform, we wanted to examine the behavior of the clients in a real training scenario and compare that behavior to clients in Flower. We chose to run the FedAvg algorithm on MNIST for this experiment.

We performed two experiments. The first included 100 clients, 20 selected each round for 30 rounds. We increased the number to 200 in the second experiment with 50 clients per round without changing other experiment parameters. The idea of using a small portion of the clients each round is to show the advantage of using FaaS in training FL models. Besides, in a large-scale scenario with millions of clients, a small subset of clients participate each round [24, 88]. We compared FedLess against Flower [87] in terms of the system's memory utilization.

We used the statistics for clients' invocation time, duration, and average memory utilization to calculate the memory utilization for the whole system. We utilized a sliding window approach with one-second increments to obtain the number of active client functions at a specific time during the training session. Each client function was allowed a five-second cooldown window (after execution) before being considered inactive. We obtained the system's memory utilization by multiplying the client's average memory by the number of active clients at a given instance.

For Flower, each client runs in a container, and the central server uses high performance Remote Procedure Call *gRPC* to communicate with the clients. We obtained the memory utilization during the experiment by aggregating the statistics of the running flower containers.

Figure 5.17 shows the difference in median memory utilization between FedLess and Flower. In both scenarios, FedLess has 50% less memory utilization. This demonstrates the advantages of a run-on-demand model for FaaS compared to IaaS clients. In addition, we

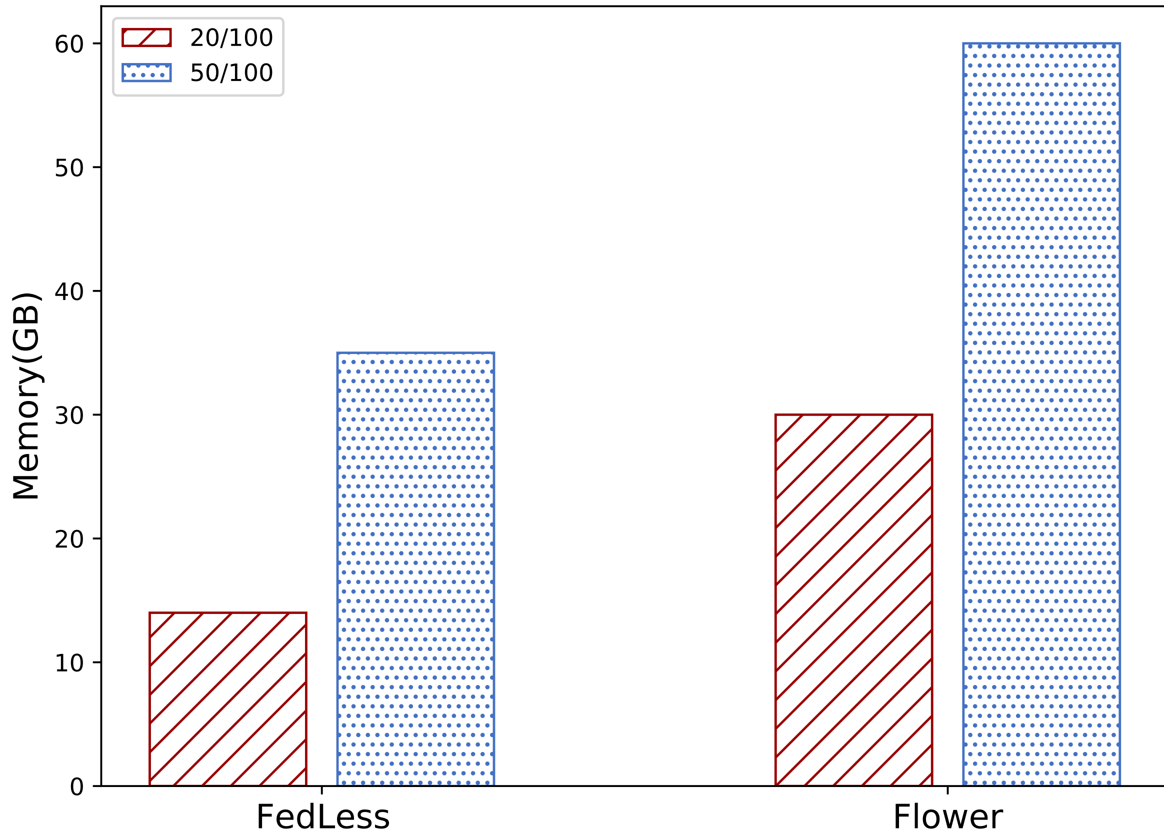


Figure 5.17: Median memory utilization for FedLess and Flower. The first experiment shows the result for 100 clients 20 of which are selected per round. The second experiment shows 200 clients with 50 clients selected per round.

noticed that the memory usage of a client function was larger than the memory usage of a client using Flower. This is due to the additional infrastructure cost that the function requires. For instance, the function must listen to requests and parse request objects, resulting in higher memory utilization. We argue that the infrastructure cost becomes less noticeable for higher client workloads as the additional memory becomes insignificant to the memory required for training. Furthermore, Edge devices tend to have lighter components. Their memory footprint is usually smaller than public or self-hosted FaaS platforms; hence the infrastructure load on these devices will not be as significant.

5.6 Discussion

FedLesScan has a few advantages that set it apart from other approaches. While other strategies may rely on a multi-threading or computationally capable central server, FedLesScan does not require substantial computational power to run or manage the FL clients. FedLesScan

provides an efficient way to use serverless functions in FL. Although synchronous approaches are reliable, even a small number of stragglers might significantly impact their training time. Asynchronous techniques, on the other hand, may tolerate client heterogeneity and the effect of stragglers. However, the high communication cost or persistent communication link between the central server and the clients prevents the full benefit of serverless architecture from being realized. The semi-asynchronous nature of FedLesScan attempts to combine the benefits of both approaches by providing best-effort synchronization guarantees while tolerating failures and long delays.

We did an exhaustive evaluation of our new approach on four different datasets. Furthermore, we compared FedLesScan to two popular FL training approaches FedAvg and FedAvg, in terms of round utilization, accuracy, training time and cost. One of the aspects we wanted to evaluate is how our approach can dynamically adapt to systems with different ratios of stragglers. We evaluated the performance in standard scenarios for each dataset without changing the clients' behavior. Moreover, we performed controlled experiments by simulating a system with different percentages of stragglers. The evaluation findings show how the strategy behaves in different scenarios. Furthermore, they show the effect of stragglers on the system performance and our approach to mitigate this effect.

In Standard scenarios, FedLesScan showed better accuracy, shorter experiment duration, and lower cost. FedLesScan outperformed both FedAvg and FedProx on all datasets with the exception of Shakespeare dataset. Our analysis of the Shakespeare dataset shows that some of the clients might contribute to the model accuracy more than others, especially clients with longer training time. Figure 5.13 confirms that hypothesis; FedLesScan has lower accuracy in the early rounds because we prioritize fast clients early in training. Moreover, the training session was slightly short, with only 25 rounds, due to budget constraints. We argue that in a more realistic scenario with more rounds, the difference in accuracy will decrease.

In our simulated scenarios, FedLesScan delivered better accuracy in a shorter time and lower experiment cost on MNIST, FEMNIST, and Speech Commands datasets. On the Shakespeare dataset, FedLesScan outperformed FedAvg and FedProx in scenarios with 30%, 50%, and 70% stragglers. From our results, we can define a proper usecase for FedLesScan. Although FedLesScan was designed as a straggler-resilient variation of FedAvg on serverless FL, the strategy does not try and reduce or eliminate stragglers from the system. That said, FedLesScan works best in long training sessions because there are more data collected about the clients' behavior that will help group clients better. In case of repeated failures, FedLesScan should perform well due to its three-tier classification system. Furthermore, we notice that in the case of small-scale experiments with a small number of clients per round, strategies based on random client selection might perform better than FedLesScan. We notice this effect in the Shakespeare experiment as the benefits of grouping clients with similar behavior appear in large-scale experiments.

The overall evaluation showed that FedLesScan achieves an average of 2% better accuracy in 8% shorter round time and up to 20% lower cost compared to FedAvg and FedProx.

We also compared the resource utilization between FedLess (a FaaS-based FL platform) and Flower (an IaaS FL platform). The results we showed in Section 5.5 demonstrates the

resource-efficiency of a scale-to-zero model in FL compared to IaaS platforms. Alongside the results provided in [21], we argue that FaaS environments can be an efficient alternative in terms of resources and cost to IaaS-based FL systems.

6 Conclusion and Future Work

In our work, we focused on two main contributions. Our first contribution consists of numerous enhancements to the FedLess platform. We made some architectural modifications to enable FedLess to run different training strategies and facilitate the deployment of the platform. Furthermore, we integrated a mocking system that can simulate the behavior of all system components for easier development of FL models. Few improvements can be integrated intoFedLess. Currently, the platform is implemented using the Tensorflow library, which limits the platform's capability to run models or training based on other libraries such as PyTorch [83]. We faced this limitation when we tried using the FedScale benchmark to test our strategy because the benchmark was entirely based on Pytorch. One improvement to address this issue is to move towards library agnostic FL by abstracting the platform implementation from library dependencies. Furthermore, the current architecture of the platform uses MongoDB [60] as the parameter server and client database. Although MongoDB is easy to use and supports horizontal scaling, the platform is locked in to use this specific database. Another limitation is the usage of a single aggregation function. In large-scale systems with thousands of clients, the aggregation needs significant memory and computation power. To address this issue, we can explore multi-function aggregation approaches such as *λ-FL* [59].

Our second contribution presented FedLesScan, a clustering-based training strategy designed for FL on FaaS platforms. FedLesScan can adapt to system behavior dynamically. We proposed an intelligent client selection algorithm based on clustering clients with similar behavior. The selection strategy can dynamically adapt to the client's performance to provide better system utilization. Furthermore, we integrated a staleness-aware aggregation mechanism to mitigate wasted contributions for slow clients.

We did an extensive evaluation of FedLesScan comparing its performance to two popular strategies on multiple datasets. We analyzed its behavior in different system states with different straggler ratios. Our experiments showed that FedLesScan achieves overall better results in terms of accuracy, time, and cost by better utilizing participating clients. Although the strategy produced good results, some enhancements can be made to enhance its performance. First, FedLesScan uses a simple staleness-aware aggregation module. In some cases, stale updates affect model accuracy. Therefore, a more sophisticated aggregation scheme to aggregate valuable updates and discard unnecessary ones might provide better results. An example of this aggregation algorithm is the one presented by [64].

In conclusion, the fast development of serverless technologies demonstrates a new direction for FL. We hope that our efforts to tackle the problem of stragglers in serverless FL motivate further research in this area.

List of Figures

1.1 Model accuracy (Left) and average FL training round duration (Right) for different ratios of stragglers in the serverless FL system [21] for the Speech Commands dataset [23] using the `FedAvg` algorithm [24]. 3

2.1 FL system architecture and training flow 5

3.1 FedLess System Architecture [21]. 14
3.2 FedLesstraining workflow [21]. 15

4.1 Modified Architecture of the FedLess platform. The highlighted components shows the modified components and additions to the system. 23
4.2 The training workflow of a typical training round in FedLess 24

5.1 EUR comparison between FedAvg [24], FedProx [26] and FedLesScan on the MNIST dataset. 36
5.2 Distribution of client’s invocation frequency on the MNIST dataset. 38
5.3 EUR comparison between FedAvg [24], FedProx [26] and FedLesScan on the FEMNIST dataset. 39
5.4 Distribution of client’s invocation frequency for the FEMNIST dataset. 40
5.5 EUR comparison between FedAvg [24], FedProx [26] and FedLesScan on Shakespeare dataset for language modeling task. 41
5.6 Distribution of client’s invocation frequency on Shakespeare Dataset. 42
5.7 EUR comparison between FedAvg [24], FedProx [26] and FedLesScan on the Speech Commands dataset. 43
5.8 Distribution of client’s invocation frequency on the Speech Commands dataset. 44
5.9 Comparison between test accuracy for FedAvg [24], FedProx [26], and FedLesScan on MNIST dataset with different straggler ratios. 45
5.10 Comparison between loss for FedAvg [24], FedProx [26], and FedLesScan on MNIST dataset with different straggler ratios. 46
5.11 Comparison between test accuracy for FedAvg [24], FedProx [26], and FedLesScan on FEMNIST dataset with different straggler ratios. 47
5.12 Comparison between loss for FedAvg [24], FedProx [26], and FedLesScan on FEMNIST dataset with different straggler ratios. 48
5.13 Comparison between test accuracy for FedAvg [24], FedProx [26], and FedLesScan on Shakespeare dataset with different straggler ratios. 49
5.14 Comparison between loss for FedAvg, FedProx, and FedLesScan on the Shakespeare dataset with different straggler ratios. 50

5.15 Comparison between test accuracy for FedAvg, FedProx, and FedLesScan on the Speech Commands dataset with different straggler ratios. 51
5.16 Comparison between loss during training for FedAvg, FedProx, and FedLesScan on the Speech Commands dataset with different straggler ratios. 52
5.17 Median memory utilization for FedLess and Flower. The first experiment shows the result for 100 clients 20 of which are selected per round. The second experiment shows 200 clients with 50 clients selected per round. 56

List of Tables

5.1 Model hyperparameters used for each dataset. 35
5.2 Total number of clients and clients per round used for each dataset. 35
5.3 Analysis of round duration on the Shakespeare dataset for normal behavior with no straggler simulation. Furthermore, we show the time taken by the three strategies to reach the same accuracy. 50
5.4 Experiment duration comparison between FedAvg, FedProx, and FedLesScan across all datasets. 53
5.5 Cost analysis of the three approaches in real-world and synthesized scenarios across all datasets. 54

Acronyms

BSP bulk-synchronous processing. 12

DBSCAN Density Based Spatial Clustering of Applications with Noise. 9, 10, 29

EMA exponential moving average. 29

EUR Effective Update Ratio. 32, 36–41, 43, 60

FaaS Function-as-a-Service. iv, 1–4, 11–15, 17, 20–22, 33, 35, 55–59

FedAvg FedAverage. 3, 4, 15–18, 20, 24, 32, 34, 36–39, 41, 43, 45–55, 57, 60–62

FedLess Serverless Federated learning Platform. 2–4, 11, 13–15, 22–27, 31, 32, 34, 35, 48, 55–57, 59–61

FedLesScan *FedLess Clustering based algorithm*. iv, 3, 4, 22, 24, 25, 31, 32, 35–41, 43, 45–57, 59–62

FedProx FedProx. 3, 4, 17, 18, 32, 36–41, 43, 45–55, 57, 60–62

FL Federated Learning. iv, 1–7, 10, 11, 13–18, 20–22, 32, 33, 55–60

HC Hierarchical clustering. 7, 8, 10

IaaS Infrastructure-as-a-Service. 2, 4, 11–13, 55, 57, 58

IID Independent Identically Distributed. 5, 6

ML Machine Learning. iv, 1, 4, 5, 11–13

SGD Stochastic Gradient Descend. 6

Bibliography

[1] K. Siau and W. Wang. "Building trust in artificial intelligence, machine learning, and robotics". In: *Cutter business technology journal* 31.2 (2018), pp. 47–53.

[2] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer. "A survey on distributed machine learning". In: *ACM Computing Surveys (CSUR)* 53.2 (2020), pp. 1–33.

[3] P. Regulation. "Regulation (EU) 2016/679 of the European Parliament and of the Council". In: *Regulation (eu)* 679 (2016), p. 2016.

[4] L. Braubach and A. Pokahr. "Addressing challenges of distributed systems using active components". In: *Intelligent Distributed Computing V*. Springer, 2011, pp. 141–151.

[5] M. Van Steen, G. Pierre, and S. Voulgaris. "Challenges in very large distributed systems". In: *Journal of Internet Services and Applications* 3.1 (2012), pp. 59–66.

[6] Q. Yang, Y. Liu, T. Chen, and Y. Tong. "Federated machine learning: Concept and applications". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 10.2 (2019), pp. 1–19.

[7] M. Aledhari, R. Razzak, R. M. Parizi, and F. Saeed. "Federated learning: A survey on enabling technologies, protocols, and applications". In: *IEEE Access* 8 (2020), pp. 140699–140725.

[8] M. Shaheen, M. S. Farooq, T. Umer, and B.-S. Kim. "Applications of Federated Learning; Taxonomy, Challenges, and Research Trends". In: *Electronics* 11.4 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11040670. URL: https://www.mdpi.com/2079-9292/11/4/670.

[9] P. M. Mammen. "Federated learning: Opportunities and challenges". In: *arXiv preprint arXiv:2101.05428* (2021).

[10] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith. "Federated Learning: Challenges, Methods, and Future Directions". In: *IEEE Signal Processing Magazine* 37.3 (2020), pp. 50–60. DOI: 10.1109/MSP.2020.2975749.

[11] S. Allen, C. Aniszczyk, C. Arimura, 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, and C. Zhang. *CNCF Serverless Whitepaper v1.0*. 2022. URL: https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf.

[12] A. W. Services. *Serverless Computing - AWS Lambda - Amazon Web Services*. 2022. URL: https://aws.amazon.com/lambda/ (visited on 04/14/2022).

[13] G. Cloud. *Cloud Functions*. 2022. URL: https://cloud.google.com/functions (visited on 04/14/2022).

[14] M. Azure. *Azure Functions – Serverless Apps and Computing*. 2022. URL: https://azure.microsoft.com/services/functions/ (visited on 04/14/2022).

[15] I. Cloud. *IBM Cloud Functions*. 2022. URL: https://cloud.ibm.com/functions/ (visited on 04/14/2022).

[16] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. "The rise of serverless computing". In: *Communications of the ACM* 62.12 (2019), pp. 44–54.

[17] V. Ishakian, V. Muthusamy, and A. Slominski. "Serving deep learning models in a serverless platform". In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2018, pp. 257–262.

[18] M. Chadha, A. Jindal, and M. Gerndt. "Towards federated learning using faas fabric". In: *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. 2020, pp. 49–54.

[19] OpenFaasS. *OpenFaaS - Serverless Functions Made Simple*. 2019. URL: https://www.openfaas.com/ (visited on 04/14/2022).

[20] T. A. Foundation. *Apache OpenWhisk is a serverless, open source cloud platform*. 2018. URL: https://openwhisk.apache.org/ (visited on 04/14/2022).

[21] A. Grafberger, M. Chadha, A. Jindal, J. Gu, and M. Gerndt. "FedLess: Secure and Scalable Federated Learning Using Serverless Computing". In: *2021 IEEE International Conference on Big Data (Big Data)*. IEEE. 2021, pp. 164–173.

[22] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. "Deep learning with differential privacy". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 308–318.

[23] P. Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. 2018. DOI: 10.48550/ARXIV.1804.03209. URL: https://arxiv.org/abs/1804.03209.

[24] 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.

[25] F. Lai, Y. Dai, X. Zhu, H. V. Madhyastha, and M. Chowdhury. "FedScale: Benchmarking model and system performance of federated learning". In: *Proceedings of the First Workshop on Systems Challenges in Reliable and Secure Federated Learning*. 2021, pp. 1–3.

[26] A. K. Sahu, T. Li, M. Sanjabi, M. Zaheer, A. Talwalkar, and V. Smith. "On the convergence of federated optimization in heterogeneous networks". In: *arXiv preprint arXiv:1812.06127* 3 (2018), p. 3.

[27] J. Xu, B. S. Glicksberg, C. Su, P. Walker, J. Bian, and F. Wang. "Federated learning for healthcare informatics". In: *Journal of Healthcare Informatics Research* 5.1 (2021), pp. 1–19.

[28] N. Rieke, J. Hancox, W. Li, F. Milletari, H. R. Roth, S. Albarqouni, S. Bakas, M. N. Galtier, B. A. Landman, K. Maier-Hein, et al. "The future of digital health with federated learning". In: *NPJ digital medicine* 3.1 (2020), pp. 1–7.

[29] S. R. Pokhrel and J. Choi. "A decentralized federated learning approach for connected autonomous vehicles". In: *2020 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*. IEEE. 2020, pp. 1–6.

[30] S. R. Pokhrel and J. Choi. "Federated learning with blockchain for autonomous vehicles: Analysis and design challenges". In: *IEEE Transactions on Communications* 68.8 (2020), pp. 4734–4746.

[31] 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". In: *arXiv preprint arXiv:1811.03604* (2018).

[32] S. Ramaswamy, R. Mathews, K. Rao, and F. Beaufays. "Federated learning for emoji prediction in a mobile keyboard". In: *arXiv preprint arXiv:1906.04329* (2019).

[33] A. Naeem, T. Anees, R. A. Naqvi, and W.-K. Loh. "A Comprehensive Analysis of Recent Deep and Federated-Learning-Based Methodologies for Brain Tumor Diagnosis". In: *Journal of Personalized Medicine* 12.2 (2022), p. 275.

[34] L. Yi, J. Zhang, R. Zhang, J. Shi, G. Wang, and X. Liu. "SU-net: an efficient encoder-decoder model of federated learning for brain tumor segmentation". In: *International Conference on Artificial Neural Networks*. Springer. 2020, pp. 761–773.

[35] S. Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[36] X. Li, K. Huang, W. Yang, S. Wang, and Z. Zhang. "On the convergence of fedavg on non-iid data". In: *arXiv preprint arXiv:1907.02189* (2019).

[37] J. Konečn'y, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. "Federated learning: Strategies for improving communication efficiency". In: *arXiv preprint arXiv:1610.05492* (2016).

[38] W. Luping, W. Wei, and L. Bo. "CMFL: Mitigating communication overhead for federated learning". In: *2019 IEEE 39th international conference on distributed computing systems (ICDCS)*. IEEE. 2019, pp. 954–964.

[39] R. C. Geyer, T. Klein, and M. Nabi. *Differentially Private Federated Learning: A Client Level Perspective*. 2017. DOI: 10.48550/ARXIV.1712.07557. URL: https://arxiv.org/abs/1712.07557.

[40] V. Tolpegin, S. Truex, M. E. Gursoy, and L. Liu. "Data poisoning attacks against federated learning systems". In: *European Symposium on Research in Computer Security*. Springer. 2020, pp. 480–501.

[41] O. A. Abbas. "Comparisons between data clustering algorithms." In: *International Arab Journal of Information Technology (IAJIT)* 5.3 (2008).

[42] R. Xu and D. Wunsch. "Survey of clustering algorithms". In: *IEEE Transactions on neural networks* 16.3 (2005), pp. 645–678.

[43] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *kdd*. Vol. 96. 34. 1996, pp. 226–231.

[44] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. "Clustering algorithms and validity measures". In: *Proceedings Thirteenth International Conference on Scientific and Statistical Database Management. SSDBM 2001*. IEEE. 2001, pp. 3–22.

[45] P. H. Sneath. "The application of computers to taxonomy". In: *Microbiology* 17.1 (1957), pp. 201–226.

[46] P. Dawyndt, H. D. Meyer, and B. D. Baets. "The complete linkage clustering algorithm revisited". In: *Soft Computing* 9.5 (2005), pp. 385–392.

[47] J. MacQueen et al. "Some methods for classification and analysis of multivariate observations". In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 14. Oakland, CA, USA. 1967, pp. 281–297.

[48] G. H. Ball and D. J. Hall. "A clustering technique for summarizing multivariate data". In: *Behavioral science* 12.2 (1967), pp. 153–155.

[49] L. Rdusseeun and P. Kaufman. "Clustering by means of medoids". In: *Proceedings of the statistical data analysis based on the L1 norm conference, neuchatel, switzerland*. Vol. 31. 1987.

[50] M. Verma, M. Srivastava, N. Chack, A. K. Diswar, and N. Gupta. "A comparative study of various clustering algorithms in data mining". In: *International Journal of Engineering Research and Applications (IJERA)* 2.3 (2012), pp. 1379–1384.

[51] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. "When is "nearest neighbor" meaningful?" In: *International conference on database theory*. Springer. 1999, pp. 217–235.

[52] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. "A case for serverless machine learning". In: *Workshop on Systems for ML and Open Source Software at NeurIPS*. Vol. 2018. 2018.

[53] H. Lee, K. Satyam, and G. Fox. "Evaluation of production serverless computing environments". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 442–450.

[54] A. W. Services. *Cloud Object Storage-Amazon S3-Amazon Web Services*. 2022. URL: `https://aws.amazon.com/s3/` (visited on 04/14/2022).

[55] R. Ltd. *Redis*. 2022. URL: `https://redis.io/` (visited on 04/14/2022).

[56] H. Wang, D. Niu, and B. Li. "Distributed Machine Learning with a Serverless Architecture". In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 2019, pp. 1288–1296. DOI: `10.1109/INFOCOM.2019.8737391`.

[57] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[58] R. Technologies. *Messaging that just works — RabbitMQ*. 2022. URL: `https://www.rabbitmq.com/` (visited on 04/14/2022).

[59] K. Jayaram, V. Muthusamy, G. Thomas, A. Verma, and M. Purcell. "λ-FL: Serverless Aggregation For Federated Learning". In: (2022).

[60] MongoDB. *MongoDB: The Application Data Platform | MongoDB*. 2022. URL: https://www.mongodb.com/ (visited on 04/14/2022).

[61] A. W. Services. *Amazon Cognito Simple and Secure User Sign-Up, Sign-In, and Access Control*. 2022. URL: https://aws.amazon.com/cognito/ (visited on 04/14/2022).

[62] Y. Chen, Y. Ning, and H. Rangwala. "Asynchronous online federated learning for edge devices". In: *arXiv preprint arXiv:1911.02134* (2019).

[63] C. Xie, S. Koyejo, and I. Gupta. "Asynchronous federated optimization". In: *arXiv preprint arXiv:1903.03934* (2019).

[64] G. Damaskinos, R. Guerraoui, A.-M. Kermarrec, V. Nitu, R. Patra, and F. Taiani. "Fleet: Online federated learning via staleness awareness and performance prediction". In: *Proceedings of the 21st International Middleware Conference*. 2020, pp. 163–177.

[65] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečn'y, H. B. McMahan, V. Smith, and A. Talwalkar. "Leaf: A benchmark for federated settings". In: *arXiv preprint arXiv:1812.01097* (2018).

[66] Y. Zhang, M. Duan, D. Liu, L. Li, A. Ren, X. Chen, Y. Tan, and C. Wang. "CSAFL: A clustered semi-asynchronous federated learning framework". In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–10.

[67] W. Wu, L. He, W. Lin, R. Mao, C. Maple, and S. Jarvis. "Safa: a semi-asynchronous protocol for fast federated learning with low overhead". In: *IEEE Transactions on Computers* 70.5 (2020), pp. 655–668.

[68] J. Jiang, B. Cui, C. Zhang, and L. Yu. "Heterogeneity-aware distributed parameter servers". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 463–478.

[69] W. Zhang, S. Gupta, X. Lian, and J. Liu. "Staleness-aware async-sgd for distributed deep learning". In: *arXiv preprint arXiv:1511.05950* (2015).

[70] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar. *Slow and Stale Gradients Can Win the Race: Error-Runtime Trade-offs in Distributed SGD*. 2018. DOI: 10.48550/ARXIV.1803.01113. URL: https://arxiv.org/abs/1803.01113.

[71] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. R. Yu, and T. Huang. "When Serverless Computing Meets Edge Computing: Architecture, Challenges, and Open Issues". In: *IEEE Wireless Communications* 28.5 (2021), pp. 126–133. DOI: 10.1109/MWC.001.2000466.

[72] S. G. Kulkarni, G. Liu, K. K. Ramakrishnan, and T. Wood. "Living on the Edge: Serverless Computing and the Cost of Failure Resiliency". In: *2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. 2019, pp. 1–6. DOI: 10.1109/LANMAN.2019.8846970.

[73] G. Cloud. *Cloud Functions Service Level Agreement (SLA)*. 2022. URL: https://cloud.google.com/functions/sla (visited on 04/14/2022).

[74] A. W. Services. *AWS Lambda Service Level Agreement*. 2022. URL: https://aws.amazon.com/lambda/sla/ (visited on 04/14/2022).

[75] G. Cloud. *Compute Engine Service Level Agreement (SLA)*. 2022. URL: https://cloud.google.com/compute/sla (visited on 04/14/2022).

[76] F. Klinker. "Exponential moving average versus moving exponential average". In: *Mathematische Semesterberichte* 58.1 (2011), pp. 97–107.

[77] T.Caliński and J. Harabasz. "A dendrite method for cluster analysis". In: *Communications in Statistics* 3.1 (1974), pp. 1–27. DOI: 10.1080/03610927408827101. eprint: https://www.tandfonline.com/doi/pdf/10.1080/03610927408827101. URL: https://www.tandfonline.com/doi/abs/10.1080/03610927408827101.

[78] scikit-learn developers. *sklearn.cluster.DBSCAN — scikit-learn 1.0.2 documentation*. 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html (visited on 04/14/2022).

[79] G. Cloud. *Cloud Functions*. 2022. URL: https://cloud.google.com/functions/docs/2nd-gen/overview (visited on 04/14/2022).

[80] G. Cloud. *Cloud Functions pricing*. 2022. URL: https://cloud.google.com/functions/pricing (visited on 04/14/2022).

[81] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik. "EMNIST: Extending MNIST to handwritten letters". In: *2017 international joint conference on neural networks (IJCNN)*. IEEE. 2017, pp. 2921–2926.

[82] W. Shakespeare. *The complete works of William Shakespeare*. Race Point Publishing, 2014.

[83] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[84] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[85] S. Hochreiter and J. Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[86] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[87] 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". In: *arXiv preprint arXiv:2007.14390* (2020).

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