FedLesScan: Mitigating Stragglers in Serverless Federated Learning
Mohamed Elzohairy*, Mohak Chadha*, Anshul Jindal*, Andreas Grafberger*, Jianfeng Gu*,
Michael Gerndt*, Osama Abboud†
*Chair of Computer Architecture and Parallel Systems, Technische Universität München
Garching (near Munich), Germany
†Huawei Technologies, Munich, Germany
Email: {mohamed.elzohairy, mohak.chadha, anshul.jindal, andreas.grafberger, jianfeng.gu}@tum.de,
gerndt@in.tum.de, osama.abboud@huawei.com

Abstract—Federated Learning (FL) is a machine learning paradigm that enables the training of a shared global model across distributed clients while keeping the training data local. While most prior work on designing systems for FL has focused on using stateful always running components, recent work has shown that components in an FL system can greatly benefit from the usage of serverless computing and Function-as-a-Service technologies. To this end, distributed training of models with serverless FL systems can be more resource-efficient and cheaper than conventional FL systems. However, serverless FL systems still suffer from the presence of stragglers, i.e., slow clients due to their resource and statistical heterogeneity. While several strategies have been proposed for mitigating stragglers in FL, most methodologies do not account for the particular characteristics of serverless environments, i.e., cold-starts, performance variations, and the ephemeral stateless nature of the function instances. Towards this, we propose FedLesScan, a novel clustering-based semi-asynchronous training strategy, specifically tailored for serverless FL. FedLesScan dynamically adapts to the behavior of clients and minimizes the effect of stragglers on the overall system. We implement our strategy by extending an open-source serverless FL system called FedLess. Moreover, we comprehensively evaluate our strategy using the 2nd generation Google Cloud Functions with four datasets and varying percentages of stragglers. Results from our experiments show that compared to other approaches FedLesScan reduces training time and cost by an average of 8% and 20% respectively while utilizing clients better with an average increase in the effective update ratio of 17.75%.

Index Terms—Federated learning, Deep learning, Serverless computing, Function-as-a-service, FaaS

I. INTRODUCTION

The number of heterogeneous edge devices such as modern smartphones and IoT have significantly increased over the past few years. Due to the presence of several smart sensors, their increasing popularity, and computation power, these edge devices are able to accumulate and process enormous amounts of data each day [1]. Coupled with distributed machine learning (ML) techniques, the data generated by these devices can be used to solve challenging AI tasks. With the increasing privacy concerns of the data holders and recent legislations on data protection and privacy such as the European General Data Protection Regulation (GDPR), Federated Learning (FL) has emerged as a novel distributed ML training paradigm that enables collaborative on-device training of ML models [2]. In contrast to the traditional centralized ML approach, devices (clients) in FL never share their private data directly but learn a shared global model by optimizing its parameters on each device locally and sending back only the updated parameters. Following this, the local model updates are aggregated to form the shared global model.

A traditional FL system consists of two main entities, i.e., clients and a central server. While most prior work on designing FL systems has relied on using stateful always running components [3], recent work [4], [5], [6] has shown that both entities in a traditional FL system can greatly benefit from the usage of stateless serverless computing technologies, particularly Function-as-a-Service (FaaS). FaaS is an emerging cloud-based programming paradigm that enables developers to focus on the application logic, while responsibilities such as infrastructure management, resource provisioning, and scaling are handled by the cloud service providers. In serverless FL, clients are independent functions deployed on a FaaS platform and capable of computing their model updates.

Resource and statistical heterogeneity restrict the collaborative training process in large-scale FL systems [7]. Resource heterogeneity is caused by the difference in the computational and communication capabilities of the clients involved in the FL training process. On the other hand, statistical heterogeneity is caused by the presence of unbalanced non-IID data on clients in FL, i.e., different clients can have a different number of data records and one client’s dataset is not representative of the full data distribution across all clients. As a result, the slower clients, i.e., stragglers greatly affect the global model training process. The usage of FaaS functions as FL clients greatly reduces the effect of stragglers on the overall training costs since client functions are only billed for their execution. However, the presence of stragglers in a serverless FL system can still reduce the accuracy of the trained global model and significantly increase the total FL round duration as shown in Figure 1.

To mitigate stragglers in FL, several synchronous [8], [9] and asynchronous techniques [10] have been proposed. Synchronous strategies have the disadvantages of increased training time and resource utilization, while asynchronous
strategies suffer from higher communication costs and often require a persistent communication link between the FL server and the clients. Moreover, most prior strategies [11], [12], [13] do not account for the particular characteristics of serverless environments, i.e., cold-starts, performance variations [14], and the ephemeral stateless nature of the function instances. To this end, this paper advances the state-of-the-art in Serverless FL by proposing FedLesScan, a novel clustering-based semi-asynchronous training strategy, specifically tailored for serverless FL. Our strategy consists of two main components. First, an adaptive clustering-based client selection algorithm that selects a subset of clients for training based on their previous behavior. Second, a staleness-aware aggregation scheme to mitigate slow model updates and avoid wasted contribution of clients. Towards efficient serverless FL, our key contributions are:

- We propose FedLesScan, a novel training strategy designed for serverless FL.
- We implement FedLesScan by extending an open-source serverless FL system called FedLess [5]. This represents a real system that can be used by data holders for FL. Our implementation can be found here1.
- We comprehensively evaluate our methodology using the 2nd generation Google Cloud Functions [15] for up to 200 concurrent clients on four different datasets against two other popular FL training approaches wrt accuracy, round utilization, training time, and cost. We demonstrate that FedLesScan constantly outperforms the other strategies with the different metrics for varying number of stragglers in the system.

II. RELATED WORK

A. Stragglers in Serverless Federated Learning

Despite several advantages of the FaaS model, such as elasticity, resource-efficiency, and costs, such systems have weak reliability guarantees [17], [18]. Node failures can cause requests to be dropped or even executed multiple times. For instance, the Service Level Objective (SLO) for GCF is an uptime percentage of 99.95% [19] as compared to 99.99% for multi-zone compute instances [20]. As a result, FL clients hosted on a FaaS platform can frequently fail. Furthermore, with scale-to-zero, the FL clients might frequently undergo cold starts which can significantly impact the FL training round duration. Moreover, with most commercial FaaS platforms, the FL client function instances are launched on the FaaS platform’s automatically provisioned virtual machine (VM) offerings. However, the user is not aware of the details of the provisioned VMs such as the CPU architecture. To this end, the performance of the FL client functions can significantly vary depending on the characteristics of the provisioned VM [14]. To the best of our knowledge, FedLesScan is the first strategy designed to facilitate efficient FL in serverless environments.

III. SYSTEM DESIGN

Figure 2 provides an overview of the modified system architecture of FedLess [5], with the different added system components highlighted in blue. The first significant enhancement is to the FedLess controller. Previously, the controller required a complex deployment configuration and ran on a Kubernetes (K8s) cluster with deployed OpenWhisk (OW). To overcome this, we completely removed the dependencies on OW and K8s. To this end, the controller is now a lightweight process that can run on any system that includes its dependencies without any infrastructure management. A major challenge with distributed systems is the cost, ease of development, and debugging. Previously, the development and debugging of client/aggregation functions in FedLess required their deployment on an actual FaaS platform. To mitigate this, we implemented a mocking system in FedLess that enables developers to run and debug the entire system on a single machine. We achieved this by adding different mock components to the controller that simulate the behavior of the actual components. It should be noted that using mock clients and aggregator does not require any additional effort from the user’s side and everything is handled internally by the Mock Invoker. The mocking system is activated by passing the -mock flag to the controller. Moreover, we abstracted the aggregation support to enable its implementation with any FaaS platform.

To implement our strategy with FedLess, we added a Strategy Manager component to the controller. It is responsible for controlling the behavior of the selected strategy and contains two sub-components, i.e., the client selection and the aggregation scheme. The former is responsible for selecting

1https://tinyurl.com/3utduuu
the clients involved in a particular training round, while the latter is responsible for the type of aggregation algorithm used. Furthermore, to store the behavioral data of the clients, such as the number of failures, training duration, and the training rounds missed (§IV-B), we added a client history collection to the database. The overall FL training workflow with our strategy remains similar to the one described in [5] with minor changes required for fetching/updating the behavioral data of clients.

IV. FedLESScan

In this section, we describe our methodology for mitigating stragglers in serverless FL.

A. Partitioning Clients

Our selection strategy separates reliable clients that do not miss their training rounds due to slow updates, timeouts, or failures from stragglers. Towards this, we partition the clients into three groups, i.e., (i) rookies, (ii) participants, and (iii) stragglers. Rookies are clients which have not participated in the FL training process and for which no behavioral data exists (§IV-B). Participants are the group of clients that can participate in the clustering process (§IV-B, §IV-C). In our strategy, we use clustering to group reliable clients with similar behavior together. Finally, stragglers are clients that have missed one or more consecutive training rounds. They have the lowest priority in our selection process (§IV-C) and are characterized by using a variable called cooldown as described in §IV-B. Note that, our client selection strategy (§IV-C) adapts to the behavior of the clients as the FL training progresses. As a result, tier-2 clients can move to tier-3 and vice-versa.

B. Gathering Behavioral Data

Our client selection strategy (§IV-C) depends on the data collected from the clients’ behavior in previous training rounds. Towards this, for each client we collect three attributes, i.e., training time, missed rounds, and cooldown. Training time is the time taken by the client to complete local model training. Missed rounds is a list that contains the round numbers of the rounds missed by a client. We use this to calculate a client’s penalty as described in §IV-C. Cooldown represents the number of rounds a client has to stay in the last tier and cannot participate in clustering (§IV-A). We evaluate the cooldown period from the client’s last missed round using Equation 1. For instance, if a client missed round 2, the cooldown is set to 1. Moreover, if the same client missed round 4, the cooldown is multiplied by two. As a result, the clients with cooldown greater than zero are characterized as stragglers. Towards this, using cooldown can reduce the impact of temporarily slow or unavailable clients by lowering their priority for a certain number of rounds (§IV-A, §IV-C).

\[
\text{cooldown} = \begin{cases} 
0 & : \text{if client completed training in time} \\
1 & : \text{if cooldown = zero} \\
\text{cooldown} \times 2 & : \text{otherwise}
\end{cases}
\] (1)

To gather and update the different client attributes, we modify the FedLess controller (§III) and FL client routines as shown in Algorithm 1. In each FL training round, the controller runs the Train_Global_Model routine, while the clients run the Client_Update routine to locally train the model. nClientsPerRound represents the maximum number of clients that must be selected in every round and maxRounds represents the maximum number of allowed training rounds.

Initially, the controller selects a subset of clients, invokes them, and then waits until they finish or a timeout occurs (Lines 3-4). Following this, we iterate over each successful client response and reset the cooldown variable (Lines 5-8). Since the controller doesn’t know if the client was slow or crashed, it assumes that the remaining invoked clients failed to finish. Subsequently, we update their missed rounds and cooldown attributes (Lines 9-12). At the client-side, we first load its behavioral history from previous rounds (Line 17). Following this, we measure the time for the client to load the global model, its local dataset, and train the model with its local data (Lines 18-21). Following this, each client sends its local model updates to the database (Line 22) and updates its training time for the current round (Line 23). Furthermore, slower clients that finished a round later can correct information about their missed rounds. As described before, the controller considers clients that did not finish the round in time as crashed. Therefore, distinguishing between crashed and slow clients is done on the client side. This is done by deleting the current round from their missed rounds list (Lines 24-26). Finally, the client updates its information in the database (Line 27).

C. Selecting Clients

Algorithm 2 describes our strategy for client selection in a particular FL training round. Our strategy promotes fair
Algorithm 2: Client selection

```
1 Function Select_Clients(clients, round, maxRounds, clientsperRound):
2     Characterize clients as rookies, participants, and stragglers.
3     if #rookies ≥ clientsperRound then
4         return Randomly sample clientsperRound from rookies.
5     end
6     Calculate #cluster_clients required from participants.
7     Calculate #straggler_clients required from stragglers.
8     Randomly sample #straggler_clients from stragglers.
9     clusteringData = []
10    for each client in participants do
11        Calculate EMA of client based on its training time.
12        Calculate missed round ratios and missedRoundEma
13        Update clusteringData.
14    end
15    Obtain cluster labels using DBScan and clusteringData.
16    Sort Clusters.
17    Sample #cluster_clients from sorted clusters.
18    return [rookies + cluster_clients + straggler_clients]
```

selection for reliable clients while involving stragglers less in the training process. Initially, we characterize the clients as rookies, participants, or stragglers (§IV-A) based on their previous behaviour (Line 2). First, we randomly select the required number of clients in a round from the pool of rookie clients. If the number of available rookie clients is greater than the number of required clients, the algorithm terminates (Line 3-5). If not, we calculate the number of clients required from the participants and the stragglers (Lines 6-7). Note that, clients from stragglers are only selected if the number of clients selected from the first and second tiers is not sufficient. Following this, we randomly sample the required number of straggler clients from the stragglers (Line 8).

The reason for prioritizing rookies first is to guarantee that every client gets a chance to contribute to the global model. Moreover, it provides data on clients’ behaviour which is used to cluster clients in future rounds. Note that the number of rookies decreases as the training progress, until it reaches zero. In this case, the algorithm fully relies on the dynamic clustering of tier two clients.

\[
\text{totalEMA} = \text{trainingEMA} + \text{missedRoundEMA} \times \text{maxTrainingTime} \quad (2)
\]

For the clients that will participate in clustering, we calculate two features. First, trainingEMA which represents an exponential moving average (EMA) on the previously recorded training times (Line 11). We use EMA since a weighted average better represents the current behavior of the client by giving higher weight to the recently recorded times. Second, missedRoundEMA which is a penalty factor based on the previous rounds missed by the client (§IV-B). This satisfies two objectives. First, recent failures should have higher penalties. Second, the penalty should decrease as the training progresses if the client becomes more reliable. To compute missedRoundEma, we divide the round numbers in the missed rounds list by the current round number to get a list of ratios. Following this, we take an EMA on the computed ratios (Line 12). As the training progresses, the effect of a specific missed round decreases because the current round number increases. The computed features are added to the data required for clustering (Line 13). Following this, we use the collected data as input to the DBSCAN [21] algorithm for partitioning it into separate clusters, each with a specific label (Line 15). For generating clusters, DBSCAN relies on an \( \epsilon \) parameter that represents the maximum distance between two samples to be considered in the neighborhood of each other. For simplicity, we treat outliers as a single cluster. To find a value for \( \epsilon \), we do a grid search and select the parameter value that yields the highest Calinski-Harabasz index [22]. 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 and will not impact our running time. Moreover, due to the low time complexity of DBSCAN, i.e., \( O(N \log(N)) \) the time for computing clusters multiple times is insignificant compared to the overall round time. Following this, we sort the clusters according to the increasing order of the average totalEMA of their members (Line 16). We calculate the totalEMA using Equation 2. For sampling clients from the sorted clusters, we start by first choosing the clients belonging to the faster clusters and gradually move to the slower clusters in the sorted list. Moreover, to provide maximum information to the global model and avoid sampling from the same cluster in every training round, we start choosing from the cluster corresponding to our current training progress. This is determined by using the ratio between the current round and the maximum number of rounds. We continue sampling from the clusters until we reach the required number of clients or there are no more clients to choose from. Finally, our algorithm returns a list of clients selected from rookies, clusters, and stragglers.

\[
w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{t_k}{t} \times \frac{n_k}{n} w_k^k
\]

\[D. \text{ Staleness-aware Aggregation}\]

Although our intelligent client selection strategy (§IV-C) improves the efficiency of the system, stragglers are not completely eliminated. Stragglers might push their local model updates to the parameter server after the completion of an FL training round. Moreover, these updates might contain valuable information that can improve the performance of the global model. Towards this, we aggregate the delayed updates of the clients with a dampening effect asynchronously, i.e., delayed updates are considered the next time the aggregation function is called. Equation 3 shows our updated aggregation function used to include stale updates. \( w_k^k \) is the local model of client \( k \) at round \( t_k \) and \( w_{t+1} \) is the global model after aggregation at round \( t \). Furthermore, \( n_k \) 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 similar to FedAvg [2]. 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 \( \tau \) to dictate the maximum age of updates included in the aggregation. Updates with \( t - t_k \geq \tau \) are discarded by the aggregator. In our experiments, we use a value of two for \( \tau \).
V. EXPERIMENTAL RESULTS

A. Experiment Setup

1) Datasets: For our experiments, we use four datasets from different domains. These include image classification, i.e., (MNIST, FEMNIST), language modeling, i.e., (Shakespeare), and speech recognition, i.e., (Google Speech [16]). We choose the FEMNIST and the Shakespeare datasets from a benchmarking framework for FL called LEAF [23], while the Google Speech dataset was chosen from the FedScale [24] benchmark suite.

2) Model Architectures and Parameters: For each dataset, we use a DNN model architecture suited for the particular task. For the MNIST, FEMNIST, and the Shakespeare datasets, we use the same model architecture used in LEAF [23]. On the other hand, for the Google Speech dataset we use two identical blocks followed by an average pooling layer and an output layer with 35 neurons. A block contains two convolutional layers with a 3x3 kernel followed by a max-pooling layer. However, with our model we achieve similar results for accuracy as compared to [24].

3) Experiment configuration: To scale our experiments, we deployed FedLess (§III) on a VM hosted by the LRZ compute cloud. The VM was configured with 40 virtual CPUs (vCPUs) and 177GiB of RAM. For the aggregator function, we used a self-hosted OpenFaaS cluster deployed on a VM with 10vCPUs and 45GiB of RAM. We limit the memory of the aggregation function to 7GiB. Furthermore, for hosting our datasets we use a nginx store running on a VM with 10vCPUs and 45GiB of RAM.

For all our experiments, we deployed the FaaS based FL clients on the 2nd generation GCFs [15] based on Knative. Each client function had a memory limit of 2048MiB and a timeout of 540 sec. For the MNIST, FEMNIST, and Google Speech datasets, we use 200/300, 175/300, and 200/542 concurrent clients per round respectively. On the other hand, for Shakespeare, we use 50/100 clients per round.

4) Experiment scenarios: With our experiments, we aim to evaluate the performance of our strategy against delays and client function dropouts. Although the deployed functions will show delays, failures, and cold starts, it does not indicate how our strategy behaves in extreme situations, i.e., with a significantly high number of stragglers. Towards this, we consider two scenarios in our experiments.

Standard Scenario. In this, we perform the experiments on the deployed client functions without any modifications. Furthermore, we adjust the FL training round time to ensure that the clients can finish their training before the training round ends. This scenario demonstrates the performance in a more real-world situation when clients on commercial FaaS platforms are used for training.

Straggler (%) Scenario. In this, we simulate varying percentages of stragglers in the FL system. Although there might be different reasons for FaaS client failures, such as memory limit, function timeout, or communication issues, these failures can only have one of two effects on the clients. Clients can either push their updates after the training round is finished (slow updates) or can completely crash (not push their updates). To simulate slow updates, we limit the training round time to only fit clients with no issues or delays. Meaning, that clients which experience cold starts, bandwidth limitations, or communication delays do not finish the round in time; therefore, pushing their updates later. On the other hand, 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 in the system.

5) Metrics: For comparing the different strategies, we use the standard metrics accuracy, experiment duration, and cost. For calculating accuracy of the trained global model, we randomly choose a set of clients and evaluate it on their test datasets. Following this, we multiply the obtained accuracy for a particular client by the ratio of its test set cardinality to the total cardinality of the test dataset. The final accuracy value is obtained by averaging the obtained accuracy values. While accuracy describes model performance, it does not provide insights about the performance of the strategy and the contributions of the clients to the global model. Towards this, we also use the metrics Effective Update Ratio (EUR) [13] and Bias [13]. EUR is defined as the ratio between the successful clients and the subset of selected clients. It shows the effect of stragglers on round utilization. A higher value of EUR represents less wasted resources since clients requested to participate in a certain round end up contributing to the global model. Furthermore, to provide insights into the bias of the client selection schemes, we use variance plots. This is done by showing the frequency of selection for each client across the FL training process. Bias is defined as the difference between the frequency of the least called client and the most called client [13]. For scenarios with low number of stragglers, we target low bias, while for scenarios with high number of stragglers the bias should be higher due to the prioritization of reliable clients in training. Experiment duration represents the total time required for training the model. For computing training costs, we use the cost computation model [25] used by Google to estimate the cost for each client function based on the number of invocations, allocated memory, and execution duration.

B. Comparing accuracy and round utilization

The obtained accuracy and EUR values across all our experiments is summarized in Table I. For the standard scenarios (§V-A4), we obtained better accuracy and round utilization for our strategy as compared to FedAvg and FedProx.
across all datasets except Shakespeare. This is because with
the Shakespeare dataset, some of the clients contribute to
the model accuracy more than others, especially clients with
longer training times. Moreover, due to budget constraints and
significantly high training costs, we train for a slightly less
number of rounds, i.e., 25 for Shakespeare. However, we argue
that in a more realistic scenario with more number of rounds,
the difference in accuracy will decrease. This is because our
strategy utilizes clients more efficiently, i.e., higher EUR. As
a result, with more rounds the number of invocations per
client will increase and more clients will contribute to the
global model, leading to a higher accuracy. Similarly, for
the straggler (%) scenarios, we obtained better results for
accuracy with our strategy as compared to the other two for
the MNIST, FEMNIST, and the Google Speech datasets. On the
Shakespeare dataset, our strategy outperformed FedAvg and
FedProx in scenarios with 30%, 50%, and 70% stragglers. In
terms of EUR, our strategy constantly outperforms the other
two strategies across all scenarios and datasets since they use
random selection for selecting clients for an FL training round.

For most standard scenarios (Table I), FedLesScan ob-
tained better accuracy as compared to the other two strategies
due to the better distribution of client invocations. This is
because our strategy prioritizes clients with the least number
of invocations in a selected client cluster (§IV-C) leading to more
balanced contributions among the participating clients. On the
other hand, with straggler (%) scenario, our strategy reached better accuracies by relying more on robust and reliable clients.
Furthermore, the utilization of a staleness-aware aggregation
scheme (§IV-D) avoids wasting valuable contributions, which
in turn increases accuracy.

To offer detailed insights, we present results for the Google
Speech dataset [16] wrt the metrics accuracy and EUR, across
the FL training session as shown in Figures 3a and 3b
respectively. For the standard scenarios, we ran the FL training
session for 35 rounds, while for the straggler (%) scenarios,
we ran the experiments for 60 rounds. For the standard sce-
cenario, our strategy reached an accuracy of 79.4% as compared
to 76.6% and 77.4% for FedAvg and FedProx respectively.
Furthermore, our strategy showed faster convergence by reach-
ing an accuracy of 70% in 19 rounds as compared to 21
and 22 for FedAvg and FedProx respectively. With 10% stragglers, our strategy and FedAvg had a similar convergence rate, while FedProx was slightly behind. Moreover, our
strategy reached an end accuracy of 76% which is a 6% and
10% increase over FedAvg and FedProx respectively. For
an FL system with 30% stragglers, our strategy consistently
outperforms FedAvg by around 8% towards the end of the
training, while outperforming FedProx by a smaller margin
of 1%. We observe a similar trend for our experiments with
50% and 70% stragglers in the system, where our strategy
outperforms the other two. From our experiments, we observe
that the presence of stragglers affects the convergence speed
of the DNN models in an FL training session. DNN models
trained in standard scenarios converge faster as compared to
the models trained with straggler (%) scenarios as shown in
Figure 3a. We observe a similar behaviour for the different
datasets (§V-A1). Moreover, for the straggler (%) scenarios,
increasing the percentage of stragglers in the system does not
consistently decrease the accuracy of the trained DNN model
as shown in Table I. This is especially true for experiments
with a higher number of stragglers, i.e., 70%. This behavior is
not exclusive to our experiments and was also reported by Li
et al. [8] and Wu et al. [13]. While the authors do not provide
a clear explanation for this behaviour, we argue that due to the
non-IID nature of the data, clients do not contribute evenly to
the test accuracy. In addition, having fewer number of reliable
clients reduces the varying effect of local model updates and
local model deviations from the global model, making it easier
to reach a consensus on the global model. Therefore, we can
reach situations where a system with more stragglers reaches
higher overall accuracy at the end.
Figure 3b shows the EUR comparison among the three strategies for the different scenarios. In the standard scenario, the three strategies performed similarly, achieving more than 99% average EUR ratio during training. In the straggler (%) scenarios, our strategy consistently achieves higher EUR as compared to the other strategies. Furthermore, as the number of stragglers in the system increases, the difference between the average EUR of our strategy as compared to the other two also increases. We observe occasional drops in EUR for our strategy with varying number of stragglers in the system. These drops demonstrate the effect of clustering of the clients. Distributing slow clients across the training rounds will affect the efficiency of more rounds. Our strategy utilizes dynamic clusters to combine clients with similar behavior together. Recomputing the clusters each round based on clients’ recent behavior reduces the impact of stragglers on training the other clients. However, this leads to occasional drops in the EUR, which happens when an unreliable subset of clients is invoked. The subsequent rounds, involving the rest of the clients, maintain higher EUR values, thereby decreasing the total training time.

Although EUR demonstrates 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 will have a higher EUR but will underutilize the rest of the clients. To this end, we use violin plots as shown in Figure 3c to provide insights into the bias encountered by our strategy (§V-A5). 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 points in the distribution. A greater difference (height) represents that the algorithm is biased towards a specific subset of clients, while a smaller difference (height) represents that the difference between the most and least invoked clients is low. A bigger width indicates that certain clients were invoked more frequently. For the standard scenario, we observe that FedAvg and FedProx show similar behaviour without distinguishing between stragglers and reliable clients. This is because they use random client selection. On the other hand, our strategy adapts to stragglers and promotes fair selection of clients. This is apparent since the distribution of our strategy is centered around similar values, i.e., most clients have the same number of invocations. Furthermore, we observe that with our strategy few clients have 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. For the straggler (%) scenarios, we observe that our strategy prioritizes reliable clients while relying on stragglers less during training. We see a distinction between the number of invocations of reliable clients and stragglers.

### C. Comparing time and cost

Although model accuracy is an important metric, fast convergence wrt the number of training rounds does not provide a complete picture of the efficiency of the system. Towards this, in this section, we provide a collective analysis of all experiments wrt total duration and costs. To compute the total experiment duration, we aggregate the total round time during training. For all the three strategies, the round time is determined by the slowest invoked client. As a result, the round time depends on either the response of the slowest client or a predetermined timeout (§V-A4). Furthermore, for straggler (%) scenarios, we simulate real-world behavior by assuming that the stragglers will not respond, thus forcing the controller (§III) to wait until the round timeout.

Table II shows the total aggregated time per experiment. For the standard scenario, training a model with our strategy is significantly faster as compared to FedAvg and FedProx across all datasets. For instance, for the MNIST dataset, our strategy takes 40% less time as compared to the other strategies. For the straggler (%) scenarios, we observe the effect of stragglers on experiment duration. In the scenarios with 10% and 30% stragglers, we observe that FedLesScan maintains a lower duration across all experiments. However, when the number of stragglers in the system is significantly higher, they must be invoked in almost all training rounds to meet the minimum number of clients required per round. We notice this behavior for our strategy with greater than 50% stragglers in the system for the MNIST, FEMNIST, and the Shakespeare dataset. However, for the Google Speech dataset, our strategy still has an 18% lower experiment duration as compared to the other two. This is because the total number of clients for the Google Speech dataset is 542 with 200 concurrent clients participating in a training round (§V-A3). For the scenario with 70% stragglers in the system, 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 fail. However, their running cost still factors into the experiment cost. In the worst-case scenario, stragglers can increase costs by wasting resources doing computations on...
wasted contributions. As a result, we estimate the cost of stragglers as the cost of running the functions for the entire round duration.

Table III shows the cost for the different strategies, datasets, and scenarios. For the standard scenario with MNIST, we observe a 6.8% and 50% cost reduction for our strategy as compared to FedAvg and FedProx. Similarly for the FEM-NIST and Google Speech datasets in the standard scenario, we observe cost reductions of about 2%, 20% and 12%, 27% as compared to FedAvg and FedProx respectively. On the other hand, for the Shakespeare dataset, FedProx performed better than our strategy and FedAvg by 4% and 6% respectively. We observe that the efficiency of our strategy becomes more visible as the number of stragglers in the system increases. For all the scenarios with a varying number of stragglers, our strategy has the minimum cost as compared to the other two strategies. For the straggler (%) scenarios, our strategy consistently achieved lower experiment cost across all datasets with an average cost reduction of 25% and 32% as compared to FedAvg and FedProx respectively.

VI. CONCLUSION & FUTURE WORK

In this paper, we made two main contributions. First, we made several extensions to an open-source system and framework for serverless FL called FedLess. Second, we proposed and implemented FedLessScan, a novel clustering-based training strategy designed for FL on FaaS platforms. We comprehensively evaluated FedLessScan by comparing its performance against two popular strategies on multiple datasets. Overall, our experiments showed that FedLessScan achieved better results in terms of accuracy, training time, effective update ratio, and training costs by better utilizing the participating clients. In the future, we plan to explore dynamically adapting the number of clients selected each round in FedLessScan based on the current system state.

VII. ACKNOWLEDGEMENT

The research leading to these results was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)-Projektnummer 146371743-TRR 89: Invasive Computing. Google Cloud credits in this work were provided by the Google Cloud Research Credits program with the award number 64e92de5-fb62-4386-8c5b-ff3f480390bb.

REFERENCES

[1] M. Chiang and T. Zhang, “Fog and iot: An overview of research opportunities,” IEEE Internet of things journal, vol. 3, no. 6, pp. 854–864, 2016.
[2] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data,” in Artificial Intelligence and Statistics. PMLR, 2017, pp. 1273–1282.
[3] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, T. Parcollet, N. D. Lane, P. P. B. de Gusmão, and N. D. Lane, “Flower: A Friendly Federated Learning Research Framework,” arXiv, pp. 1–22, jul 2020. [Online]. Available: http://arxiv.org/abs/2007.14390
[4] M. Chadha, A. Jindal, and M. Gerndt, “Towards federated learning using faas fabric,” in Proceedings of the 2020 Sixth International Workshop on Serverless Computing, ser. WoSC’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 49–54. [Online]. Available: https://doi.org/10.1145/3429880.3430100
[5] A. Graßberger, 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), 2021, pp. 164–173. [Online]. Available: https://doi.org/10.1109/BigData52589.2021.9672067
[6] K. Jayaram, V. Muthusamy, G. Thomas, A. Verma, and M. Purcell, “Adaptive aggregation for federated learning,” arXiv preprint arXiv:2203.12163, 2022.
[7] W. Y. B. Lim, N. C. Luong, D. T. Hoang, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, and C. Miao, “Federated learning in mobile edge networks: A comprehensive survey,” IEEE Communications Surveys & Tutorials, vol. 22, no. 3, pp. 2031–2063, 2020.
[8] A. K. Sahu, T. Li, M. Sanjabi, M. Zaheer, A. Talwalkar, and V. Smith, “On the convergence of federated optimization in heterogeneous networks,” arXiv preprint arXiv:1812.06127, vol. 3, p. 3, 2018.
[9] C. T. Dinh, N. H. Tran, T. D. Nguyen, W. Bao, A. Y. Zomaya, and B. B. Zhou, “Federated learning with proximal stochastic variance reduced gradient algorithms,” in 46th International Conference on Parallel Processing - ICPP, ser. ICPP ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3404397.3404457
[10] C. Xie, S. Koyejo, and I. Gupta, “Asynchronous federated optimization,” arXiv preprint arXiv:1903.03934, 2019.
[11] Z. Chai, Y. Chen, A. Anwar, L. Zhao, Y. Cheng, and H. Rangwala, “Fedal: A high-performance and communication-efficient federated learning system with asynchronous tiers,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1109/SC5358817.3476211
[12] Y. Zhang, M. Duan, D. Liu, L. Li, A. Ren, X. Chen, Y. Tan, and C. Wang, “Coffaf: A clustered semi-asynchronous federated learning framework,” in 2021 International Joint Conference on Neural Networks (IJCNN). IEEE, 2021, pp. 1–10.
[13] 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,” IEEE Transactions on Computers, vol. 70, no. 5, pp. 655–668, 2020.
[14] M. Chadha, A. Jindal, and M. Gerndt, “Architecture-specific performance optimization of compute-intensive faas functions,” in 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), 2021, pp. 478–483. [Online]. Available: https://doi.org/10.1109/CLOUD53861.2021.00062
[15] Google Cloud, “Cloud Functions Second Generation— Google Cloud,” 2022. [Online]. Available: https://cloud.google.com/functions/docs/2nd-gen/overview
[16] P. Warden, “Speech commands: A dataset for limited-vocabulary speech recognition,” 2018. [Online]. Available: https://arxiv.org/abs/1804.03209
[17] 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,” IEEE Wireless Communications, vol. 28, no. 5, pp. 126–133, 2021.
[18] 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.
[19] G. Cloud, “Cloud functions service level agreement (sla),” 2022. [Online]. Available: https://cloud.google.com/functions/sla
[20] —, “Compute engine service level agreement (sla),” 2022. [Online]. Available: https://cloud.google.com/compute/sla
[21] 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, no. 34, 1996, pp. 226–231.
[22] T. Caliński and J. Harabasz, “A dendrite method for cluster analysis,” Communications in Statistics, vol. 3, no. 1, pp. 1–27, 1974. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/03610927408827101
[23] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar, “LEAF: A Benchmark for Federated Settings,” in Workshop on Federated Learning for Data Privacy and Confidentiality, NeurIPS, 2018, pp. 1–9. [Online]. Available: http://arxiv.org/abs/1812.01097
[24] F. Lai, L. Dai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, “Fedscale: Benchmarking model and system performance of federated learning,” in Proceedings of the First Conference on System Challenges in Reliable and Secure Federated Learning, 2021, pp. 1–3.
[25] G. Cloud, “Cloud functions pricing,” 2022. [Online]. Available: https://cloud.google.com/functions/pricing