Apodotiko: Enabling Efficient Serverless Federated Learning in Heterogeneous Environments

Mohak Chadha*, Alexander Jensen*, Jianfeng Gu*, Osama Abboud†, Michael Gerndt*
*Chair of Computer Architecture and Parallel Systems, Technische Universität München
Garching (near Munich), Germany
†Huawei Technologies, Munich, Germany
Email: {mohak.chadha, alexander.jensen, jianfeng.gu}@tum.de, osama.abboud@huawei.com, gerndt@in.tum.de

arXiv:2404.14033v1 [cs.DC] 22 Apr 2024

***Abstract*—Federated Learning (FL) is an emerging machine learning paradigm that enables the collaborative training of a shared global model across distributed clients while keeping the data decentralized. Recent works on designing systems for efficient FL have shown that utilizing serverless computing technologies, particularly Function-as-a-Service (FaaS) for FL, can enhance resource efficiency, reduce training costs, and alleviate the complex infrastructure management burden on data holders. However, current serverless FL systems still suffer from the presence of *stragglers*, i.e., slow clients that impede the collaborative training process. While strategies aimed at mitigating stragglers in these systems have been proposed, they overlook the diverse hardware resource configurations among FL clients. To this end, we present `Apodotiko`, a novel *asynchronous* training strategy designed for serverless FL. Our strategy incorporates a scoring mechanism that evaluates each client's hardware capacity and dataset size to intelligently prioritize and select clients for each training round, thereby minimizing the effects of stragglers on system performance. We comprehensively evaluate `Apodotiko` across diverse datasets, considering a mix of CPU and GPU clients, and compare its performance against five other FL training strategies. Results from our experiments demonstrate that `Apodotiko` outperforms other FL training strategies, achieving an average speedup of 2.75x and a maximum speedup of 7.03x. Furthermore, our strategy significantly reduces cold starts by a factor of four on average, demonstrating suitability in serverless environments.**

***Index Terms*—Federated learning, Deep learning, Serverless computing, Function-as-a-service, Straggler mitigation**

I. INTRODUCTION

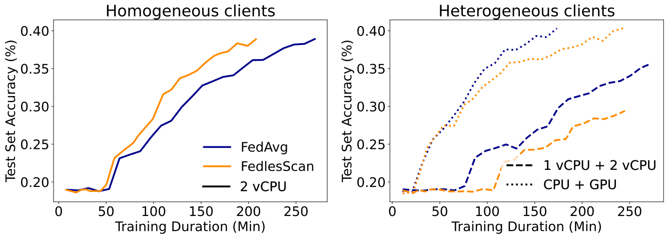
Increasing concerns about data privacy and recent legislations such as the Consumer Privacy Bill of Rights in the U.S. [1] prevent the training of ML models using the traditional centralized learning approach. With the goal of not exposing raw data as in centralized learning [2], an emerging distributed training paradigm called Federated Learning (FL) [3] has gained significant popularity in various application domains, such as medical care [4] and mobile services [5].

FL enables the collaborative training of a shared global ML model across remote devices or `clients` while keeping the training data decentralized. The traditional FL training process [3] is *synchronous* and occurs in multiple rounds. A main component called the `central server` organizes the training process and decides which clients contribute in a new round. During each round, clients improve the shared global model by optimizing it on their local datasets and sending back only the updated model parameters to the central server. Following this, the local model updates from all participating clients are collected and aggregated to form the updated consensus model. Recent works on designing systems for efficient FL have shown that both components in an FL system, i.e., the `clients` and the `central server`, can immensely benefit from an emerging cloud computing paradigm called *serverless computing* [6], [7], [8], [9], [10], [11], [12].

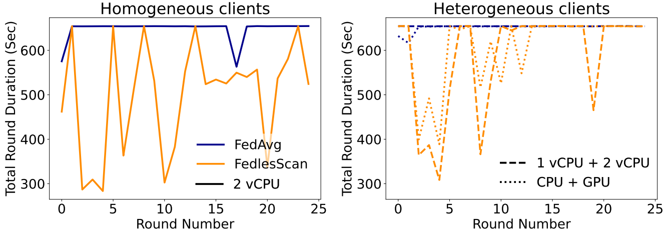
Function-as-a-Service (FaaS) is the computational concept of serverless computing and has gained significant popularity and widespread adoption in various application domains such as machine learning [13], [14], edge computing [15], heterogeneous computing [16], [17], [18], [19], and scientific computing [20], [21]. In FaaS, developers implement fine-grained pieces of code called *functions* that are packaged independently in containers and uploaded onto a FaaS platform. These functions are *ephemeral*, *event-driven*, and *stateless*. Several open-source and commercial FaaS platforms, such as OpenFaaS [22] and Google Cloud Functions (GCF) [23], are currently available. Clients in serverless FL are independent functions deployed onto a FaaS platform and capable of performing their model updates.

The FaaS computing model offers several advantages, such as no infrastructure management, automatic scaling to zero when resources are unused, and an attractive fine-grained *pay-per-use* billing policy [24]. Incorporating FaaS functions as `clients` in FL systems can improve resource efficiency and reduce training costs [7], [12]. In addition, utilizing FaaS technologies for the aggregation process in the FL `server` can enhance aggregation performance, scalability, and resource efficiency [7], [9], [10], [11].

Large-scale practical FL systems encounter different fundamental client-level challenges that limit collaborative training. These include computational heterogeneity and statistical data heterogeneity. FL clients in the wild [5] can vary from small edge devices to high-performant GPU-enabled systems with varying memory, compute, and storage capacities. In addition, clients in practical FL systems have *unbalanced non-IID* data distributions, i.e., the private data samples held by individual clients exhibit variations in their statistical properties, such as feature distributions, class imbalances, or data biases [25]. These two challenges often result in the presence of `stragglers`, i.e., slower clients within the FL system. Stragglers tend to increase the duration and costs of the FL training process while diminishing the accuracy of the trained global model [8], [26], [27].



(a) Comparing trained model accuracy.



(b) Comparing FL round durations.

Fig. 1: Comparing `FedAvg` [3] and `FedLesScan` [8] across various client-hardware resource configurations using *FedLess* [7]. The results are obtained using the non-IID data partitions of the *Shakespeare* dataset [32] with 100 clients deployed on `OpenFaaS` [22].

To mitigate stragglers in FL, several strategies have been proposed in the literature [3], [27], [28], [29], [30], [31]. However, most of these techniques fail to consider the unique characteristics of serverless environments, such as function cold starts, performance variations, and the transient stateless nature of function instances. To this end, `FedLesScan` [8] represents the current state-of-the-art for straggler mitigation in serverless FL. It is a *semi-asynchronous* training strategy that dynamically adapts to the behavior of clients to minimize the impact of stragglers on the FL system. It consists of two key components: a clustering-based client selection algorithm and a staleness-aware aggregation scheme. The former is responsible for selecting a subset of clients for training based on their previous training round durations, while the latter accounts for delayed client round updates.

Figure 1 compares the performance of `FedLesScan` with `FedAvg` across three different client hardware resource distribution scenarios. In the first scenario, all 100 clients have the same hardware resource configuration of 2vCPUs. In the second scenario, we configured 60 clients with 1vCPU and 40 clients with 2vCPUs, while in the third scenario, we had a mix of 50 clients with 1vCPU, 30 clients with 2vCPUs, and 20 clients running on GPUs. In each round, 50 clients are selected for training. With homogeneous clients, we observe that `FedLesScan` requires 30% less time than `FedAvg` to achieve an accuracy of 40% for the global model as shown in Figure 1a. However, with heterogeneous clients, `FedlesScan` struggles and lags behind `FedAvg` in performance. For the second scenario, `FedlesScan` requires 40% more training time than `FedAvg` to reach an accuracy of 30%, while for the third scenario `FedAvg` is 43% faster. To provide deeper insights, Figure 1b offers a detailed breakdown of the training duration for individual rounds. In the homogeneous scenario, `FedlesScan` effectively clusters clients with adequate sizes, enabling efficient client selection for each training round. This leads to sporadic peaks in round durations, occurring when stragglers are included. However, in the heterogeneous scenarios, more rounds reach their maximum duration, resembling the round times observed with `FedAvg`. This is because the clustering method adopted by `FedlesScan` fails to provide enough clients per cluster for selection in each round, resulting in the inclusion of stragglers as replacements. The presence of stragglers during training rounds causes the FL server to time out, resulting in relatively consistent round durations. Our experiments and observations demonstrate that `FedlesScan` fails to accommodate the increasing heterogeneity in FL client hardware under non-IID data distributions, leading to decreased global model accuracy and increased training times. To address these limitations and advance the state-of-the-art in serverless FL, our key contributions are:

- We present `Apodotiko`[1], a novel *asynchronous* scoring-based strategy that enables efficient serverless FL across clients with varying hardware resource configurations and data distributions.
- We implement `Apodotiko` by extending a popular open-source serverless FL framework called *FedLess* [7], [33]. This enables greater adoption and accessibility of our strategy in the community.
- We demonstrate with extensive experiments the performance of our strategy against five other popular FL training approaches across multiple datasets using various metrics, including accuracy, selection bias, and costs.

The rest of the paper is structured as follows. In §II, we describe the previous approaches related to our work. §III describes our strategy in detail. In §IV, we present our experimental results. Finally, §V concludes the paper and presents an outlook.

II. RELATED WORK

A. *Serverless Federated Learning*

Using serverless computing technologies, particularly FaaS, for designing efficient systems for FL is a relatively new research direction. Existing works in this domain can be categorized into two groups: (i) systems that employ serverless functions exclusively in the `central server` [9], [10], [11] and (ii) systems that leverage serverless functions in both entities of an FL system [6], [7], [12]. In [9], Jayaram et al. propose λ-FL, a serverless aggregation strategy for FL to improve fault tolerance and reduce resource wastage. The authors use serverless functions as aggregators to optimize the aggregation of model parameters in conventional `FedAvg` [3] over several steps. They implement their prototype using message queues, `Kafka`, and use `Ray` [34] as the serverless platform. In [10] and [11], the authors extend their previous strategy to enable adaptive and just-in-time aggregation of client model updates using serverless functions. In the second group, *FedKeeper* [6] was the first serverless FL system that enabled the training of Deep Neural Network (DNN) models using FL for clients distributed across a combination of heterogeneous FaaS platforms. However, it lacked crucial features required for practical FL systems, such as security and

---

[1] https://github.com/Serverless-Federated-Learning/FedLess

Attribute Strategy	Type	FaaS Support	Asynchronous Aggregation	Performance-based Selection	Client Efficiency Scoring	Adaptive Penalty
FedProx [37]	Synchronous	👎	👎	👎	👎	👎
FedNova [38]	Synchronous	👎	👎	👎	👎	👎
SCAFFOLD [29]	Synchronous	👎	👎	👎	👎	👎
TiFL [26]	Synchronous	👎	👎	👍	👎	👍
Aergia [39]	Synchronous	👎	👎	👎	👎	👎
Oort [40]	Synchronous	👎	👎	👍	👎	👎
SAFA [30]	Semi-asynchronous	👎	👎	👎	👎	👎
FedAT [41]	Semi-asynchronous	👎	👎	👍	👎	👍
FedAsync [42]	Asynchronous	👎	👍	👎	👎	👎
FedBuff [31]	Asynchronous	👎	👍	👎	👎	👎
Pisces [27]	Asynchronous	👎	👍	👍	👎	👍
FedlesScan [8]	Semi-asynchronous	👍	👎	👍	👎	👍
Apodotiko (This work)	Asynchronous	👍	👍	👍	👍	👍

TABLE I: Comparing strategies for straggler mitigation in FL. 👍Supported. 👎No support.

support for large DNN models. To address these drawbacks, *FedLess* [7] was introduced as an evolution of *FedKeeper* with multiple new enhancements. These include: (i) support for multiple open-source and commercial FaaS platforms, (ii) authentication/authorization of client functions using AWS Cognito, (iii) training of arbitrary homogeneous DNN models using the `Tensorflow` library, (iv) the privacy-preserving FL training of models using Differential Privacy [35], and (v) the aggregation of model updates using serverless functions. A more recent work by Kotsehub et al. [12] introduces *Flox*, a system built on the `funcX` [36] serverless platform. *Flox* aims to separate FL model training/inference from infrastructure management, providing users with a convenient way to deploy FL models on heterogeneous distributed compute endpoints. However, its tight integration with `funcX` restricts its compatibility with other open-source or commercial FaaS platforms, limiting its applicability and generality.

B. Stragglers in Federated Learning

Addressing the impact of slow clients during training in FL is an active research area. Towards this, several *synchronous* [26], [29], [37], [38], [39], [40], *semi-asynchronous* [8], [30], and *asynchronous* [27], [31], [41], [42] strategies have been proposed in the literature.

`FedProx` [37], a popular strategy, builds on `FedAvg` with two essential modifications. First, it introduces a specialized loss function at the client level to regulate fluctuations in local updates, enhancing the model's stability across varied data distributions. Second, it enables clients to adjust their workload based on hardware and network constraints, ensuring adaptability by varying the number of local updates performed. Similar to `FedProx`, `FedNova` [38] tackles statistical heterogeneity in FL by merging the concepts of `FedAvg` with momentum-based optimization. It introduces a momentum term to stabilize convergence and alleviate the impact of noisy local updates. To accommodate varying local updates for clients, it introduces a new weighting scheme that normalizes client local updates with the number of local steps. However, `FedNova` is tailored only for `SGD`, limiting its broader applicability. In [29], Karimireddy et al. propose `SCAFFOLD`, an FL strategy designed to address the challenges of varied local client data distributions and biased updates. The authors employ *control variates*, a technique from convex optimization, to reduce stochastic gradient variance. This minimizes local update variability, stabilizing the aggregation process. In addition, it enables identifying and eliminating client-specific biases pre-aggregation, enhancing global model accuracy and stability. Unlike `FedProx`, `SCAFFOLD` doesn't accommodate varying local progress and mandates uniform local update counts across clients. Moreover, it relies on full client participation for peak performance, diminishing its effectiveness with reduced client sampling ratios per round, as shown in [43]. `TiFL` [26] is a tier-based system designed to address heterogeneity challenges in FL (§I). It organizes clients into tiers, selecting same-tier clients per round to mitigate the impact of stragglers. In addition, it incorporates an adaptive tiering mechanism that dynamically adjusts tiers based on observed training performance. However, the authors limit their experiments to only CPU-based clients without exploring extreme heterogeneous environments with a mix of both CPU and GPU-based clients. In [40], Lai et al. propose `Oort` [40] a strategy that aims to optimize FL training by prioritizing clients that offer the most valuable contributions to model accuracy. It assesses clients based on their utility in improving model accuracy and their ability to train efficiently while preserving privacy. To select high-utility clients, `Oort` employs an online exploration-exploitation strategy that dynamically adapts the selection process to account for outliers and achieve a balance between statistical and system efficiency. Unlike other *synchronous* FL strategies, `Aergia` [39] freezes the computationally intensive part of a slow client's model and offloads it to a faster client that trains it using its own dataset. It leverages the spare computational capacity from robust clients and achieves high accuracy in relatively low training times by effectively matching clients' performance profiles and data similarity.

`SAFA` [30] is a *semi-asynchronous* training strategy that targets improved round efficiency and convergence, especially in scenarios with frequent client dropouts. It introduces innovative client selection and global aggregation methods, including a caching mechanism to prevent wasted client contributions. `FedAT` [41] uses a tiering mechanism that partitions clients into logical tiers based on their response latencies. Faster tiers update the model synchronously, while slower tiers send updates asynchronously. It employs a weighted aggregation approach to avoid bias toward faster tiers and uses compression to reduce communication costs. In [42], Xie et al. propose `FedAsync` [42], an *asynchronous* FL strategy that utilizes a parameter server architecture to synchronize and coordinate client invocations. It employs a scheduler thread to trigger client training and an updater thread for aggregating client updates into the global model. To address scalability concerns in practical FL systems, `FedBuff` [31] introduces buffered asynchronous aggregations. In this strategy, clients train and communicate asynchronously with the server, storing their updates in a buffer until a server update triggers aggregation once a specific number of client updates are collected. In [27], Jiang et al. propose `Pisces` [27], an *asynchronous* FL strategy that utilizes guided participant selection and adaptive aggregation pacing to mitigate slow clients. It merges techniques from `FedBuff` and `Oort` to enhance performance, focusing on prioritizing participants with high data quality as `Oort`. This enables more efficient utilization of clients compared to `FedBuff`.

Table I provides a comprehensive comparison between `Apodotiko` and the different strategies for straggler mitigation in FL. We differentiate these strategies based on five attributes: support for FaaS environments, asynchronous aggregation, performance-based selection, client-efficiency scoring, and adaptive penalty. FaaS support indicates compatibility with serverless environments. Asynchronous aggregation reflects the flexibility to separate client updates from training. Performance-based selection involves choosing clients based on their training duration. Client efficiency scoring accounts for hardware diversity during selection, while adaptive penalty reflects adjustments in client selection based on performance and availability over time. While `FedProx`, `FedNova`, `SCAFFOLD`, and `Aergia` primarily focus on optimizing the local training process and aggregation methods, they do not incorporate intelligent client selection to optimize round performance as done in `Apodotiko`. `SAFA` tracks the status of clients' local models to ensure their synchronization with the global model but tends to overutilize clients and lacks suitability for FaaS environments. `TiFL`, `FedAT`, and `FedlesScan` group clients based on training duration into clusters, but they overlook the hardware and data heterogeneity during the client selection process and lack support for asynchronous aggregation. Although `Oort` considers data size and training duration in client selection, it overlooks the correlation between these factors and diverse hardware configurations, enforcing a strict penalty on slower clients. `FedAsync` and `FedBuff` focus on optimizing FL with asynchronous aggregation but adopt random client selection. In contrast, `Pisces` combines the methods from `Oort` and `FedBuff`, refining the scoring approach, yet it still overlooks clients' efficiency during scoring (§III-C) and selection. `Apodotiko` overcomes these limitations by incorporating comprehensive scoring metrics that account for both hardware and data heterogeneity, ensuring intelligent client selection and efficient round performance.

III. APODOTIKO

In this section, we describe our strategy for *asynchronous* serverless FL in detail. First, we provide an overview of `Apodotiko`, followed by our methodology for *asynchronous aggregation* of model updates. After this, we present our novel client scoring strategy, followed by our probabilistic client selection technique.

A. Overview

The *FedLess* [7] framework includes a central component known as the controller, which oversees and orchestrates the entire FL training process. It incorporates a Strategy Manager subcomponent [8] that is responsible for controlling the behavior of the selected strategy. This involves managing client selection and choosing the aggregation technique used. To implement `Apodotiko`, we extend the Strategy Manager subcomponent in *FedLess*. In addition, to enable asynchronous aggregation (§III-B) and collect different client attributes required for our scoring strategy (§III-C), we modify the controller and client routines as shown in Algorithm 1. At the start of each FL training round, the controller runs the `Train_Global_Model` routine, while the selected clients execute the `Client_Update` routine to locally train the global model.

Algorithm 1: Modifed *FedLess* [7] controller and client routines.

```
1  Fedless Controller:
2  Function Train_Global_Model(clients, round):
3      client_selection = Select_Clients(clients, clientsPerRound)
4      Invoke selected clients
5      for each client in client_selection do
6          Save invocation record to database (client, round)
7          Set client invocation status to runnning // Busy client
8      end
9      while $\#results \geq (clientsPerRound * concurrencyRatio)$ do
10         Sleep(0.1)
11     end
12     Aggregate Model
13 return
14 Fedless Client:
15 Function Client_Update(hyperParameters, round):
16     Load model and dataset.
17     Start_Timer()
18     Train model
19     Stop_Timer()
20     Save updated model to database.
21     Add measured time to invocation record in database.
22     Update invocation status to complete. // Available client
23 return
```

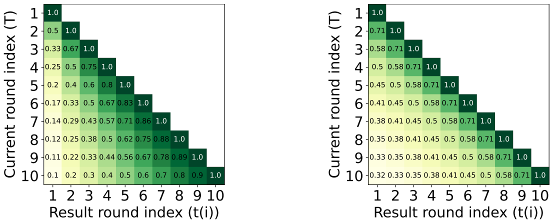
Initially, the controller selects the required number of clients using our client selection strategy (§III-D) and invokes them (Lines 3-4). Following this, we store the information regarding the invoked clients, such as the current round number, in the *FedLess* database (Line 6). Moreover, we mark the currently running clients as *busy* (Line 7). This is required to prevent selecting already running clients in the next FL training round. After this, the controller periodically checks for the availability of results from a fraction of clients in the database and then triggers the global model aggregation function (§III-B, §II-A) (Lines 9-12). On the client side, the selected clients retrieve the most recent global model along with their local datasets and perform local model updates for a specified number of configured epochs (Lines 16-19). Following this, they store the updated model and the measured training time in the database (Lines 20-21). Finally, we mark the finished clients as *complete* in the database, making them available for selection in the next FL training rounds (Line 22).

$$w_{t+1} \longleftarrow \sum_{i=1}^{N} \frac{t_i}{T} \times \frac{n_i}{n} w_{t_i}^{i} \tag{1}$$

$$w_{t+1} \longleftarrow \sum_{i=1}^{N} \frac{1}{(T - t_i + 1)^{0.5}} \times \frac{n_i}{n} w_{t_i}^{i} \tag{2}$$

B. Asynchronous Aggregation

In *synchronous* FL, implementing per-round timeouts is a common strategy to prevent exceedingly long round times [3], [7], [29]. This timeout mechanism ensures that the central server does not wait indefinitely for all clients to send their updates before initiating the global model aggregation. However, slow clients might push their local model updates to the parameter server after the completion of an FL training round. These updates, often discarded, can potentially contain valuable information that can improve the performance of the



(a) `FedLesScan`. (b) `Apodotiko`.

Fig. 2: Comparing weighting functions for aggregating stale client model updates.

global model. To reduce training times and improve convergence rates, we extend the *FedLess* [7] controller to aggregate client model updates asynchronously without waiting for all current client model updates to be available in the database. Towards this, the modified controller in `Apodotiko` only waits for a fraction of client updates, referred to as the *concurrencyRatio* ((0,1]), before invoking the aggregator function. For instance, with 100 clients per round and a ratio of 0.6, the controller only waits for updates from 60 clients. These client updates can be from the current or previous training rounds. Aggregating model updates from previous rounds can lead to reduced convergence rate and higher variance in the global model [42]. Moreover, the older the update, the higher the risk to the quality of the global model. To mitigate this, most *asynchronous* strategies utilize a staleness weighting function to dampen updates from previous rounds during aggregation. This weighting function should assign a weight of 1 to the current round's results and show a monotonically decreasing pattern with increasing round numbers. With `Apodotiko`, we experimented with different staleness functions shown in Equations 1 and 2. The former is used by `FedLesScan` [8], while the latter is adopted from [42]. In these equations, $w^i_{t_i}$ represents the local model of the client i at round t_i, while w_{t+1} is the global model after aggregation at round T. Furthermore, n_i represents the cardinality of the dataset at client i while n is the total cardinality of the aggregated clients. With Equation 1, we observe that the weight of one round of late updates gradually increases as the round number increases, as shown in Figure 2a. Moreover, in Figure 2a, the weight values derived from Equation 1 exhibit inconsistency for results with identical staleness levels, contrasting Figure 2b with Eq. 2, where the weight values maintain consistency along the diagonal axis. As a result, we use Eq. 2 with our strategy for aggregating model updates. In our experiments, we only considered client updates from a maximum of five previous rounds but observed that most delayed updates arrived within two rounds. Although conceptually similar, our *asynchronous aggregation* mechanism in `Apodotiko` differs from the buffering technique used in `FedBuff` [31]. In `FedBuff`, client model updates are stored in-memory, whereas in `Apodotiko`, we utilize an external database for this purpose. Furthermore, our approach to aggregating stale model updates differs from the method outlined in [31].

C. Scoring Clients

Prior FL training strategies [26], [41], including `FedLesScan` [8] only consider the training duration of clients for selection. This approach is suitable in homogeneous hardware environments, where training time correlates directly with a client's data size. However, this assumption no longer holds true in a heterogeneous setting, as shown in Figure 3. To this end, `Apodotiko` introduces a scoring-based probabilistic client selection approach considering the client's training duration and data size.

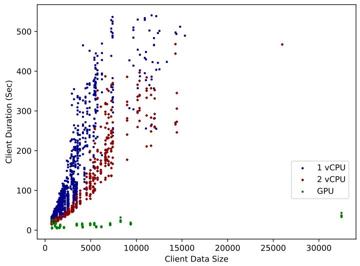


Fig. 3: Client training durations for different hardware resource configurations with non-IID data partitions for the Shakespeare dataset [32].

Algorithm 2: Weighted Average Client Score. β: client's current booster value; T: list of training duration for each round; N_c: local client data cardinality; E: number of local epochs; B: client local batch size; λ: global defined decay rate; k: FL training rounds.

```
1 Function Calculate_Score(β, T, N_c, E, B):
2     $\#updates \leftarrow \frac{N_c \times E}{B}$
3     $weighted\_sum \leftarrow \sum_{i=0}^{k-1} (\lambda^i \times (N_c \times \frac{\#updates}{T_i}))$
4     $weighted\_average\_score \leftarrow \beta \times \frac{weighted\_sum}{\sum_{i=0}^{k-1} \lambda^i}$ // multiple
       with β to promote clients
5 return weighted_average_score
```

To score clients, we collect five attributes during each training round: *training duration*, *local client data cardinality*, *batch size*, *number of local epochs*, and a *booster* value. Training duration represents the time required for executing *model.fit* (Algorithm 1: Line 17-19), excluding the time spent on network communication and model initialization. Data cardinality represents the total size of a client's local dataset. It is not considered confidential information in FL as it is essential for model aggregation [3] (§III-B). The booster value is a floating point number used in our strategy for promoting fairness during client scoring and selection.

Algorithm 2 describes our strategy for scoring clients. It generates a weighted average score by considering a client's participation in the different FL training rounds. Following this, the calculated scores are then used to select clients that participate in each training round as described in §III-D. Evaluating a client's machine learning performance solely based on hardware specifications such as micro-architecture, frequency, and core count can be challenging. Traditional methods often require benchmarking to obtain a score, which can be expensive and inefficient. Towards this, we utilize the client's training performance as a hardware benchmark score, referred to as the *Client Efficiency Score* (CEF). Our scoring algorithm calculates the CEF by determining the number of updates a client makes on the local model. This is done by multiplying the data size (N_c) with the epoch size (E) and dividing the result by the batch size (B) (Line

Algorithm 3: Client selection routine.

```
1  Function Select_Clients(clients, clientsPerRound):
2      Characterize clients as uninvoked_clients and invoked_clients
3      Exclude busy clients from invoked_clients
4      if #uninvoked_clients ≥ clientsPerRound then
5          return Randomly sample clientsPerRound from uninvoked_clients.
6      end
7      client_scores = []
8      for each client in invoked_clients do
9          Calculate Weighted Score for client.
10         Append Client Score to client_scores.
11     end
12     Calculate probability for all invoked_clients $\frac{score}{\sum client\_scores}$
13     client_selection ← Sample invoked_clients based on probability
14     Reset booster value for all clients in client_selection
15     Increase booster value for all clients NOT in client_selection
16 return client_selection
```

1). Following this, we calculate the number of updates per second by dividing the total number of updates by the training time (Line 2). A higher CEF score indicates a more powerful hardware resource configuration. Moreover, to account for data heterogeneity, we multiply the CEF score with N_c. To obtain a score that considers the performance of the client along with the duration of the FL training process, we utilize an exponentially decreasing weighting technique. This technique assigns the highest weight to a client's most recent result ($i = 0$) while gradually decreasing the weights for older results (Line 2). The weights are calculated using a globally defined decay rate denoted by λ. To promote fairness and participation from slow clients in training rounds, we introduce a booster value (β) that is multiplied by the final score before selection (Line 4). Initially, the booster value is set to 1 for all clients. If a client is available (not busy) (§III-A) but not selected for the training round, its booster value is increased by multiplying it with a promotion value greater than one. This ensures that slow clients have a higher probability of being selected in future rounds (§III-D). However, if a client is selected, its booster value is reset to one. The decay rate λ and promotion rate β are determined by an adjustment rate ρ in the range $0 < \rho \leq 1$. The adjustment rate controls the extent to which the score weight is increased or decreased. Specifically, the decay rate is calculated as $\lambda = 1 - \rho$, while the promotion rate is calculated as $\beta = 1 + \rho$. By default, the value of ρ is set to 0.2, which is also used in all of our experiments.

D. *Selecting Clients*

Algorithm 3 describes our strategy for client selection. The goal is to sample a given number of clients (*clientsPerRound*) from a list of available clients (*clients*). Initially, we differentiate between clients who have participated in at least one training round and those who have not yet been invoked (Line 2). Following this, we remove the currently running clients (§III-A) from the list of clients that have been invoked once (Line 3). Initially, our algorithm prioritizes uninvoked clients to gather data and enable scoring (§III-C). If sufficient uninvoked clients are available, the required number of clients is randomly selected from this pool (Lines 4-6). When the number of required clients exceeds the available uninvoked clients, we sample the remaining clients using our scoring strategy (§III-C). We calculate the score for each available client and append it to the score list. After calculating all scores, we normalize them into values between zero and one. Following this, we transform these normalized scores into probabilities by summing up all scores and dividing each client's normalized score by the total score (Line 12). The higher the client score, the higher the probability, and the more likely the client will be selected for the next round. If a GPU-based client has scored higher than a CPU client, its normalized probability will also be higher, giving it a greater chance of being selected. After obtaining the probability of each client, we randomly sample the required number of clients from the list based on the probability (Line 13). We reset the booster value to 1 for the selected clients so that we do not keep promoting them (Line 14). Finally, we increase the booster value (β) for the available (§III-A) but not selected clients by multiplying it with the promotion rate (Line 15).

IV. EXPERIMENTAL RESULTS

In this section, we present the performance results for our strategy `Apodotiko` against other FL training approaches across multiple datasets. For all our experiments, we follow best practices while reporting results.

A. *Experiment Setup*

1) Datasets: For our experiments, we utilize four datasets from various application domains, including image classification, speech recognition, and language modeling, to provide a conclusive evaluation of our strategy's effectiveness. The first dataset we use is the popular handwritten image database called *MNIST*. It contains 60,000 training images and 10,000 images for central evaluation. To simulate a non-IID setting with MNIST, we sort the images by label, split them into 300 shards of 200 images each, and distribute the shards across clients. From the `LEAF` FL benchmarking framework [32], we utilize the *FEMNIST* and the *Shakespeare* datasets. The *FEMNIST* dataset is an extended version of the *MNIST* dataset and contains over 800,000 images. On the other hand, the *Shakespeare* dataset consists of sentences from *The Complete Works of William Shakespeare*, each of length 80 characters. We employ existing non-IID data partitions available within `LEAF` for these datasets. From the `FedScale` [44] FL benchmark, we utilize the real-world *Google Speech* dataset. This dataset is designed to create simple and useful voice interfaces for applications that use common words like "Yes," "No," and directions. It consists of 105,000 1-second audio files distributed across $2,618$ clients.

2) Model Architectures and Parameters: For our experiments with the four datasets, we utilize different model architectures that have been used by several previous works in this domain [6], [7], [8], [32], [44]. For the *MNIST* dataset, we employ a two-layer Convolutional Neural Network (CNN) with a $5x5$ convolutional kernel. Each convolutional layer is followed by a $2x2$ max pooling layer. The model ends with a fully connected layer with 512 neurons and a ten-neuron output layer. The model comprises $582,026$ trainable parameters in total. Similar to *MNIST*, we use a 2-layer CNN for *FEMNIST*. However, in this case, the network concludes with a fully connected layer comprising 2048 neurons and an output layer containing 62 neurons, resulting in $6,603,710$ trainable model parameters. For the *Shakespeare* dataset, we use a Long Short Term Memory (LSTM) recurrent neural network. The model contains an embedding layer of size eight, followed by two LSTM layers with 256 units and an output layer with a size of 82. This model has $818,402$ trainable parameters. The model architecture for the *Google Speech* dataset consists of two identical blocks, followed by an average pooling layer and an output layer with 35 neurons. Each block consists of two convolutional layers with a $3x3$ convolutional kernel and a max-pooling layer. To prevent overfitting, a dropout layer follows the max-pooling layer, with a rate of 0.25. In this case, the model has $67,267$ trainable parameters. Across all layers, we use RelU as the activation function, except in the output layer, where the softmax function is utilized. The clients for the *MNIST*, *FEMNIST*, and the *Google Speech* datasets train for five local epochs with a batch size of ten, ten, and five respectively. On the other hand, for the *Shakespeare* dataset, the clients train for one local epoch with a batch size of 32 [7], [8]. For the *MNIST*, *FEMNIST*, and the *Google Speech* datasets, we use `Adam` as the optimizer with a learning rate of $1e-3$. On the other hand, for *Shakespeare*, we use `SGD` with a learning rate of 0.8.

3) Experiment Configuration: To effectively scale our experiments and eliminate potential bottlenecks, we set up *FedLess* on a dedicated virtual machine (VM) hosted on our institute's compute cloud. The VM is configured with 40vCPUs and 177GiB of RAM. This machine also hosts the file server, providing 200GiB of storage to accommodate the four datasets utilized in our study. We deployed the aggregator function (§II-A) on a self-hosted, single-node VM Kubernetes (K8s) cluster with OpenFaaS [22] as the FaaS platform. We configure the VM with 45GiB of RAM and 10vCPUs to ensure sufficient resources for efficient operation.

In all our experiments, we deploy the FaaS-based FL clients using the OpenFaaS platform based on K8s. We used OpenFaaS rather than commercial FaaS offerings since it provides us with maximum flexibility for configuring the different clients. In addition, none of the current commercial FaaS offerings support the execution of FaaS functions with GPUs. To enable GPU-based FL clients with OpenFaaS and K8s, we use the 4paradigm's K8s `vGPU scheduler` [45]. Standard K8s does not support fine-grained allocation or the sharing of GPUs, often leading to underutilization. In contrast, the `vGPU scheduler` balances GPU usage across nodes and allows users to allocate resources based on device memory and core usage, thereby increasing GPU utilization.

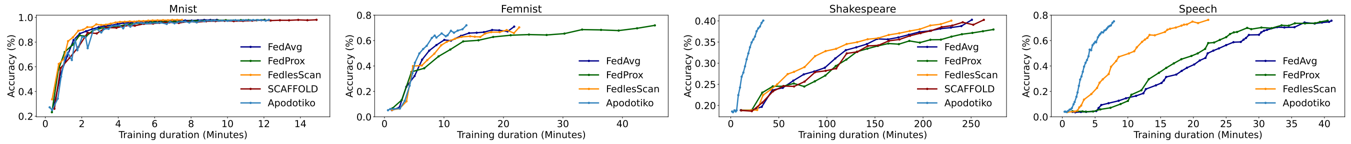
Across all datasets (§IV-A1), we use 200 clients and sample 100 clients per round unless otherwise specified. We configure 130 clients with 1vCPU and a memory limit of 2048MiB, 50 clients with 2vCPUs and a memory limit of 4096MiB, and an additional 20 clients spread across five Nvidia P100 GPUs, each configured with 0.4 vGPU [45]. This diverse client setup, ranging from CPU to GPU configurations, allows for a comprehensive exploration of our strategy's behavior and performance in heterogeneous environments, offering insights into its practical efficacy in serverless FL systems.

4) Baseline Strategies: To effectively evaluate `Apodotiko`, we compare it against five other strategies: `FedAvg` [3], `FedProx` [37], `SCAFFOLD` [29], `FedLesScan` [8], and `FedBuff` [31]. `FedProx` and `SCAFFOLD` are significantly popular conventional FL strategies in heterogeneous environments, while `FedBuff` is utilized in production at Meta [5]. For `Apodotiko`, we fix the *concurrencyRatio* to 0.3 in all our experiments unless otherwise specified.

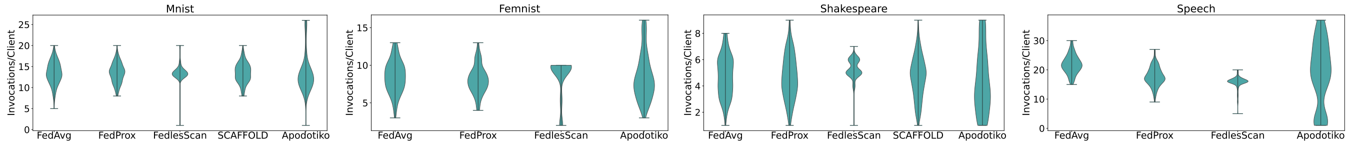
5) Evaluation Metrics: For comparing the different strategies (§IV-A4), we use metrics that cover four aspects: *model performance*, *client selection bias*, *cold start ratio*, and *strategy efficiency*. To evaluate model performance, we calculate model accuracy and track the accuracy progress throughout the FL training process. To ensure a fair evaluation of the newest global model, we use a distributed evaluation approach. Towards this, we randomly select clients after each FL training round to evaluate the global model on their test datasets. Following this, we calculate a weighted average of the obtained accuracy values from the different clients to obtain the final model accuracy. Client selection bias represents the variations in client invocations during the FL training process, offering insights into the effectiveness of the selection strategy. We quantify bias by calculating the difference between the least-called and most-called clients [8], [30]. Low bias is desirable for scenarios with minimal stragglers, while prioritizing reliable clients may be necessary for straggler-heavy environments, leading to increased bias. A key characteristic of the FaaS computing model is `scale-to-zero`, i.e., idle function instances are automatically terminated if there are no function invocation requests within a given time frame. In serverless FL, this can cause cold starts for client function instances, leading to increased training durations and expenses [7], [8], [9]. To compute the cold start ratio, we calculate the total number of client cold start invocations and divide it by the total number of invocations across all clients. We differentiate between cold and warm start invocations by monitoring the client function instances in our K8s cluster. In our experiments, we configured the function instances to scale down after remaining inactive for ten minutes. To assess the efficiency of our strategy, we calculate the total training time and costs. The total time represents the duration required to achieve the target global model accuracy. For estimating costs for our experiments, we use the cost model provided by GCP [46], [47]. or CPU-based clients, we calculate costs based on the allocated memory, CPU, and function duration. In contrast, for GPU clients, we calculated costs by considering the hourly rate of the GPU model and the proportion of GPU resources utilized during the training process.

B. Comparing Accuracy

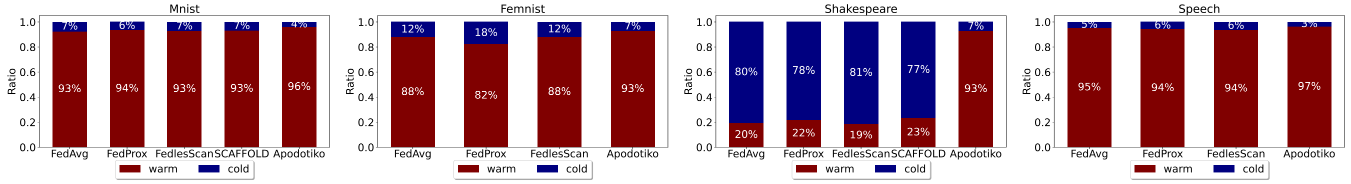
In this subsection, we focus on demonstrating the improved convergence rate of `Apodotiko` as compared to other FL strategies rather than pursuing state-of-the-art model accuracies on these tasks. Towards this, we limit target model accuracies to 0.98 for *MNIST*, 0.70 for *FEMNIST*, 0.40 for *Shakespeare*, and 0.75 for the *Google Speech* dataset. Figure 4a illustrates the model accuracies throughout the FL training



(a) Comparing *model accuracy*.

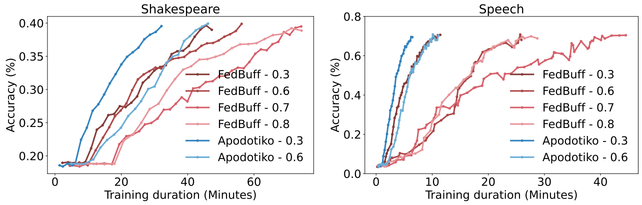


(b) Comparing *client selection bias*.



(c) Comparing *cold start ratios*.

Fig. 4: Comparing different evaluation metrics across the different FL strategies.



(a) *Shakespeare*. (b) *Google Speech*.

Fig. 5: Comparing `Apodotiko` with `FedBuff`.

process across various strategies. For the *MNIST* dataset, we observe that all strategies have similar performance, while `FedlesScan` sightly outperforms other training strategies. This can be attributed to two reasons. First, *MNIST* is a relatively small dataset with low training iteration durations. Second, *MNIST*'s balanced non-IID data distribution allows `FedLesScan` to benefit from its client clustering strategy based solely on training times. On the other hand, for the *FEMNIST* dataset, we observe that `FedAvg` performs better than `FedlesScan` due to unbalanced non-IID data distributions and increased training times. `Apodotiko` outperforms all other strategies and achieves the target model accuracy with a speedup of 1.73x compared to `FedAvg`. For the *Shakespeare* dataset, we observe a significant performance difference across the different training strategies. `Apodotiko` outperforms all other strategies with a speedup of 7x as compared to `FedAvg`, 6.63x against `FedLesScan`, 7.2x against `SCAFFOLD`, and 7.82x against `FedProx`. The non-IID data partitions provided by LEAF for the *Shakespeare* dataset introduce significant variations in training durations across different hardware resource configurations, as shown in Figure 3. This variance can lead to situations where clients with limited resources might fail to complete their training before a FL round ends, resulting in wasted contributions. However, our client selection strategy (§III-C), driven by scoring based on data size and hardware resources, prioritizes clients with larger data sizes and more powerful hardware, ensuring greater contributions to the global model. Moreover, our asynchronous aggregation technique with a stateless weighting function (§III-B) accommodates late contributions, leading to better global model accuracy. These two factors combined contribute to the superior performance of `Apodotiko`. Similarly, for the *Google Speech* dataset, `Apodotiko` achieves a speedup of 6.19x compared to `FedAvg` and 3.3x compared to `FedLesScan`. In our experiments, `SCAFFOLD` did not converge for the *FEMNIST* and the *Speech* datasets. As a result, we omit it in Figure 4.

While Figure 4a compares `Apodotiko` with *synchronous* and *semi-asynchronous* strategies, Figure 5 presents a comparative analysis with the *asynchronous* strategy `FedBuff` [31] for the *Shakespeare* and *Google Speech* datasets. For `FedBuff`, we vary the buffer ratio from 0.3 to 0.8, while for our strategy, we present results for the *concurrencyRatios* (CR) of 0.3 and 0.6. For the *Shakespeare* dataset, our strategy using CRs of 0.3 and 0.6 outperforms `FedBuff` with a buffer ratio of 0.3, achieving a speedup of 1.43x and 1.06x respectively. Similarly, for the *Google Speech* dataset, our strategy achieves a speedup of 1.74x and 1.01x with CRs of 0.3 and 0.6 respectively, compared to `FedBuff` with a buffer ratio of 0.3. The better performance of our strategy can be attributed to our intelligent client selection methodology that prioritizes clients that contribute more to the global model, in contrast to `FedBuff`, which selects clients randomly.

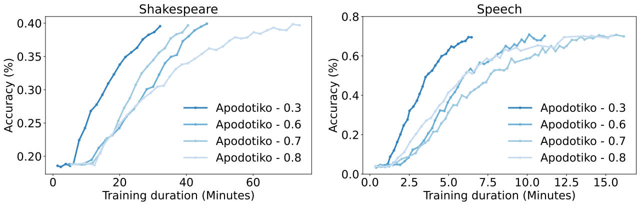
.

C. Comparing Client Selection Bias and Cold Start Ratios

Figure 4b presents insights into the client selection biases among various FL training strategies across multiple datasets. Using violin plots, we visualize a distribution based on the number of invocations for each client (y-axis). The height difference between the highest and lowest points in the distribution represents the degree of bias. A greater height indicates a stronger bias towards a specific subset of clients, while a lower height suggests a more balanced distribution of invocations. Moreover, a wider width in the distribution indicates that certain clients were invoked more frequently. Across all datasets, we observe that `FedAvg`, `FedProx`,

Strategy \ Dataset		**MNIST (min)**		**FEMNIST (min)**		**Shakespeare (min)**		**Speech (min)**	
FedAvg [3]		10.98	(1.00x)	22.44	(1.00x)	245.98	(1.00x)	49.78	(1.00x)
FedProx [37]		15.03	(0.73x)	39.46	(0.57x)	273.58	(0.90x)	53.20	(0.94x)
FedLesScan [8]		**9.69**	**(1.13x)**	25.88	(0.87x)	232.18	(1.06x)	26.59	(1.87x)
SCAFFOLD [29]		14.31	(0.77x)		-	252.07	(0.98x)		-
Apodotiko	CR = 0.3	11.83	(0.93x)	**12.95**	**(1.73x)**	**34.98**	**(7.03x)**	**8.04**	**(6.19x)**
	CR = 0.6	11.65	(0.94x)	18.28	(1.23x)	69.04	(3.56x)	12.18	(4.09x)
	CR = 0.7	12.21	(0.90x)	20.21	(1.11x)	51.28	(4.80x)	12.74	(3.91x)
	CR = 0.8	10.92	(1.01x)	22.72	(0.99x)	72.66	(3.39x)	16.42	(3.03x)

TABLE II: Comparing total training duration (min) across various FL strategies and datasets. The highlighted values represent the best-performing strategy for a particular dataset.



(a) *Shakespeare*. (b) *Google Speech*.

Fig. 6: Impact of different *concurrencyRatios* (CRs).

and `SCAFFOLD` show normal distributions due to the random selection of clients. For the *MNIST* dataset, we observe that `FedLesScan` appears more centralized, indicating a balanced allocation of training among clients, with only a few outliers. In contrast, our strategy shows a relatively normal distribution of client invocations, with most clients being invoked between five and 15 times. However, the line stretches out more than other strategies, with some clients receiving over 25 invocations and few with as little as five. As the *MNIST* dataset involves balanced non-IID data distributions among clients, the scoring strategy in `Apodotiko` primarily differentiates based on the training time, which is mainly influenced by clients' hardware resources. As a result, GPU-based clients receive more frequent invocations than other clients. For the *FEMNIST* dataset, we observe that `FedLesScan` is more concentrated in the middle as it prioritizes clients with lower training durations. Moreover, it overlooks clients with larger data sizes and limited computational resources, often characterizing them as stragglers and selecting them less frequently in training. In contrast, our strategy maintains a relatively balanced distribution due to the probabilistic client selection approach described in §III-D. This method ensures that every client retains a chance of being selected, even those with longer training times. For the *Shakespeare* dataset, `FedLesScan` shows a distribution similar to that for the *FEMNIST* dataset. On the other hand, `Apodotiko` exhibits a slightly fatter end on the bottom compared to the *FEMNIST* dataset. This distribution arises from the significant differences in training durations observed across various hardware resource configurations, as shown in Figure 3 (§IV-B). For the *Google Speech* dataset, `FedLesScan` shows a normal distribution, similar to the results for the *MNIST* dataset, but with lower variance compared to both `FedAvg` and `FedProx`. In contrast, our strategy portrays a distribution distinct from the one seen in the *MNIST* dataset. This divergence stems from the unbalanced non-IID data distribution in the *Google Speech* dataset, causing client data size to hold a more significant role in the scoring process.

Figure 4c compares the cold start ratios among different FL strategies across multiple datasets. We observe that cold starts are significantly limited for the *MNIST* and *Google Speech* datasets due to the relatively brief round durations and the short intervals between each round and client invocation. These factors contribute to a higher probability of a client being invoked again within a short timeframe, consequently minimizing the occurrence of cold starts. However, for the *FEMNIST* and the *Shakespeare* datasets, cold starts are more prominent due to larger average training round durations. In our experiments, we observe that `Apodotiko` consistently achieves a low cold start ratio across all datasets. This can be attributed to two reasons. First, the *asynchronous aggregation* strategy used in `Apodotiko` triggers the invocation of the next round of clients as soon as a portion of the results becomes available. Second, our strategy incorporates a well-designed promotion mechanism (§III-C) that effectively prevents clients from missing multiple rounds over an extended period.

Strategy \ Dataset		**MNIST (USD)**	**FEMNIST (USD)**	**Shakespeare (USD)**	**Speech (USD)**
FedAvg [3]		1.13	**2.74**	8.86	2.14
FedProx [37]		1.90	3.83	10.63	2.37
FedLesScan [8]		**1.11**	3.68	10.28	**1.85**
SCAFFOLD [29]		1.52	-	8.41	-
Apodotiko	CR = 0.3	11.97	5.99	**6.68**	2.72
	CR = 0.6	7.65	9.05	8.91	4.05
	CR = 0.7	7.35	4.92	9.47	3.91
	CR = 0.8	5.94	9.71	11.11	3.14

TABLE III: Comparing total training cost (USD) across various FL strategies and datasets. The highlighted values represent the minimum costs for a particular dataset.

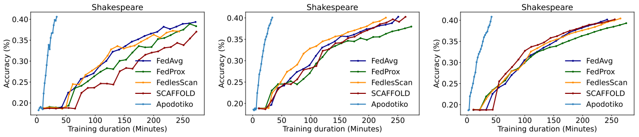
D. *Comparing Time and Cost*

Table II compares the total training time for the different FL strategies across multiple datasets. In our experiments, we observe that `Apodotiko` consistently outperforms other training strategies, particularly with a CR value of 0.3. Table III compares the total training cost in USD for the different FL strategies and datasets. Although `Apodotiko` isn't the most cost-effective strategy, it remains competitive with other methods. For instance, our strategy with a CR of 0.3 incurs a total training cost of 6.68 USD for the *Shakespeare* dataset and 2.72 USD for the *Google Speech* dataset. The increased costs can be attributed to the invocation of more clients, a consequence of the asynchronous nature of our strategy.

E. *Sensitivity Analysis*

We examine `Apodotikos`' effectiveness across various environments and configurations.

Impact of different *concurrencyRatios*. Figures 6a and 6b show the effect of different CRs (§III-B) on `Apodotiko` for the *Shakespeare* and the *Google Speech* datasets respectively. Our experiments on both datasets demonstrate that `Apodotiko` with a CR of 0.3 exhibits the fastest convergence rate compared to other concurrency ratios. For the *Shakespeare* dataset, we observe a speedup of 1.34x with a CR of 0.3 compared to the ratio of 0.6. For the *Speech* dataset, this speedup factor further increases to 1.7x. This accelerated convergence is attributed to the controller's ability to trigger model aggregation with only 30 clients, significantly reducing the time between model updates. Additionally, our client selection algorithm (§III-D) and stale weight aggregation function (§III-B) effectively identify and select high-quality clients, ensuring that the aggregated model does not diverge.



(a) 50/200 clients. (b) 100/200 clients. (c) 200/200 clients.

Fig. 7: Impact of different client sample sizes per round.

Table II and III also highlight the performance/cost of our strategy with different CR values against other FL approaches.

Impact of different client sample sizes per round.. Figures 7a, 7b, and 7c show the performance of the different FL strategies with 50, 100, and 200 clients selected for training in each round for the *Shakespeare* dataset. Our experiments show that `FedAvg` and `FedProx` perform similarly regardless of the client sample size. Moreover, we observe `FedLesScan` performs best with 100 clients per round. This could be attributed to smaller cluster sizes, enhancing the probability of choosing sufficient clients from the same cluster for each training round. In contrast, we observe that `SCAFFOLD` performs better with increasing client sample size as the global variant [29] becomes more accurate with more clients' results. Our observations about `SCAFFOLD` also align with the results presented in [43]. `Apodotiko` shows no significant impact on convergence rate with different sample sizes, as it only waits for a specific ratio of clients to complete the training before aggregating results and invoking new clients from the pool. We omit results for other datasets dues to space constraints but observe similar results.

V. CONCLUSION AND FUTURE WORK

In this paper, we presented `Apodotiko`, a novel *asynchronous* scoring-based strategy that enables efficient serverless FL across clients with varying hardware resource configurations. We comprehensively evaluated our strategy against five other popular FL training approaches on multiple datasets. Our experiments highlight that `Apodotiko` converges faster and consistently minimizes cold starts in client function invocations. In the future, we plan to investigate the usage of an adaptive *concurrencyRatio* based on the historical behavior of selected clients. This adaptive approach would potentially enable the inclusion of more results in the aggregation round by slightly delaying the model aggregation, thus preventing results from becoming stale.

VI. ACKNOWLEDGEMENTS

The research leading to these results was funded by the German Federal Ministry of Education and Research (BMBF) in the scope of the Software Campus program under the grant agreement 01IS17049.

REFERENCES

[1] B. M. Gaff *et al.*, "Privacy and big data," *Computer*, vol. 47, no. 6, pp. 7–9, 2014.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[3] B. McMahan *et al.*, "Communication-efficient learning of deep networks from decentralized data," in *Artificial Intelligence and Statistics*. PMLR, 2017, pp. 1273–1282.

[4] N. Rieke *et al.*, "The future of digital health with federated learning," *NPJ digital medicine*, vol. 3, no. 1, p. 119, 2020.

[5] D. Huba *et al.*, "Papaya: Practical, private, and scalable federated learning," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 814–832, 2022.

[6] M. Chadha *et al.*, "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

[7] A. Grafberger *et al.*, "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

[8] M. Elzohairy *et al.*, "Fedlesscan: Mitigating stragglers in serverless federated learning," in *2022 IEEE International Conference on Big Data (Big Data)*, 2022, pp. 1230–1237. [Online]. Available: https://doi.org/10.1109/BigData55660.2022.10021037

[9] K. Jayaram *et al.*, "λ-fl: Serverless aggregation for federated learning," 2022. [Online]. Available: http://tinyurl.com/3pkd2zks

[10] K. R. Jayaram *et al.*, "Just-in-time aggregation for federated learning," in *2022 30th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2022, pp. 1–8.

[11] K. Jayaram *et al.*, "Adaptive aggregation for federated learning," in *2022 IEEE International Conference on Big Data (Big Data)*, 2022, pp. 180–185.

[12] N. Kotsehub *et al.*, "Flox: Federated learning with faas at the edge," in *2022 IEEE 18th International Conference on e-Science (e-Science)*, 2022, pp. 11–20.

[13] J. Gu, Y. Zhu, P. Wang, M. Chadha, and M. Gerndt, "Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 635–644. [Online]. Available: https://doi.org/10.1145/3605573.3605638

[14] M. Chadha, P. Khera, J. Gu, O. Abboud, and M. Gerndt, "Training heterogeneous client models using knowledge distillation in serverless federated learning," *arXiv preprint arXiv:2402.07295*, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.07295

[15] C. P. Smith, A. Jindal, M. Chadha, M. Gerndt, and S. Benedict, "Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters," in *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, 2022, pp. 17–25. [Online]. Available: https://doi.org/10.1109/ICFEC54809.2022.00010

[16] A. Jindal, M. Chadha, S. Benedict, and M. Gerndt, "Estimating the capacities of function-as-a-service functions," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC '21. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3492323.3495628

[17] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen, "Function delivery network: Extending serverless computing for heterogeneous platforms," *Software: Practice and Experience*, vol. 51, no. 9, pp. 1936–1963, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2966

[18] A. Jindal, J. Frielinghaus, M. Chadha, and M. Gerndt, "Courier: Delivering serverless functions within heterogeneous faas deployments," in *2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC'21)*, ser. UCC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3468737.3494097

[19] A. Jindal, M. Chadha, M. Gerndt, J. Frielinghaus, V. Podolskiy, and P. Chen, "Poster: Function delivery network: Extending serverless to heterogeneous computing," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 1128–1129. [Online]. Available: https://doi.org/10.1109/ICDCS51616.2021.00120

[20] 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

[21] M. Kiener, M. Chadha, and M. Gerndt, "Towards demystifying intra-function parallelism in serverless computing," in *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, ser. WoSC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 42–49. [Online]. Available: https://doi.org/10.1145/3493651.3493672

[22] OpenFaaS, “OpenFaaS - Serverless Functions Made Simple,” 2019. [Online]. Available: https://www.openfaas.com/https://docs.openfaas.com/

[23] Google Cloud, “Cloud Functions Second Generation— Google Cloud,” 2022. [Online]. Available: http://tinyurl.com/3yapa4pv

[24] P. Castro *et al.*, “The rise of serverless computing,” *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[25] K. Hsieh *et al.*, “The non-iid data quagmire of decentralized machine learning,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 4387–4398.

[26] Z. Chai *et al.*, “Tifl: A tier-based federated learning system,” in *Proceedings of the 29th international symposium on high-performance parallel and distributed computing*, 2020, pp. 125–136.

[27] Z. Jiang *et al.*, “Pisces: Efficient federated learning via guided asynchronous training,” in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 370–385.

[28] A. K. Sahu *et al.*, “On the convergence of federated optimization in heterogeneous networks,” *arXiv preprint arXiv:1812.06127*, vol. 3, p. 3, 2018.

[29] S. P. Karimireddy *et al.*, “Scaffold: Stochastic controlled averaging for on-device federated learning,” *CoRR*, vol. abs/1910.06378, 2019. [Online]. Available: http://arxiv.org/abs/1910.06378

[30] W. Wu *et al.*, “Safa: A semi-asynchronous protocol for fast federated learning with low overhead,” *IEEE Transactions on Computers*, vol. 70, no. 5, pp. 655–668, 2020.

[31] J. Nguyen *et al.*, “Federated learning with buffered asynchronous aggregation,” *CoRR*, vol. abs/2106.06639, 2021. [Online]. Available: https://arxiv.org/abs/2106.06639

[32] S. Caldas *et al.*, “LEAF: A Benchmark for Federated Settings,” in *Workshop on Federated Learning for Data Privacy and Confidentiality, NeurIPS*, 2018, pp. 1–9.

[33] “Fedless: Secure and scalable serverless federated learning,” https://osseu2023.sched.com/event/1OGe2.

[34] P. Moritz *et al.*, “Ray: A distributed framework for emerging AI applications,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577.

[35] V. Mothukuri *et al.*, “A survey on security and privacy of federated learning,” *Future Generation Computer Systems*, vol. 115, pp. 619–640, feb 2021.

[36] R. Chard *et al.*, “Funcx: A federated function serving fabric for science,” in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 65–76. [Online]. Available: https://doi.org/10.1145/3369583.3392683

[37] T. Li *et al.*, “Federated optimization in heterogeneous networks,” 2020.

[38] J. Wang *et al.*, “Tackling the objective inconsistency problem in heterogeneous federated optimization,” 2020.

[39] B. Cox *et al.*, “Aergia: leveraging heterogeneity in federated learning systems,” in *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, 2022, pp. 107–120.

[40] F. Lai *et al.*, “Oort: Informed participant selection for scalable federated learning,” *CoRR*, vol. abs/2010.06081, 2020. [Online]. Available: https://arxiv.org/abs/2010.06081

[41] Z. Chai *et al.*, “Fedat: 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*, 2021, pp. 1–16.

[42] C. Xie, S. Koyejo, and I. Gupta, “Asynchronous federated optimization,” *CoRR*, vol. abs/1903.03934, 2019. [Online]. Available: http://arxiv.org/abs/1903.03934

[43] Q. Li *et al.*, “Federated learning on non-iid data silos: An experimental study,” in *IEEE International Conference on Data Engineering*, 2022.

[44] F. Lai *et al.*, “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.

[45] S. Li, Pei and Zheng, “4paradigm/k8s-vgpu-scheduler: Open aios vgpu scheduler for kubernetes,” 2022. [Online]. Available: https://github.com/4paradigm/k8s-vgpu-scheduler

[46] (2023) Pricing, compute engine: Virtual machines (vms), google cloud. [Online]. Available: https://cloud.google.com/compute/all-pricing?authuser=2#gpus

[47] (2023) Pricing, cloud functions, google cloud. [Online]. Available: https://cloud.google.com/functions/pricing