LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Studienabschlussarbeiten
Fakultät für Mathematik, Informatik und Statistik

Jensen, Jan Alexander:

Enabling Efficient Serverless Federated Learning in Heterogeneous Environments

Masterarbeit, Sommersemester 2023

Fakultät für Mathematik, Informatik und Statistik
Institut für Informatik

Ludwig-Maximilians-Universität München

https://doi.org/10.5282/ubm/epub.96273

DEPARTMENT OF INFORMATICS

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Enabling Efficient Serverless Federated Learning in Heterogeneous Environments

Effizientes serverloses föderiertes Lernen in heterogenen Environments

Author:	Jan Alexander Jensen
Supervisor:	Prof. Dr. Michael Gerndt (TUM)
Advisor:	M.Sc. Mohak Chadha (TUM)
Submission Date:	16.05.2023

DEPARTMENT OF INFORMATICS

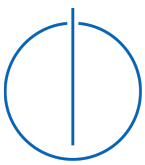
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Enabling Efficient Serverless Federated Learning in Heterogeneous Environments

Effizientes serverloses föderiertes Lernen in heterogenen Environments

Author:	Jan Alexander Jensen (LMU)
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	M.Sc. Mohak Chadha
Submission Date:	16.05.2023



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

Munich, 16.05.2023 Jan Alexander Jensen

Acknowledgments

I am deeply grateful to Prof. Dr. Michael Gerndt, my supervisor, for giving me the opportunity to work on such an exhilarating project and for creating a conducive work atmosphere. I would like to extend my sincere appreciation and thanks to my advisor, Mohak Chadha, for his unwavering guidance and patience during the thesis. His support has been instrumental in bringing me to this point. I would also like to acknowledge Mohamed Elzohairy for his previous contributions and help during the early stages of the project.

Abstract

Federated Learning (FL) has emerged as a promising approach to distributed machine learning. It enables multiple clients to collaborate in training models without compromising their data privacy by avoiding a centralized server. Previous research has suggested that Function-as-a-Service (FaaS) platforms can effectively address certain challenges, thereby facilitating an efficient training process among heterogeneous clients. However, the heterogeneous computational capabilities of clients may lead to stragglers, impeding scalability and prolonging the runtime. In our work, we propose an asynchronous score-based strategy, namely *FedlesScore*, which accommodates variable client hardware and data sizes, thereby mitigating the impact of stragglers on the system's overall performance. We extensively evaluate our strategy and compare it to novel FL approaches on four different datasets; *FedlesScore* achieves an average speedup in training time of 2.75x while incurring only a minor increase in cost. Moreover, the proposed strategy significantly reduces client cold starting, with an average reduction factor of four.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Research Objectives . . . 5
1.3 Thesis Overview . . . 5

2 Theoretical Background **7**
2.1 Federated Learning (FL) . . . 7
2.1.1 Client Selection . . . 8
2.1.2 Model Aggregation . . . 9
2.1.3 Challenges . . . 9
2.2 Serverless Computing . . . 10
2.2.1 Function As a Service (FaaS) . . . 11
2.2.2 Kubernetes . . . 11

3 Related Work **13**
3.1 Serverless Federated Learning . . . 13
3.2 Stragglers in Federated Learning . . . 14
3.2.1 Asynchronous Federated Learning . . . 17
3.3 Stragglers in Serverless Federated Learning . . . 19
3.4 Strategy Comparison . . . 20

4 System and Strategy Design **22**
4.1 Fedless . . . 22
4.1.1 Enabling GPU on Fedless . . . 24
4.1.2 Mock Cold Start . . . 26
4.2 FedlesScore . . . 28
4.2.1 Asynchronous Aggregation . . . 29
4.2.2 Score-based selection . . . 30

5 Experiments **35**
5.1 Experiment Setup 35
5.1.1 Benchmarks and Datasets 36
5.1.2 Model Configuration 36
5.1.3 Evaluation Metrics 38
5.2 Accuracy and Model Performance 40
5.3 Client Selection Bias 42
5.4 Cold Start Ratio 45
5.5 Client Size 46
5.6 Ablation Studies 49
5.7 Time & Cost Analysis 51
5.8 Discussion 54

6 Conclusion and Future Work **55**

List of Figures **56**

List of Tables **57**

Bibliography **58**

1 Introduction

In recent years, Federated learning (FL) has emerged as a promising approach to distributed machine learning. It allows multiple clients to train a machine learning model collaboratively without sharing their private data with a central server.

Instead of computing on large, centralized machines, federated learning (FL) distributes models over multiple devices for computation. Subsequently, it summarizes the changes as a small update, typically containing the model parameters and corresponding weights. This enables the devices to collaboratively learn a shared prediction model while keeping all the training data on the device, decoupling the ability to do machine learning from the need to store all the data on a central server. [1]

As federated learning provides privacy and evades using considerable bandwidth for data transfer, it requires each party to manage their instance and network to train and update the model. That is precisely where serverless computing comes into play. Serverless computing is a model of cloud computing that provides a miniature architecture where the end customer does not need to deploy, configure or manage server services. Leveraging serverless environments can bring about advantages for FL in terms of resource efficiency and cost-effectiveness in many settings. In conventional FL, only a subset of clients participates in the training round, leaving the remaining clients idle and consuming hardware resources. By contrast, a FaaS environment ensures that clients only utilize hardware resources when participating in the current training round, thereby reducing resource usage and associated costs. Additionally, FaaS offers a solution to the challenges of rapid scaling and infrastructure management for clients, which are core benefits of serverless architectures. This translates to a more straightforward setup and maintenance of a serverless-based FL platform. Accordingly, Chadha [2] and Grafberger [3] have composed a framework to migrate FL into serverless, called *Fedless*. Furthermore, it also provides additional security to protect against insecure serialization and injection attacks [3].

While providing multiple benefits, FL also brings its challenges. The heterogeneity of client computation speed negatively affects the scalability and significantly slows down its runtime due to the presence of stragglers. In previous work *FedLesScan* [4], a clustering-based strategy has been proposed to mitigate the effect of stragglers in serverless-based FL. We now want to incorporate GPU into the current system.

GPUs were designed from the beginning to be used almost exclusively for rendering

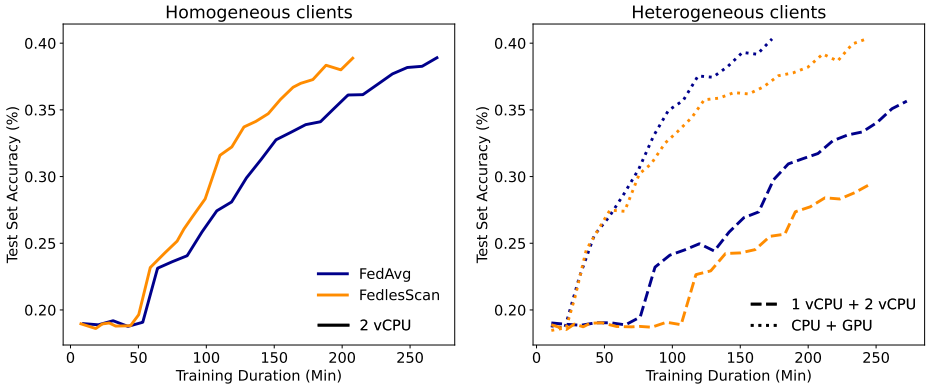
high-resolution images and graphics, which does not require much context switching. Instead, GPUs focus on concurrency or breaking down complex tasks into smaller subtasks that can be executed continuously alongside each other. Since GPUs are good at solving complex tasks and are widely used for machine learning, we would also like to leverage them. To do so, we want to enable GPU usage for *Fedless* by migrating the current function calls to assigning the task to the GPUs on the machine. Then, we want to investigate the straggler effect further after enabling GPUs. Moreover, we would like to extend the current strategy from *FedLesScan* to be even more resilient against stragglers.

The next chapter will discuss the challenges and related works tackling these issues. After that, we will present the possible approach to improve the current version of the system and how to evaluate the difference.

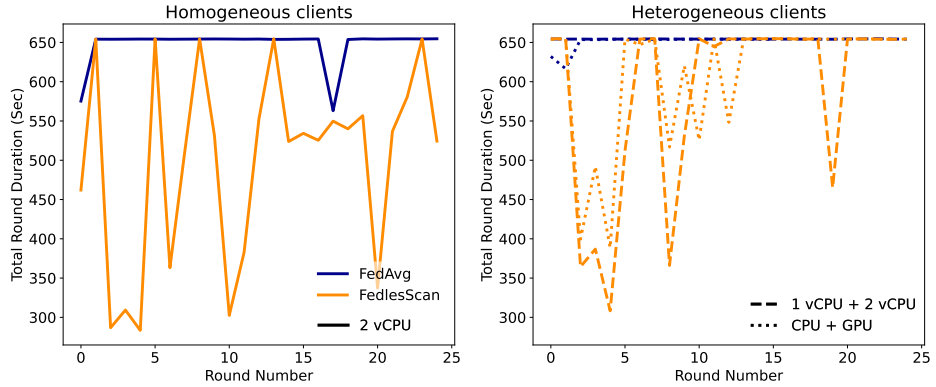
1.1 Problem Statement

The straggler effect is one of the main bottlenecks in federated learning. All selected clients who have completed their tasks must wait for the slowest client to end the current round, significantly increasing training time and cost. A recently proposed strategy, *FedlesScan*, has demonstrated effectiveness in mitigating the straggler effect, which can significantly hinder the performance of federated learning systems. However, *FedlesScan* primarily targets homogeneous clients and relies on clustering based on training duration. This approach fails to accommodate the increasing heterogeneity in client hardware and data size, leading to inadequate cluster sizes for sampling clients in each round. Consequently, stragglers are inadvertently included in the round, negating the benefits of *FedlesScan*.

Figure 1.1 illustrates the deficiency of *FedlesScan* in heterogeneous hardware and data size settings. To demonstrate this drawback, we conducted experiments by deploying 100 client functions on *OpenfaaS* [5] and managed the experiment with *Fedless* [3]. 50 clients were selected per round in various hardware settings and ran the experiment for 25 rounds. Specifically, we used clients with three different hardware resource distributions with the same heterogeneous data distribution. Initially, we ran both *FedAvg* and *FedlesScan* in a homogeneous hardware setting, where all 100 clients had 2 vCPUs. Subsequently, we added two scenarios, one with GPUs and one without, to test the performance of *FedlesScan* in heterogeneous environments. In the first scenario, we had a mix of 60 clients with 1 vCPU and 40 clients with 2 vCPUs, while in the second scenario, we had a mix of 50 clients with 1 vCPU, 30 clients with 2 vCPUs, and 20 clients running on a GPU. These scenarios were designed to represent a more realistic setting where clients have varying levels of computational resources. The experiments



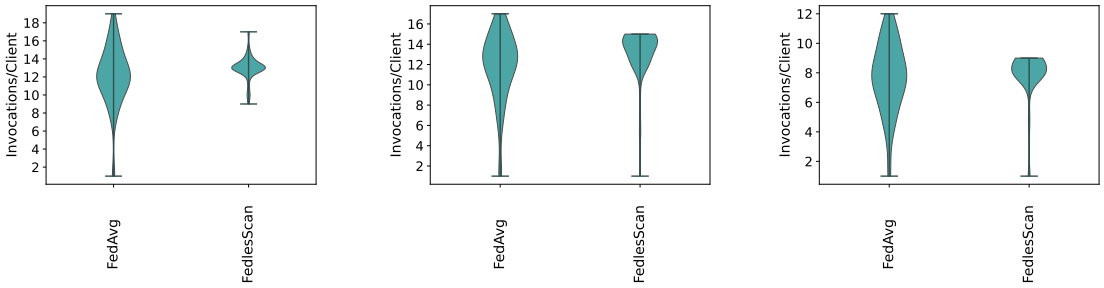
(a) Accuracy comparison across different client hardware settings



(b) Round Duration comparison across different client hardware settings

Figure 1.1: Training efficiency Comparison between *FedAvg* [1] and *FedlesScan* [4] in different hardware settings on the Shakespeare dataset

were conducted to demonstrate the impact of heterogeneous hardware settings on the performance of *FedlesScan* compared to *FedAvg*. These experiments provide insights into the impact of heterogeneous hardware settings on *FedlesScan* performance and highlight the importance of selecting appropriate hardware configurations to improve system performance. As depicted in Figure 1.1a demonstrates that both *FedAvg* and *FedlesScan* achieve an accuracy of 0.4, but *FedlesScan* requires 30% less time than *FedAvg*. However, as client hardware becomes more heterogeneous, *FedlesScan* struggles and falls behind *FedAvg*. For instance, when we add clients with 1 vCPU to the mixture, *FedlesScan* requires 40% more training time than *FedAvg* to achieve an accuracy of 0.3. Furthermore, with additional GPU clients, both strategies attain an accuracy of 0.4



(a) 2 vCPU (b) 1 vCPU + 2 vCPU (c) 1 vCPU + 2 vCPU + GPU

Figure 1.2: Selection Bias across different client hardware settings

within 25 rounds, but *FedAvg* is 43% faster in terms of total training time. Figure 1.1b provides a more detailed breakdown of the training duration for each round. In the homogeneous setting, *FedlesScan* succeeds in clustering clients with sufficient cluster size to select clients for each round and shows a high peak of round time only every few rounds when stragglers are selected. However, in the heterogeneous setting displayed on the right of Figure 1.1b, we see more rounds that max out and have similar round durations as *FedAvg*. This is because the clustering approach fails due to insufficient clients per cluster for selection in each round, and stragglers are selected as substitutes.

Figure 1.2 illustrates the distribution of invocations per client for *FedAvg* and *FedlesScan* on the *Shakespeare* dataset. In the first scenario (Figure 1.2a), with a homogeneous hardware setup where all clients have 2 vCPU, the invocation is fairly distributed among all clients, and it is also much more concentrated than in *FedAvg* with random selection. However, as we increase the heterogeneity among clients' hardware, as shown in Figure 1.2b and Figure 1.2c, we observe that the invocations of clients on *FedAvg* remain normally distributed. At the same time, a more substantial bias starts to form with *FedlesScan*. *FedlesScan* favors clients with lower training duration, indicating smaller data sizes and more powerful computational resources, as seen at the top of the curve. Conversely, clients with higher training durations get penalized for missing the round before the timeout. These clients have less powerful computational power and more extensive data size, as seen at the bottom of the curve. These observations motivate the need for a more effective client selection method in federated learning systems with heterogeneous hardware settings and varying client data sizes, which we address in our proposed research objectives.

1.2 Research Objectives

The proposed research objectives aim to improve federated learning systems' performance by investigating the impact of heterogeneous client hardware on system performance and the straggler effect and address the limitations of *FedlesScan* in the context of heterogeneous hardware settings and varying client data sizes. The objectives also aim to develop an asynchronous aggregation approach that can accommodate varying client hardware capabilities and data sizes, minimize the impact of stragglers on the overall system performance, propose a comprehensive client selection method that accounts for hardware heterogeneity and data size differences, and evaluate the effectiveness of the proposed asynchronous aggregation and client selection techniques in mitigating the straggler effect. The research objectives aim to improve the efficiency and effectiveness of federated learning systems, enabling their use in a broader range of applications.

The research will focus mainly on the following points:

1. Investigate the impact of heterogeneous client hardware, such as GPUs, on the performance of federated learning systems and the straggler effect.
2. Develop an asynchronous aggregation approach that can accommodate varying client hardware capabilities and data sizes, minimizing the impact of stragglers on the overall system performance.
3. Propose a comprehensive client selection method that accounts for hardware heterogeneity and data size differences, ensuring more efficient and representative sampling for each federated learning round.
4. Evaluate the effectiveness of the proposed asynchronous aggregation and client selection techniques in mitigating the straggler effect in federated learning systems with heterogeneous hardware settings.
5. Compare the performance of the proposed methods with existing strategies, such as *FedlesScan*, to assess their potential benefits and drawbacks in real-world federated learning scenarios.

1.3 Thesis Overview

This thesis explores the combination of two emerging technologies, FL and Serverless computing, to improve the scalability and efficiency of FL. Chapter 2 provides the theoretical background for FL and Serverless computing. It covers the core concepts

of client selection and model aggregation in FL and the advantages and challenges of Serverless computing, including FaaS and *Kubernetes*. Chapter 3 reviews related work in FL and Serverless FL to provide a comprehensive overview of the current state-of-the-art in the field. Chapter 4 presents the design of *Fedless*, a Serverless FL system that addresses the limitations of traditional FL. It also introduces *FedlesScore*, a strategy that uses an asynchronous aggregation method and a score-based selection approach to optimize the performance of *Fedless*. Chapter 5 presents the experimental results of the proposed system and strategy. It evaluates the accuracy and performance of the models, examines the client selection bias, analyzes the impact of the client size, and conducts ablation studies to understand the performance of the system components. Moreover, it provides time and cost analyses of the *Fedless* system with different strategies. Finally, Chapter 6 concludes the thesis by summarizing the research contributions and suggesting further directions for future work to improve the performance and applicability of Serverless FL.

2 Theoretical Background

2.1 Federated Learning (FL)

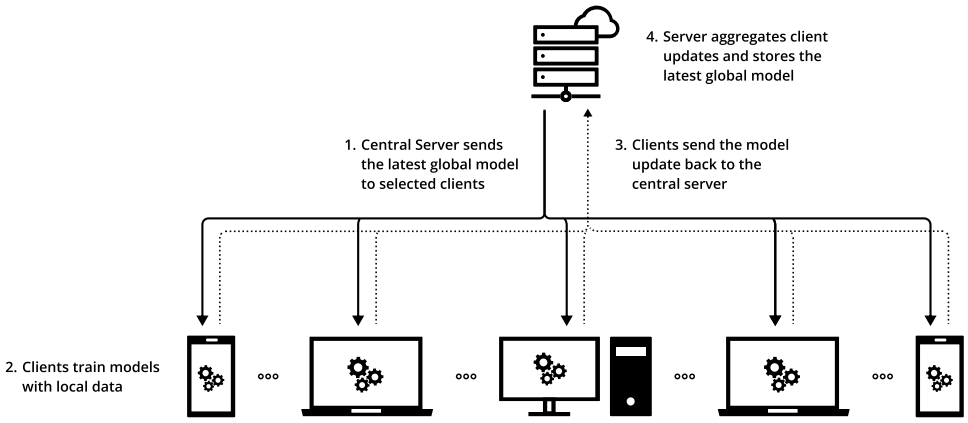


Figure 2.1: A schematic architecture of a general federated learning system

Federated learning (FL) is a distributed approach to machine learning that allows multiple devices or servers to collaboratively train a shared model while keeping the training data locally on each device. This approach preserves data privacy, as sensitive data does not need to be transferred to a central server for model training. [1]

In FL, the global model's training occurs through the following steps: A central server initializes and shares a global model with the participating devices (clients). Each client trains the model using its local data, generating model updates (e.g., gradients or weights) without sharing the raw data itself. The clients send their model updates to the central server. The central server aggregates the received updates and updates the global model accordingly. The updated global model is sent back to the clients for further training. This process is repeated until the model converges or a predetermined stopping criterion is met. FL has several advantages, such as:

- **Data privacy**: It allows training models on sensitive data without exposing the

raw data to the central server or other devices.

- **Reduced data transfer**: By keeping data on local devices, FL minimizes the need for data transmission, which can save bandwidth and reduce latency.
- **Scalability**: FL can scale to a large number of clients, leveraging the power of distributed computing.
- **Adaptability**: The approach can be adapted to various scenarios, including mobile devices, edge computing, and even leveraging cloud computing services like FaaS.

2.1.1 Client Selection

In federated learning, selecting clients to participate in model training is critical in achieving high model accuracy and reducing communication costs. There are several methods for client selection in FL, which can be broadly classified into three categories:

- **Random Selection**: The most straightforward approach is randomly selecting clients to participate in model training. This method is easy to implement and requires little computation. However, it may lead to suboptimal model accuracy and communication inefficiency if the selected clients have significantly different data distributions from the rest of the clients.
- **Sampling-based on Data Characteristics**: Another approach is to select clients based on the characteristics of their data. For example, clients with similar data distributions or data with a high degree of heterogeneity may be selected. This approach can improve model accuracy and reduce communication costs, but it requires prior knowledge of the data distribution, which may only sometimes be available.
- **Sampling-based on Hardware Characteristics**: This approach involves selecting clients based on their characteristics, such as computational capabilities or the amount of data they possess. For example, clients with high computational capabilities may be selected to reduce training time, or clients with a large amount of data may be selected to improve model accuracy. This approach can also improve model accuracy and reduce communication costs, but it requires knowledge of client characteristics, which may only sometimes be available. The selection method used in FL depends on the specific use case and available resources.

2.1.2 Model Aggregation

There are two main types of federated learning: synchronous and asynchronous.

- **Synchronous**: In synchronous FL, clients simultaneously train the model and communicate with the central server during each round of training. In synchronous FL, the central server must wait for all participating clients to complete their training and send their updates before aggregating the updates and moving to the next round. This means that the overall training process can be slowed down by slower clients or "stragglers" who may have limited computational resources or poor network connectivity. The straggler effect can lead to longer training times and reduced overall efficiency in synchronous FL.
- **Asynchronous**: Asynchronous FL aims to address the stragglers effect by allowing clients to train and communicate with the central server independently and without waiting for other clients. While this approach improves training efficiency by not waiting for slower clients, it also creates a scenario where different clients are working with different versions of the global model. When a client sends its update to the central server, the global model may have already been updated by other clients, making the client's update stale or out-of-date.

Straggling clients, which exhibit considerably slower processing compared to non-stragglers, can severely impact the performance of synchronous FL. This issue is particularly pronounced in cross-device settings, where clients vary significantly regarding computing power and data volume. As a result, synchronous FL is susceptible to performance bottlenecks caused by the presence of slow clients. This highlights the drawbacks of synchronous FL and underscores the advantages of asynchronous FL.

2.1.3 Challenges

However, federated learning also faces challenges, such as heterogeneity, communication overhead, privacy and security concerns, and resource constraints on participating devices. [6]

- **Expensive Communication**: FL relies on frequent communication between local devices and a centralized server to exchange model updates. This can lead to high communication overhead and bandwidth consumption, especially when dealing with large-scale networks and massive amounts of data. Consequently, it can slow the learning process and increase overall system latency, making it less efficient and practical for specific use cases.

- **Systems Heterogeneity**: In FL, participating devices often have varying hardware capabilities, processing power, storage capacity, and battery life. This can lead to imbalances in the learning process, as more powerful devices may contribute more significantly to the global model than weaker ones. Moreover, device-specific constraints limit their ability to participate in the learning process, affecting the overall performance of the FL system.
- **Statistical Heterogeneity**: The data distribution across participating devices in FL may be uneven or skewed, resulting in varying local model quality. This can be due to different data collection processes, user behaviors, or regional differences. Statistical heterogeneity can lead to challenges in model convergence and negatively impact the performance of the global model, as it may need to accurately represent the underlying data distribution.
- **Privacy Concerns**: While FL aims to maintain privacy by keeping raw data on local devices, it can still be vulnerable to privacy risks. An attacker could infer sensitive information about individual users during the model aggregation by analyzing the model updates shared between devices and the central server. Additionally, model inversion and membership inference attacks are possible, posing significant privacy risks to the FL system. Various privacy-preserving techniques, such as differential privacy and secure multi-party computation, can be employed to mitigate these concerns.

2.2 Serverless Computing

Serverless computing has gained significant popularity and adoption in various domains such as linear algebra [7], heterogeneous computing [8, 9, 10], high-performance computing [11, 12], and edge computing [13]. It is a cloud computing execution model where the infrastructure and servers needed to run an application or service are directly provided by a cloud provider. The cloud provider handles the provisions for necessary resources and scales them up or down as required to take the workload, making it more cost-effective and scalable than traditional server-based approaches. Serverless computing accelerates the development process by letting developers focus on developing applications rather than worrying about the supporting infrastructure. This speeds up experimentation and iteration. It offers better cost-effectiveness, scalability, and productivity and is becoming increasingly popular for various applications and services.

2.2.1 Function As a Service (FaaS)

Functions as a Service (FaaS) is a cloud computing service that enables the execution of code without worrying about building and maintaining the infrastructure required for developing and launching an application. [14] The code is directly deployed with FaaS, and the cloud provider automatically configures all the necessary resources to run the code in response to specific events or triggers. One of the main benefits of FaaS is its scalability. Because of the cloud provider's automated resource adaption on demand, the code can run concurrently on multiple servers, allowing it to scale up quickly to handle heavy workloads. FaaS is frequently more cost-effective than traditional server-based alternatives because you only pay for the computing resources required for code execution. This increases the cost-effectiveness of FaaS for applications with variable or unpredictable workloads. FaaS has numerous benefits, including scalability, cost, and usability. Henceforth, it is getting more popular for various applications and services.

Cold Start

A cold start is the delay or latency experienced when a function is called for the first time or after it has been inactive for some time. Scaling to zero allows containers to be run only when there is demand. Before starting the execution of a function, the FaaS platform must allocate resources such as CPU, memory, and network and then load the function's code into those allocated resources. This can be a time-consuming process, especially if the function's codebase is large or complex or if the platform must deploy additional infrastructure to support the function's execution. If the function has been inactive for an extended period of time, the platform may have reduced the resources allocated to it in an effort to reduce costs. As a result, the following invocation will require the platform to allocate resources and reload the function code, resulting in another cold start. Cold starts can be a performance issue for functions that require low latency or high throughput, as they can add significant overhead to function invocations. [15] To mitigate the cold starts, pre-warming or keeping functions warm involves keeping a function's resources allocated and its code loaded in memory to reduce the delay experienced on subsequent invocations. [16]

2.2.2 Kubernetes

Kubernetes is an adaptable, expandable, open-source container orchestration system that manages containerized workloads and services. [17] *Kubernetes* simplifies containerized applications' deployment, scaling, and management for organizations of varying scopes. The platform operates on the principle of declarative configuration,

allowing developers to define their applications' desired state while *Kubernetes* ensures the attainment and maintenance of that state. This approach simplifies managing complex, distributed applications, letting developers concentrate on coding rather than the underlying infrastructure. In summary, *Kubernetes* is a robust platform for governing containerized workloads and services, widely utilized by organizations of all sizes in production environments.

OpenFaas

OpenFaaS [18][5] is a framework that enables the creation and deployment of serverless functions utilizing *Docker* [19] containers. Designed for simplicity and a lightweight experience, it empowers developers to promptly develop and deploy event-responsive functions triggered by a broad spectrum of events. These functions can be crafted in any programming language compatible with *Docker* containers and are built to be highly scalable and efficient. [20]

By leveraging *OpenFaaS*, functions can be conveniently deployed as *Docker* containers in any cloud or on-premises setting that supports *Docker*. [5] This demonstrates the exceptional portability of *OpenFaaS*, making it suitable for various environments such as public clouds, private clouds, and on-premises infrastructure [21].

3 Related Work

The existing body of research focuses on addressing the challenges stragglers pose in traditional FL and serverless FL settings. This chapter explores the advancements in serverless FL algorithms and architectures, emphasizing their scalability, cost-effectiveness, and ease of deployment. Various techniques proposed to mitigate the impact of stragglers in FL are discussed, including the introduction of Asynchronous FL, which decouples the training process from synchronous execution to accelerate convergence. Furthermore, the chapter examines the specific challenges encountered in dealing with stragglers within the context of Serverless FL. It presents strategies that are specifically designed to accommodate the resource constraints and scalability features of serverless environments. This chapter establishes the foundation for the proposed solutions presented in the subsequent chapters by providing a comprehensive overview of the existing research.

3.1 Serverless Federated Learning

Utilizing serverless technologies for distributed ML training has been extensively researched in the literature, but exploring FaaS functions for FL remains a relatively new research direction. The first work in this domain was conducted by Chadha et al., who proposed *FedKeeper* [2], a tool for orchestrating Deep Neural Network (DNN) model training using FL for clients distributed across a combination of heterogeneous FaaS platforms. Building on *FedKeeper*, Grafberger et al. introduced *FedLess* [3], a system and framework for scalable FL using serverless computing technologies. *FedLess* is cloud-agnostic, supports all major commercial and open-source FaaS platforms, and enables the training of arbitrary DNN models using the *TensorFlow* library. Additionally, it incorporates several important security features, such as authentication/authorization of client functions using *AWS Cognito*, and supports privacy-preserving FL training of models using Differential Privacy. Explicitly designed for serverless environments, *FedLess* includes performance optimizations like global namespace caching, running average model aggregation, and federated evaluation. This work also uses *Fedless* as a system to perform FL on a FaaS environment, and the architecture and workflow will be discussed in further detail in Chapter 4 (Figure 4.1, Figure 4.2).

JIT [22] is an approach to FL aggregation called "just-in-time" aggregation proposed by researchers at IBM Research AI. This approach maximizes the utilization of compute resources by deferring the aggregation process, allowing for allocating resources to other FL tasks or data center workloads. The authors demonstrate that using JIT aggregation can reduce resource usage and does not increase the latency of FL jobs.

AdaFed [23] is an FL parameter aggregation mechanism that dynamically uses serverless/cloud functions to scale aggregation resource-efficient and fault-tolerant manner. *AdaFed* reduces state in aggregators, leverages serverless technologies, and is efficient and expressive for programmers.

λ-FL [24] is an FL aggregation architecture that leverages serverless technology/cloud functions to deploy and scale the aggregation process as necessary dynamically. λ-FL reduces the state in aggregators and achieves fault tolerance with minimal effort. It uses a message queue to record the presence of model updates and eliminates persistent network connections between aggregators for increased reliability and fault tolerance.

3.2 Stragglers in Federated Learning

Resource and data heterogeneity pose challenges in large-scale FL systems. Maintaining consistent performance and reliable communication across all clients throughout the training process becomes impractical. In real-world scenarios, clients may experience periods of offline status due to network limitations or resource constraints. Moreover, the training speed of clients is influenced by the volume of data they possess and their computational capabilities. Synchronous training mechanisms like *FedAvg* [1] are susceptible to the presence of stragglers, where the slowest client dictates the overall training pace. Additionally, offline clients that fail to respond in a timely manner can lead to significant training delays.

Li et al. proposed a strategy called *FedProx* [25] to address the challenges of heterogeneity in FL. This algorithm builds upon *FedAvg* with two essential modifications. Firstly, it introduces a customized loss function at the client level, incorporating a proximal term to mitigate the impact of local updates' fluctuations. This proximal term helps regulate the deviation of the local model from the global model by considering the squared difference in model weights during the loss computation, preventing them from diverging significantly and enhancing the model's robustness to heterogeneous data distributions across devices. Secondly, *FedProx* introduces the concept of tolerating partial work, allowing clients to adapt their workload based on hardware, network, and battery constraints. This flexibility enables clients to perform a variable number of local epochs to accommodate their resource limitations.

Karimireddy et al. proposed an FL strategy designed to address the challenges of

varying local data distributions and the potential for biased updates, called *SCAFFOLD* [26]. To mitigate the bias, they incorporate control variates, a method derived from standard convex optimization literature that aims to reduce the variance of stochastic gradients in finite sum minimization problems, thereby accelerating convergence [27]. By leveraging control variates, *SCAFFOLD* effectively diminishes the variance of local updates, leading to a more stable aggregation process. Consequently, integrating control variates enables identifying and eliminating device-specific biases from updates prior to aggregation, resulting in improved accuracy and stability of global model updates.

The *SCAFFOLD* algorithm utilizes a persistent state associated with each client, known as a control variate, denoted as c_i for client i. This control variate estimates the gradient of the loss concerning the client's local data. The server maintains all client control variates' averages as the global control variate, denoted as c, which is communicated to the selected clients in each round. Clients undertake multiple stochastic gradient descent (*SGD*) steps in each round, similar to the FedAvg approach while incorporating an additional correction term, $c - c_i$, to each stochastic gradient. This correction term effectively reduces the bias of the update step on each client, ensuring that the updates are much closer to the global update. This characteristic enables *SCAFFOLD* to achieve provably faster convergence compared to vanilla *FedAvg*, without imposing any assumptions on data heterogeneity.

When implementing *SCAFFOLD*, various options exist for selecting the control variates. For instance as shown in Equation 3.1, one can choose to utilize the averaged local gradients from the previous round as c_i. Specifically, after performing local updates, the local control variate is updated as follows:

$$c_i^{(t+1)} = c_i^{(t)} - c^{(t)} + \frac{1}{K\eta_l}(x - y_i) \tag{3.1}$$

where the superscript t denotes the index of communication round, and K the number of local updates, and η_l being the local learning rate. Compared with *FedAvg*, intuitively, SCAFFOLD doubles the communication size per round due to the additional control variates.

However, *SCAFFOLD* has several drawbacks. Firstly, it does not account for heterogeneous local progress, requiring clients to have the same number of local updates or utilize dynamic batch sizes to maintain consistency. Secondly, it results in double the communication size per round as both the local model weight and the local control variates need to be transmitted. Moreover, *SCAFFOLD* relies on the participation of all clients for optimal performance, and its effectiveness decreases as the client sample ratio per round is reduced, as demonstrated in the work of Li and Diao [28].

Wang et al. [29] proposed a framework called *FedNova* for unifying *FedAvg* and

FedProx. This showed that the computational heterogeneity across clients influences the weighting scheme and argues that local updates should be normalized while averaging to minimize heterogeneity induced by a different number of local update steps.

Similarly to *FedProx* [25], *FedNova* [29] addresses the non-IID data challenge by combining *FedAvg*'s ideas with a momentum-based optimization technique. This approach updates the global model using a weighted average of local models. It incorporates a momentum term to improve convergence by maintaining a sense of direction and reducing the effect of noisy local updates.

When clients perform varying local steps in vanilla *FedAvg* and other FL algorithms, it leads to an implicit assignment of higher weights to clients with more local steps. To address this inconsistency between the surrogate loss and the original loss, *FedNova* [29] proposes a solution by normalizing the local updates with the number of local steps. This normalization can be seen as a new weighting scheme that assigns lower weights to clients with more local steps. However, *FedNova*'s method is specifically designed for *SGD*, which limits its applicability. To address this limitation, Wang et al. [30] present a generalized analysis that demonstrates the presence of inconsistency even when using adaptive methods like *Adagrad* or *Adam*, even with the same number of local steps.

Chai, Ali, and Zawad proposed a Tier-based FL System called *TiFL* [31] to address the impact of heterogeneity on training time and model accuracy in conventional FL systems. *TiFL* employs a tier-based client selection approach to address the challenges posed by heterogeneity in resource availability and data quantity. By dividing clients into tiers and selecting clients from the same tier in each training round, *TiFL* mitigates the impact of stragglers. Additionally, *TiFL* incorporates an adaptive tier selection mechanism that dynamically updates the tiering based on observed training performance and accuracy. This adaptive approach effectively manages the heterogeneity arising from non-IID data and resource variations, improving the overall performance of the FL system.

Cox, Chen, and Decouchant proposed a new approach to address slow clients in FL called *Aergia* [32]. Unlike other approaches that use deadlines or send partially trained models, *Aergia* freezes the computationally intensive part of the slow client's model and offloads it to a faster client that trains it using its own dataset. *Aergia* leverages the spare computational capacity of strong clients and achieves high accuracy in low training time by effectively matching clients' performance profiles and data similarity.

Oort [33] is an FL strategy that enhances the efficiency of model training and testing by employing a guided participant selection approach. It aims to identify and prioritize valuable participants for FL training and testing, giving preference to clients whose data can contribute the most to improving model accuracy and having the ability to perform training quickly. The scoring mechanism in *Oort* evaluates clients based on two main criteria: their utility in enhancing model accuracy and their capacity to execute training

efficiently, all while maintaining 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. The authors propose a practical approximation of statistical utility by utilizing the gradient norm derived from the training loss. This statistical utility is then combined with a global system utility, which considers the duration of each training round to strike an optimal trade-off between statistical accuracy and system efficiency.

$$U_i^{Oort} = \underbrace{|B_i| \sqrt{\frac{1}{|B_i|} \sum_{k \in B_i} \text{Loss}(k)^2}}_{\text{Statistical utility } U(i)} \times \underbrace{\left(\frac{T}{t_i}\right)^{1(T<t_i)\times\alpha}}_{\text{Global sys utility}} \tag{3.2}$$

The clients are evaluated and selected based on the utility score shown in Equation 3.2. With B_i being the local data on client i, the aggregated training loss will be multiplied by the local data size and the system utility, which is defined by the client's duration t_i and the developer's preferred duration T. α being the developer-specified factor that the clients will be penalized by. Experimental results demonstrate that this approach significantly improves both time-to-accuracy performance and final model accuracy while respecting privacy constraints.

3.2.1 Asynchronous Federated Learning

Asynchronous online federated learning for edge devices with non-iid dataAsynchronous FL has gained significant attention due to its potential to address the challenges of heterogeneity, scalability, and privacy in large-scale cross-device FL systems. AFL combines the benefits of synchronous and asynchronous training mechanisms to achieve improved efficiency and convergence. This section discusses several notable approaches and their essential contributions.

SAFA [34], a semi-asynchronous protocol for fast FL with low overhead, is proposed by Wu et al. The algorithm focused on low round efficiency and slow convergence in scenarios where clients drop frequently. In order to mitigate the impact of stragglers, the authors proposed novel designs for client selection and global aggregation. They introduced a caching mechanism to prevent wasted contributions, storing a mapping between clients and their updates in a cache maintained by the central server. Through their experiments, they observed improvements in both accuracy and efficiency, albeit at a slightly higher communication cost. While the custom client selection algorithm proved effective in addressing the stragglers' problem, it is worth noting that this approach may encounter challenges when applied in a FaaS environment. One of the drawbacks is overutilizing the clients, which could increase the experiment's cost.

Deprecated clients may persist throughout the training session, wasting contributions and resource consumption. Additionally, their selection strategy does not leverage the scale-to-zero capabilities of a serverless infrastructure, as it involves all clients in every round without considering their current relevance or efficiency. Clients will always run in some scenarios since they are called every round, increasing cost and resource utilization.

FedAT [35] combines synchronous and asynchronous FL training using a tiering mechanism. Clients are partitioned into logical tiers based on their response latency, and each tier proceeds at its own pace. Faster tiers update the model synchronously, while slower tiers asynchronously send model updates to the server. *FedAT* uses a weighted aggregation heuristic to prevent bias toward faster tiers and a polyline encoding compression algorithm to minimize communication costs.

Xie, Koyejo, and Gupta introduced *FedAsync* [36], an asynchronous federated optimization algorithm. *FedAsync* leverages the parameter server architecture to coordinate client invocations and synchronization. It employs a scheduler thread responsible for periodically triggering clients to conduct training using the most latest global model. Additionally, an updater thread is employed to receive client updates and perform direct aggregation into the global model.

Chen et al. [37] proposed an Asynchronous Online FL framework *ASO-Fed* that addresses the limitations of Federated Averaging in heterogeneous device environments. *ASO-Fed* enables continuous local data streaming by edge devices, allowing for better model convergence than synchronous FL frameworks. The framework includes regularization and a central feature learning module to learn inter-client relatedness effectively. It also has a dynamic learning strategy for step-size adaptation on local devices to mitigate the impact of stragglers.

To address scalability and privacy challenges in cross-device FL systems, *FedBuff* [38] presents an asynchronous optimization framework. Utilizing buffered asynchronous aggregation, *FedBuff* achieves scalability while maintaining compatibility with privacy-preserving technologies such as Secure Aggregation and differential privacy. In *FedBuff*, clients train and communicate asynchronously with the server, and client updates are stored in a buffer. A server update occurs once a certain number of client updates are in the buffer, allowing the server to choose the model update frequency instead of coupling concurrency with the server model update as in synchronous FL. *FedBuff* is compatible with privacy-preserving technologies such as Secure Aggregation and differential privacy, with theoretical convergence guarantees in a smooth, non-convex setting. The method outperforms synchronous and asynchronous FL approaches and applies them in real-world settings. *FedBuff* provides further protection against inference attacks on clients' data using trusted execution environments.

Pisces is an asynchronous FL strategy that addresses the challenges of slow clients

and data quality by conducting guided participant selection. It uses a novel participant selection strategy that prioritizes clients with high-quality data and avoids stale computation. The algorithm also adopts an adaptive aggregation pace control that dynamically adjusts the aggregation interval to match the running client's speed. *Pisces* combines techniques from *FedBuff* [38] for asynchronous FL and *Oort* [33] for client selection to improve performance. However, unlike *FedBuff*, *Pisces* prioritizes participants with high data quality as *Oort* does. By doing so, *Pisces* makes more efficient use of concurrency quotas than FedBuff does.

Regarding scoring, *Pisces* measures the data quality of each participant by approximating its importance and weight using a loss function. The loss function estimates the probability that a participant's update will be selected for aggregation in each round of FL training. In Equation 3.3, formulate the utility of a client, which is used as scoring by respecting the roles that it is data quality and staleness play in improving the global model. The first part for data quality remains the same as in *Oort* [33]; it uses the aggregated training loss. However, the second part does not include the training duration in the scoring. It uses the estimated staleness of the client's updates $\tilde{\tau}_i$ and moderates the penalty of the staleness with the factor of β. Based on the profiled utilities, *Pisces* sorts clients and selects the clients with the highest utilities to train.

$$U_i^{\text{Pisces}} = \underbrace{|B_i| \sqrt{\frac{1}{|B_i|} \sum_{k \in B_i} \text{Loss}(k)^2}}_{\text{Data qualit } y} \times \underbrace{\frac{1}{(\tilde{\tau}_i + 1)^{\beta}}}_{\text{Staleness}} \tag{3.3}$$

Pisces demonstrates superior performance compared to *Oort* [33] and *FedBuff* [38], achieving a 1.2x and 2.0x improvement in time-to-accuracy performance, respectively. This significant advantage in training efficiency is achieved through an automated participant selection and model aggregation process that is designed in a principled manner.

3.3 Stragglers in Serverless Federated Learning

FedLesScan [4] proposed by Elzohairy, is an extension of the *Fedless* framework [3], introduces a novel clustering-based semi-asynchronous training strategy designed explicitly for serverless FL systems. Its primary objective is to mitigate the impact of stragglers, which are slow clients characterized by resource and statistical heterogeneity, on the overall system performance. The *FedLesScan* strategy consists of two key components: an adaptive clustering-based client selection algorithm and a staleness-aware aggregation scheme. *FedLesScan* employs the Density-Based Spatial Clustering of Applications with Noise (*DBSCAN*) algorithm to enable efficient client selection. This

algorithm partitions the behavioral data collected from clients into distinct clusters, considering the ϵ parameter, which determines the maximum distance between two samples for them to be considered in each other's neighborhood. Outliers are separately assigned to a single cluster. The clusters are then sorted based on their members' average total Exponential Moving Average (*EMA*) in ascending order. By sampling clients from the sorted clusters, *FedLesScan* ensures a gradual transition from faster to slower clusters, considering both the client's behavior and their current training progress. An essential advantage of *FedLesScan* is that it does not require additional computation on the client side. Clients are only required to be active during the training process, making it well-suited for serverless environments. The strategy effectively addresses the challenges posed by stragglers in FL systems, enhancing the overall efficiency and robustness of the training process.

In Chapter 1.1, we extensively discussed the limitations and challenges associated with *FedlesScan*. To address these issues, we present a novel asynchronous FL strategy called *FedlesScore*. This new strategy is designed to overcome the difficulties encountered with *FedlesScan* and provides an improved approach to asynchronous FL. By introducing *FedlesScore*, we aim to enhance the performance and effectiveness of FL in challenging environments.

3.4 Strategy Comparison

Table 3.1 provides a comprehensive comparison of various strategies aiming to mitigate the impact of stragglers in synchronous and asynchronous FL approaches. We listed the following features and compared the support of the strategies:

- FaaS Support: compatible with FaaS environments, leveraging the scalability and cost-effectiveness of serverless computing
- Asynchronous Aggregation: decouples the training process from the synchronization of client updates, allowing for more flexible and efficient communication.
- Performance-based Selection: Select clients based on their training duration.
- Client Efficiency Scoring: Takes the hardware heterogeneity into account during selection, specifically, the relationship and inverse correlation between data size and training time.
- Adaptive Penalty: Adjust the probability of selecting a client based on performance and availability over time.

	FaaS Support	Asynchronous Aggregation	Performance based Selection	Client Efficiency Scoring	Adaptive Penalty
FedProx [25]					
FedNova [29]					
SCAFFOLD [26]					
TiFL [31]			✓		✓
Aergia [32]					
Oort [33]			✓		
SAFA [34]					
FedAT [35]			✓		✓
FedAsync [36]		✓			
FedBuff [38]		✓			
Pisces [39]		✓	✓		✓
FedlesScan [4]	✓		✓		✓
FedlesScore	✓	✓	✓	✓	✓

Table 3.1: Strategies Feature Comparison

While *FedProx* [25], *FedNova* [29], *SCAFFOLD* [26], and *Aergia* [32] primarily focus on optimizing the local training process and aggregation methods, they do not incorporate intelligent client selection to optimize round performance as demonstrated in our work. *SAFA* [34] tracks the status of clients' local models to ensure their synchronization with the global model but tends to overutilize clients and lacks suitability for the FaaS environment. *TiFL* [31], *FedAT* [35], and *FedlesScan* [4] group clients based on training duration tiers or clusters. However, they overlook the hardware and data heterogeneity during the client selection process and lack full asynchrony regarding aggregation. *Oort* [33] refines the client selection approach by considering both data size and training duration but does not account for the correlation between these factors and heterogeneous hardware and imposes a strict penalty on slow clients. *FedAsync* [36] and *FedBuff* [38] focus on optimizing FL with asynchronous aggregation but adopt random client selection. In contrast, *Pisces* [39] combines the methods from *Oort* and *FedBuff*, refining the scoring approach, yet it still overlooks clients' efficiency during scoring and selection. Our proposed strategy overcomes these limitations by incorporating comprehensive scoring metrics that account for both hardware and data heterogeneity, ensuring intelligent client selection and efficient round performance.

4 System and Strategy Design

4.1 Fedless

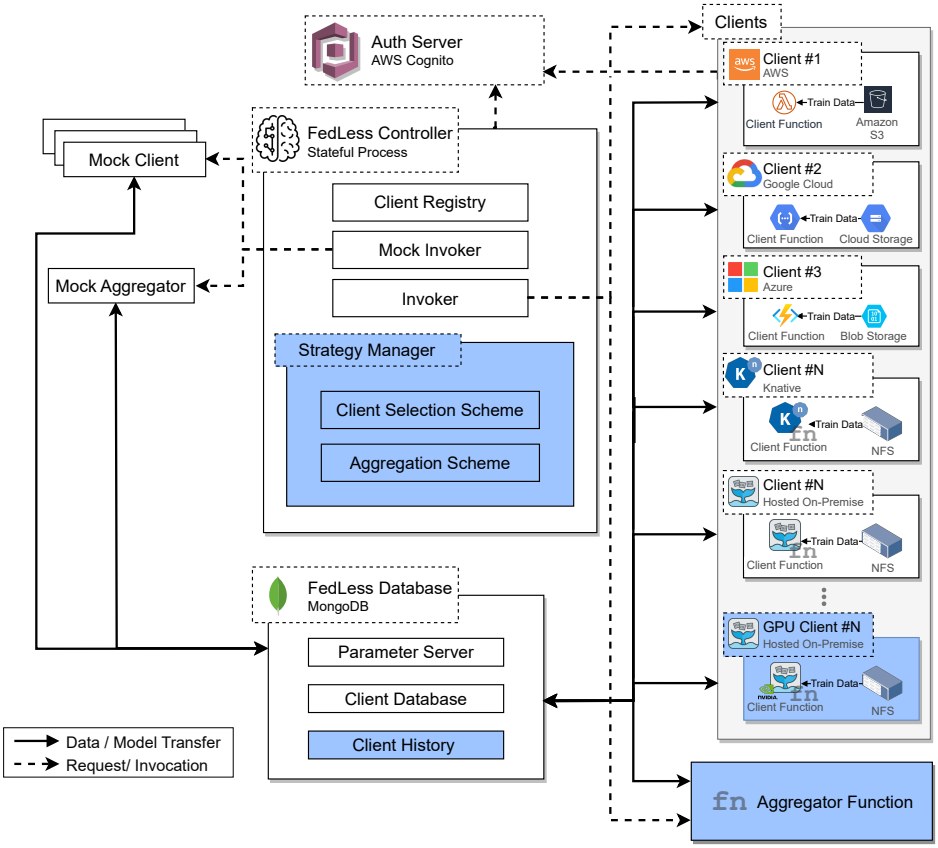


Figure 4.1: Modified Architecture of the *FedLess* platform. The highlighted components shows the modified components and additions to the system.

FedLess [3] represents a significant improvement over its predecessor, *FedKeeper* [2], in terms of both performance and security. As shown in Figure 4.2, the platform architecture of *FedLess* consists of several components, including the client registry,

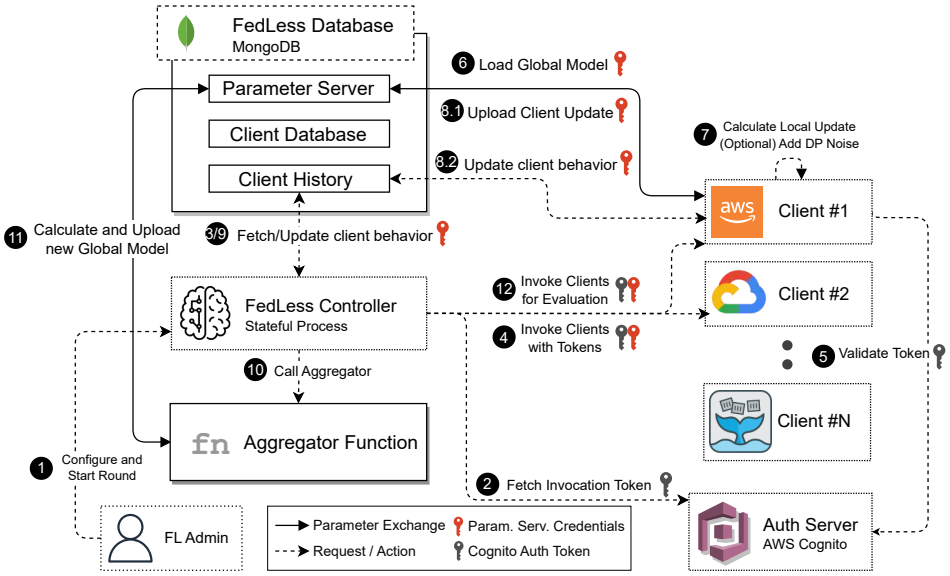


Figure 4.2: The training workflow of a typical training round in *FedLess*

client invoker, parameter server, and aggregator. A crucial addition to the *FedLess* architecture is an authentication entity powered by *AWS Cognito* [40], which ensures that only the *FedLess* Controller can call the client functions and that functions are verified before participating in the training. During a training round, the *FedLess* Controller performs the necessary steps to prepare the training environment. The FL admin configures the model, dataset, and hyperparameters before the start of training. The controller fetches invocation tokens from the authentication server and invokes the client function using these tokens. The client function contacts the authentication server to validate the invocation token, and upon successful validation, the clients fetch the latest global model from the parameter server and train locally. Once training is complete, clients upload their new local models to the parameter server and notify the controller. The controller invokes the local aggregation function, which combines the clients' results into a new global model. A subset of clients is selected for evaluation, and the process repeats for subsequent rounds. *FedAvg*, *FedProx*, and *FedlesScan* are currently the training strategies available for training client functions within *FedLess*. Clients must submit a registration request to the authentication server to participate in the training, which is approved by the FL admin who runs the *FedLess* Controller.

The fundamental concept of *FedLess* remained unchanged, but modifications were made to the training workflow to support the asynchronous approach, as described in detail in Section 4.2.1. The protocols for invoking and authenticating clients remain the

same. In a typical training round, communication occurs among various components of *FedLess*, as illustrated in Figure 4.2. Prior to training, the FL admin sets the model, dataset, and hyperparameters. The controller obtains invocation tokens from the authentication server and retrieves the clients' behavioral data from the *FedLess* database based on the selected training strategy. A subset of clients is then selected, and their functions are invoked using the obtained tokens. The clients perform local training using the latest global model from the parameter server. They upload their updated local models and training progress information to the parameter server and client history collection. After training completion, the client function signals the controller. The controller updates the clients' behavioral attributes based on the strategy and invokes the aggregation function to create a new global model. Finally, the controller invokes a subset of clients for evaluation, which is repeated for subsequent rounds.

4.1.1 Enabling GPU on Fedless

The default *Kubernetes* architecture does not support GPU splitting or fine-grained allocation and sharing of GPU resources, which makes assigning GPU resources as floating points challenging. [41] To address this issue, plugins like *KubeShare* [42] and 4paradigm's k8s vGPU scheduler [43] have been developed. *KubeShare* extends *Kubernetes* to allow the allocation and sharing of GPUs as first-class resources for scheduling. However, at the time of writing, *KubeShare* did not work, and we opted for the *4paradigm k8s vGPU scheduler*, which enables GPU sharing among tasks, device memory control, virtual device memory, and GPU type specification. 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. Although the vGPU scheduler is based on the *NVIDIA device plugin* and retains its official features, it lacked proper limitations to GPU memory on pods at the time of writing. To address this issue, we restricted memory allocation at the *TensorFlow* level, ensuring that one client would not occupy all memory on the GPU.

After integrating 4paradigm's k8s vGPU scheduler on the *Kubernetes* cluster, the default setting splits the GPU into ten slots. This implies that despite only being able to request one nvidia.com/gpu per pod, ten pods can utilize the GPU simultaneously. Without the vGPU scheduler from *4paradigm*, this results in inadequate GPU resources. Additionally, it is now possible to use GPUs directly on our pods and *OpenFaaS* functions. Configuration files for different hardware settings for *OpenFaaS* functions are provided in Listing 4.1, 4.2, and 4.3. For clients with one vCPU and two vCPU, we simply request and limit the resources to the same amount to ensure that the function has the exact amount of resources it needs without excess or deficit. However, for GPU clients, specifying both the core percentage (nvidia.com/gpucores)

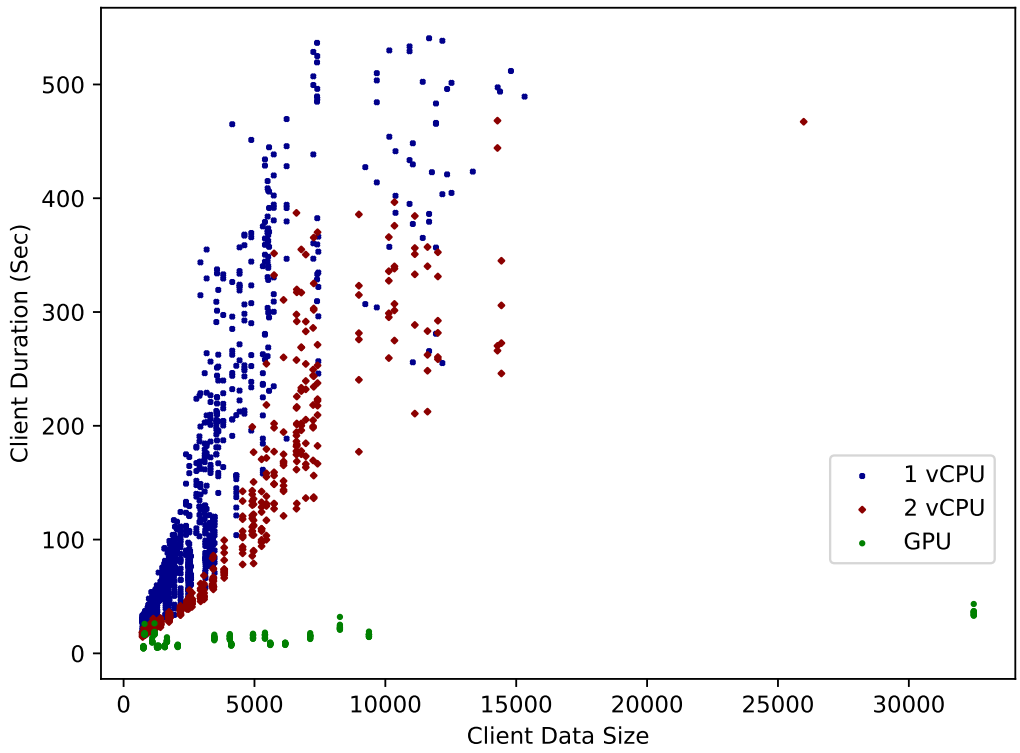


Figure 4.3: Training duration based on client hardware setting and Data size.

and memory percentage (nvidia.com/gpumem-percentage) is necessary if we do not want the pod to utilize too many resources. Nevertheless, the GPU memory percentage limit does not work as intended, so we use an additional environment variable called gpu_memory_fraction to restrict the GPU memory usage at the *TensorFlow* level. It is important to note that the vGPU scheduler is used solely to simulate GPU FaaS functions and is not directly related to the study.

Figure 4.3 illustrates the impact of data size on training duration across different hardware settings on the *Shakespeare* dataset. The results were directly taken from the experiments that will be described in detail in chapter 5, here the client data size and average training time of each invocation across the whole experiment are plotted in a scatterplot to visualize how the data size and hardware resources affect the training duration. The results indicate that clients with only one virtual CPU (vCPU) experience performance degradation as the data size increases. Similarly, clients with two vCPUs can handle more data than their counterparts with one vCPU but eventually face performance bottlenecks. However, clients equipped with GPUs can handle larger data sizes with ease, even if they only leverage a fraction of the GPU resources. Notably, utilizing two-fifths of the GPU resources is sufficient to maintain the training duration and improve the efficiency of the training process. These findings demonstrate the potential of GPUs to accelerate machine learning workloads and highlight the importance of selecting appropriate hardware for data-intensive workloads.

4.1.2 Mock Cold Start

The mock cold start feature for *Fedless* is a functionality that introduces an invocation delay if a serverless function has not been invoked within a specified time. This delay is designed to mimic the cold start time when a function is first invoked, and the serverless platform needs to allocate resources to execute it. The cold start duration is randomly sampled from a Gauss distribution [44], which is slightly different for each invocation.

Since GPU is not available on any cloud provider's FaaS to provide a baseline for how long GPU cold start might take, we make the cold start behavior the same as regular CPU clients. The primary purpose of this feature is to observe the cold starting effect and provide a more realistic testing environment for serverless functions that may not be invoked frequently. By introducing a synthetic delay for the first invocation after a certain period has elapsed, we can better understand how the functions will perform in a real-world scenario. When this feature is enabled, the controller will track the last time each function was invoked. If a function has not been invoked for a specified amount of time, *Fedless* will introduce a delay before executing the function on the following invocation.

```yaml
version: 1.0
provider:
  name: openfaas
  gateway: http://127.0.0.1:8080
functions:
  vcpu1-client:
    lang: python3-http-debian
    handler: ./client
    image: fedless-cpu-client
    limits:
      cpu: 1000m
      memory: 2048Mi
    requests:
      cpu: 1000m
      memory: 2048Mi
```

Listing 4.1: YAML configuration for 1 vCPU client

```yaml
version: 1.0
provider:
  name: openfaas
  gateway: http://127.0.0.1:8080
functions:
  vcpu2-client:
    lang: python3-http-debian
    handler: ./client
    image: fedless-cpu-client
    limits:
      cpu: 2000m
      memory: 4096Mi
    requests:
      cpu: 2000m
      memory: 4096Mi
```

Listing 4.2: YAML configuration for 2 vCPU client

```yaml
version: 1.0
provider:
  name: openfaas
  gateway: http://127.0.0.1:8080
functions:
  gpu-client:
    lang: python3-http-debian
    handler: ./client
    image: fedless-gpu-client
    limits:
      cpu: 2000m
      memory: 3000Mi
      nvidia.com/gpu: 1
      nvidia.com/gpucores: 18
      nvidia.com/gpumem-percentage: 18
    requests:
      nvidia.com/gpu: 1
      nvidia.com/gpucores: 18
      nvidia.com/gpumem-percentage: 18
    environment:
      gpu_memory_fraction: 0.18
```

Listing 4.3: YAML configuration for GPU client

Algorithm 1: Modifed *FedLess* [3] controller and client routines.

Fedless Controller:
Function `Train_Global_Model(`*Clients, round*`)`**:**
    selected = Select_Clients (*clients*, *clientsPerRound*)
    Invoke_Clients (*selected*)
    for *each client in selected* **do**
        save invocation record to database (*client*, *round*, *isColdStart*)
        set *client* invocation status to *runnning* `// Busy client`
    end
    while #*results* $\geq$ (*clientsPerRound* $*$ *bufferRatio*) **do**
        sleep
    end
    Invoke_aggregator()
return
Fedless Client:
Function `Client_Update(`*hyperParameters, round*`)`**:**
    Load model and dataset.
    `Start_Timer()`
    Train model
    `Stop_Timer()`
    Save updated model to database. (*results*)
    Add measured time to invocation record.
    Update *invocation* status to *complete*. `// Available client`
return

4.2 FedlesScore

In the following sections, we will comprehensively describe our proposed asynchronous strategy called *FedlesScore* for the *Fedless* framework, detailing the client selection strategy and algorithm modifications necessary for effective asynchronous communication. Furthermore, we will demonstrate the benefits of our approach through a series of experiments and comparisons with existing synchronous federated learning methods. By introducing an asynchronous strategy to *FedLess*, we hope to contribute to the ongoing advancements in federated learning, ultimately enabling more efficient and scalable distributed machine learning across diverse and dynamic networks of devices.

We slightly modified the *Fedless* controller and client to implement our proposed

strategy effectively. The resulting changes in the *FedLess* controller and FL client routines are depicted in Algorithm 1, providing a comprehensive overview of the strategy's implementation. In the subsequent sections, we will delve into each aspect of the strategy in detail.

4.2.1 Asynchronous Aggregation

To enhance the efficiency of the *FedLess* system, we aim to make the entire process asynchronous. We propose a novel mechanism for triggering the global model aggregation. Previously, *FedLess* invoked multiple clients and waited for them to complete their local training before aggregating the global model. The only way to limit the waiting time was to set a timeout on the sockets. However, slow clients might push their updates to the parameter server after the round ends, and these updates are considered valuable contributions. To address this issue, *FedlesScan* introduced a staleness-aware aggregation scheme that allows delayed updates to be included during the next aggregation function call, with a staleness weight based on how late the results arrive.

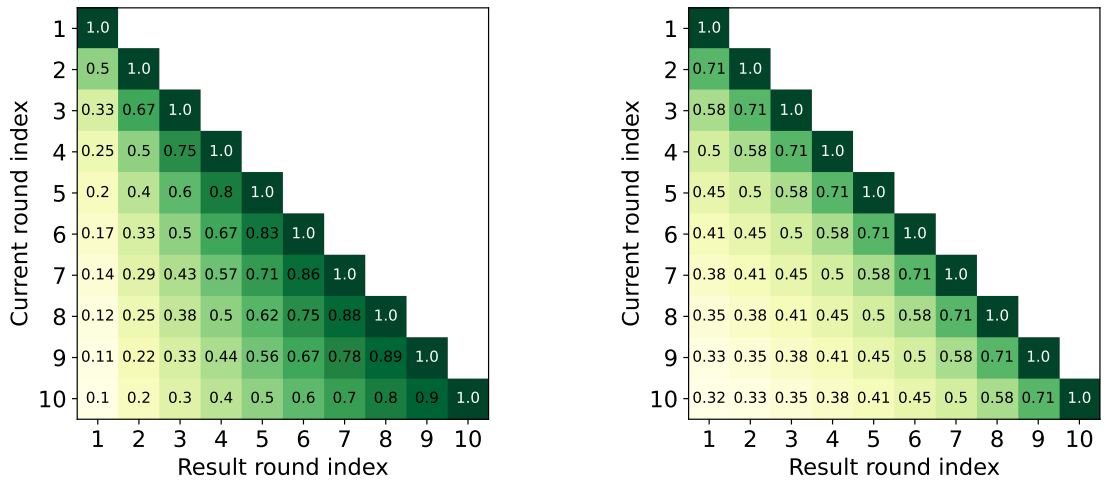
We aim to aggregate delayed updates asynchronously without waiting for all results to come in. To achieve this, we modify the trigger for aggregation to only wait for a fraction of client results, known as the buffer ratio. The buffer ratio, a floating-point value between 0 and 1, determines how many results we wait for before triggering the aggregation process. For instance, if we select 100 clients per round and set the buffer ratio to 0.6, we wait for only 60 client results, regardless of whether the result is delayed or not. The client results can be from the current round, previous rounds, or even further back in time. To prevent outdated results from contributing, we set a threshold for the age of client results. In our experiments, we set the threshold to 5, but we observed that most delayed results arrive within two rounds.

Staleness Weighting Function

$$weight = \frac{t_i}{T} \tag{4.1}$$

$$weight = \frac{1}{(T - t_i + 1)^{0.5}} \tag{4.2}$$

Intuitively, larger staleness leads to an increased error in the global model updates [36]. The weighting function should assign a weight of 1 to the current round's results and show a monotonically decreasing pattern with increasing round numbers. We have experimented with different staleness weighting functions, including the one used in



(a) Equation 4.1

(b) Equation 4.2

Figure 4.4: Evaluation of staleness weights in the relationship between the current round index and result round index

FedLesScan, as shown in Equation 4.1, with T being the current round index, and t_i round index of the client's result. This function gradually increases the weight of one round of late results as the round number increases. Therefore, we decided to adopt a different weighting function proposed by Xie, Koyejo, and Gupta [36] (Equation 4.2). As shown in Figure 4.4, we see the Staleness weighting in relationship to the current and result round indexes. Here we can see that the weighting value from Equation 4.1 shown in Figure 4.4a is not consistent for the result with the same staleness, as we see the values do not remain consistent in the diagonal axis like in Figure 4.4b for Equation 4.2.

In the next section, we will introduce an additional selection strategy to accompany the asynchronous aggregation approach.

4.2.2 Score-based selection

In *FedlesScan*, clustering was performed solely based on training time, which is appropriate in a homogeneous hardware setting where the training time is directly proportional to the client's data size, given that all clients share the same hardware and model specifications. However, this assumption no longer holds true in a heterogeneous environment, as illustrated in Figure 4.3, which shows that training duration can vary across different hardware settings. Therefore, we want to do more than just sample and select our clients based on one factor. We want to sample our clients based on both data

and hardware characteristics (as mentioned in section 2.1.1). Therefore, we propose a probabilistic client selection approach considering the client's training duration and data size.

Having made the process asynchronous, we are no longer directly impacted by the presence of slow clients, often referred to as stragglers. Consequently, our focus shifts to selecting clients that can contribute significantly within a reasonable timeframe. Ideally, clients with large data sizes and access to powerful hardware resources should be selected, as they can provide the most contributions in the shortest amount of time. To this end, we introduce a scoring system for each client based on the size of their data and hardware resources.

Gathering Behavioral Data

To score clients, information on their training behavior is necessary. We record and collect the pure training time for each client's invocation to evaluate the hardware used. This only includes the duration of executing *model.fit* (Algorithm 1: Line 17-19), excluding time spent on network communication and model initialization. To score clients based on data cardinality, we also require the local data size for each client, which is not a secret as it is required for aggregation with FedAvg. The batch size and the number of epochs are also recorded to calculate the total number of local updates on the clients during training, which remains static across all clients as listed in Table 5.2. The booster value of each client is also recorded to determine the level of promotion. This floating value is initialized to 1 and is updated each time the client is not selected. When the client is finally selected again, the booster value is reset to 1. In the next section, we will further describe the method of scoring and how we used the collected information to select clients in more detail.

Client Scoring

Algorithm 2 presents the algorithm for scoring clients that would be used to select clients each round to participate in the training round. It can be challenging to determine a client's machine learning performance based solely on their hardware specifications (such as CPU model, frequency, and the number of cores), making it necessary to conduct a benchmark to acquire a score. However, this approach can be both expensive and inefficient. Alternatively, we use the training performance as a hardware benchmark score (Client Efficiency Score). The algorithm calculates the number of updates made by the client on the local model by multiplying the data size N_c by the epoch size E and dividing the result by the batch size B. It then calculates the number of updates per second by dividing the number of updates by the training

Algorithm 2: Calculate Weighted Average Score: The β is the clients' current booster value, and T is the list of each training duration of the clients, and N_c is the cardinality of the clients' data; E is the number of epochs of the clients, B is the client local batch size, and λ is the global defined decay rate

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

time (in seconds). A higher score indicates a more powerful hardware configuration.

To obtain a score that considers both client performance and training duration, we record the duration of every training session and calculate an exponentially weighted score. The exponentially weighted average uses exponentially decreasing weights as the data point ages. The weights are assigned based on their relative recency in an exponential decay fashion ($\lambda^0, \lambda^1, \lambda^2, ..., \lambda^i$), meaning the weights decrease exponentially as the data points move further into the past. λ is calculate with a smoothing factor α $1 - \alpha$. The most recent data points are assigned higher weights, while older data points receive lower weights. In terms of client scoring, we assign a higher weight to the most recent invocations' score and decrease the weight (λ^i, where i increases as the entry gets older) for older entries.

However, we also want to promote slow clients and provide them an additional opportunity to be selected. For this purpose, we introduce a booster value, which is multiplied by the final score before sampling. The booster value is initially set to 1 for all clients. Whenever a client is available (not busy training) but not selected for the round, we increase its booster value by multiplying it with a promotion value greater than 1; otherwise, the client will receive an unfavorable promotion, and the chances of getting selected will have come even lower every time it did not get selected and got the score got multiplied again by the promotion rate. This gradual increment of the booster value ensures that the client's score increases over time as it keeps being left out of the selection. If the client is eventually selected, the booster value is reset to 1.

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. If ρ is set to 1, both damping and promotion are

disabled, resulting in equal weighting for all scoring and no promotion for clients that have been consistently missing out.

Client Selection

Algorithm 3: Client selection

```
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      Calculate #Scored_clients required from invoked_clients.
8      client_scores = []
9      for each client in invoked_clients do
10         Calculate Weighted Score for client.
11         Append Client Score to client_scores.
12     end
13     Calculate probability for all invoked_clients  score / ∑ client_scores
14     client_selection ← Sample invoked_clients based on probability
15     Reset booster value for all clients in client_selection
16     Increase booster value for all clients NOT in client_selection
         (invoked_clients − client_selection)
17 return client_selection
```

The selection strategy is based on the score of each client, and it is described in Algorithm 3. The goal is to sample a given number of clients (*clientsPerRound*) from a list of available clients (*clients*). First, we distinguish clients who have already been invoked from those who have not. Since clients must be invoked at least once to calculate a score, we prioritize uninvoked clients first. If there are enough uninvoked clients, we always choose them first. When we need more clients than the available uninvoked ones, we have to sample based on the scoring system. Before selecting the already-invoked clients, we exclude the busy clients still running (clients are marked as busy in Algorithm 1 on Line 7, and marked available after completion on Line 22), as we cannot assume that all functions autoscale when invoked again. We then calculate the score for each remaining client as described in Algorithm 2 and append it to the score list. After obtaining all scores, we need to normalize them into values between 0

and 1; in Line 13, we convert them into the probability by summing up all the scores and dividing each client's score by the total score. (The sum of all clients' probability must equal 1.) 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 has obtained a higher score than a CPU client, the normalized probability would also be higher than the CPU client and, therefore, have a higher chance of getting selected. After obtaining the probability of each client, we randomly sample the required number of clients from the list based on the probability. We reset the booster value to 1 for the selected clients so that we do not keep promoting them. We increase the booster value for the available but not selected clients by multiplying it with a promotion value.

Model Evaluation

To ensure a fair evaluation of the newest global model, we maintain a random selection of clients during the evaluation process. The model's performance is assessed by testing it locally on each client's data, and metric results such as accuracy and loss are collected. To prevent bias in the evaluation metrics, we wait for all client results until the timeout, even if some clients experience delays. Consequently, the presence of stragglers may increase the total round duration, particularly for larger models.

5 Experiments

With the system improvements made to *FedLess*, we were able to implement additional strategies and compare their performance to *FedlesScore*. Specifically, we compared *FedlesScore* to three novel training strategies: *FedAvg*, *FedProx*, and *SCAFFOLD*, as well as a previous work focused on serverless FL, *FedlesScan*. Through this evaluation, we sought to gain insights into the performance expectations and limitations of *FedLesScan*.

5.1 Experiment Setup

To effectively scale our experiments and eliminate potential database bottlenecks, we set up the *FedLess* parameter server with *MongoDB* [45] on a dedicated machine equipped with 10 vCPUs and 45GB of RAM. This machine also hosts the file server, providing 200GB of storage to accommodate the four datasets utilized in our study.

We deployed the aggregator function on a self-hosted, single-node *Kubernetes* cluster with *OpenFaaS* deploy, which was configured with 45GB of RAM and 10 vCPUs to ensure sufficient resources for efficient operation.

For the client functions, we aimed to simulate the environment of Google Cloud Functions by configuring each client's hardware based on the specifications provided by *Google Cloud*. In total, we created 200 clients with varying resource allocations:

Client Type	vCPU	RAM	vGPU	# Clients	Cost based on	Cost Per 100 seconds
CPU	1 vCPU	2048MB	-	130	1 vCPU GPC function	0.0029
CPU	2 vCPU	4096MB	-	50	2 vCPU GPC function	0.0058
GPU	-	-	0.2 vGPU	20	NVIDIA Tesla P100	0.0406

Table 5.1: Client type distribution and cost estimate

As listed in Table 5.1, 130 clients were assigned 1 vCPU and a memory limit of 2048MB, mimicking the configuration of lower-end Google Cloud Functions. 50 clients were allocated 2 vCPUs and a memory limit of 4096MB, representing mid-range configurations. The remaining 20 clients were hosted on 5 GPU machines, each equipped with a single *Nvidia P100 GPU*, to simulate higher-end hardware capabilities found in

more demanding federated learning scenarios. This diverse range of client configurations enabled us to study the behavior and performance of our proposed methods under various hardware settings, thereby providing a comprehensive evaluation of their effectiveness in real-world federated learning applications.

5.1.1 Benchmarks and Datasets

In our study, we utilized four datasets from various benchmarks to evaluate our proposed strategy. These datasets are chosen from different domains, including image classification, speech recognition, and language modeling, to provide a conclusive evaluation of the new strategy's effectiveness.

The first dataset we used is the widely known *MNIST* [46] Handwritten Image Database, containing 60,000 training images and 10,000 central evaluation images. These images are randomly distributed among 200 clients.

Two datasets were obtained from the *LEAF* [47] benchmarking framework for Federated Learning: *FEMNIST* for image classification and the *Shakespeare* dataset for character prediction. The *FEMNIST* dataset is an extended version of the *MNIST* dataset and contains over 800,000 images distributed among clients. The *Shakespeare* dataset consists of sentences from the complete works of *William Shakespeare* [48], with over 4 million samples, each of length 80 characters, distributed among clients.

The *FedScale* [49] benchmark, which includes various large-scale realistic benchmark datasets for object detection, word prediction, and speech recognition, is also used in our study. We chose the *Google Speech Commands* dataset [50] from this benchmark, which focuses on real-world speech recognition. This dataset is designed to create simple and useful voice interfaces for applications that use common words like "Yes," "No," and directions and comprises 105,000 1-second audio files distributed among 2,618 clients.

By employing these diverse datasets, we ensure that our evaluation is comprehensive and can robustly assess the proposed strategy's performance across different domains.

5.1.2 Model Configuration

The model architecture used for each experiment is tailored to match its corresponding task. Table 5.2 provides a comprehensive overview of all configurations, such as the number of epochs, batch size, learning rate, and optimizer employed in each experiment. The *MNIST* [46], *FEMNIST*, and *Shakespeare* experiments all utilize the same model architecture outlined in the original *LEAF* [47] benchmark paper. For the *MNIST* experiment, we employ a 2-layer CNN with a 5x5 kernel, and a 2x2 max pooling layer follows each convolutional layer. The model concludes with a fully connected

Dataset	Hyper Parameters			
	Epochs	Batch Size	Optimizer	Learning Rate
MNIST [46]	5	10	Adam	0.001
FEMNIST [47]	5	10	Adam	0.001
Shakespeare [47][48]	1	32	SGD	0.8
Speech Command [50]	5	5	Adam	0.001

Table 5.2: Model hyperparameters by Datasets

Benchmark	Dataset	# Clients	# Clients Per Round	Target Accuracy	Training Timeout
-	MNIST	200	100	0.98	100
LEAF	FEMNIST	200	100	0.7	150
LEAF	Shakespeare	200	100	0.4	550
FedScale	Speech Command	200	100	0.75	60

Table 5.3: Training configuration by Datasets

layer with 512 neurons and a ten-neuron output layer with softmax activation. In total, the model contains 582,026 trainable parameters.

Two datasets were chosen from the *LEAF* benchmark, with *FEMNIST* employing a 2-layer CNN with a 5x5 kernel. Similar to *MNIST*, a 2x2 max-pooling layer is applied to each convolutional layer. The model ends with a fully connected layer with 2048 neurons, followed by an output layer with 62 neurons and softmax activation. This final model has 6,603,710 trainable parameters.

For the *Shakespeare* dataset, a Long Short Term Memory (*LSTM*) [51] recurrent neural network is utilized. The model comprises an embedding layer with a size of eight, followed by two *LSTM* layers with 256 units, and an output layer with a size of 82 and softmax activation. The final model has 818,402 trainable parameters.

In contrast to replicating models from the *FedScale* benchmark, a simple CNN is designed to compare different strategies for the *FedScale* benchmark experiment. Although the designed model only serves to demonstrate any differences, it is as accurate as the models from the original *FedScale* paper. The model architecture consists of two identical blocks, followed by an average pooling layer and an output layer with 35 neurons and softmax activation. Each block consists of two convolutional layers with a 3x3 kernel and a max-pooling layer. To prevent overfitting, a dropout layer follows the max-pooling layer, with a rate of 0.25. The model has 67,267 trainable parameters.

We maintained consistency in the number of clients across all datasets and models used in the experiment, with a total of 200 clients and 100 clients sampled per round.

Additionally, we set different target accuracies and training timeouts for each model since the training duration varies across models. Detailed specifications are provided in Table 5.3.

5.1.3 Evaluation Metrics

This section provides an overview of the metrics and evaluation methodologies used to assess *FedlesScore* and compares its results to those of *FedAvg*, *FedProx*, *SCAFFOLD*, and *FedlesScan*. Our evaluation covers four aspects, namely:

- **Model performance**, which includes accuracy and loss.
- **Selection bias**, which measures the variation in client invocations during training.
- **Cold start ratio**, which assesses the proportion of client functions that have been inactive for a given amount of time, must reinitialize its runtime.
- **Strategy efficiency**, which evaluates the total time and cost.

By examining these aspects, we aim to provide a comprehensive evaluation of the effectiveness of *FedlesScore* in comparison to other federated learning strategies.

In order to evaluate the performance of our model, we utilize two key evaluation metrics: final accuracy and accuracy progress throughout the training process. The final accuracy is measured by the achieved accuracy of the model after a predetermined number of training rounds or upon reaching the target accuracy. To ensure a realistic evaluation in a highly scalable FL system, we adopt a distributed evaluation approach. Randomly selected clients are chosen to evaluate the model on their respective test datasets. The number of clients participating in the testing phase corresponds to the number of clients called per round. Each client calculates its own accuracy using Equation 5.1. To obtain a weighted average accuracy, we consider the clients' relative dataset cardinalities in relation to the total cardinality of the test dataset (N), as depicted in Equation 5.2. This evaluation approach provides a comprehensive assessment of the model's performance, accounting for the varying sizes of client datasets.

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Number\ of\ predictions} \tag{5.1}$$

$$\bar{Accuracy} = \sum_{\forall c \in C} (\frac{n_c}{N} \times Accuracy) \tag{5.2}$$

In order to assess the potential bias of the client selection scheme, we make use of variance plots to display the frequency with which each client is selected throughout

the training session. To measure bias, we compute the difference between the frequency of the least-called client and the most-called client [34]. In light-straggler scenarios, low bias is ideal, while in straggler-heavy situations, it may be necessary to prioritize reliable clients and thus increase bias. By examining the variance plots, we can gain insights into the effectiveness of the client selection scheme and identify any potential biases that may affect the performance of the system.

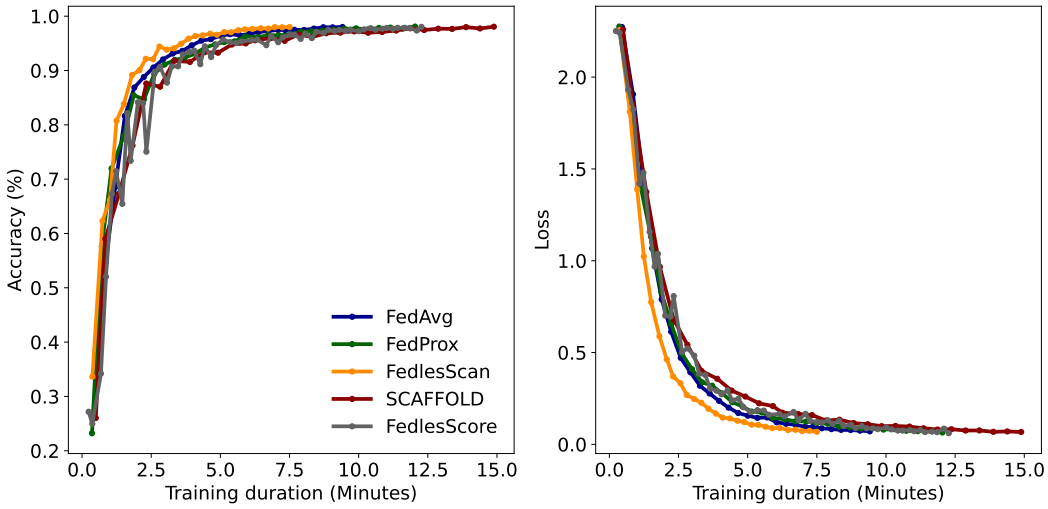
$$Cold\ Start\ Ratio = \frac{\sum_{\forall c \in C} \#Cold\ invocations\ of\ Client\ c}{\sum_{\forall c \in C} \#Total\ invocations\ of\ Client\ c} \tag{5.3}$$

Given that our experiment was conducted on *OpenFaaS* on our own server, the cold start issue was not a concern as the containers were always ready as a function due to the absence of scale to zero (Only available with *OpenFaaS* Pro), which allows systems to reduce resource allocation to minimize costs dynamically. To simulate the cold start effect and compare it across different strategies, we implemented a mock cold start in *Fedless*. For each client invocation, we included a boolean tag indicating whether it was cold-started. Using the time of the last call, we calculated the total number of cold-start invocations and divided it by the total number of invocations across all clients, as shown in Equation 5.3. It is important to note that this metric allows us to assess the impact of cold starting on the overall system performance and compare the effectiveness of different strategies.

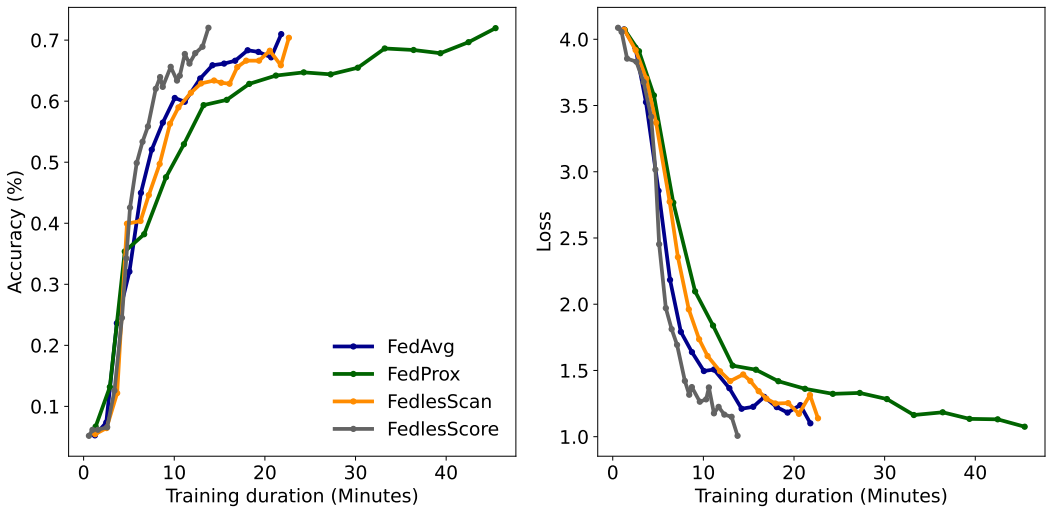
Finally, we conducted a comprehensive analysis of our experiments' time and cost to evaluate the system's efficiency. Regarding time analysis, we measured the total time each strategy took to reach the targeted accuracy from the beginning to the end of the training process. For cost evaluation, we utilized *OpenFaaS* functions on our own server. However, since the cost for this configuration is not directly calculable using standard FaaS providers, we employed the Google Cloud Platform (*GCP*) cost computation model [52][53] to obtain a more accurate cost estimate. To provide more transparency and accuracy in our cost analysis, we created a table that lists the pricing and hardware configurations for each type of client, which can be found in Table 5.1. To ensure accurate cost evaluation, we measured the runtime for each client invocation and used different cost plans for Tier 1 based on the hardware resources provided for the client. Specifically, for clients with one vCPU, we configured them with identical hardware and utilized the pricing of GCP functions with 1 vCPU and 2048MB Memory. For clients with 2 vCPU, we employed pricing with the equivalent hardware resource on GPC Function [52]. For GPU clients, we had to calculate the cost based on the hourly cost for the GPU unit and the fraction of GPU resources utilized during training [53]. The time and cost analyses provide valuable insights into the system's efficiency, especially in heterogeneous hardware and data size settings. These analyses help to determine the most suitable strategy for the given scenario, which can significantly

impact the overall cost of the system in real-world applications where resources are limited.

5.2 Accuracy and Model Performance

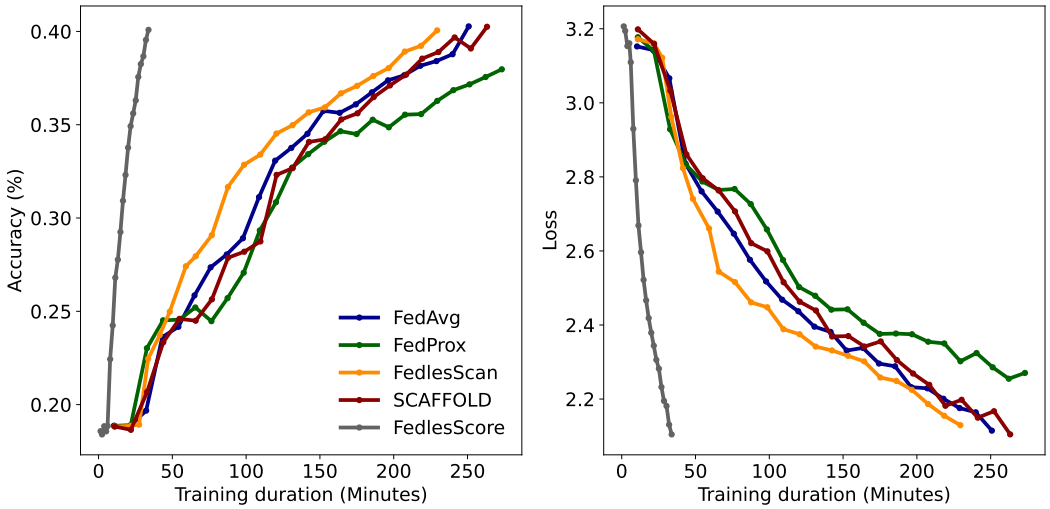


(a) MNIST

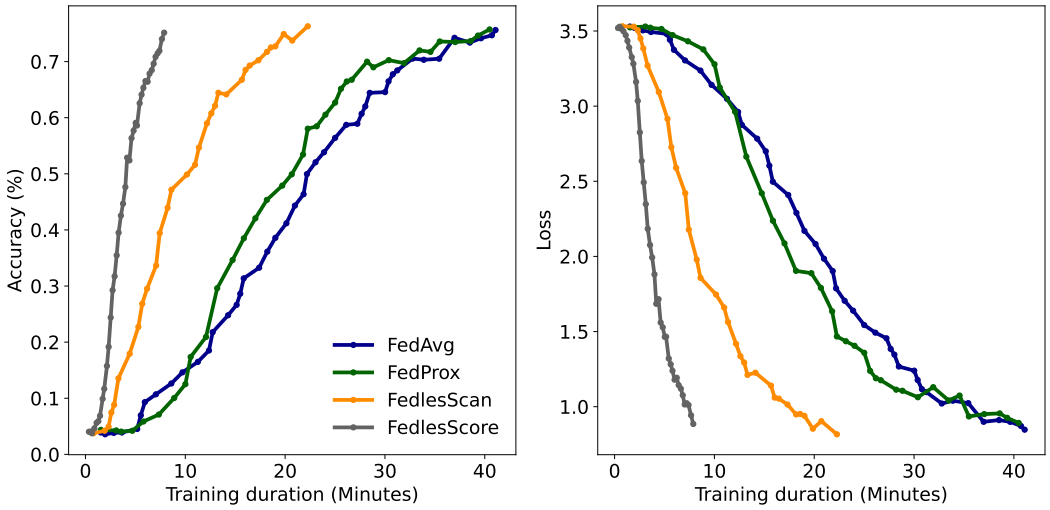


(b) FEMNIST

Figure 5.1: Comparison of perforamace metrics (Accuracy, Loss) between strategies on different datasets.



(c) Shakespeare



(d) Speech Command

Figure 5.1: Comparison of performance metrics (Accuracy, Loss) between strategies on different datasets.

In this section, we will assess the performance of different strategies by comparing their model accuracy and loss across various datasets. We will analyze the convergence of the models and how different strategies affect the overall performance. This analysis will provide insights into the effectiveness of each strategy in achieving high-quality models while maintaining low loss values.

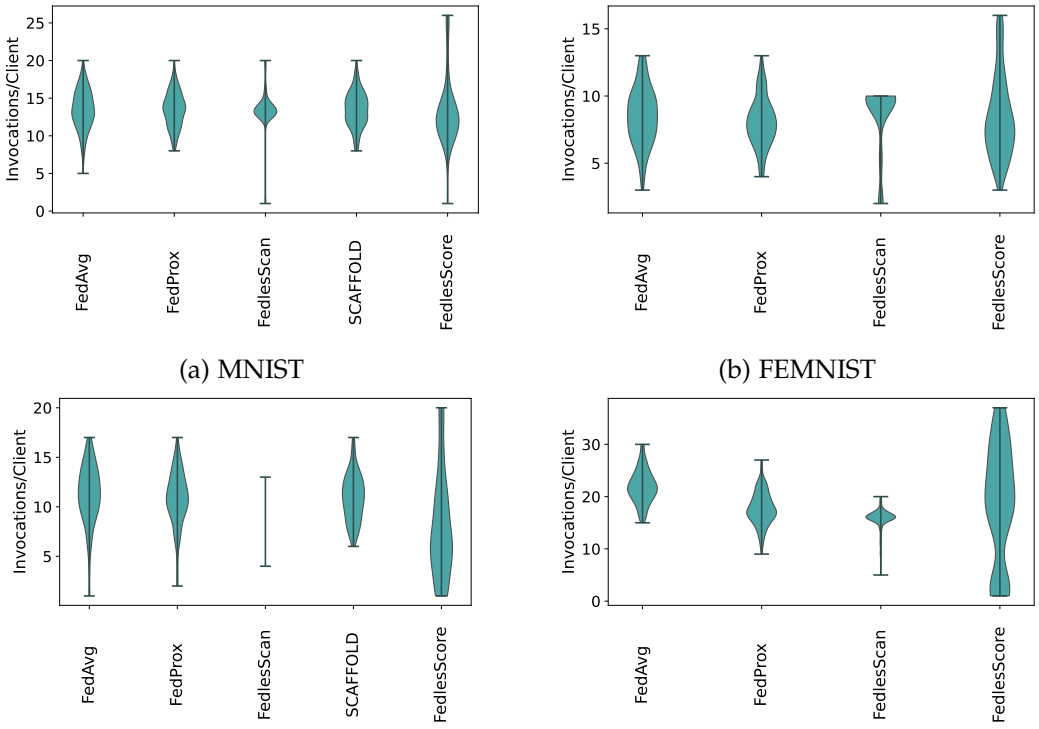
Figure 5.1a shows the performance of different strategies on the *MNIST* dataset, where all strategies have similar results as the clients have only 300 samples of data and the training iteration duration is relatively low. Among the strategies, FedlesScan slightly outperforms others as the data is uniformly distributed, and clustering based on training time makes sense. On the other hand, for the *FEMNIST* dataset shown in Figure 5.1b, *FedAvg* performs better than *FedlesScan* as the data and hardware distribution is heterogeneous, and clustering based solely on training time is no longer appropriate. *SCAFFOLD* is intended explicitly for homogeneous local training steps, where all clients have the same data size or the batch size is dynamically adjusted to the data size to ensure consistent updates across all clients, which is not the case for our setup; therefore, we excluded *SCAFFOLD* in the experiment on *FEMNIST*. *FedlesScore* outperforms all other strategies and achieves the targeted accuracy with a speedup of 1.73x compared to *FedAvg*. There is a significant difference between strategies for the *Shakespeare* dataset, as shown in Figure 5.1c. *FedlesScore* outperforms *FedAvg* with a speedup of 7x, while *FedlesScan* slightly exceeds *FedAvg* with a speedup of only 1.06x. In Figure 5.1d, we show the performance of different strategies on the *Speech Command* dataset. *SCAFFOLD* is again excluded from the experiment due to the heterogeneous number of local steps. *FedlesScan* is faster than *FedAvg* with a speedup of 1.87x, while *FedlesScore* has a speedup of 6.19x, which is almost tripled compared to *FedAvg*. Overall, the results show that *FedlesScore* outperforms other strategies in most cases, except for *MNIST* where all strategies have similar performance. *FedlesScan* also performs well in certain cases, especially when the data is uniformly distributed.

Compared to other strategies, *FedlesScore* consistently achieves the target accuracy faster. This can be attributed to its asynchronous approach and intelligent selection mechanism, which prioritizes clients with larger data sizes and more powerful computational resources, including GPUs.

5.3 Client Selection Bias

This subsection focuses on evaluating the client selection process within the strategies. We will use violin plots to visualize the distribution of selected clients and identify any potential biases or skewness in the selection process. The violin plots presented in Figure 5.2 offer valuable insights into the bias encountered by our strategy. The distribution depicted on the y-axis represents the number of invocations per client. The height difference between the highest and lowest points in the distribution is a measure of bias, with a greater height indicating a bias towards a specific subset of clients. Conversely, a lower height indicates a minor difference between the most and least invoked clients. Moreover, the plot's width at a particular point X can be used to

discern the proportion of clients called X times compared to an arbitrary point Y. If the width at point X is more significant than at point Y, it indicates that more clients were called X times than Y times. This explanation is instrumental in comprehending the performance of our strategy on the four datasets.



(a) MNIST (b) FEMNIST

(c) Shakespeare (d) Speech Command

Figure 5.2: Comparison of client's invocation frequency distribution between strategies

Figure 5.2a depicts the bias observed by our selection strategy for various federated learning methods, including *FedAvg*, *FedProx*, *SCAFFOLD*, *FedLesScan*, and *FedlesScore*, on the *MNIST* dataset. Random client selection is employed by *FedAvg*, *FedProx*, and *SCAFFOLD*, resulting in similar behavior that does not differentiate between stragglers and reliable clients. In contrast, *FedLesScan* is more concentrated in the middle, indicating an equitable distribution of training among clients, with only a few outliers at the top and bottom. The *FedlesScore* approach also shows a relatively normal distribution of client invocations, with most clients being invoked between 5-15 times. However, the line stretches out more than other strategies, with some clients receiving over 25 invocations and few with as little as 5. As the experiment with *MNIST* involves the equal distribution of data among all clients, with each client possessing

300 samples, the scoring primarily differentiates based on the training time, which is mainly impacted by the hardware resources of the clients. Notably, the rise in the curve's upper region is more pronounced for GPU clients, as they are faster and receive more frequent invocations than other clients. However, since *MNIST* is a relatively easy dataset to train, performance is similar among different hardware configurations.

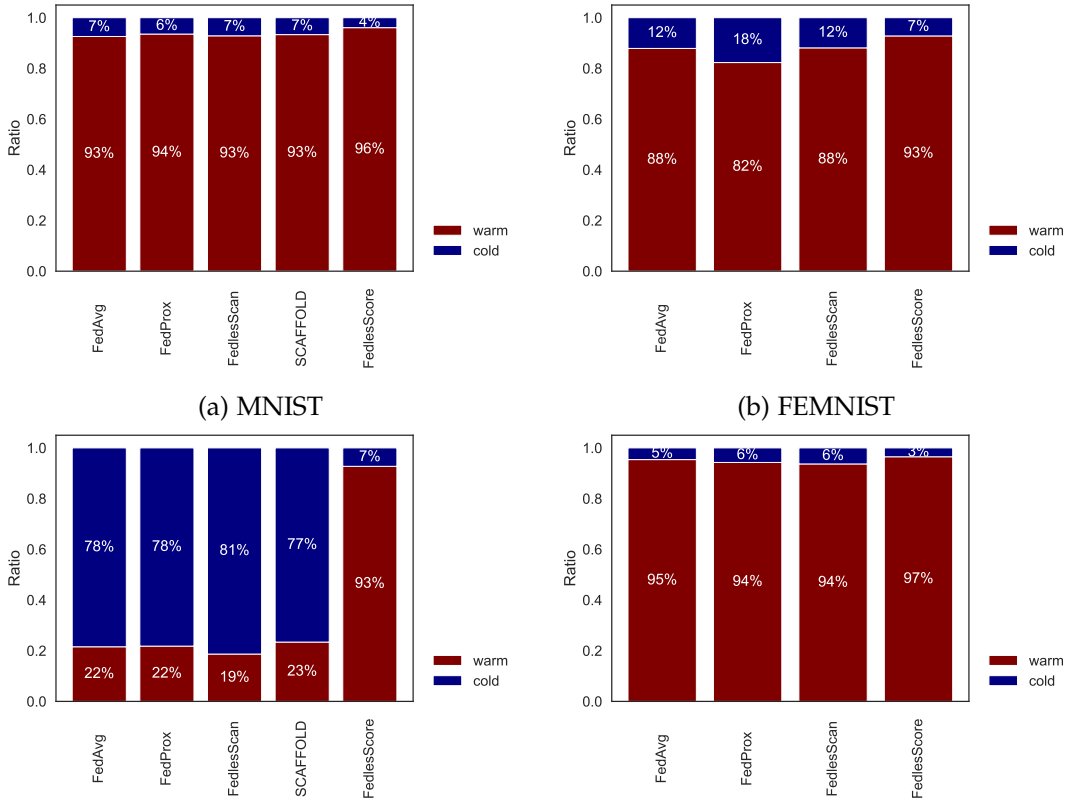
Figure 5.2b displays the distribution of clients based on the number of invocations per client for various federated learning methods, including *FedAvg*, *FedProx*, and *FedlesScan*, on the *FEMNIST* dataset. *FedAvg* and *FedProx* exhibit no clear bias as they randomly select clients. In contrast, *FedlesScan* is concentrated in the middle. However, it starts favoring clients with lower training durations, such as clients with smaller data sizes or more computational resources, like GPU clients. Conversely, clients with more extensive data sizes and fewer computational resources are treated as stragglers and are not included as frequently. However, *FedlesScore* favors some clients at the top, mainly GPU clients and clients with 2 vCPUs. Yet, the remaining plot is relatively smooth, as the probabilistic approach ensures that clients always have a chance of being selected, even with longer training times.

Figure 5.2c demonstrates the distribution of invocations per client for various federated learning methods on the *Shakespeare* dataset, which is similar to the previous datasets analyzed. *FedAvg*, *FedProx*, and *SCAFFOLD* exhibit mostly normal distributions, while *FedlesScan* shows a distribution similar to that of the *FEMNIST* dataset. In contrast, *FedlesScore* exhibits a slightly fatter end on the bottom compared to the *FEMNIST* dataset due to the model's relative difficulty and the noticeable impact of data size. As shown in Figure 4.3, the relationship between training time and hardware resources is significant, especially as the data size increases for clients with only one CPU. Such clients may not complete training before the round ends and, consequently, be available for selection again. In contrast, clients with more computational resources complete training earlier and are available for selection again. This finding underscores the importance of selecting appropriate hardware for data-intensive workloads and optimizing the selection strategy to improve overall system performance.

Figure 5.2d highlights the differences in behavior across various scenarios for different federated learning methods on the *Speech Commands* dataset. *FedAvg* and *FedProx* exhibit a normal distribution, which is expected. *FedlesScan* also shows a normal distribution but with lower variance compared to both *FedAvg* and *FedProx*. The results are similar to those obtained with the *MNIST* dataset, as the model for the *Speech Command* dataset requires relatively small computational resources to compute and optimize across different data sizes. Interestingly, *FedlesScore* exhibits a distribution that differs from that of the *MNIST* dataset. This is because the data is not equally distributed among clients in this experiment, and training duration is one of many factors considered in the selection process. The data size of the clients plays a more significant role, as the

training duration is similar among clients compared to the experiment on *Shakespeare*. These findings underscore the importance of selecting appropriate hardware and optimizing the selection strategy to improve overall system performance, particularly in scenarios with uneven data distributions.

5.4 Cold Start Ratio



(a) MNIST

(b) FEMNIST

(c) Shakespeare

(d) Speech Command

Figure 5.3: Cold start ratio between strategies

In this part, we will examine the cold start ratio of the serverless federated learning strategies. The cold start ratio represents the proportion of invocations that occur after a client has not been called for a certain period. We will evaluate the impact of this phenomenon on model training and identify which strategies are more prone to cold starts, which can affect the overall efficiency of the learning process.

Figure 5.3 presents the effect of cold start on different strategies across various datasets. The figure shows that the cold start ratio is minimal for *MNIST* and Speech Command datasets, as the round duration is relatively short, and the time between each round and client invocation is also short, resulting in a higher chance of a client being invoked again within a short period. However, for *FEMNIST*, as shown in Figure 5.3b, the cold start ratio is more prominent, as the average round duration is longer than the other datasets, as can also be seen in Figure 5.8. *FedlesScore* outperforms other strategies, especially for *FEMNIST*, as it has a shorter average round duration than other strategies.

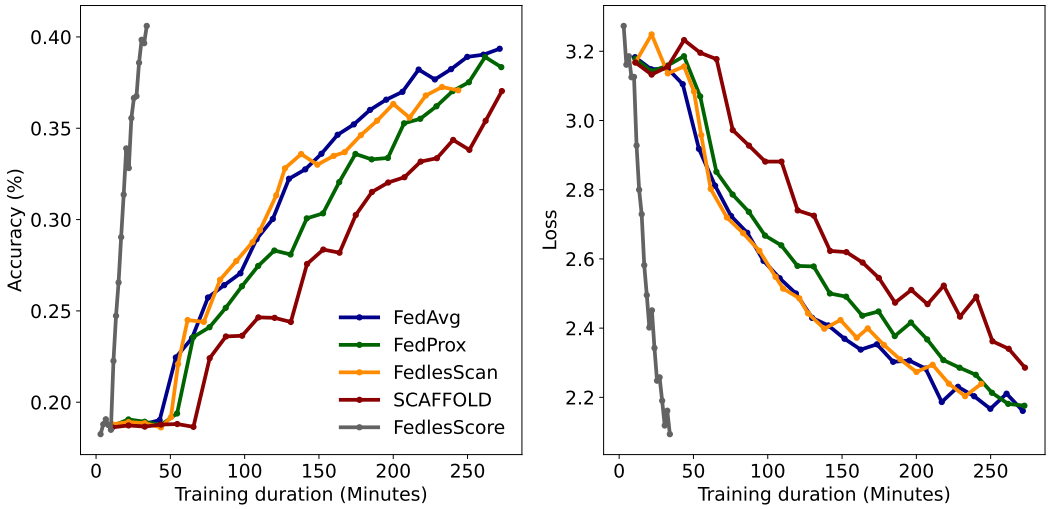
Furthermore, Figure 5.8c shows a significant difference between *FedlesScore* and other strategies, as the round duration remains lower than other methods. The client selection approach in *FedlesScore* also promotes clients with a factor every round it is not selected, contributing to the reduced round duration.

FedlesScore consistently achieves a low cold start ratio across all datasets in our experiments, thanks to its asynchronous approach and well-designed promotion mechanism that prevents clients from missing multiple rounds for an extended period of time.

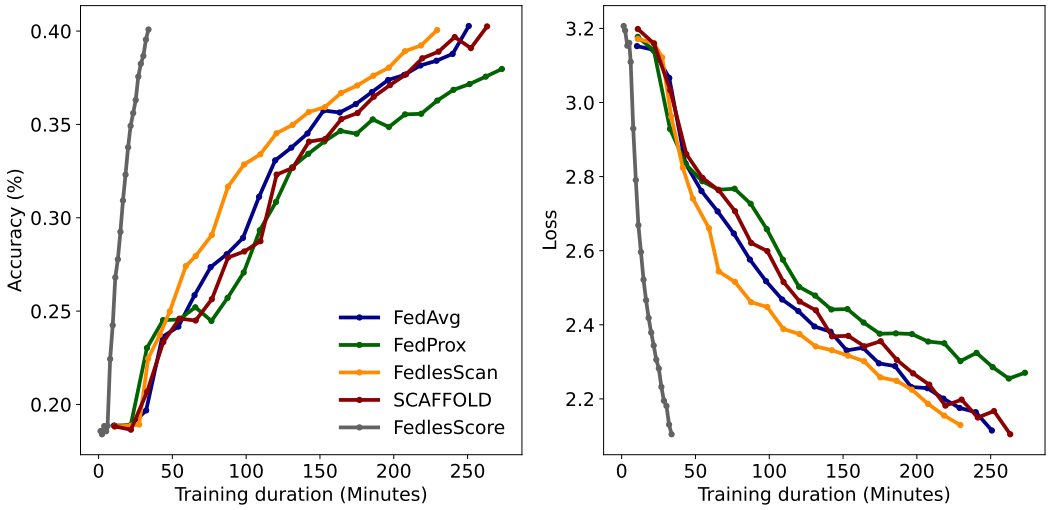
5.5 Client Size

Here, we will evaluate the performance of strategies with different client sample sizes per round (50, 100, 200). By analyzing the impact of client size on the model's performance, we can determine the optimal sample size for each strategy and dataset.

Figure 5.4 shows the accuracy and loss over time for three different sample sizes, where *FedAvg* and *FedProx* perform similarly regardless of the client sample size. *FedLesScan* performs better with a smaller sample size, as the cluster sizes in a heterogeneous hardware setting are smaller. It is more likely to have enough clients in the same cluster to select for the training round. Increasing the sample size may include clients outside the cluster, leading to potential stragglers in the round. In contrast, as discussed in Chapter 3, *SCAFFOLD* performs better with a larger sample size as the global variant becomes more accurate with more clients' results. Li and Diao [28] conducted a more in-depth comparison, revealing that *SCAFFOLD* fails to converge when only 10% of the total clients are selected per round. *FedlesScore*, on the other hand, shows no significant impact on convergence speed 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.

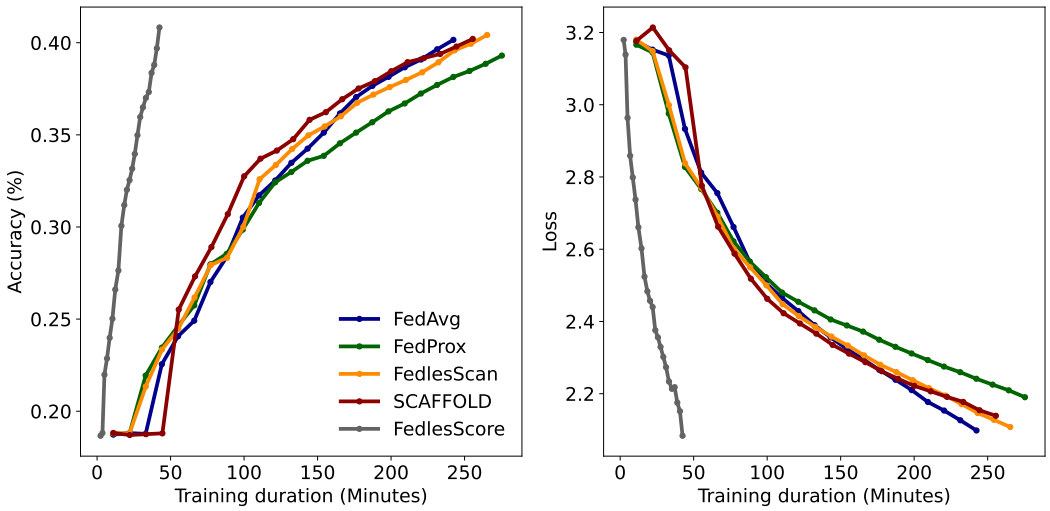


(a) 50 clients per round with a total of 200 clients.



(b) 100 clients per round with a total of 200 clients.

Figure 5.4: Comparison of performance metrics (Accuracy, Loss) between strategies on Speech Command dataset with different client sampling per round.



(c) 200 clients per round with a total of 200 clients.

Figure 5.4: Comparison of performance metrics (Accuracy, Loss) between strategies on Speech Command dataset with different client sampling per round.

5.6 Ablation Studies

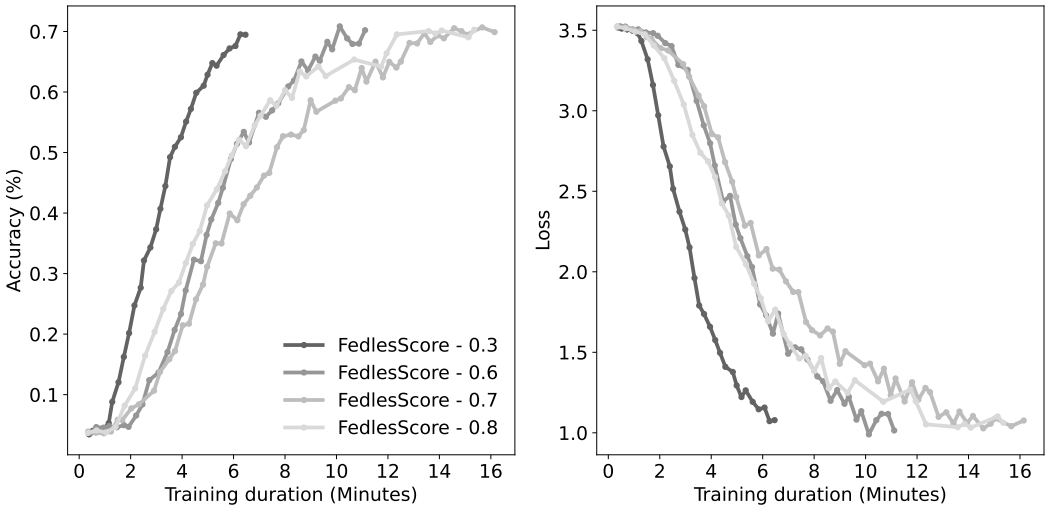


Figure 5.5: Ablation Study: Performance metrics on Speech Command dataset

In this subsection, we will conduct ablation studies to compare the performance of the newly proposed strategy with score-based selection and random selection. We will examine various plots and metrics in depth to understand the strengths and weaknesses of each approach. This comparison will allow us to identify the most effective strategy and understand the key factors contributing to its success.

Due to the synchronous evaluation process, it takes considerable time to evaluate the *Shakespeare* dataset, given some clients' low computational resources and large data sizes. This allows slower clients sufficient time to catch up, minimizing the difference between score-based and random selection. Hence, we choose to use the *Speech Commands* dataset for the ablation study. Before comparing the results of score-based selection with random selection, we first examine the impact of varying buffer ratios on *FedlesScore*. Figure 5.5 presents the performance of *FedlesScore* with score-based selection for different buffer ratios. The best performance is achieved with a buffer ratio of 0.3, outperforming the next best with a buffer ratio of 0.6, with a speedup of 1.7x. The slowest experiment in *FedlesScore* (Buffer ratio = 0.8) has a speedup of 2.4x. Given that 0.3 is the best buffer ratio for score-based selection, we use this ratio to compare with the random selection method.

In Figure 5.6, we compare the performance of score-based and random selection. We observe that *FedlesScore* outperforms all experiments with random selection, with a speedup of 1.73x over the fastest result from random selection. Even when compared

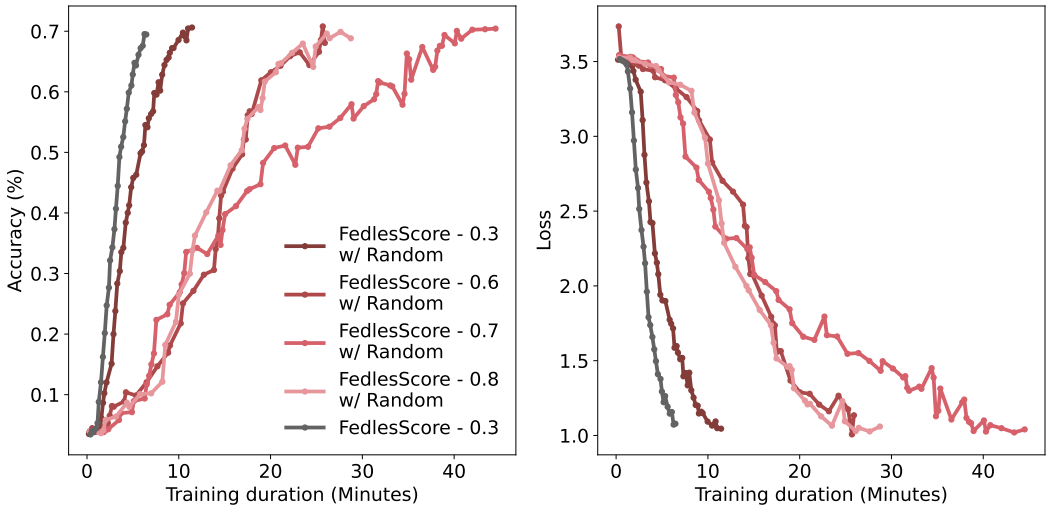


Figure 5.6: Ablation Study: Performance metrics on Speech Command dataset

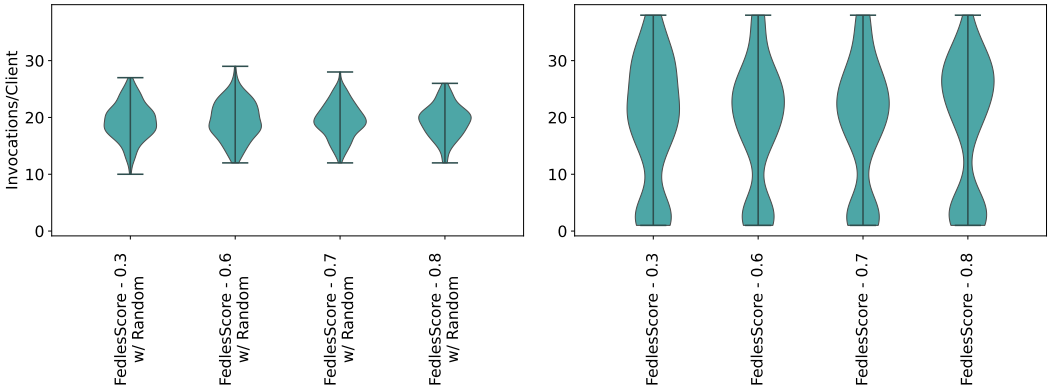


Figure 5.7: Ablation Study: Selection Bias on Speech Command dataset

to the best result from random selection with a buffer size of 0.8, *FedlesScore* with the score-based selection still achieves a speedup of 1.87x. These findings indicate that score-based selection outperforms random selection in all scenarios.

Figure 5.7 provides further insights into the selection bias of each experiment. In the case of random selection, the experiments are consistent across different buffer ratios. The shape is comparable to FedAvg in Figure 5.2d, as all client selections are chosen randomly. As for score-based selection, the experiments also exhibit a similar level of consistency across different buffer ratios. However, as depicted in Figure 5.2d, *FedlesScore* displays a biased selection toward clients with more powerful computational

resources and larger data sizes, resulting in less frequent invocations of clients with smaller data sizes and slower hardware.

Based on our experimental results, it is evident that score-based selection is superior to random selection in the asynchronous setting. *FedlesScore* outperformed all experiments with random selection, even when compared to the best result obtained from random selection. Hence, score-based selection is the preferred approach in this setting.

5.7 Time & Cost Analysis

The round duration distribution for different strategies on various datasets is displayed in Figure 5.8. Figure 5.8a illustrates that there is no significant difference between strategies, with all round durations ranging from 10 to 30 seconds. *SCAFFOLD* appears to be slightly slower than other strategies. It can be argued that the communication overhead required for local and global variants in *SCAFFOLD* is more noticeable in models with shorter training times. For the *FEMNIST* and *Speech* datasets, the round durations for *FedAvg* and *FedProx* are comparable. *FedlesScan* shows slightly better performance on the *Speech* dataset due to its clustering selection approach not including stragglers in every round. *FedlesScore* consistently has shorter rounds than other strategies, particularly for the *Shakespeare* dataset, where it outperforms *FedAvg* with a total speedup of 6.19 (see Table 5.4).

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

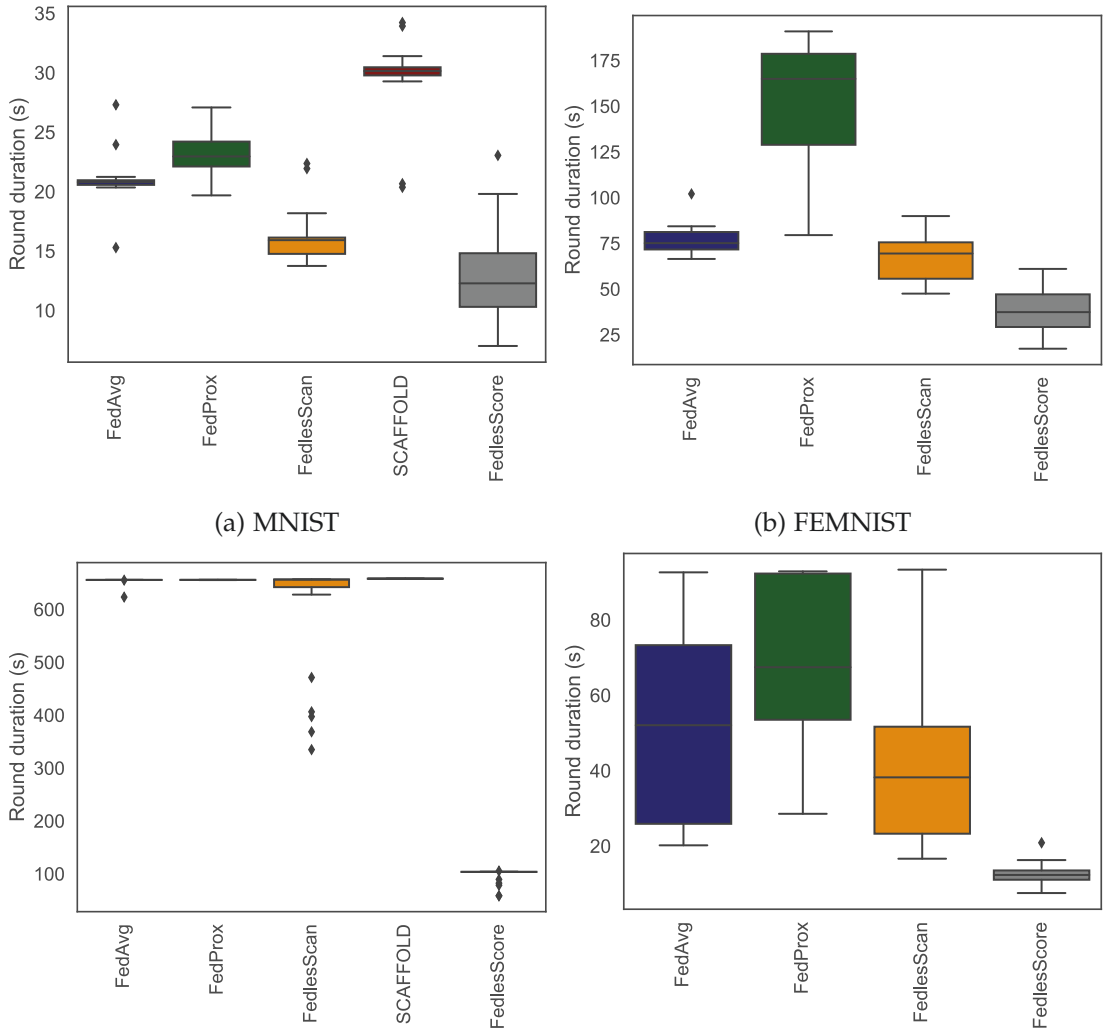
Table 5.4: Full Experiment duration comparison between all strategies across all datasets.

Lastly, we present two tables that provide a summary of the experiments conducted in this study. Table 5.4 compares the duration of various federated learning strategies across different datasets, while Table 5.5 compares their total training cost in USD. Table 5.4 compares the speedup achieved by different federated learning strategies,

Strategy \ Dataset		MNIST	FEMNIST	Shakespeare	Speech
FedAvg		1.13	**2.74**	8.86	2.14
FedProx		1.90	3.83	10.63	2.37
FedLesScan		**1.11**	3.68	10.28	**1.85**
SCAFFOLD		1.52	-	8.41	-
FedlesScore	0.3	11.97	5.99	**6.68**	2.72
	0.6	7.65	9.05	8.91	4.05
	0.7	7.35	4.92	9.47	3.91
	0.8	5.94	9.71	11.11	3.14

Table 5.5: Total experiment cost (USD) comparison between all strategies across all datasets.

including *FedAvg*, *FedProx*, *FedLesScan*, *SCAFFOLD*, and *FedlesScore*, using different buffer values (0.3, 0.6, 0.7, and 0.8) across four datasets: *MNIST*, *FEMNIST*, *Shakespeare*, and *Speech*. *FedlesScore* consistently outperforms other strategies regarding speed, particularly when using a buffer ratio of 0.3. *FedlesScore* (buffer ratio = 0.3) achieves a remarkable speedup of 7.03x for the *Shakespeare* dataset and 6.19x for the Speech dataset. These results suggest that *FedlesScore* is an optimal choice when considering the time efficiency of a federated learning strategy. Table 5.5 compares the total training cost in USD for the same federated learning strategies and datasets. *FedlesScore* is not the least expensive strategy; however, it still remains competitive with other methods. For instance, *FedlesScore* (buffer ratio = 0.3) has a total training cost of 6.68 USD for the *Shakespeare* dataset and 2.72 USD for the *Speech* dataset. In conclusion, the findings from Tables 5.4 and 5.5 suggest that *FedlesScore*, especially with a buffer ratio of 0.3, is a favorable federated learning strategy due to its superior speed compared to other methods and a total training cost that remains competitive.



(a) MNIST

(b) FEMNIST

(c) Shakespeare

(d) Speech Command

Figure 5.8: Round duration with different strategies on different datasets.

5.8 Discussion

In this work, a series of experiments were conducted to evaluate and compare the performance of various federated learning strategies, including *FedAvg*, *FedProx*, *SCAFFOLD*, *FedlesScan*, and *FedlesScore*, in the context of accuracy, model performance, potential bias in client selection, cold start effect, and time and cost analysis. The experiments were conducted with 200 clients with diverse hardware and data sizes. Overall, *FedlesScore* demonstrated superior performance in most cases, except for the *MNIST* dataset, where all strategies exhibited similar results. Moreover, *FedlesScore* consistently had shorter rounds than other strategies, making it an optimal time-efficient choice. Although not the least expensive strategy, *FedlesScore* remained competitive in total training cost. Furthermore, the cold start ratio was minimal for the *MNIST* and *Speech Command* datasets while more prominent for *FEMNIST*. *FedlesScore* outperformed other strategies by having a shorter average round duration, especially for the *FEMNIST* dataset. *FedlesScore* also showed no significant impact on convergence speed with different sample sizes, making it robust across various client sizes. The ablation studies indicated that score-based selection outperformed random selection in all scenarios, with *FedlesScore* proving superior to random selection in the asynchronous setting. These results emphasize the importance of selecting appropriate hardware and optimizing the selection strategy to improve overall system performance, particularly in scenarios with uneven data distributions.

Although we did not directly implement and compare our *FedlesScore* strategy with *FedBuff* [38], our ablation study in Section 5.6 reveals a significant performance difference between our score-based selection method and the random selection method employed by *FedBuff* [38] in our asynchronous approach. While *Pisces* [39] achieves a training time speedup of 1.4x compared to *FedBuff* [38], the ablation study demonstrates that our score-based selection approach consistently outperforms random selection, yielding a training time speedup ranging from 1.7 to 2.4 across different buffer ratios. These findings firmly establish the superiority of our scoring approach in improving the training efficiency of asynchronous federated learning, surpassing both *FedBuff* and *Pisces* in terms of performance.

In conclusion, the experiments demonstrate the advantages of *FedlesScore* in terms of accuracy, model performance, and time and cost efficiency. The findings suggest that *FedlesScore* is a favorable choice for federated learning applications in real-world settings with limited resources and heterogeneous environments.

6 Conclusion and Future Work

In this study, we have significantly enhanced the *FedLess* platform by incorporating two significant advancements. First, we integrated support for the *SCAFFOLD* framework [26] and introduced our novel strategy, *FedlesScore*, specifically designed for heterogeneous environments. *FedlesScore* considers each client's hardware capabilities and data size to score and select clients for the training round. Our experiments on various datasets demonstrate that *FedlesScore* achieves superior accuracy, time, and cost efficiency by effectively leveraging participating clients based on their performance.

Second, we implemented support for simulating the cold start as in FaaS service with major cloud providers, enabling us to evaluate *FedLess*'s performance under realistic cloud provider conditions and assess the impact of a cold start on the training process. As *FedlesScore* currently only factors in hardware and data size for client scoring, potential enhancements could incorporate additional metrics such as data distribution or variance. These metrics could help gauge the diversity of clients' data and determine its potential contribution to the global model without disclosing any private information. Furthermore, an adaptive buffer ratio could be introduced to adjust the waiting time based on the historical behavior of selected clients. This approach would allow for the inclusion of more results in the aggregation round by extending the waiting time by a few seconds, thus preventing results from becoming stale.

Deploying *FedLess* entails a one-time overhead, including setting up various components such as *Kubernetes* clusters, parameter servers, file servers, and *OpenFaaS*. Automating this process could significantly optimize deployment. Additionally, a GUI dashboard for live-tracking the training process and issuing alerts if clients fail would greatly simplify client management and large-scale experiments. This could be achieved using the *Grafana Stack* [54, 55]. It is important to note that *FedLess* currently only supports training with *TensorFlow* [56], while *PyTorch* is widely used in research (e.g., *SCAFFOLD* [26]). Consequently, adding *PyTorch* support to *FedLess* would facilitate faster integration of new strategies, provide more flexible model selection, and enable researchers to test their models in a serverless-based FL environment more rapidly. In conclusion, our work underscores the potential advantages of serverless-based FL and lays the groundwork for future research and refinements in this domain.

List of Figures

1.1 Training efficiency Comparison between *FedAvg* [1] and *FedlesScan* [4] in different hardware settings on the Shakespeare dataset 3
1.2 Selection Bias across different client hardware settings 4

2.1 A schematic architecture of a general federated learning system 7

4.1 Modified Architecture of the *FedLess* platform. The highlighted components shows the modified components and additions to the system. . . 22
4.2 The training workflow of a typical training round in *FedLess* 23
4.3 Training duration based on client hardware setting and Data size. . . . 25
4.4 Evaluation of staleness weights in the relationship between the current round index and result round index 30

5.1 Comparison of perforamace metrics (Accuracy, Loss) between strategies on different datasets. 40
5.1 Comparison of performance metrics (Accuracy, Loss) between strategies on different datasets. 41
5.2 Comparison of client's invocation frequency distribution between strategies 43
5.3 Cold start ratio between strategies 45
5.4 Comparison of performance metrics (Accuracy, Loss) between strategies on Speech Command dataset with different client sampling per round. 47
5.4 Comparison of performance metrics (Accuracy, Loss) between strategies on Speech Command dataset with different client sampling per round. 48
5.5 Ablation Study: Performance metrics on Speech Command dataset . . . 49
5.6 Ablation Study: Performance metrics on Speech Command dataset . . . 50
5.7 Ablation Study: Selection Bias on Speech Command dataset 50
5.8 Round duration with different strategies on different datasets. 53

List of Tables

3.1 Strategies Feature Comparison 21

5.1 Client type distribution and cost estimate 35
5.2 Model hyperparameters by Datasets 37
5.3 Training configuration by Datasets 37
5.4 Full Experiment duration comparison between all strategies across all datasets. 51
5.5 Total experiment cost (USD) comparison between all strategies across all datasets. 52

Bibliography

[1] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas. "Federated Learning of Deep Networks using Model Averaging". In: *CoRR* abs/1602.05629 (2016). arXiv: 1602.05629. URL: http://arxiv.org/abs/1602.05629.

[2] M. Chadha, A. Jindal, and M. Gerndt. "Towards Federated Learning Using FaaS Fabric". In: *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. WoSC'20. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 49–54. ISBN: 9781450382045. DOI: 10.1145/3429880.3430100. URL: https://doi.org/10.1145/3429880.3430100.

[3] A. Grafberger, M. Chadha, A. Jindal, J. Gu, and M. Gerndt. "FedLess: Secure and Scalable Federated Learning Using Serverless Computing". In: *2021 IEEE International Conference on Big Data (Big Data)*. 2021, pp. 164–173. URL: https://doi.org/10.1109/BigData52589.2021.9672067.

[4] M. Elzohairy, M. Chadha, A. Jindal, A. Grafberger, J. Gu, M. Gerndt, and O. Abboud. "FedLesScan: Mitigating Stragglers in Serverless Federated Learning". In: *arXiv preprint arXiv:2211.05739* (2022). URL: https://doi.org/10.48550/arXiv.2211.05739.

[5] *OpenFaaS Github*. Apr. 1, 2023. URL: https://github.com/openfaas/faas (visited on 04/01/2023).

[6] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith. "Federated Learning: Challenges, Methods, and Future Directions". In: *CoRR* abs/1908.07873 (2019). arXiv: 1908.07873. URL: http://arxiv.org/abs/1908.07873.

[7] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman. "Serverless Linear Algebra". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 281–295. ISBN: 9781450381376. DOI: 10.1145/3419111.3421287. URL: https://doi.org/10.1145/3419111.3421287.

[8] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen. "Function delivery network: Extending serverless computing for heterogeneous platforms". In: *Software: Practice and Experience* 51.9 (2021), pp. 1936–1963. DOI: https://doi.org/10.1002/spe.2966. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2966.

[9] 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)*. UCC '21. Leicester, United Kingdom: Association for Computing Machinery, 2021. ISBN: 978-1-4503-8564-0/21/12. DOI: 10.1145/3468737.3494097. URL: https://doi.org/10.1145/3468737.3494097.

[10] 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. DOI: 10.1109/ICDCS51616.2021.00120.

[11] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard. "FuncX: A Federated Function Serving Fabric for Science". In: *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '20. Stockholm, Sweden: Association for Computing Machinery, 2020, pp. 65–76. ISBN: 9781450370523. DOI: 10.1145/3369583.3392683. URL: https://doi.org/10.1145/3369583.3392683.

[12] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler. "rfaas: Rdma-enabled faas platform for serverless high-performance computing". In: (). URL: http://www.unixer.de/publications/img/2021_copik_rfaas_preprint.pdf.

[13] 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. DOI: 10.1109/ICFEC54809.2022.00010. URL: https://doi.org/10.1109/ICFEC54809.2022.00010.

[14] M. Shahrad, J. Balkind, and D. Wentzlaff. "Architectural implications of function-as-a-service computing". In: *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 2019, pp. 1063–1075.

[15] J. Manner, M. Endreß, T. Heckel, and G. Wirtz. "Cold Start Influencing Factors in Function as a Service". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018, pp. 181–188. DOI: 10.1109/UCC-Companion.2018.00054.

[16] D. Bermbach, A.-S. Karakaya, and S. Buchholz. "Using application knowledge to reduce cold starts in FaaS services". In: *Proceedings of the 35th annual ACM symposium on applied computing*. 2020, pp. 134–143.

[17] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. "Borg, omega, and kubernetes". In: *Communications of the ACM* 59.5 (2016), pp. 50–57.

[18] *OpenFaaS: Serverless Functions Made Simple*. Apr. 1, 2023. URL: https://www.openfaas.com (visited on 04/01/2023).

[19] D. Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583.

[20] *OpenFaaS Templates*. Apr. 1, 2023. URL: https://github.com/openfaas/templates (visited on 04/01/2023).

[21] D.-N. Le, S. Pal, and P. K. Pattnaik. "OpenFaaS". In: *Cloud Computing Solutions*. John Wiley & Sons, Ltd, 2022. Chap. 17, pp. 287–303. ISBN: 9781119682318. DOI: https://doi.org/10.1002/9781119682318.ch17. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119682318.ch17. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119682318.ch17.

[22] K. Jayaram, A. Verma, G. Thomas, and V. Muthusamy. "Just-in-Time Aggregation for Federated Learning". In: *arXiv preprint arXiv:2208.09740* (2022).

[23] K. Jayaram, V. Muthusamy, G. Thomas, A. Verma, and M. Purcell. "Adaptive Aggregation For Federated Learning". In: *arXiv preprint arXiv:2203.12163* (2022).

[24] K. Jayaram, V. Muthusamy, G. Thomas, A. Verma, and M. Purcell. "Lambda FL: Serverless Aggregation For Federated Learning". In: *International Workshop on Trustable, Verifiable and Auditable Federated Learning*. 2022, p. 9.

[25] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith. *Federated Optimization in Heterogeneous Networks*. 2020. arXiv: 1812.06127 [cs.LG].

[26] S. P. Karimireddy, S. Kale, M. Mohri, S. J. Reddi, S. U. Stich, and A. T. Suresh. "SCAFFOLD: Stochastic Controlled Averaging for On-Device Federated Learning". In: *CoRR* abs/1910.06378 (2019). arXiv: 1910.06378. URL: http://arxiv.org/abs/1910.06378.

[27] P. Glasserman. *Monte Carlo methods in financial engineering*. Vol. 53. Springer, 2004.

[28] Q. Li, Y. Diao, Q. Chen, and B. He. "Federated Learning on Non-IID Data Silos: An Experimental Study". In: *IEEE International Conference on Data Engineering*. 2022.

[29] J. Wang, Q. Liu, H. Liang, G. Joshi, and H. V. Poor. *Tackling the Objective Inconsistency Problem in Heterogeneous Federated Optimization*. 2020. arXiv: 2007.07481 [cs.LG].

[30] J. Wang, Z. Xu, Z. Garrett, Z. Charles, L. Liu, and G. Joshi. "Local adaptivity in federated learning: Convergence and consistency". In: *arXiv preprint arXiv:2106.02305* (2021).

[31] Z. Chai, A. Ali, S. Zawad, S. Truex, A. Anwar, N. Baracaldo, Y. Zhou, H. Ludwig, F. Yan, and Y. Cheng. "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.

[32] B. Cox, L. Y. Chen, and J. Decouchant. "Aergia: leveraging heterogeneity in federated learning systems". In: *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. 2022, pp. 107–120.

[33] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury. "Oort: Informed Participant Selection for Scalable Federated Learning". In: *CoRR* abs/2010.06081 (2020). arXiv: 2010.06081. URL: https://arxiv.org/abs/2010.06081.

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

[35] Z. Chai, Y. Chen, A. Anwar, L. Zhao, Y. Cheng, and H. Rangwala. "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.

[36] C. Xie, S. Koyejo, and I. Gupta. "Asynchronous Federated Optimization". In: *CoRR* abs/1903.03934 (2019). arXiv: 1903.03934. URL: http://arxiv.org/abs/1903.03934.

[37] Y. Chen, Y. Ning, M. Slawski, and H. Rangwala. "Asynchronous online federated learning for edge devices with non-iid data". In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE. 2020, pp. 15–24.

[38] J. Nguyen, K. Malik, H. Zhan, A. Yousefpour, M. Rabbat, M. Malek, and D. Huba. "Federated Learning with Buffered Asynchronous Aggregation". In: *CoRR* abs/2106.06639 (2021). arXiv: 2106.06639. URL: https://arxiv.org/abs/2106.06639.

[39] Z. Jiang, W. Wang, B. Li, and B. Li. "Pisces: Efficient Federated Learning via Guided Asynchronous Training". In: *Proceedings of the 13th Symposium on Cloud Computing*. SoCC '22. San Francisco, California: Association for Computing Machinery, 2022, pp. 370–385. ISBN: 9781450394147. DOI: 10.1145/3542929.3563463. URL: https://doi.org/10.1145/3542929.3563463.

[40] *Amazon Cognito - secure, frictionless customer identity and access management*. Apr. 1, 2023. URL: https://aws.amazon.com/cognito/ (visited on 04/01/2023).

[41] *Schedule GPUs on Kubernetes*. Apr. 1, 2023. URL: https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/ (visited on 04/01/2023).

[42] T.-A. Yeh, H.-H. Chen, and J. Chou. "Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud". In: *Proceedings of the 29th international symposium on high-performance parallel and distributed computing*. 2020, pp. 173–184.

[43] S. Li Pei and Zheng. *4paradigm/k8s-vgpu-scheduler: Open AIOS vGPU scheduler for Kubernetes*. 2022. URL: https://github.com/4paradigm/k8s-vgpu-scheduler.

[44] D. G. Altman and J. M. Bland. "Statistics notes: the normal distribution". In: *Bmj* 310.6975 (1995), p. 298.

[45] *MongoDB: The Developer Data Platform*. Apr. 1, 2023. URL: https://www.mongodb.com (visited on 04/01/2023).

[46] L. Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142. DOI: 10.1109/MSP.2012.2211477.

[47] 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*. 2019. arXiv: 1812.01097 [cs.LG].

[48] W. Shakespeare. *The Complete Works of William Shakespeare*. Jan. 1, 1994. URL: https://www.gutenberg.org/ebooks/100.

[49] F. Lai, Y. Dai, S. Singapuram, J. Liu, X. Zhu, H. Madhyastha, and M. Chowdhury. "FedScale: Benchmarking Model and System Performance of Federated Learning at Scale". In: *Proceedings of the 39th International Conference on Machine Learning*. Ed. by K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato. Vol. 162. Proceedings of Machine Learning Research. PMLR, July 2022, pp. 11814–11827. URL: https://proceedings.mlr.press/v162/lai22a.html.

[50] P. Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. 2018. arXiv: 1804.03209 [cs.CL].

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

[52] *Pricing, Cloud Functions, Google Cloud*. Apr. 1, 2023. URL: https://cloud.google.com/functions/pricing (visited on 04/01/2023).

[53] *Pricing, Compute Engine: Virtual Machines (VMs), Google Cloud*. Apr. 1, 2023. URL: https://cloud.google.com/compute/all-pricing?authuser=2#gpus (visited on 04/01/2023).

[54] M. Chakraborty and A. P. Kundan. "Grafana". In: *Monitoring Cloud-Native Applications: Lead Agile Operations Confidently Using Open Source Software*. Berkeley, CA: Apress, 2021, pp. 187–240. ISBN: 978-1-4842-6888-9. DOI: 10.1007/978-1-4842-6888-9_6. URL: https://doi.org/10.1007/978-1-4842-6888-9_6.

[55] E. Betke and J. Kunkel. "Real-time I/O-monitoring of HPC applications with SIOX, elasticsearch, Grafana and FUSE". In: *High Performance Computing: ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^ 3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers 32*. Springer. 2017, pp. 174–186.

[56] *TensorFlow: end-to-end open source machine learning platform*. Apr. 1, 2023. URL: https://www.tensorflow.org/ (visited on 04/01/2023).