DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

High Performance Federated Learning Using Serverless Computing

Evgeni Kiradzhiyski

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

High Performance Federated Learning Using Serverless Computing

Hochperformantes Föderiertes Lernen mit Serverless Computing

Author:	Evgeni Kiradzhiyski
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	M.Sc. Mohak Chadha
Submission Date:	15.07.2022

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

Munich, 15.07.2022 Evgeni Kiradzhiyski

Abstract

Smartphone ownership and internet usage are growing at an extraordinary rate nowadays. In fact, more than 3 billion mobile and more than 12 billion Internet of Things (IoT) devices such as autonomous vehicles and smart homes are used as primary computing devices for the accomplishment of many different daily business tasks. These devices have access to a wealth of data and, thus, provide promising opportunities for crafting machine learning (ML) models, due to their rich data nature. Much of the data generated, and collected by the devices, however, is private in nature, requiring special attention. Dealing with heterogeneous and, more importantly, sensitive data, necessities careful handling as any improper usage and data mishandling might have severe consequences, affecting millions of users worldwide. This, however, contradicts traditional distributed ML training approaches, which require the training data to be collected at a central place, making these inappropriate choices for privacy-sensitive domains. This gives a rise to decentralized learning techniques such as Federated Learning (FL), a novel paradigm, which is gaining a lot of popularity in both industry and academia recently. FL enables training on a large corpus of decentralized data, overcoming many of the privacy-related issues by allowing devices participating in the training process to update the parameters of a shared model locally, and send only the model updates to a central server. Thus, the actual training data is never sent to a central server, which means that no local record leaves the device at any point in time. As a new approach on the horizon, FL is an active area of research and experiments. Scientists are heavily studying the FL domain by examining and applying different techniques in the domain. Recent research works propose the application of serverless computing for ML/DL training in the FL context, arguing that components involved in FL can benefit from this new computing paradigm. This work sets on top of the foundations of a framework for serverless FL. We improve and evaluate the performance of an existing serverless system for FL training by replacing the technology used for the parameter server. With the use of *Redis* and the *redisAI* module, we achieve a significant performance speedup and outperform the existing parameter server solution by orders of magnitude. We evaluate the performance of the system in a large-scale scenario considering up to 300 clients. Moreover, in order to examine the system's capability of handling larger models, we increase the model size 8 times and successfully perform FL training. In addition to that, we simplify the client as well as the FL server-side workflows by eliminating the need for serialization of the model parameters. Lastly, we evaluate the running times of the system with multiple replicated parameter servers and compare the results with the existing measurements. The introduced improvements allow us achieve more than **3 times** better client-side performance, and up to **4 times** better performance on the FL server-side, once the aggregator combines the client updates. As a result of that, we are furthermore able to reduce the costs for the use of *FedLess* up to **50%**.

Contents

1 Introduction

As a result of the technological advancement over the past decades, a decent amount of today's world population uses mobile phones and tablets as primary computing devices on a daily basis [1, 2]. Such devices nowadays serve for the accomplishment of a various number of daily tasks and activities in many diverse areas, for example - education, business, entertainment, and social networking among many others [3]. In fact, the usage of smart devices has drastically increased, especially in the last two years, driven and primarily impacted by the global pandemic caused by Covid-19 and the resulting shift towards digitization in many different fields with school and higher education, for instance, being among the most heavily influenced ones [4]. Recent studies show a vast increase in the usage of mobile devices for diverse learning activities over the last years among students such as library services, access to learning material systems, writing texts for examination papers, internet research during lessons, reviewing grades, sending emails to teachers, and others [5].

In general, smartphone ownership and internet usage continue to grow in emerging economies at an extraordinary rate - a phenomenon that has been observed over the past decade. The rise in the use of mobile devices around the globe between 2013 and 2015 is in fact *16%*, climbing from *21%* to *37%* in just two years [6, 1]. According to a recent survey conducted by the *Pew Research Center*[1] in *2021*, the share of people among the Americans that own a smartphone has even grown to *85%*, marking a huge increase in just less than ten years [7]. Along with smartphone usage, there has also been an increasing number in the use of other powerful end-user, and network edge devices such as smart home applications, connected vehicles, connected industrial equipment, and many others [8]. Currently, there are nearly 12 billion connected Internet of Things (IoT) devices and more than 3 billion smartphones worldwide that are being used daily [9].

With the technological evolution, many of these modern mobile and IoT devices - autonomous vehicles, smart homes, smart wearable devices, and many more have access to a wealth of data [10, 11] such as photos [12], videos [13], location information [14], etc. This rapid development of new technologies and the rise of social networking applications, in particular, have led to an exponential and unprecedented growth of the data generated at the network edge [15, 16], with the prediction that the data generation rate will exceed the capacity of today's Internet in the near future [17]. These smart devices generate manifolds of data each day [8], which, opens up various possibilities for meaningful research and industry applications [2]. Due to the devices' rich data nature, they provide opportunities for crafting machine learning (ML) models that can be used to analyze data sets, build decision-making systems, empower more intelligent applications, and thus, improve user experience on the

[1] `https://www.pewresearch.org/`

device [18, 19, 11, 20, 16]. Examples of potential advancements that can be achieved as a result of applying various machine learning techniques include obtaining information about image detection and classification, controlling self-driving cars [21], recognizing speech [22], predicting both customer behavior [23] and future events [15], medical screening [24] and many more. Gained insights are then used to improve workflows in various use cases - mobile keyboard predictions on smartphones [25, 26], traffic condition monitoring on IoT devices [27], patient mortality and hospital stay time predictions on health-card devices [28], on-demand traffic condition estimation [29], and many more [30, 31, 32, 33].

Referring to a sub-field of computer science concerned with computer programs that are able to "learn" from experience and thus, improve in performance over time on a specific task [34, 35], machine learning-based algorithms are mainly used to detect data patterns in order to automate complex tasks or make predictions [36]. Machine learning systems find applications in many domains of science, business, and government - they are used to identify objects in images, transcribe speech into text, select relevant results of a search to name a few [37]. In particular, the word "learning" is a metaphor and does not imply that computer systems are artificially replicating the advanced cognitive systems thought to be involved in human learning [38], but rather refers to learning these algorithms in a functional sense as they can change their behavior to enhance performance on some task based on previous experiences [36]. The formal definition of machine learning, however, is quite broad, varying from simple data summarization with linear regression to multi-class classification with support vector machines (SVMs) and deep neural networks (DNNs) [39, 40]. The latter are recently showing extremely promising results on various complex tasks such as image classification, natural language understanding [41], sentiment analysis and question answering [42], language translation [43, 44], etc. Neural networks have countless applications nowadays in many different areas and are used by many people worldwide in their daily routines [45]. One key enabler of all these machine learning techniques is the ability to learn models given a very large amount of data [15]. Being one of the reasons behind the rapidly growing interest among academia in smart devices that are able to generate huge amounts of data, this promises great improvements in many different applications.

Nevertheless, there are some important aspects that require careful consideration when dealing with such heterogeneous and more importantly, quite sensitive data generated by these devices. The large-scale collection of data entails some risks of improper usage and data mishandling, which might have huge consequences [46]. In fact, much of the data generated and collected by these devices is private in nature [47] and often contains privacy-sensitive information, making data owners reluctant to upload their data to a central server for learning purposes [46, 19] as usually done in the standard ML setting nowadays. Examples include personally identifiable information (e.g. passport information), payment data (e.g. bank accounts, credit card information, payment invoices, financial documents), protected health information (e.g. medical records containing disease information), and others [45]. Additionally, by lacking serious privacy and security considerations, sensitive data are highly exposed to undesirable disclosure, attacks, and cyber risks [45], especially when dealing with a large-scale data collection. Such threats might result in disclosing a huge amount

of private data to the public and, therefore, affect millions of users worldwide with *eBay*[2] (145 million users affected in 2014), *Yahoo*[3] (3 billion users affected in 2013-2014), *LinkedIn*[4] (millions of SHA-1 passwords breached by an attacker in 2016), and *Equifax*[5] (147.9 million customers affected in 2017) being among the worst examples of data breaches recorded in the 21st century [48].

Thus, due to the growing concerns about data privacy and confidentiality, in many industry sectors e.g. banking and healthcare domains [49], sharing data is even strictly forbidden as it might violate many regulations [50, 51, 52] like the European Commission's *General Data Protection Regulation* (GDPR) [53, 54], the Consumer Privacy Bill of Rights in the U.S. [55], the *Health Insurance Portability and Accountability Act* (HIPAA) [56], etc. Nonetheless, these data-related restrictions conflict with the traditional setting in which machine learning is applied, which usually requires all training data to be stored at a centralized location [57, 10] as it might compromise users' privacy with potential eavesdropping attacks. Therefore, an alternative way to benefit from the rich-data devices without violating confidentiality and, thus, ensuring their data privacy has to be considered.

A new, emerging technology that arises on the horizon and deals with many of these issues is Mobile Edge Computing (MEC). In the MEC framework, cloud computing capabilities are provided to mobile devices [58], meaning that these devices have the ability to offload their tasks to the MEC servers [59]. It consists of edge nodes, backed with storage and computation capabilities that work together with a remote cloud, in order to successfully perform distributed tasks at scale [15]. The data is stored locally and processed to a server with global coordination [60, 61]. There exist some research works that have considered training machine learning models centrally, but serving and storing them locally instead, e.g. in mobile user modeling and penalization [62, 63]. However, due to the growing computational and storage capabilities of the devices within distributed networks, it is possible to benefit from these strengthened resources on each device [64]. Thus, by making use of the enhanced capabilities [65] of the end devices, the main idea is to bring the model training closer to where the data is being produced [66], which has led to a growing interest in *federated learning*.

Federated learning (FL) is a new distributed machine learning approach that enables training on a large corpus of decentralized data [67, 68] and overcomes many of the above-stated issues. This paradigm enables multiple edge clients to collaboratively train a common model and produce a global inference without sharing their local private data [69, 70]. The locally trained and optimized model parameters are being aggregated and contribute to the global model. The raw training data is collected and stored *only* at the edge nodes and *never* sent to a central place [15], which means that no local record leaves any device at any point in time during the process. FL decouples the model training from the need of directly accessing the training data and allows training of machine learning models even on private, strongly user-based information, without compromising privacy as the data never leaves the device [10, 18]. Additionally, FL provides security guarantees with security techniques such

[2] `https://www.ebay.de/`
[3] `https://de.yahoo.com/?p=us`
[4] `https://www.linkedin.com/`
[5] `https://www.equifax.com/`

as differential privacy [71] and secure multiparty computation [46]. This privacy-preserving learning technique allows researchers and practitioners to benefit from shared models that are trained under the coordination of a central server (e.g. cloud), having a large amount of data at hand and without violating any regulations [10, 18]. Thus, it can be considered an instance of the general approach of *bringing the code to the data, instead of the data to the code* [67], allowing training of ML models by addressing fundamental problems in terms of privacy, ownership, and locality of data [72, 73], which are hard to avoid in the standard ML setting.

Nonetheless, along with solving issues related to the privacy of the data, FL comes at some cost and introduces new challenges that require special attention as well as handling [74]. For instance, the involvement of many resource-constrained edge and cloud devices and a shared model makes the management of FL clients challenging [75], turning the communication between all involved devices into a potential bottleneck of FL systems. Additionally, due to the rapid growth of clients in the FL setting, dealing with a large number of clients requires special attention as scaling the FL training process becomes more difficult [75].

This work sets on top of the foundations from Chadha, Jindal, and Gerndt [75], who applied Serverless Computing in an FL setting to address challenges such as efficient management of participating workers (*clients*) and further argues that both components in the FL setting - the server and the clients can benefit from a new computing paradigm called Serverless Computing (*Serverless*). Serverless computing has recently emerged as a new type of computation infrastructure [76], which abstracts away operational concerns such as provisioning and management of cloud computing resources from infrastructure engineers and hands these over to a Function-as-a-Service (FaaS) platform [77, 78]. Applications are developed and deployed in small pieces of cloud-native code - *functions*, that respond to a specific event [78].

The problems that serverless technologies try to solve align closely with many of the challenges that occur in current FL systems, which is a compelling motivation behind the work throughout this paper. By applying serverless in an FL setting there is a potential to achieve an improvement in terms of costs on the FL-client side and performance when dealing with a sudden increase of computationally expensive workloads on the FL server side. An example of such is the aggregation of the latest clients' results from a specific FL round.

Thus, the focus throughout this work is on the intersections of these two technological trends - it is concerned with distributed ML training on top of serverless computing, aiming to overcome some of the above-stated issues in a typical FL setting. It builds on top of the foundations set in the development of *FedKeeper*[6] - a framework for efficiently managing FL over a combination of connected *Function-as-a-Service* (FaaS) platforms, i.e., FaaS fabric [75] and its successor, called *FedLess*, and aims at addressing known issues with the existing solution. The first part of this work addresses known issues in terms of performance and scalability and aims at replacing the existing parameter server [79] solution and, therefore, achieve better efficiency at FL training. Moreover, it looks at different weights serialization approaches to adopt the newly introduced parameter server implementation and achieve a performance speedup. The second part of this work is concerned with evaluating the system's performance with the consideration of multiple parameter servers, aiming at a

[6] `https://github.com/ansjin/fl_faas_fabric`

further-increased efficiency (e.g. in terms of network latencies). Based on the stated objectives, the study answers the following two research questions:

- **Research Question 1**: *Which technology should the current parameter server be replaced with, in order to increase the system's performance and scalability?*
- **Research Question 2**: *How would the system perform under the consideration of multiple FL servers with replicated parameter servers?*

By approaching and answering these two research questions, we aim at delivering the following key contributions as an outcome:

- We improve *FedLess*' performance by replacing the technology used for the parameter server of the system, solving the problem with the slow model uploads, identified as the system's bottleneck. With the proposed solution, we not only achieve faster model uploads, but faster running times for both the clients and the aggregator functions in general, resulting in reduced costs. We illustrate the performance speedup and the cost reduction by comparing the two parameter server approaches.
- We evaluate the system's performance while eliminating the need for weights serialization by comparing it with the existing solution, which uses a specific serialization technique. We show that this further contributes to the total running time reduction
- We demonstrate the ability of the system to handle larger models by increasing the size up to **8 times** and performing the corresponding evaluations
- We examine the system under the consideration of multiple FL servers by integrating parameter server replication nodes

In general, the improved *FedLess* system achieves more than **3 times** faster running times on the client-side and more than **4 times** on the FL-server side, once aggregating the latest model updates. The exact difference varies depending on the corresponding test case considering the model size as well as the number of clients involved in the training. Nonetheless, the new solution is always superior to the existing system's parameter server technology. As a result of that, we are able to significantly cut the costs associated with FL training using *FedLess*.

The remainder of this paper is structured as follows: Section 2, introduces the concepts of the technological trends at the core of this work in depth, namely FL and serverless, by defining the key terminology used throughout this study. It furthermore describes a typical FL and serverless workflow; Section 3 presents different applications of FL and serverless in the industry as well as related academical findings; Section 4 describes the problem statement and the approach in detail. Additionally, it presents the changes introduced within this work in terms of the system design and workflow. It concludes with details concerned with the implementation of the system; Section 5 presents the results of this study comparing the two technologies for the parameter server in terms of their running times as well as associated costs, presenting the exact improvements in detail. The section concludes with a brief discussion of the findings; Section 6 concludes the whole study with a summary of the exact system evaluations and discusses potential improvements and future work.

2 Background

The following sections provide background information about the concepts that are used throughout this work. Section 2.1 introduces the origin, popularity, definition, and challenges of *Federated Learning*. It furthermore presents the typical workflow in an FL setting. Similarly, Section 2.2 gives information about what is being referred to and understood as *Serverless Computing* (or simply *Serverless*) nowadays, the popularity of the term over the last decades as well as its most recent applications.

2.1 Federated Learning

Introduced by McMahan in 2016 [10], the concept of *Federated Learning* has recently gained popularity among researchers and practitioners [80, 81, 82, 83]. In fact, the interest in federated learning keeps constantly growing over the past years. Figure 2.1 shows the increasing popularity of the search term "federated learning" over the past five years (2017 - 2022), according to *Google Trends*[1]. It is quite that federated learning is getting more and more attention and is rapidly becoming a hot topic of interest.

Figure 2.1: Popularity of the term *Federated Learning* as reported by Google Trends

Ever since its first mention [46], the concept of FL has received great interest among both academia and industry and has been thoroughly studied as it promises great improvements in many different applications [84, 10, 85, 86], providing a solution to the problem of input data privacy [87]. In practice, FL demonstrated its success in a wide range of applications in many different areas [11, 81, 88, 82, 89, 90, 91] including learning sentiment, semantic location, adapting to pedestrian behavior in autonomous vehicles, activities of mobile phone users, face detection, voice recognition, and others. *Google*[2], for example, made use of the Federated

[1] `https://trends.google.com/`
[2] `https://www.google.com/`

Averaging algorithm (*FedAvg*) [10] and applied it to the Google's Gboard, in order to improve next-word [11] as well as suitable emojis [92] predictions based on users' historical text data. Another privacy-sensitive domain where the training data are distributed at the edge and that benefits from federated learning is healthcare, with a wide range of concrete applications such as diagnosis prediction [93], meta-analysis of brain data [94], predicting health events like heart attack risk from wearable devices [95, 72], and many more [96, 97, 98, 99].

2.1.1 Definition and Workflow

Federated machine learning (FL) can be described as a novel paradigm shift toward enabling ML model training in a collaborative manner [2]. The main idea behind the concept is to build machine learning models based on data sets that are distributed across multiple devices while highly concentrating on data privacy and, therefore, preventing possible data leakage [100]. FL decouples the model training from the need to directly access the raw training data [10, 18] with the use of a loose federation of participating devices (*clients*[3]) under the coordination of a central *server* (e.g. cloud). The clients jointly train a global machine learning model while keeping all the training data local and private, unlike the training approach in the traditional, distributed ML training setting with a central storage system [87]. In contrast to uploading its local data set to the central server, each client computes an incremental update to the current global model and communicates only that model update to the server instead [10]. This proceeds in multiple rounds of communication in which a fraction of the clients is selected to participate in the corresponding training round at random. The updates from all the participating clients are then combined in accordance with a predefined aggregation scheme [87].

The optimization problem for training ML models with the use of a training set with n samples can be formally described as a finite-sum minimization problem [101], aiming at finding some optimal, d-dimensional model parameters $w \in \mathbb{R}^d$ by minimizing the corresponding objective function $f : \mathbb{R}^d \to \mathbb{R}$. It takes the following form

$$\min_{w \in \mathbb{R}^d} f(w) \quad \text{where} \quad f(w) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^{n} f_i(w) \tag{2.1}$$

where f_i typically denotes some loss $l(x_i, y_i; w)$ of the prediction on example (x_i, y_i). Also known as empirical risk minimization, it is widely used in various number of both convex and non-convex problems such as logistic-regression, multi-kernel learning, and neural networks [101].

However, the optimization problem in the FL setting, referred to as *Federated Optimization* by McMahan et al. [10], differentiates from the distributed optimization by the following properties: non-IID training data, unbalanced similarity as some users are using a specific service or an app to a much heavier extend compared to others, massive distribution since the expectations are such that the number of clients exceeds the average number of samples per client, and limited communication [10]. In an FL setting, it is to be assumed the data is

[3] "Devices", "clients", and "workers" are used interchangeably throughout this paper.

partitioned over K different clients. Thus, in order to consider the multiple partitions in which all the clients are located, one needs to modify the expression by splitting up the objective function f, which results in the following updated formulation of the original optimization problem

$$f(w) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w) \quad \text{where} \quad F_k(w) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(w) \tag{2.2}$$

This contrasts with the IID assumption made by distributed optimization algorithms, namely, that the partition is formed by a uniform, random distribution of the training examples over the involved clients [10].

The application of stochastic gradient descent (SGD) and its variants seem a natural choice in the FL setting, given the number of its successful applications in the context of deep learning in recent times [10]. Even though SGD can be applied in its original form to the federated optimization providing computationally efficient results as reported by [10], it requires a quite large number of training rounds, in order to produce good model results, a downside examined thoroughly by Ioffe and Szegedy [102]. Within their experiments, the authors trained MNIST for 50000 steps on mini-batches of size 60 even with the use of batch normalization, showing the incredibly large number of rounds involved.

In order to address this disadvantage and increase performance, McMahan et al. [10] apply the large-batch synchronous SGD, an approach evaluated by Chen et al. [103] in the data center setting, and successfully apply this approach in the federated setting with some necessary modifications, referring to this algorithm as *FederatedSGD* (or *FedSGD*) [10]. In *FederatedSGD*, instead of sending back the gradients, each client, by using the current model and its local data, takes a single step of gradient descent. The server then computes a weighted average of the resulting models [10]. A more formal definition of this approach takes the form

$$w_{t+1}^k \leftarrow w_t - \eta \nabla F_k(w_t), \text{ followed by } w_{t+1}^k \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k \tag{2.3}$$

where η is the corresponding learning rate. This can be then further extended by adding more computation to each client by iterating over the local update $w^k \leftarrow w^k - \eta \nabla F_k\left(w^k\right)$ multiple times before the averaging step, which is termed by the authors as *FederatedAveraging* or *FedAvg* [10] and can be considered as the current state-of-the-art method for federated learning [104].

Due to great interest in federated learning among researchers, there exist many variations of the *FedAvg* nowadays, studying primarily the drawbacks of the approach and seeking potential improvements. In fact, the ways how the participating clients update their local models to the server is of great importance for both model convergence and system efficiency [18], giving a rise to the main differentiation between synchronous FL and asynchronous FL. In the original synchronous federated learning approach, clients suffer waiting times, implying more sensitivity to local model training times and irregular or missed model updates, thus, having a negative impact not only on the client-server communication but also on the scalability of the system [105].

Therefore, providing the possibility for clients to calculate their new local updates without the need to wait for other, delayed clients from previous rounds can be seen as a great improvement in terms of the system's performance. Unlike a synchronous update scheme such as *FedAvg*, an asynchronous one would allow for an increased amount of data that is being processed simultaneously and, thus, tends to achieve a higher system efficiency [18]. In fact, asynchronous training is widely used in traditional distributed SGD [106, 107, 108, 109] and usually converges faster than synchronous SGD in real time due to parallelism. Nonetheless, it is impractical in the FL context given the existing unreliable and slow communication as it directly sends gradients to the server after each local update [110]. Thus, one can find some interesting research works dealing with the application of an asynchronous scheme in the FL context, which are presented in Section 3.

2.1.2 Categories and Use Cases

Typically, Federated Learning is divided into two distinct categories based on the distribution characteristics of the data - *Horizontal Federated Learning* and *Vertical Federated Learning* [111]. Horizontal federated learning, also referred to as simple-based federated learning, refers to scenarios in which the data sets of the involved participating clients share the same feature space but in different samples [111]. An interesting horizontal federated learning solution for Android phone model updates, proposed by Google, marks an industry example within this context [10]. On the other hand, in the vertical federated learning setting, also referred to as the feature-based federated learning setting, the data is assumed to be partitioned based on the features - e.g. two different data sets of two distinct participating clients share the same sample but differ in feature space or in other words, the clients hold different information about the very same example [87, 111]. In fact, many of the existing privacy-preserving machine learning algorithms have already been proposed for vertically partitioned data such as secure linear regression, for example [112, 113].

One can additionally distinguish between two different types of domains of FL based on the setting in which federated learning concepts are being applied, namely *Cross-device* and *Cross-silo* learning [74]. In the former setting, a very large number of different, low-powered clients such as mobile or IoT devices are to be observed [87, 74]. The latter is typically concerned with a number of organizations that have an incentive to train a model based on all of their data but are hindered from sharing their data directly, due to legal constraints or confidentiality [74]. The model is trained on siloed data [74] and the clients usually have more local computation power and storage capabilities at hand, in contrast to the characteristics of cross-device federated learning. The participating clients can, for example, be different organizations or geographically distributed data centers [87, 74].

As in many other areas in software engineering, the trade-off of choosing one approach over the other results in a gain in terms of a specific aspect, or several ones, however, at the same time, it comes at the cost of disregarding some of the existing benefits. Even though cross-silo federated learning provides more flexibility in certain aspects of the overall design, it comes at a certain cost and presents a setting in which achieving other properties can be harder [74].

2.1.3 Challenges

Even though FL appears to be similar, in terms of its structure, to the distributed optimization approach for statistical learning [114, 115] in a data center or traditional private data analyses, for example, there exist a number of important differences, resulting in some new challenges that arise in the FL-setting [74]. These challenges, therefore, necessitate an additional effort and consideration of new, concrete solutions when applying federated learning techniques, in order to implement an efficient FL system and benefit from all the advantages in the federated learning context. The following list contains some of the most recurring, key challenges faced in the FL context:

- **Communication cost**: Since training a model over decentralized data heavily relies on communication, one of the core challenges in the federated learning setting is the cost of that communication occurring [116]. Communication can be seen as a serious bottleneck in that setting, being limited by the wide area of mobile network bandwidth [117, 10]. This might then result in a drop in the network by many orders of magnitude as federated networks are usually comprised of a huge number of clients [118, 119]. As a result, the slower network communication affects the performance of the system [64, 120, 19]. Potential approaches to overcome the expensive communication issue include reducing the total number of communication rounds or reducing the size of the messages that are being transmitted each round [64].

- **Heterogeneity**: *Cross-device* FL usually involves a massive amount of clients such as mobile or IoT devices with various computing capacities [57, 47], implying a severe heterogeneity that is rarely found in a data center distributed ML. This wide range of participating clients (e.g. up to 10^{10}), that differ in their computational capabilities (hardware resources) and data quantity, might significantly impact and worsen the training performance and the model accuracy in FL with *FedAvg* [47, 64, 120, 19]. Heterogeneity is, therefore, a significant challenge in the implementation of federated networks. One can differentiate between two different types of heterogeneity - heterogeneity in terms of the systems involved in the federated network (in the following referred to as *hardware heterogeneity*) as well as statistical heterogeneity (in the following referred to as *data heterogeneity*), concerned with the local data available at every device:
  - **Hardware heterogeneity**: As many different devices are included in the inference step in FL, there exists a significant variability in the hardware characteristics among the clients involved in the federated setting [64]. Therefore, it is quite common that the devices differ a lot in their storage, computation as well as communication capabilities, due to many diverse factors such as hardware variability (e.g. CPU and memory), network connectivity (e.g. 3G, 4G, wifi), and power (e.g. battery level), which might result in potential client unavailabilities that can occur at any time [87]. This, however, contrasts with distributed learning, which usually occurs on reliable cluster infrastructure with high network performance and low failure rate [87]. Furthermore, since the communication between client and server

in FL takes place over a high-latency network, the size of the network might additionally cause only a small portion of the devices to be active at once [121], which could hinder these devices from participating in an FL round. Thus, heterogeneous hardware toleration and robustness to dropped devices in the network are to be considered thoroughly when dealing with federated learning [64].

  - **Data heterogeneity**: In FL there is no control over the data distribution of the participating computational nodes [87], which results in heterogeneity in the training data available at the devices. The local data distributions on the clients are usually based on a single user or a group of users. These different contexts lead to significant data distribution skew across all the devices [116]. This, however, has the following consequences that the data set of a single client is not representative of the total population across all clients and, thus, violates the most-commonly used independent and identically distributed (I.I.D.) assumption. Additionally, the number of data points collected across devices may also vary significantly. As some clients might have significantly larger local data sizes than others, heterogeneity may cause stragglers [122, 123] (slow clients), especially in large-scale FL training scenarios, that can potentially impact both training throughput and model accuracy [19, 124, 125]. There exist several approaches that might be helpful to overcome this issue, nonetheless, there are no concrete guidelines for every specific case to the best of our knowledge. One potential approach would be to modify existing algorithms, for example, through hyper-parameter tuning [74]. Additionally, data augmentation might be an appropriate choice to handle the problem and make the data across participating clients more similar in some applications [126].

- **Privacy**: Even though privacy is one of the core motivations behind the use of FL contrasting to the standard distributed ML approach, there are some concerns that require special consideration in FL applications as well. FL can be seen as the better choice in terms of data protection as it shares the model updates instead of the raw data [127, 128], nevertheless, sensitive information can still be revealed throughout the communication involved within the training process, when communicating with the central server, for instance [129].

This list is far from complete as it contains only the most recurring problems in the FL context identified in the literature. In addition to the above-mentioned challenges, we can further note lacking convergence of the shared model, efficient hyper-parameter tuning, and more [74, 18, 130, 131, 132, 133, 47] as issues that require special attention and handling in the FL setting. Similar to the core problems listed above, these issues are an area of active research among academia. Based on the amount of research work in the form of scientific papers and books, one can expect interesting findings and concrete solutions to these problems in near future.

2.2 Serverless Computing

Cloud Computing refers to two overlapping but, in some sense, different concepts. On the one hand, it is concerned with applications that are developed as a service over the Internet, and, at the same time, on the other, with the software as well as hardware systems located in the data centers that provide these services [134]. The former is referred to as *Software as a Service (SaaS)*, and the latter as *Cloud*. It is furthermore to be differentiated between *Public Cloud* - a *Cloud* available in a pay-as-you-go manner to the public and a *Private Cloud* - referring to internal data centers of businesses or organizations that are not made available with a public access [134].

Recently, a new type of computation infrastructure in the cloud computing context is gaining a lot of interest and popularity, namely *Serverless Computing*. Serverless computing refers to the concept of building and running applications that do not require server management. Stakeholders involved in development are provided with a simplified programming model for creating cloud applications as they no longer need to deal with operational concerns [78]. Similar to *Federated Learning*, the term *serverless* dates back to the last ten years and thus, can be classified as a fairly new concept in the technological world. In fact, the ability to use computing resources on demand was seen as a future direction of cloud computing around twelve years ago [134]. In its recent form, serverless was initially introduced and popularized by Amazon in the re:Invent 2014 session "Getting Started with AWS Lambda" [135]. The growing interest in serverless ever since can be depicted by the appearance of the term "Serverless PaaS" in the *Gartner's*[4] Hyper Technologies Report 2017 [136], classified "on the rise", among many other technological trends like "Deep Reinforcement Learning", for instance. Gartner strengthens the increasing relevance of serverless even further, reporting that "the value of serverless computing is on a trajectory of increased growth and adoption." [137]. The search trends by Google illustrate the significant uptick in attention to the topic in recent times. Queries for the term "serverless" matched the historic peak of popularity of the phrase "MapReduce" [138]. This can be seen in Figure 2.2, which shows the popularity of the term from 2004 until recent times according to *Google Trends*. One can detect a constantly growing interest over the past eight years, despite the slight decrease in attention within the last two years.

2.2.1 Definition and Key Terminology

Ever since the beginning of the usage of the term serverless, almost twenty years ago, the understanding of serverless has evolved a lot over time, drifting from its original meaning of not using servers to what we refer to as being serverless nowadays [139]. The exact applications of serverless have undergone a big change as serverless was initially applied to peer-to-peer (P2P) software or client-side only solutions back in the early times after its invention [140, 141]. In recent times, despite the name, serverless does rely on servers [142]. In the cloud context, as srverless is nowadays understood, the term simply means that

[4] `https://www.gartner.com/en`

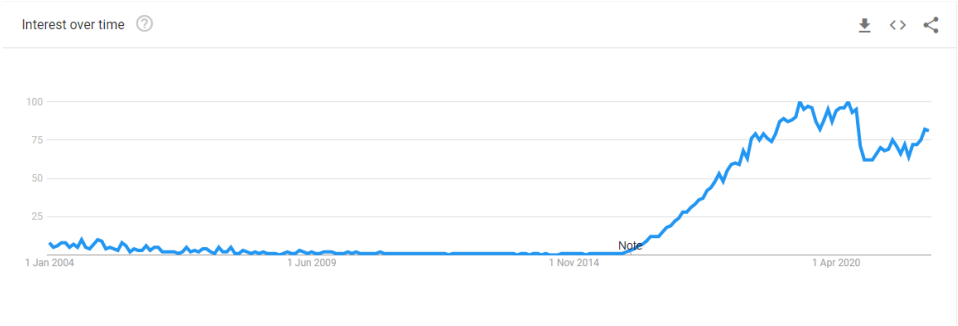


Figure 2.2: Popularity of the term *Serverless* as reported by Google Trends (2004 - 2022)

developers do not need to concern themselves with provisioning or operating servers since serverless solutions are server-hidden. They are designed to hide how scaling is done, in other words, abstract servers away [139].

Nevertheless, as with many different trends in the software community, there does not exist a clear view of what serverless is and, in fact, it encompasses two different but in some sense similar and overlapping ideas [143]. On the one hand, serverless is used to describe applications that fully incorporate third-party, cloud-hosted applications and services, in order to manage server-side logic and state [143]. Prominent examples include the so- called "rich client" applications - single-page web applications (SPAs), mobile applications, authentication services (e.g. *Auth0*, *AWS Cognito*[5] [144]), databases (e.g. *Parse*, *Google's Firebase*[6] [145], *AWS' DynamoDB*[7] [146]), and so on, all being referred to as *(Mobile) Backend-as-a-Service* or shortly *(M)BaaS*. Some of these services even provide "cloud functions" - the ability to run server-side code on behalf of a mobile app, without the necessity to manage servers [78]. An example of such service is *Meta's*[8] Parse Cloud Code.

On the other hand, serverless describes applications with server-side logic still written by the application developer, however, unlike traditional architectures, it's run in stateless containers that can be described with the following characteristics: event-triggered, ephemeral (may only last for one invocation), and fully managed by a third party [143]. The latter is being referred to as *Function-as-a-Service* or in short - *FaaS*, which is the more commonly used and more descriptive name for the core of serverless offerings [138]. In fact, both *BaaS* and *FaaS* concepts are related in their operational attributes (e.g. lacking resource management) and are frequently used together [143], buildig jointly the collective term serverless.

In the cloud, serverless computing appeared as an attractive alternative to already existing cloud computing paradigms, which demand comparably more work when it comes to operational concerns such as provisioning and management of cloud computing resources as serverless does not require dedicated infrastructure [147]. Referring to the official, opening description of AWS's Lambda, it is defined as follows: *"AWS Lambda lets you run code without*

[5] `https://aws.amazon.com/cognito/`
[6] `https://firebase.google.com/`
[7] `https://aws.amazon.com/dynamodb/`
[8] `https://about.facebook.com/meta/`

provisioning or managing servers.", which compactly describes one of the core ideas of serverless - abstracting away server management. With *FaaS*, the vendor handles all underlying resource provisioning and allocation [143]. Simplifying the way in which scalable applications are being developed in a novel, cost-effective manner, serverless platforms can be seen as the next step in the evolution of cloud computing architectures [148]. Serverless describes a more fine-grained deployment model in contrast to other cloud application models, for instance, *Infrastructure-as-a-Service (IaaS)*, *Platform-as-a-Service (PaaS)* (e.g. *Google AppEngine*[9] [149], and *Elastic Beanstalk*[10] from AWS [150]), and container orchestration or *Container-as-a-Service (CaaS)*. These fine-grained, small pieces of cloud-native code, typically written in popular modern programming languages such as JavaScript, Python, Go, C#, any JVM language (e.g. Java, Clojure, Scala), etc., are also referred to as *functions*. The functions are usually triggered in response to an inbound HTTP request or some predefined event determined by the provider [78, 138] such as S3 (file/object) updates, scheduled tasks, and messages added to the message bus with AWS [143], for instance.

As a cloud application development model, however, the exact definition of serverless might still more or less overlap with other cloud models such as *PaaS* and *Software-as-a-Service (SaaS)* [78]. Figure 2.3 provides an overview of different existing cloud computing approaches and visualizes what resources are being provided with each of these different services: for *Infrastructure-as-a-Service (IaaS)*, only a network and a computer are provided, typically a virtual machine (VM); in addition to that, *Container-as-a-Service (CaaS)* provides a (host) operating system as well as a container run-time; *Platform-as-a-Service (PaaS)* adds tools and libraries as well as application run-time to the already available resources from the preceding service. On top of that, in *PaaS*, serverless adds service events and a service mesh. In the case of *Software-as-a-Service (SaaS)*, additionally, the application and its service are given in the hands of the consumer.

In fact, Baldini et al. [78] explains the term Serverless by considering the varying levels of developer control over the cloud infrastructure with the *IaaS* model being the most customizable among the different services, providing the most control to the developer and, therefore, making that person responsible for provisioning the hardware or virtual machines (VMs). That person is in charge of customizing every single aspect of how an application is being deployed and executed, whereas *PaaS* and *SaaS* are on the exact opposite side. This means that the developer has the least control and is unaware of any infrastructure, having only access to prepackaged components or full applications.

Even though the *PaaS* model is often also seen as an approach for abstracting away the management of servers, there are some distinct characteristics when it comes to comparing it to what serverless (*FaaS*) provides. The serverless model provides a "stripped down" programming model based on stateless functions and unlike *PaaS*, it allows developers to implement code snippets not restricted to using pre-packaged applications [78]. *FaaS* is about running back-end code without managing own server systems or own long-lived server applications, which is the key difference when it comes to comparisons with other

[9] `https://cloud.google.com/appengine`
[10] `https://aws.amazon.com/elasticbeanstalk/`

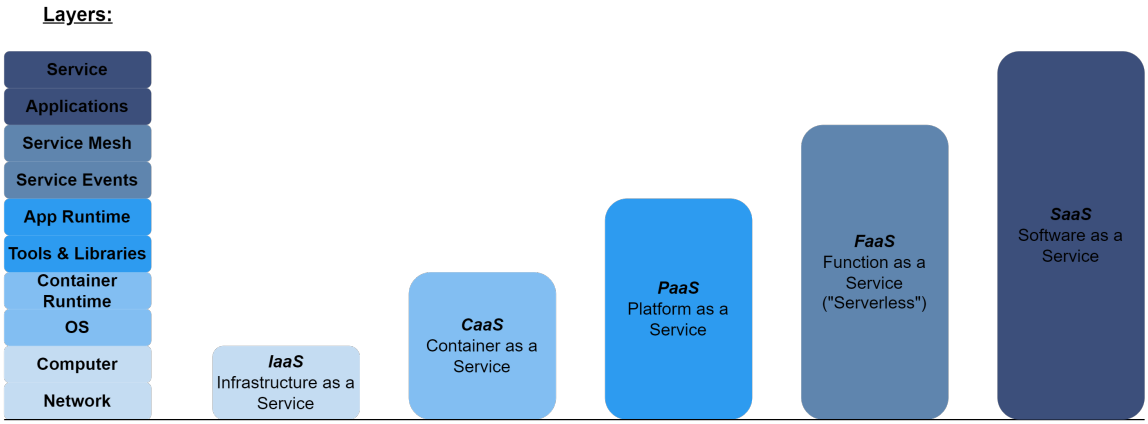


Figure 2.3: Cloud approaches based on the SEIP lecture from *msg group* for TUM [151]

modern architectural trends like containers and *PaaS* [143]. The functions are not meant to be long-running, just the opposite, they are designed from the start to be short-lived [147], usually in accordance with some provider-specific, predefined timeouts for their execution duration. In other words, in contrast to *PaaS*, serverless applications are geared to bring up and down entire applications in a response to some event [143].

Furthermore, unlike *SaaS* or *PaaS*, which are always running but scale on-demand, serverless workloads rather run on-demand, and consequently, scale on-demand [139], which makes them an attractive alternative, due to their cost-saving billing model. In contrast to serverless, most of the other services in the cloud-computing era follow the pay-as-you-go billing method, which charges for allocated resources rather than actual usage [152], forcing cloud users to eventually pay for initially-defined resources that they might not fully use. In serverless, the cloud customer never pays for procured resources that are never used, making use of the 0.1 seconds as a charging metric, in contrast to many VM servers that are using an hourly charge metric [152]. Applications in the serverless context are uploaded to a platform, executed, scaled, and billed based on the exact demand needed [153]. Thus, customers are paying only once a corresponding resource has been consumed. In fact, from a cost perspective, the benefits of serverless architecture are most attractive for bursty, compute-intensive workloads [154]. The latter are appropriate as the price of a function invocation is proportional to its running time. Additionally, bursty workloads are suitable for serverless as functions are able to scale to zero, which means that there is no cost to the customer when the system is idle [154].

In spite of the name, servers are still needed in serverless, however, decisions such as the number of these servers and their capacity are taken care of by the serverless platform [78]. Though, from a functionality perspective, serverless and more traditional architectures may be used interchangeably [154]. The determination of when to use serverless will likely be influenced by other non-functional requirements such as the amount of control over operations required, cost as well as application workload characteristics [154].

2.2.2 Applications

Initially introduced for web microservices and IoT applications [76] and largely influenced by the recent shift of enterprise application architectures to containers and microservices [139], the concept has been utilized to support a wide variety of applications [154]. It is nowadays favored by many other applications such as event processing, API composition, data flow control, and many more [78, 155]. Concrete examples of applications that benefit from serverless include building a chatbot [147], programming for mobile applications [156], and securing Linux containers [157] among many others.

The stateless nature of the serverless functions puts these somewhere near to the functional reacting programming in terms of the application structure, due to the many similarities, when looking from a programming model perspective [158]. Therefore, typical use cases for serverless would include applications that exhibit event-driven and flow-like processing patterns [143]. The following lists some further interesting examples of the use of serverless among practitioners within the industry: *Coca-Cola*[11] makes use of serverless in their vending machine and loyalty program, which lets the company achieve 65% cost savings at 30 million requests per month. In fact, a serverless framework is a core component of the company's initiative to reduce IT operational costs; *Expedia*[12] is constantly moving more and more services towards the use of serverless such as integration events for their CI/CD platforms, infrastructure governance, and autoscaling. As an example, they went from 2.3 billion Lambda calls per month to 6.2 billion requests in 2017; *A Cloud Guru*[13] uses functions for performing protected actions such as payment processing and triggering group emails; *Netflix*[14] is another interesting example on the market that is making use of serverless. They use functions within an internally developed framework [159].

Serverless relies on the cloud infrastructure to automatically address the challenges of resource provisioning [160, 161, 162, 142, 163, 164, 165, 166, 167, 168]. Thus, the platform to which the code snippets (*functions*) are deployed is solely maintained by the vendor, which includes keeping the infrastructure running smoothly and scaling resources to satisfy function executions caused by changes in user demand [147]. In general, from the perspective of the cloud provider, serverless computing provides an opportunity to reduce operational costs by efficient optimization and management of cloud resources [148]. This allows developers and IT/operations teams to save time and resources on activities such as server maintenance, updates, scaling, and capacity planning and focus on writing their applications' business logic instead [153, 138]. Thus, from the perspective of the cloud consumer, serverless platforms provide developers or software architects with a simplified programming model for their applications [148]. They are only concerned with creating actions that load on-demand and are triggered to execute by system-generated events or end users [156], which means that after uploading the code for the function to the *FaaS* provider, everything in terms of provisioning resources, instantiating VMs, managing processes, etc. is in the "hands" of the provider [143].

[11] `https://www.coca-cola.com/`
[12] `https://www.expedia.com/`
[13] `https://acloudguru.com/`
[14] `https://www.netflix.com/`

This allows practitioners to not only save costs, due to the new, pay-as-you-go pricing model but also save time and human labor allowing them to concentrate on the business logic of their application.

Cloud users, however, are by far not the only stakeholders that can benefit from this novel approach. In fact, large amounts of the computing power in the data centers are allocated but unused and idle instead [169]. A study shows that around 10-30% of the data centers in the US are unused at each point in time, resulting in wasted resources. [169]. This means, that along with the cloud users, cloud providers can benefit from the automated resource management as well, allowing for an efficient distribution of the available resources.

2.2.3 Architecture and Workflow

The architecture in serverless differs from the other traditional architectures of other cloud services as the control and security is no longer managed by a central server. Instead, highly influenced by the concept of microservices, in the serverless version there is a preference for choreography over orchestration - each component plays a more architecturally aware role [143]. Such approach provides many benefits, e.g. systems built this way are often more flexible and amenable to change [170]; better division of concerns as well as significant cost-related benefits [171]. On the other hand, all these benefits come at a certain cost, related to the architectural change and imply potential downsides, e.g. customers have to re-think how to approach sessions, storage, authorization, and testing [171]. Therefore, as in many fields in software engineering, there is a design trade-off and whether the benefits of cost and flexibility are worth adding the complexity of multiple back-end components is very context-dependent [143]. Nevertheless, the latter is much beyond the scope of this research work and only aims at providing the reader with some foundations for the described setting and experiments.

The key functionality of a *FaaS* framework is that of an event processing system [158]. The set of functions defined by the user is being managed by a service. A typical, high-level workflow can be then described as follows (primarily based on [158]): first, an event is received from an event data source (a.k.a. *trigger*), for instance, an HTTP request; then the system determines which action, or multiple ones, should handle this event, creates a new container instance, sends the event to the function, and waits for a response; once the response has been successfully received, it makes it available to the users and stops the function, in case it is no longer needed.

Deployment and horizontal scaling are other two aspects that are handled differently in serverless. In the former, the code for the function is uploaded to the *FaaS* provider's platform and after that, everything else is out of the scope of the cloud consumer, being automatically handled by the corresponding provider. Horizontal scaling is also completely automated and elastic, thus, no extra configuration is required from the consumer as all the resource provisioning and allocation is handled by the vendor as well [143]. *FaaS* offerings, furthermore, do not rely on a specific framework or library, these functions are regular applications and can be implemented in any modern programming language as already introduced in Section 2.2.1.

FaaS functions have several distinguishable characteristics that need to be considered when dealing with serverless and, especially, when it comes to certain design choices in various use cases. The functions are by nature stateless, ephemeral, and even-driven. These can be further seen as limitations or restrictions, making these functions inappropriate in certain situations. The following list aims at explaining and describing these characteristics in more detail, even though some of these might have already been introduced:

- **State**: *FaaS* functions are often described as stateless as there exist some restrictions when it comes to the local state. In fact, these functions do have storage available, however, there are no guarantees that the state is persisted across multiple invocations [143]. Thus, the assumption that the state would be available to another invocation of the same function is not correct. If there is a need to persist the state of any *FaaS* function, it has to be externalized outside of the function instance so that it is available for further invocations.

- **Duration**: Code deployed as a serverless function on a *FaaS* platform is run in stateless containers that are ephemeral and thus, may only last for one invocation, resulting in a limited execution duration. This means that the *FaaS* provider creates and destroys the "compute containers" executing the functions driven by the exact runtime needed, which is with many cloud providers limited to some predefined timeout. This makes *FaaS* functions not the most appropriate choice when it comes to long-lived tasks [143].

- **Event-driven**: A key capability of a serverless platform is that of an event-driven processing system [78]. The service takes an event received from an event source, executes a function by launching one or more containers, and sends the event to the function instance [148]. Therefore, the functions in *FaaS* are usually triggered by event types, and depending on the provider, the event source might vary. Examples include HTTP requests as the most commonly used trigger among all providers, S3 (file or object) updates (in AWS), time scheduled tasks, and others.

- **Startup latency**: Another key differentiator of serverless, as already discussed above, is the ability to scale to zero and not charge customers for idle times [78], making it an attractive option in terms of costs. Nevertheless, while this is advantageous as compared to other cloud offerings in terms of pricing, it leads to the so-called problem of *cold-starts* [78]. Creating, instantiating, as well as destroying a new container for each function invocation can be expensive as it requires more time in contrast to reusing an existing function instance (*warm-start*) [158, 172]. The duration of a *cold-start* can vary significantly, and, thus, can increase the overall latency [158]. The *cold-start* phenomenon might be caused due to many different factors such as the programming language used for the function, the number of included libraries, the amount of code within the function, the function configuration, etc. [143]. And even though it is often possible for developers to control and configure many of these aspects, in order to reduce the startup latency as much as possible, there are no guarantees that the creation of the container instance would not cost an undesired amount of time.

In terms of their programming model, serverless frameworks allow for the execution of a single main function, which typically takes a dictionary as an input (such as a JSON object) and produces a dictionary as an output [158, 78]. In addition to that, a function is further able to access a context object, representing the environment in which the function is running (e.g. security context) [158]. A *Hello World!* example of a function that takes as an input a JSON object as the first parameter, and a context as a second is shown in the Listing 2.1 below:

```
import json

def handler(event, context):

    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Serverless!')
    }
```

Listing 2.1: Example of serverless function

The example above is written in Python language, however, currently, cloud provider serverless offerings support a wide variety of programming languages as already introduced. The list contains modern languages such as JavaScript, Java, Swift, C#, etc. Some of the providers go even further and support extensibility mechanisms for code written in any language as long as it is packed in a *Docker*[15] image supporting a well-defined API [158, 78].

2.2.4 Serverless Platforms

Nowadays, serverless computing capabilities are being offered by both major and prominent cloud computing providers such as *Amazon (AWS Lambda)* [173], *Microsoft (Azure Functions)* [174], *Google (Google Cloud Functions)* [175], and *IBM (IBM Functions)* [176] as well as open-source platforms such as *OpenFaaS* [177] and *Apache OpenWhisk* [178]. Furthermore, serverless has been shown to support a wide variety of applications such as linear algebra computation [179] and map/reduce-style jobs [180]. The advantages of the concept have recently motivated researchers to study and explore the role of serverless in data-intensive applications [181, 182, 183, 184].

As introduced at the beginning of this section, *Amazon's AWS Lambda* was the first serverless platform, presented at the re:Invernt "Getting Started with AWS Lambda" [135] back in 2014. They defined several key characteristics that can be used to nowadays distinguish between different platforms such as cost, programming model, deployment, resource limits, and others. AWS Lambda functions, just like the offerings of the other current providers, can be implemented in any programming language, providing to the consumer various possibilities based on the experience, preferences and needs of the corresponding individual [143].

Ever since this introduction, many of the other cloud providers, listed above, have followed the trend and introduced serverless within their stack. Even though the main principles and

[15] `https://www.docker.com/`

the structure of the functions are identical along all the existing providers, they do differ from each other in some of their characteristics [78]. There exist differences in terms of resource utilization, performance isolation efficiency, the way how users can include custom dependencies, etc. Moreover, despite the fact that consumers are not charged for unused computational resources, different *FaaS* platforms differ in their pricing models considering factors such as the actual execution time of the corresponding function, configured memory, number of function invocations as well as the size of the data that is being transferred [78]. In fact, the function's maximum memory, along with its cloud region, are the only two required configurations that the consumer needs to specify. Nevertheless, these do have an important effect on the function's performance and if misconfigured, can result in function failures such as invocation interrupts caused by an exceeded memory limit, for instance.

There exist some differences between the platforms when it comes to function compositions. Some of the providers allow higher-level mechanism for composing functions, allowing to invoke one serverless function from another and, thus, providing the possibility to construct more complex serverless applications [78]. *IBM OpenWhisk*, for example, allows chaining multiple serverless functions, in order to create composite functions [78]. Another important aspect is deployment, which is handled similarly alongside many of these providers by presenting support for both a user interface (UI) as well as a command-line interface (CLI) as options to deploy a particular function. AWS recommends the use of their open-source framework for building serverless applications, called *Serverless Application Model (SAM)*[16]. Alongside the UI, Google provides their users a similar, command-line way of deploying serverless applications with the use of the *gcloud SDK*[17], whereas Microsoft has integrated support for functions' deployment within their *Visual Studio* product.

2.3 Serverless Meets ML

The constantly growing interest in *FaaS* among both industry and academia has tempted practitioners and researchers to further investigate the benefits, when applied in data-intensive domains such as machine learning [76]. The use of serverless for model inference, for example, has been proposed by *Amazon* as an option to address scaling issues when building AI solutions [185]. In fact, *FaaS* is the natural choice for ML inference, showing good suitability to run deep learning predictions [148, 186], however, it is not yet clear whether applying serverless for ML training could add any value [76]. Another example by *Amazon* in the direction of serverless ML training shows the use of *AWS Lambda*, which aims at automating machine learning training tasks[18] and providing users a "code-free" ML experience. Therefore, given the capabilities of serverless and its use for ML inference, it is no surprise that ML training with serverless has recently attracted intensive attention from academia and is an open and active research topic nowadays, with several existing and promising research works in this direction [76]. In the following section, Section 3, we refer to some of these studies.

[16] `https://github.com/aws/serverless-application-model`
[17] `https://cloud.google.com/sdk/docs/install`
[18] `https://github.com/awslabs/autogluon`

3 Related Work

As noted previously in Section 2, federated learning is a relatively new and heavily studied area among academia in the past couple of years. Researchers are looking at major challenges involved in FL such as communication efficiency, system heterogeneity, statistical heterogeneity, and privacy among others [64]. Even though these fields enjoy quite a big popularity among both industry and academia recently, most of the studies referenced in the following date back to the last couple of years. This is, in fact, not surprising given that the concept of federated learning was first proposed in 2016 [10]. Thus, one can expect that a novel approach such as applying serverless principles in the field of federated learning might have not yet delivered a large amount of reasonable research outcomes. Nevertheless, there exist some quite interesting and promising results that worth referencing and that could turn out being the foundations of great improvements in the FL context.

In the following we present existing academical work that has been conducted in the fields of *Federated Learning* and *Serverless Computing*. In the first two sections, namely 3.1 and 3.2, work conducted in the fields considering each one on their own is presented. The last section, Section 3.3, provides an information about research works at the intersection of the two that try to benefit from serverless in the context of FL setting for training purposes.

3.1 Federated Learning

The constantly growing interest in Federated Learning over the past couple of years as depicted in Figure 2.1 in the Background section, has led to a remarkable amount of work within academia. In the most common case, the work conducted has mainly dealt with artificial neural networks [87], concerned with topics such as communication efficiency [57, 69, 47, 131], scalable system designs [121], and others [121, 187]. Nevertheless, there are works in the field of statistical learning methods as well, with Liu et al. [188] presenting a framework for a federated random forest.

There are several open-source frameworks that have been developed for the context of FL. Yang et al. [2] have thoroughly studied and listed a couple of existing FL frameworks including the following ones: *Tensorflow Federated* (TFF)[1] - a Google-developed, Tensorflow-based framework for decentralized ML [189]; *PySyft* [2] - a framework based on PyTorch that deals with privacy-preserving DL in untrusted environments [190]; *LEAF* - an open-source framework containing multiple datasets, e.g. Federated Extended MNIST (FEMNIST) and MNIST [191] that can be used as benchmarks in FL [192]; Federated AI Technology

[1] `https://github.com/tensorflow/federated`
[2] `https://github.com/OpenMined/PySyft`

Enabler (*FATE*) [3] - an open-source framework developed by WeBank [193] supporting the federated implementation of ML models. In addition to the above-stated frameworks from the referenced comprehensive survey, one can mention *FedML*[4], which is a library, aiming to facilitate the development of FL algorithms [194]; *TiFL* - a Tier-based FL system concerned with the heterogeneity of clients' performance capabilities or data quantity [19]. Due to the huge interest in the field over the last years, the list of tools developed to address open research problems and challenges in the FL context keeps growing: *PaddleFL*[5], *FLOWER*[6], *fedN*[7] [87], and many more.

3.2 Serverless in the Machine Learning Context

Serverless computing or simply Serverless is a hot topic in the software architecture world [143] on its own and as such, its application is an active area of further research and development [195]. It is still in its infancy, which makes it an attractive research topic among academia and the serverless community in recent times, not only to understand and shape it by addressing the challenges that it faces [154, 138] but also to study the benefits and success that it might bring when applied in other domains. One such example is the growing interest in the use of serverless concepts in data-intensive applications such as query processing, or machine learning. [76], allowing ML practitioners to benefit from automated resource provisioning and management [196].

There exist some research works that have studied the application of serverless in the machine learning context, aiming to address various open problems, nonetheless, the majority of these are concerned with machine learning inference. Carreira et al. [196] conduct a case and investigate the benefits of serverless in the ML context. Feng et al. [197] investigate the use of serverless runtimes in the context of large models and show that serverless can be advantageous, especially for hyper-parameter optimization of smaller deep learning models. The study by Ishakian et al. [148] investigates the latency impact of serverless for deep neural networks.

As a result of the conducted cases and research works in the area, several studies have proposed solutions to some of the open ML challenges with the help of serverless architectures. In addition to the case conducted by Carreira et al. [196], they furthermore propose a distributed ML workflow framework for serverless infrastructure, supporting various tasks in different ML workflows such as data preprocessing, model training, and hyperparameter optimization. The framework, named *Cirrus*[8], supports homogeneous cloud platforms such as AWS Lambda [161]. It automates the end-to-end management of data center resources by taking the advantage of serverless infrastructures.

[3] `https://github.com/FederatedAI/FATE`
[4] `https://fedml.ai/`
[5] `https://github.com/PaddlePaddle/PaddleFL`
[6] `https://github.com/adap/flower`
[7] `https://github.com/scaleoutsystems/fedn`
[8] `https://github.com/ucbrise/cirrus`

In order to reduce the amount of time on data serialization and, then, deserialization, *Cirrus* uses a special routine. The runtime allows a minibatch-based iterator backed by a local memory ring-buffer, enabling participating workes to access the data with low latency. Nonetheless, the lack of support for popular ML frameworks such as Tensorflow of *Cirrus'* worker run time could be seen as a disadvantage.

Siren is another asynchronous distributed ML framework that is making use of the serverless architecture. It allows the execution of stateless functions in the cloud, resulting in a higher level of parallelism and elasticity [198]. Jiang et al. [76] implemented *LambdaML*, a prototype system of FaaS-based ML training on Amazon Lambda and presented comparative results of distributed ML training between "serverfull" (IaaS) and serverless (FaaS) architectures using the implemented platform. Within their work, the authors furthermore consider several interesting aspects that relate to our work as well. They take a closer look at the effect of serialization and deserialization on the communication time. With the use of two different RPC frameworks and a Lambda function with varying memory size and, thus, CPUs, they show that increasing the CPUs can accelerate data serialization and deserialization. As a result of that, one could achieve a communication time speedup. Thus, the serialization performance and, therefore, the communication time is eventually bounded by the limited CPU resource of the corresponding function, in this case, Lambda, in accordance with the exact evaluations performed by the authors. These findings provide us with a direction to investigate, in order to improve the performance of the system. In particular, within our work, we pay special attention to the data serialization and deserialization and look for potential techniques to eliminate the need for these time consuming steps, in order accelerate the total running times of all the participating devices in an FL training.

Stratum is another a serverless platform developed for the life-cycle management of various ML-based data analytics tasks [199] that was identified throughout the research phase of this work. It allows deployment, scheduling, and dynamic management of ML-as-a-service for inference.

Serverless optimization is another topic that is being heavily studied among academia. Researches investigate the benefits of the serverless approach hired for various optimization approaches. There exist some works that have dealt with and examined the impact of serverless over diverse optimzation tasks. *OverSketched Newton*, for instance, is a randomized Hessian-based optimization algorithm that solves convex optimization problems in serverless systems [200]. Interestingly, it demonstrates a reduction of around *50%* in total running time (on AWS Lambda) as compared with the state-of-the-art distributed optimization schemes.

Even though most of the reference works and solutions have dealt with the use of serverless for ML inference, it is to be noted that training ML models with serverless is constantly gaining intensive attention from academia nowadays [76]. Given that high interest, there is a high expectation of more promising research works to focus on training ML models with the use of an underlying FaaS infrastructure in the near future [76].

3.3 Serverless Meets Federated Learning

Despite the fact that the frameworks referenced in Section 3.1 do address open issues and challenges in the context of FL, none of these are used in a serverless context. Thus, these tools do not make use of the benefits that serverless might provide, which is the key motivation behind this work.

Nevertheless, there exist some studies that have dealt with the application of serverless technologies in the FL context, in order to mitigate known problems. They have shown that making use of serverless computing can be beneficial in a distributed ML setting and that distributed ML training can be supported using serverless functions [161]. Applying serverless computing in an FL setting to address challenges such as efficient management of participating workers (*clients*), Chadha, Jindal, and Gerndt proposed a novel paradigm for running FL on a *Function-as-a-Service* (FaaS) infrastructure [75]. In their work, they introduce a tool for efficiently managing FL over a combination of connected *Function-as-a-Service* (FaaS) platforms, i.e., FaaS fabric, named *FedKeeper*[9], and demonstrate its functionality through an image classification task with a varying number of clients. Their main contributions are in manageability, simplicity, and scalability.

There exist other research works that propose high-level frameworks for orchestrating distributed ML training jobs running on serverless functions. Jonas et al. [180] make use of serverless computing and suggest that stateless functions are a natural fit for data processing [180]. The prototype system *PyWren*[10], proposed in their work, runs on AWS Lambda and investigates the trade-offs of using serverless functions for large-scale data analytics. *ExCamera* is a system that runs general-purpose parallel computations on a cloud function service and allows editing, transforming, and encoding videos with low latency [201]. *gg* is a framework that uses thousands of parallel threads on cloud-functions service that allows users to execute everyday applications - e.g., software compilation, unit tests, video encoding, or object recognition [202]. Chard et al. [203] propose a distributed function execution platform designed for scientific computing - *funcX*. Unlike the other tools, *funcX* supports various cloud platforms with underlying heterogeneous compute notes, however, lacks support for synchronous training of ML models. The already introduced *LambdaML* [76], is another existing framework that examines the benefits of serverless in distributed ML training and places it in a direct comparison with pure *IaaS* as well as hybrid approaches.

3.4 FedKeeper and FedLess

FedKeeper, a tool introduced by Chadha, Jindal, and Gerndt [75], is a novel solution that addresses common issues in the FL setting such as efficient FL-clients' management by making use of a combination of multiple heterogeneous *Function-as-a-Service* (FaaS) platforms, referred to as FaaS fabric by the authors. Throughout their work, they evaluate the functionality and performance of the framework through an image classification task with

[9] `https://github.com/ansjin/fl_faas_fabric`
[10] `https://pywren.io/`

varying configurations - multiple numbers of devices involved, different stochastic optimizers as well as local computations. In contrast to other existing frameworks that evaluate the performance implications of ML training with the use of serverless architectures, the authors are, to our best knowledge, the first to introduce support for various commercial cloud platforms including the most prominent ones - *AWS Lambda* by Amazon, *Azure Functions* by Microsoft, and *GCloud functions* by Google.

Developed in Python, *FedKeeper* propagates FL-client functions over a FaaS fabric, acting as a keeper or manager that is able to create, delete, and invoke numerous FL functions for each of the supported FaaS platforms. Unlike other existing tools that are concerned with the use of serverless architectures to support various tasks in the ML domain, for instance, *Cirrus* [161], *FedKeeper*'s worker time does support popular ML frameworks such as Tensorflow and, thus, simplifies the implementation, integration, and training of different models.

Figure 3.1 from the original paper by Chadha, Jindal, and Gerndt [75] provides an overview of the system architecture in *FedKeeper* and the involved interactions between the building components.

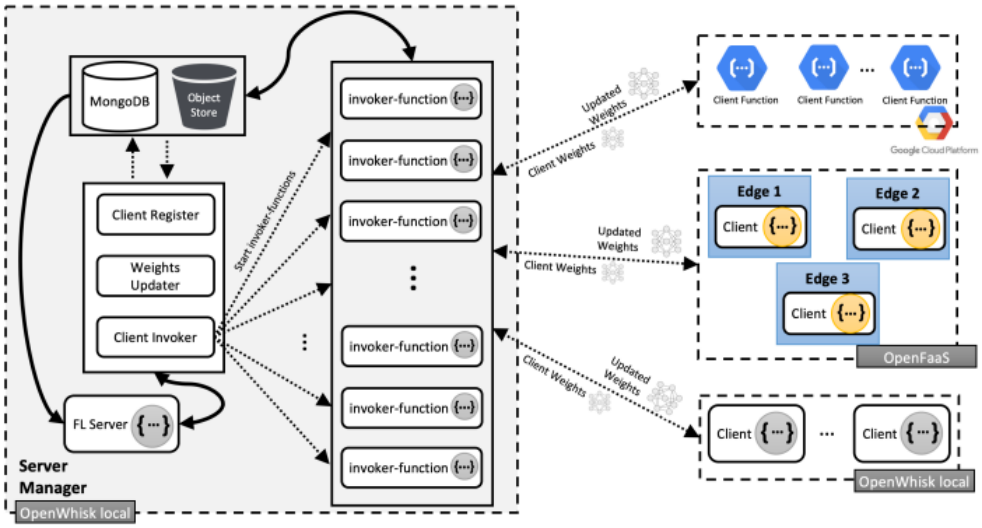


Figure 3.1: System architecture overview of *FedKeeper*

The system comprises several sub-components that interact with each other in order to execute FL training, namely - *FL-Server*, *Client Invoker*, *Weights Updater*, and *Client Register*, each being a function. The *Client Register* sub-component creates and stores all the information about the clients that are going to participate in the following training in a *MongoDB* database. The information includes the client's ID, the URL through which the corresponding client could be triggered within the training process, the type of the FaaS platform on which the client was created (e.g. gcloud) as well as additional authentication information required. The *FL-Server* sub-component is then responsible for the model and hyperparameter selection. It

decides on the model that is to be trained by specifying various parameters such as input and output size, number of layers, etc. as well as the hyperparameters such as the optimizer and the learning rate. These configurations are then used by the *Client-Invoker* for the invocation of the different clients that are going to participate in the upcoming training round. The *Client-Invoker* then creates n invoker-functions, each one responsible for the trigger of the corresponding FL-client, where n is the number of clients required in the FL training round, visualized on the right-hand side within the OpenWhisk local component in Figure 3.1. Each invoker function fetches all the necessary information about its corresponding client function as well as the current model weights and triggers the client function via an HTTP request, forwarding the data within the request body. The client then compiles the model, sets the model weights, updates them in accordance to the predefined hyperparameters, and sends the updated weights as an HTTP response to the corresponding invoker function, which then persists the client results into the parameter server. Finally, after the successful execution of all the specified clients, the *Weights-Updater* aggregates the updated clients' weights, persists them so that they are used and passed to the participating clients in the next training round, and notifies the *FL-Server* for the completion of the current training round. This process is then repeated until a certain threshold of a predefined metric (e.g. accuracy) is achieved.

With their work, the authors provide a novel approach that addresses problems in FL training by intersecting two emerging technological trends. This sets up the foundations of an interesting research area and strengthens expectations from [76], which could increase the number of applications and researchers in both industry and academia that focus on training ML models using an underlying FaaS infrastructure in near future.

FedLess [204] can be seen as the successor of *FedKeeper* that addresses open issues with the initially introduced framework. *FedLess* extends *FedKeeper* by providing additional vital features like authentication, authorization, and differential privacy. The first two come as a result of two requirements. On the one hand, only the FL server that the involved data holders interact with can call their client functions and, on the other hand, the client functions have to be approved by the FL server, in order to be included as valid ones. To achieve this, *FedLess* extends *FedKeeper* by integrating an external identity provider with the use of *AWS Cognito*. The authentication and authorization are achieved through the use of *JSON Web Tokens (JWTs)* [205] that are attached to the HTTP requests between the communicating clients and the FL-server.

In addition to that, *FedLess* focuses on strengthening and protecting the privacy of the training data, introducing *(Local) Differential Privacy* - an approach in which all the client functions distort updated parameters before communicating them to the parameter server. Moreover, looking at the performance of the system, *FedLess* is faster than its predecessor and is able to easily train more complex models across up to 200 clients and beyond. Along with these changes, *FedLess* extends the list of functions in the *FaaS* fabric and introduces support for *AWS Lambda*, *Azure Functions*, and *IBM Cloud Functions*.

Figure 3.2 provides an overview of the improved system and its most important components. All the components within the *Control Pane*, running in a Kubernetes cluster, belong to and are managed by *FedLess*. These include *Parameter Server*, *Client Database*, an OpenWhisk cluster

containing the *Aggregator Function*, responsible for the aggregation of the model updates provided by all the involved clients, and the *FedLess Controller*. In contrast to *FedKeeper*, *FedLess* lacks separation of *Invoker Functions* to call clients and introduces a new strategy for clients' calls by providing them access to the *Parameter Server*. This is as result of the large size of the models, making these inappropriate to be included in the HTTP body due to the existing payload limits. The HTTP-callable external clients, including the newly introduced providers, can be seen on the right-hand side. The external authorization and authentication provider *Auth Server* can be depicted above the *Control Pane* in the diagram.

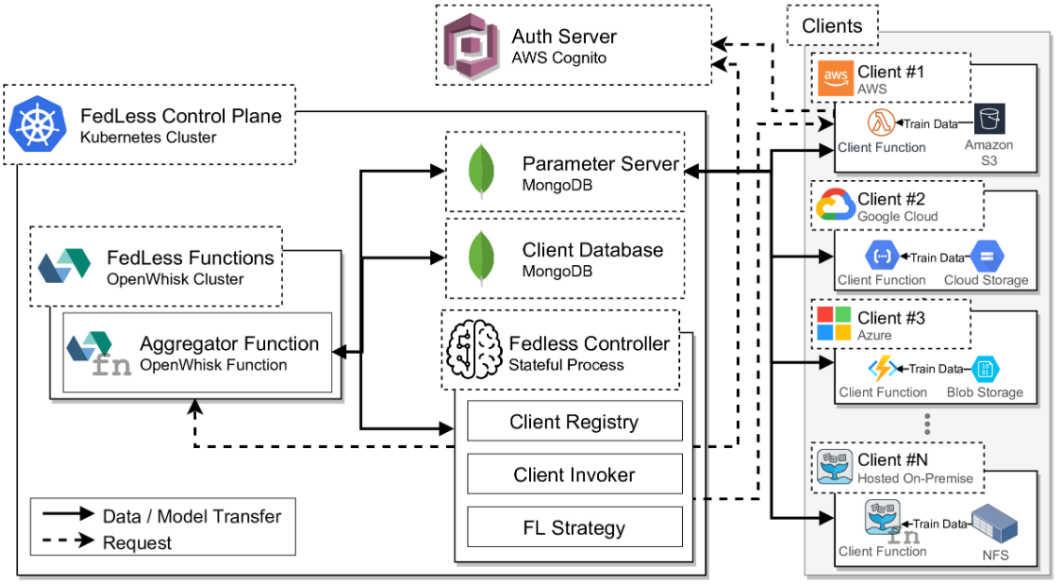


Figure 3.2: System architecture overview of *FedLess*

The evaluation results from the two frameworks motivate us to further investigate and improve the system by extending it with new features as well as improving already existing ones. With this work, we aim at addressing performance-related issues, identified as the main bottleneck of the system. These, as well as the exact evaluations as a result of the introduced improvements, are presented in the following sections.

4 Towards High Performance Federated Learning Using Serverless Computing

This work aims to address some of the major challenges involved in FL such as communication efficiency [64] and achieve a better performance of the system under development. Building on top of previous works in the field of Serverless Federated Learning [75], it consists of evaluating and improving an existing solution, in order to address some of the open challenges in the context of FL. It builds on the foundations set in the development of two consecutive frameworks for serverless FL - *FedKeeper* [75], and its successor - *FedLess*, both introduced in Section 3.4. Thus, the main objective of this study is to evaluate and improve the functionality of *FedLess* by addressing performance and scalability issues that have been identified throughout previous experiments. Based on the identified problems of the system, we define and summarize this work with two research questions (RQs). We aim at answering these research questions by further improving the system and evaluating its performance. Providing answers to the questions is considered a desired outcome of this research work.

In the following, Section 4.1, the exact definition of the problems, summarized as research questions (RQs), of this study are described and presented in detail. This is followed by Section 4.2, providing information about the methodology followed throughout this study, in order to address the problems. Section 4.3 describes the changes of the *FedLess* system made in the scope of this work in terms of both system architecture as well as the training workflow. We conclude with Section 4.4, providing information about a technology used throughout the implementation phase followed by implementation details in Section 4.5.

4.1 Problem Definition

Based on the latest examinations of the framework's performance, the first and foremost concern is addressing the parameter server as it is considered the main bottleneck of the system in terms of performance and scalability. The current technology used for the parameter server is the *NoSQL*, document-oriented database *MongoDB*[1]. Conducting various experiments of the system, it was discovered that in a large-scale scenario with multiple clients[2], the clients' networking capabilities are not the root cause of the problem that leads to degradation in their run times, but rather the parameter server's performance. It was further discovered that the parameter server is unable to handle many simultaneous uploads of large models resulting in

[1] `https://www.mongodb.com/`

[2] Throughout this study we refer to as *large-scale scenario* once we have more than 100 clients involved within an FL training round

bad scalability of the system. Thus, switching to another technology that can easily support many simultaneous connections and uploads of large models is desired outcome within the scope of this work.

The first, initial idea is to replace the *MongoDB* parameter server with an *In-memory database* (IMDB) and evaluate the system's performance in comparison with the old implementation. Indeed, we end up integrating such technology after supporting the initial beliefs with detailed literature as well as industry reviews. With this analysis we aim at gaining insights into what is being used within similar production systems in both academia and industry, and, thus, accept or reject our initial hypothesis of replacing the parameter server with an in-memory database. The exact findings gathered throughout that research and the following decisions about the technology to replace the parameter server can be found in the following.

In addition to that, aiming at increasing efficiency (e.g. in terms of network latencies), a performance evaluation under the consideration of multiple servers comprises the second desired outcome of this research work. The initial intuition that we want to investigate and approach with this research question relates to the use of multiple servers with replicated parameter servers that could reduce the costly interaction with the central server. As a result of that, the overall training time is expected to decrease, which leads to smaller execution costs as compared with the current *FedLess* solution. Given the fact that *FedLess* operates over a combination of connected *FaaS* clients, referred to as *FaaS fabric*, and, thus, relies on various cloud providers, the latter is of great interest within this research work. Reducing the training costs would result in cheaper usage of the framework. Moreover, with the consideration of multiple replicated parameter servers, one could also reduce the privacy concerns for geographically distributed clients spread over multiple different regions, which might be of great interest for the *Cross-silo* FL setting.

Based on the above-stated objectives, and as already noted in the introductory Section 1, this study aims at answering the following two research questions (RQs) by extending and executing evaluations on *FedLess*' performance:

- **Research Question 1**: *Which technology should the current parameter server be replaced with, in order to increase the system's performance and scalability?*
- **Research Question 2**: *How would the system perform under the consideration of multiple FL servers with replicated parameter servers?*

4.2 Approach

The following sections describe the approaches followed throughout this study in order to address both research questions and, therefore, provide the desired comparative outcomes related to the performance of the system. Section 4.2.1 gives information about what has been discovered while approaching the first research question including the outcomes of the literature review. Section 4.2.2, analogously, provides information about the findings while considering the second research question.

4.2.1 RQ 1: Parameter Server

In order to successfully address the first research question of this study, at first, a literature review [206] was conducted. The desired outcome of the review is to provide insights regarding the technologies used for the parameter server in other production frameworks with similar purposes within the community and take the acquired knowledge into consideration when improving *FedLess*. Moreover, it aims at strengthening the initial considerations and beliefs of replacing the current solution for the parameter server, namely *MongoDB*, with an in-memory database solution in the seek of improved communication performance. For this, we primarily focus on conducting a literature review to gain knowledge from researchers and their works in the field. Nonetheless, we further perform Internet research on our own to support our decision by looking at systems developed by practitioners.

Throughout the review, we identified various options to choose from when it comes to technologies for the implementation of a storage component within a *FaaS*-based FL system. The storage component allows stateless functions to read/write intermediate information that is generated throughout the ML training process. Each of the discovered options comes with certain trade-offs in terms of cost as well as performance. Prominent examples include *AWS S3*[3], *ElastiCache* for *Redis* and *Memcached*, *DynamoDB*, and *MongoDB*. Unlike *MongoDB* and *S3*, which are disk-based object storage services, *Redis* and *Memcached*, in contrast, are both in-memory key-value data stores.

Looking at existing FL frameworks for ML training, we found that a lot of systems with similar contexts use the *NoSQL*, document-based *MongoDB* database. Among these, there were a few examples of the use of in-memory storage such as *Redis* like *PyWren* [180], for example. Moreover, we identified other approaches such as building a customized storage layer as an alternative to using external storage services. An example of this is the *Cirrus* [161] system that has the parameter on top of a virtual machine (VM). Similar to the other disk-based storage options that were available, we did not consider this approach as suitable for our needs due to two different, but in some sense related reasons: firstly, building the storage layer on our own would introduce additional overhead in terms of parameter server maintenance, and as a result of that, this would not be a pure *FaaS* architecture, thus, conflicting with the key, distinctive property and design of our system, which is a pure *FaaS* solution, that abstracts away server management. In fact, the design approach taken by *Cirrus* is referred to as a *hybrid* by [76]. *DynamoDB* is another technology that was discovered throughout the review. However, as it is hosted by Amazon AWS, we did not consider it for further investigation as we aim at keeping our parameter server hosted by our own in the Kubernetes cluster.

We identified an existing research work that examines and compares the introduced latency between these different options for the implementation of the storage component within an FL system [76]. The authors put a disk-based object storage service, *AWS S3*, in a direct comparison with an in-memory key-value data store - *Memcached* and evaluate their performances. According to the results of the study, *Memcached* introduces lower latency

[3] `https://aws.amazon.com/s3/`

than *S3* resulting in a faster round of communication [76], which is in line with our initial considerations and strengthens the intention of replacing the current parameter server's solution with an in-memory data store technology, in order to decrease the latency and achieve a faster communication rounds.

As *AWS S3* is a disk-based object storage service just like *MongoDB*, the technology that *FedLess* is using, replacing the current storage component with *AWS S3* was not taken into further consideration as we aimed at a shift towards a technology that introduces lower latency and, thus, with faster access timings. As already argued above referring to existing research work, *Memcached* introduces lower latency than *S3* resulting in a faster FL training round [76]. Thus, being able to support our initial beliefs with concrete scientific results from existing performance examinations from academia, the decision for the parameter server replacement fell on using an in-memory data store solution in *FedLess*.

Based on the literature findings, we reduced the list of potential candidates for the parameter server replacement to two main options, namely, *Redis* and *Memcached*, which according to the referenced research work and the benchmark introduced by the authors show similar performances [76]. In fact, the authors give a slight advantage to *Memcached*, due to its underlying multi-threading architecture that is not present in *Redis*, which results in better performance and, therefore, lower communication latency throughout training when considering a larger ML model [76]. Nonetheless, we discovered several other promising advantages throughout the research for the use of *Redis* in our context. These are listed in the following.

A promising finding that we discovered through our own Internet research is the *RedisAI*[4] module by *Redis*. However, we did not find the use of *RedisAI* in any of the reviewed research works and the open-source frameworks, which might not be surprising given the fact that it is a relatively new module by *Redis*, officially introduced on May 12, 2020, according to the official repository release notes[5]. Nevertheless, given the fact that its main purpose is to serve a variety of ML tasks, we decided to take it into account within the considerations of potential parameter server replacements along with the *Memcached* option. According to the official documentation, *RedisAI* allows practitioners to *"to run inference engine where the data lives,* ***decreasing latency*** *and increasing simplicity — all coupled with the core Redis Enterprise features"*. Being referred to as one of the benefits that come with the module, reduced latency matches our needs within the scope of the first research question. Designed and built exclusively to support ML tasks such as deployment and management of ML models, *RedisAI* tends to be a good fit and a potential solution to the existing problem of high communication costs in *FedLess*. Especially interesting is the fact that *RedisAI* allows users to directly store and respectively, retrieve, model parameters as tensors, removing the need to perform any preparation steps such as serialization and deserialization of the model weights, which is the case with the current *MongoDB* solution. Removing the serialization overhead could result in a significantly improved latency and, therefore, reduced training rounds, allowing us not only to benefit from faster, in-memory accesses to the stored objects, but also to further reduce the cost by completely removing the need for any intermediate steps.

[4] `https://redis.com/modules/redis-ai/`

[5] `https://github.com/RedisAI/RedisAI/releases?page=1`

Based on these findings, the decision for the technology to replace the current parameter server solution fell on *Redis* with the use of the *RedisAI* module. Even though, as noted above, *Memcached* is considered another great in-memory data store option, which furthermore underlies a multi-threaded architecture, the benefits of the *RedisAI* seem promising. Being able to completely remove the weights serialization and deserialization steps would not only have a positive impact on the actual model training that occurs on the client-side in terms of communication costs but also significantly improve the aggregation step that occurs on the FL server side.

Thus, we decided to examine the system's performance under the consideration of two different approaches in order to address the open problem of communication cost in *FedLess*. We performed multiple experiments by replacing our storage component with *Redis* and examining two different implementations. At first, we evaluated the framework's performance with *Redis* without the use of any additional modules such as *RedisAI*. We kept the weights serialization solution as is and only replaced the underlying parameter server with *Redis*. This, however, did not provide the desired improvements. Afterward, we shifted towards the use of the *RedisAI* module and completely removed any weights serialization involved. With this change, we could not only reduce the costs of a training round for a small portion of clients involved but also managed to outperform the *MongoDB* solution even in the large-scale scenario with up to 300 clients involved. This, moreover, resulted in a more stable solution when it comes to the use of larger model sizes.

The resulting design and workflow changes can be found in Section 4.3. The implementation details as a result of the changes, in Section 4.5, and the outcomes and the corresponding exact performance results, in the following Section 5.

4.2.2 RQ 2: Replicated Parameter Servers

Aiming at increasing efficiency (e.g. in terms of network latencies), a performance evaluation under the consideration of multiple servers is to be additionally executed throughout this work. The FL server in *FedLess* comprises multiple components running in a Kubernetes cluster. Considering multiple servers might reduce costly communications with the central server and, thus, lower the overall training time compared to the current *FedLess* solution. Moreover, this could also reduce privacy concerns for geographically distributed clients spread over multiple regions.

Similar to the approach taken when working on the first research question (RQ 1), at first, we conducted a literature review prior to the actual work with the goal of gaining insights about potential ways of implementing replication within the system. As we implemented the in-memory key-value store *Redis* while working on RQ 1, the review aimed at identifying different ways of handling replications in *Redis* within literature. Nonetheless, we discovered significantly smaller amount of scientific work in the area as compared with the research executed for RQ 1. Moreover, most of the research works discovered deal with data loss problems, due to node failures in the replication context and not concretely with improving read/write throughput. However, we found several appropriate sources of information available in the official *Redis* and *AWS* documentation and refer to these in the following.

The machines providing data storage services in the *Redis* context are called "nodes" [207]. Each of the nodes takes a certain amount of memory space on each machine, configurable by the practitioner in accordance with the corresponding needs. Replication is the process that allows specific nodes to be exact copies of others, referred to as master nodes. *Redis* provides different ways of achieving replication, which mainly depends on the context in which replication is applied.

We go through the differences between the different replication techniques in *Redis* with the help of the comparison diagram in Figure 4.1, taken from the official *AWS* documentation. As noted, there exist two distinct implementations of replication in *Redis*, namely, the so-called replication with *enabled* cluster mode, which can be seen on the left-hand side of the diagram, as well as replication with *disabled* cluster mode, on the right-hand side of the diagram, respectively. The former contains a single shard including all of the cluster's data in each node, whereas the latter has its data partitioned across up to 500 shards, each having a single read/write primary node and up to 5 read-only replica nodes. The number of shards, however, is configurable as *Redis* allows for a varying shard-replica ratio - one is allowed to create a cluster with a higher number of shards and a lower number of replicas involved and vice-versa.

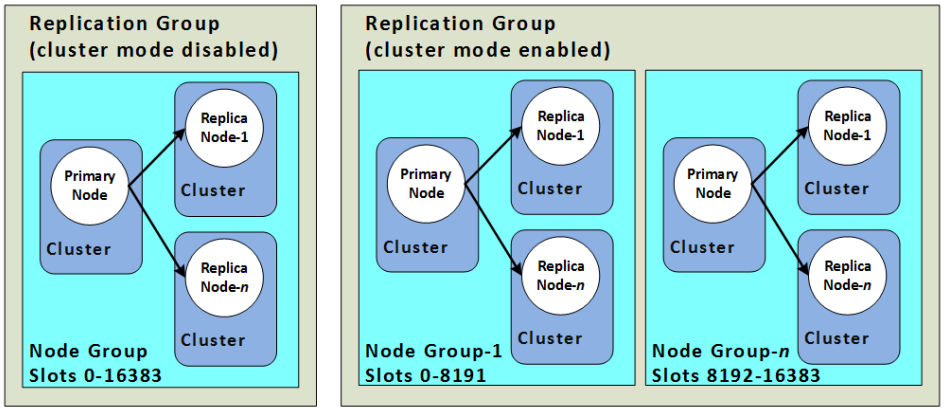


Figure 4.1: Cluster mode *disabled* (left) vs cluster mode *enabled* (right) replication in *Redis* [6]

The cluster mode *disabled* replication mode supports read-write isolation through assigning *write* operations to a single master (primary) node and designates *read* operations onto all replica nodes [208], whose number might vary between 0 and 5. It furthermore supports scaling and is considered to improve the read throughput, making it a great choice for read-intensive applications. Based on the experience made throughout the experiments concerned with RQ 1, we considered the latter as a strong argument for using the cluster mode *disabled* approach for our experiments as our system showed a good *write* performance, however, comparably worse *read* performance with *Redis* as an underlying parameter server technology.

[6]The *Redis* replication comparison diagram is taken from the AWS documentation available at: `https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/Replication.Redis-RedisCluster.html`

4.3 FedLess System Changes

This section describes the changes made to the current system solution in terms of architecture as well as workflow in detail. Section 4.3.1 provides a top-level architecture of the new system and compares it with the existing *FedLess* architecture presented in Figure 3.2. It describes the changes as well as the newly introduced components on a higher level of the system. Section 4.3.2 presents the differences in terms of the workflow caused by the new parameter server technology in a comparative manner as a result of the work related to the first research question (RQ 1). At first, we look at the workflow on the client-side of the system, followed by exact workflow description of the aggregation. The section concludes with detailed information about the replication approach that was considered and evaluated throughout this study as part of the second research question (RQ 2).

4.3.1 System Design

Figure 4.2 shows the changes in the top-level design of the system that were introduced throughout this work. As compared with Figure 3.2, displaying the existing architecture of *FedLess*, none of the components is removed, however, some are changed. In addition to that, we introduce some new components, highlighted in green. In the following, the changes, as well as the extensions, are described in more detail.

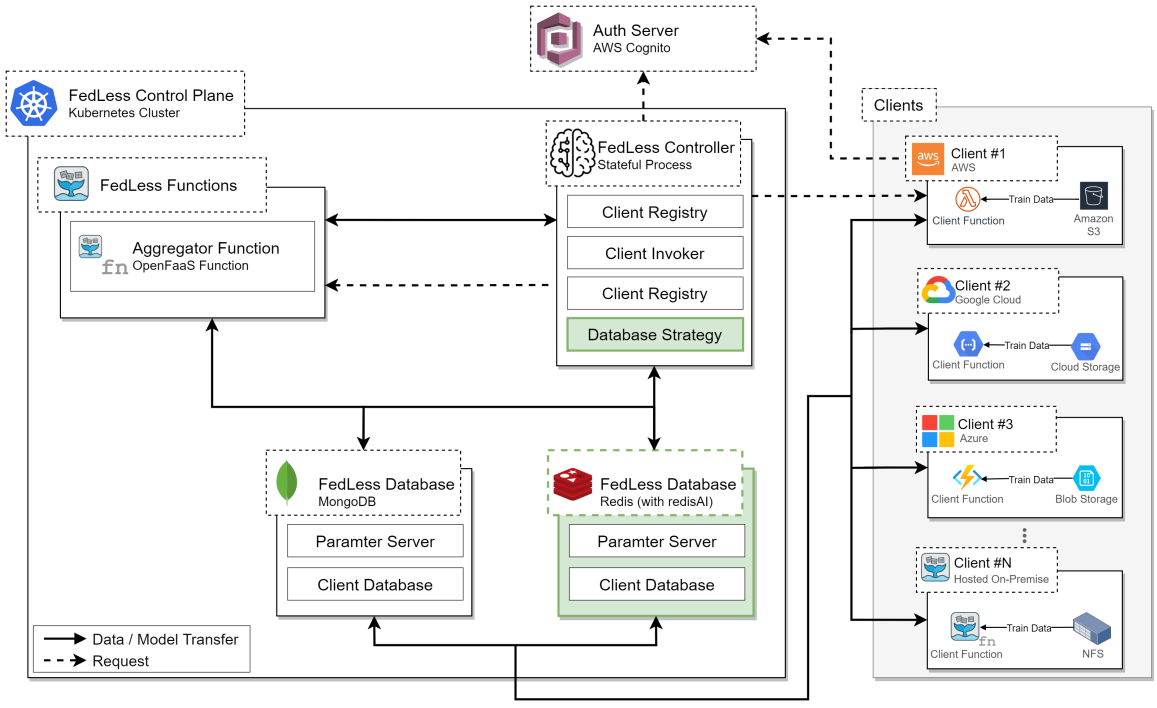


Figure 4.2: *FedLess* system architecture changes

Starting with the changes, we replace the underlying technology for the *Aggregator Function*, switching from *OpenWhisk* to *OpenFaaS*. The main reason for this is the unstable *OpenWhisk* solution. Throughout the experiments, we experienced issues with *OpenWhisk* as the aggregator function was not able to handle the aggregation of multiple clients' results, especially in the large-scale scenario that involves 100 and more clients, and results in a failing function error state. During the experiments, we could not achieve stability for the aggregator function, despite trying many different configuration parameters. Thus, we switch to an *OpenFaaS Aggregator Function*, which introduces better performance and is able to handle the aggregation of up to 300 clients even with increased ML model size. For the sake of completeness, it is to be mentioned that in contrast to *OpenFaaS*, *OpenWhisk* seems poorly maintained. More concretely, the latest *OpenWhisk* release[7] dates back to Nov 10, 2020, making it almost two years old, whereas the latest stable release of *OpenFaaS*[8] is just a couple of months old. Moreover, the community behind the *OpenFaaS* solution seems to be much more active and involved, which results in ten times fewer open issues as compared with *OpenWhisk*, giving a feeling of lacking support.

In addition to the existing three components in the *FedLess Controller*, we introduce a new one named *Database Strategy*. Just like the *FL Strategy*, the *Database Strategy* is to be selected at the beginning of each training session by the administrator, who is also responsible for the hyper-parameter configurations. The administrator is able to choose between two different options to store all the necessary information throughout an FL training session such as client and model information, client results, etc., namely *MongoDB* and *Redis*. Based on this choice at the beginning of the session, the system uses the corresponding underlying technology for all the FL training rounds throughout the whole training session.

Another noticeable difference with the *FedLess* architecture from Figure 3.2 is the newly introduced *Redis* database that comprises the *Parameter Server* as well as the *Client Database* components. These have the exact same purpose and role as the ones with *MongoDB*. However, due to the *RedisAI* module, the clients' results as well as the latest model parameters are stored directly as tensors. Detailed information on how this is achieved is provided in Section 4.5.

4.3.2 Training Workflow

In the following, a more fine-grained view of the newly introduced changes in the system are presented. Firstly, we present all the steps that occur once a client has been invoked to participate in the training of FL training round in a comparative manner. This contrasts the workflows when considering two different technologies used for the parameter server - namely, the previous, *FedLess* implementation using *MongoDB* against the newly introduced *Redis*[9] alternative. Then, we show the differences between the two parameter server technologies on the FL server-side, comparing the exact steps performed once the aggregator function has been invoked to combine the latest clients' model updates.

[7] `https://github.com/apache/openwhisk/releases`
[8] `https://github.com/openfaas/faas/releases`
[9] `https://redis.io/`

Client Side

Figure 4.3 shows the differences between the two parameter server approaches in a direct comparison. Both diagrams contain events representing concrete steps that occur throughout an FL training round of the system once a client has been called to participate. These distinct events can be observed in a top to bottom manner starting with downloading the model and the latest parameters (*Model Download*) and ending with the persistence of the calculated client results (*Model Upload*). The execution duration of these events contributes to the total duration of an FL training round and are measured and thoroughly evaluated as presented in the evaluation Section 5.

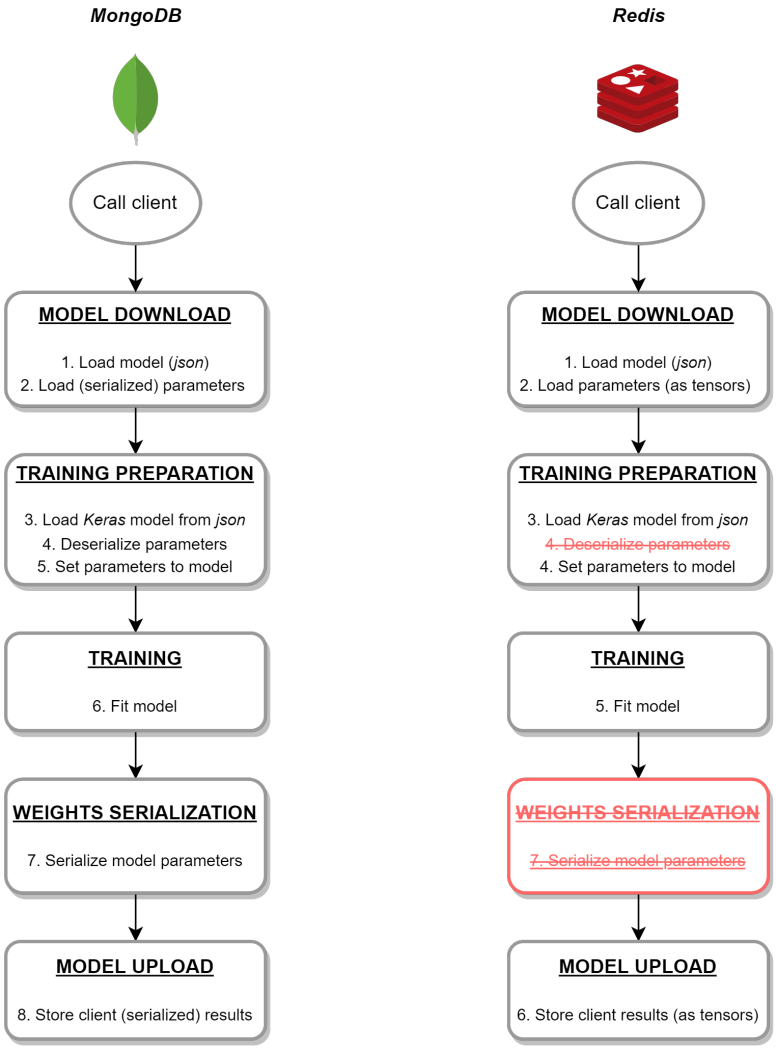


Figure 4.3: *MongoDB* vs *Redis* client-side workflow comparison for a complete FL training round

In the following, we introduce the context and provide a detailed explanation behind each of these five events from the figure above:

- *Model download* - consisting of two steps, namely, loading the *JSON*-serialized model from the database without the model weights, followed by loading the actual latest parameters (weights) from the parameter server. Depending on the technology, the weights are in a different form as described below;
- *Training preparation* - consisting of three consecutive steps depending on the underlying parameter server technology: 1) loading the *Keras* model from *JSON* using `tf.keras.models.model_from_json`; 2) deserializing (applicable for *MongoDB*) and 3) setting the weights to the *Keras* model object;
- *Training* - representing the actual `model.fit()`
- *Weights serialization* - preparation step representing the weights serialization in an appropriate format so that these can be stored in the parameter server (applicable for *MongoDB*)
- *Model upload* - persisting the latest clients' model updates (weights)

Differences: On the left-hand side of the figure, one can find the steps that are executed once a client has been invoked for the participation in an FL training round with the use of *MongoDB* and on the opposite, the ones needed with *Redis* as parameter server. The first noticeable difference can be found within the first event - *Model download*. With the *MongoDB* approach, one loads the latest parameters in a serialized form according to a predefined serialization technique, whereas the *Redis* solution, due to the *redisAI* module for ML contexts, allows for a direct persistence of the weights in the form of tensors. This allows for the elimination of some of the steps in the following events throughout the workflow.

Moving to the *Training preparation* event, one can note that the fourth step of the workflow (crossed out and highlighted in red color), namely *"Deserialize the parameters"* is no longer necessary as *redisAI* allows us to directly store the weights as tensors without the need for a prior serialization. The same holds true for the whole *Weights serialization* event, which can be completely eliminated with the use of *Redis* and *redisAI* as the clients' weights can be directly stored after the training process has been successfully executed, without the need for further modification.

With these simplifications and the elimination of the steps described above, we could achieve a total performance speedup during an FL training round on the client-side of the system. *Redis* and *redisAI* allow us to completely ignore weights serialization and deserialization processes, which contributes to the desired performance increase. Moreover, the upload times with the *Redis* solution perform better than the upload times of the current, *MongoDB* solution, allowing us to address the main bottleneck of the current system. The exact comparisons between the performance of the system with both parameter server technologies can be found in the evaluation Section 5.

FL-server side

Figure 4.4 shows the differences between the two parameter server approaches on the FL-server side. Similar to the approach in Figure 4.3, the two diagrams contain events that occur within an FL training round. However, unlike the example from above, here we show the effect that the new *Redis* parameter server has on the aggregator process in terms of the workflow. The figure below shows the subsequent events once the aggregator function has been invoked to combine the latest clients' model updates. The events are to be read top to bottom starting with the *Model Download* and ending with persisting the aggregated latest model parameters, namely, the *Model Upload* event.

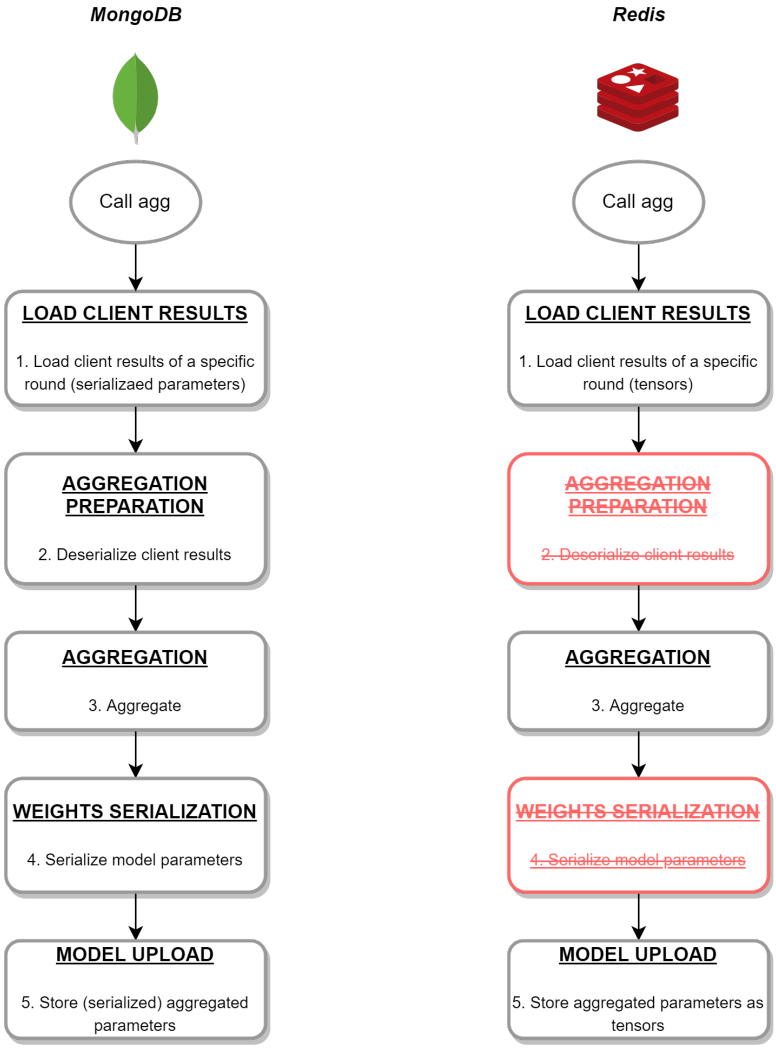


Figure 4.4: *MongoDB* vs *Redis* aggregator-side workflow comparison for a complete FL training round

In total, there are in total five events that can be observed on the aggregator side:

- *Load client results* - comprising a single step for fetching the latest clients' results (model updates);
- *Aggregation preparation* - deserializing the weights so that they can be used in the subsequent aggregation event (applicable to *MongoDB*);
- *Aggregation* - representing the actual aggregation of the clients' model updates according to a predefined aggregation scheme;
- *Weights serialization* - serializing the weights in a predefined format, in order to be stored in the parameter server as the latest actual parameters (applicable for *MongoDB*);
- *Model upload* - storing the aggregated clients' results (weights) that are to be used as latest parameters in the next FL training round

Differences: On the left-hand side of the figure, all the necessary steps that are executed on the aggregator side with *MongoDB* can be seen. These constitute the workflow, once the aggregator has been invoked to combine the latest clients' updates within an FL training round. Alternatively, the right-hand side shows the same workflow procedure with the use of *Redis* and the *redisAI* module. All differences between these two approaches are marked in red and, additionally, all the steps that could be eliminated with the use of *Redis* are crossed out.

The first noticeable difference can be seen within the first event - *Load client results*. Like in the *Model download* event on the client-side workflow, *Redis* allows us to directly fetch the clients' results as tensors, eliminating the deserialization need for the weights in any of the subsequent steps with the help of the *redisAI* module. On the contrary, *MongoDB* does not support storing the weights directly as tensors and, thus, requires some pre-processing steps such as weights serialization. As a result of that, the *Aggregation preparation* event can be completely ignored for *Redis* as we can directly proceed to the *Aggregation*, having already the weights in the desired tensor form at hand. Likewise, the event *Weights serialization* that occurs right after the actual aggregation process has been successfully executed can be eliminated.

Eliminating all the steps as described above on both client as well as FL server sides, we were able to achieve a total performance increase. As one key contribution to this improvement, one could consider the removed, time-consuming training and upload preparation events such as the weights serialization and deserialization, allowing us to reduce the total steps from 5, with the use of *MongoDB* to just 3, when using *Redis* on the FL-server side. Additionally, being an in-memory data store, *Redis* showed a better performance within the *Model upload* step on both the client as well as the FL server-side.

4.3.3 Replication

Motivation and Design

This section provides an architectural overview of the changes caused by the introduction of the parameter server replication and, thus, relates to the second research question (RQ 2) of this study. It furthermore discusses some of the design choices made throughout the implementation phase.

As already introduced in Section 4.1, *Redis* replication allows *Redis* nodes to be exact copies of specific instances called primary or also master nodes. In the *Redis* context one can observe two different replication approaches - with *enabled* as well as with *disabled* cluster mode. As already discussed in previous sections, the decision for the replication approach fell on the so-called cluster mode *disabled* way of handling replications in *Redis*. This was influenced by the findings from the evaluation of the system's performance within the scope of the first research question of this work. Throughout the conducted experiments we discovered that even though the system clearly outperforms the old *FedLess* solution using *MongoDB* as an underlying parameter server solution, there still exists room for improvement, especially considering the read throughput. In fact, the read performance of *Redis* performed even worse than *MongoDB* once an increased amount of clients was used within an FL training round.

The latter seems to be a direct result of the lacking multi-threading infrastructure in *Redis*. Interestingly, even though *Redis* is mostly single-threaded, it does provide an option to enable multiple threads for certain operations such as UNLINK and slow I/O accesses as of its 6.0 release. This can be achieved by extending the configurations and by setting the desired amount of I/O threads as follows: `io-threads 4`. According to the official documentation, it results in an improved performance per core as it speeds up the accesses, thus, removing the need for pipelining, for instance. In this example, we set 4 threads and, therefore, enable multi-threading for writes, that is, we thread the write (2) syscall and transfer the client buffers to the socket. Throughout our evaluations, we experimented with different configurations for the I/O threads starting with 4 and scaling up to 64 threads enabled.

In fact, we were able to achieve a good write times and, thus, outperform the system's performance with *MongoDB* by orders of magnitude, tackling the main bottleneck of the system identified in the preceding *FedLess* evaluations. As already noted, however, the read times that we could achieve were as bad as or even worse than the ones identified with the *MongoDB* set-up, even after following the exact configuration instructions on enabling multiple threads for read operations. The latter is achieved by setting corresponding flag in the configuration file as follows: `io-threads-do-reads yes`. Nonetheless, we regrettably discovered that this setting *usually* does not result in a performance increase as reported by the *Redis* official documentation: *"usually threading reads doesn't help much"*.

Therefore, based on these findings and the corresponding evaluations performed as a result of approaching the first research question, we aimed at a replication approach that would allow us to improve the read performance of our system and, thus, the replication decision fell on an integration of the cluster mode *disabled* way of applying replication in *Redis*.

Figure 4.5 shows what the *Redis* replication looks like in the context of *FedLess*. It consists of 6 nodes in total, which differ in the operations that these nodes could perform. It furthermore represents the clients involved and their interaction with the corresponding nodes throughout an FL training round.

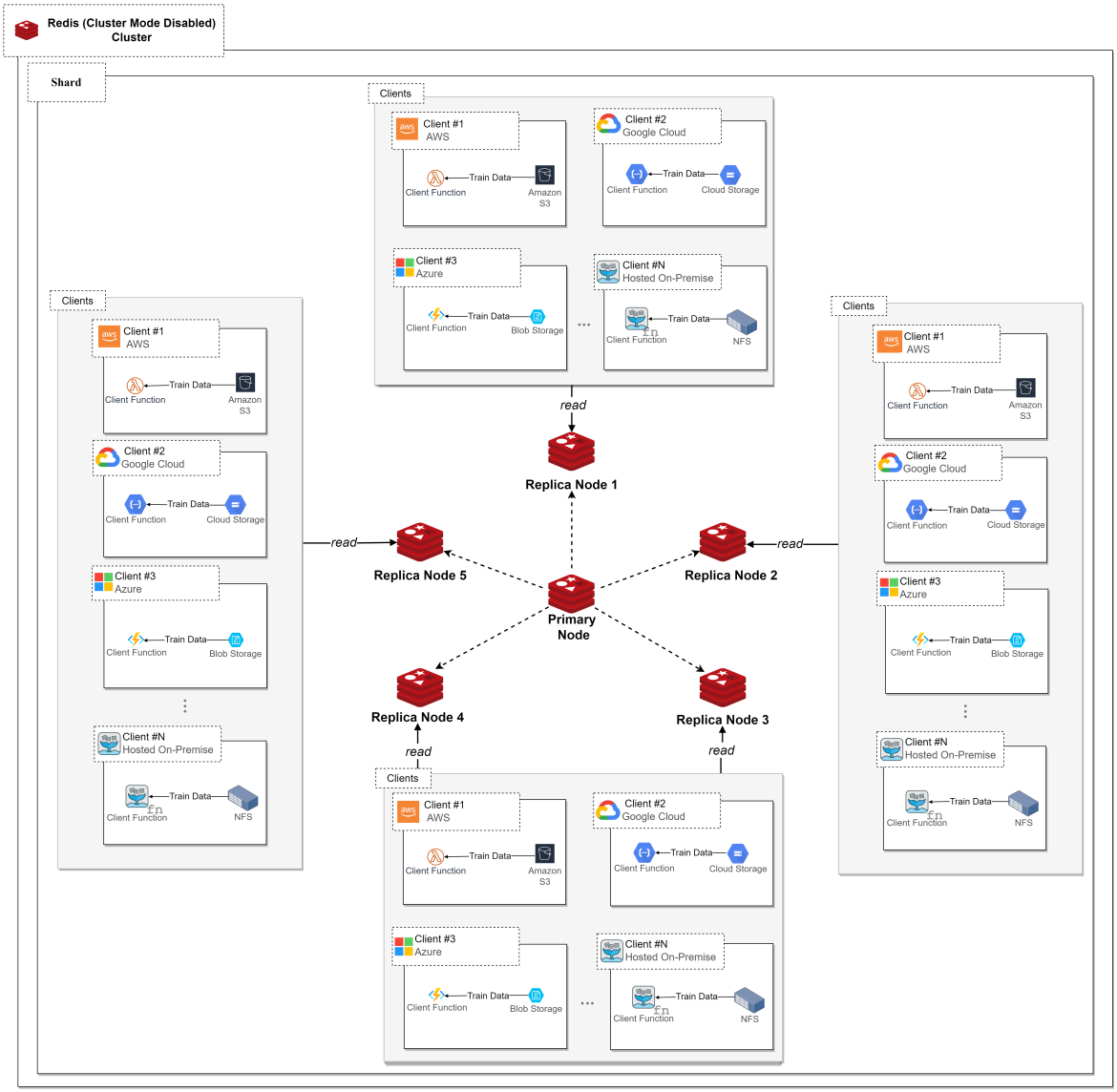


Figure 4.5: Redis parameter server replication overview

In the middle of the diagram, one can see the primary, also referred to as the master node within the single shard. The primary node is the only one that could be used for *writing* as well as *reading* operations from all the involved clients. Then, based on a predefined synchronization scheme, records are copied to the connected replication.

The number of replication nodes within a single shard could vary from 0 to 5, as already

introduced in Section 4.1. Throughout our experiments, we used three different settings and compared the performance results of the system under the consideration of these varying number of replicas, namely: 1) without any replications involved, 2) with two replications as well as 3) with a maximum number of five replications involved.

Additionally, one could see the clients involved in an FL session. As stated, these should all write their latest model updates to the primary node as it is the only one that allows *writing* operations. However, a certain amount of clients could be configured to read the latest model parameters from a specific replication node, offloading the read throughput of the primary node and, thus, increasing *read* operations' performance. The number of clients within a subset reading from the very same replication node could vary from 0 to infinity based on the needs of the *FedLess* user. These clients are different *FaaS* functions running both on-premise as well as on any public cloud provider.

Synchronous vs Asynchronous Synchronization

By default, replication in *Redis* is an asynchronous, non-blocking process on both the primary node as well as on the replication nodes. This assumes that the primary node can still handle incoming queries without the need to wait for any form of notification from a successful replica synchronization. Similarly, the replicas are able to handle incoming queries during replication, however, with the old version of the data set.

Being the default *Redis* replication behavior, we started evaluating the system's performance with an asynchronous realization of the replication process between the primary node and all the connected replication nodes. Throughout the experiments, however, we discovered that this is an obstacle, preventing the successful completion of the FL training session. The following scenario describes the problem with the help of Figure 4.5 in detail: for the sake of simplicity, we would assume to have only 2 replicas besides the primary node - *Replica Node 1*, and *Replica Node 2*, and consider all of the others as disabled for the context of this scenario. Moreover, we assume to have only 2 clients participating in the FL training round, denoting them as *Client 1*, which reads from *Replica Node 1*, and *Client 2*, which reads from *Replica Node 2* and uses the default asynchronous replication synchronization. As the *Primary Node* is the only one supporting *write* operations, our two clients write their latest model updates into that node. After successful completion of a training round, say round `i`, which involves the two clients, followed by a successful aggregation of their weights by the aggregator function, the aggregator function persists the aggregated model parameters in the parameter server. These are then to be used by the clients participating in the following, `i+1` FL training round. Nevertheless, as we are replicating the data in a non-blocking, asynchronous manner, it could happen that only one of the two replications has received the latest, aggregated model parameters at the time its corresponding client was invoked and queries the model parameters from the *Primary Node*. For example, if *Replica Node 1* has received the latest parameters replicated latest parameters from the *Primary Node*, however, the replication on the *Replica Node 2* has not yet been completed, *Client 1* would end up running the training round with the latest parameters, however, *Client 2* would have not yet received these, which is a non-desirable situation as it affects the training performance and correctness.

Based on this, we decided to apply synchronous synchronization of all the involved replication nodes when saving the latest parameters so that all the clients involved in the following FL training round are able to retrieve the latest model parameters. This was achieved with the use of the `WAIT` command in *Redis*, which ensures that a predefined number of replica nodes have received a copy of the exact latest model parameters before moving forward to the training process of the next FL round. It blocks the current client until previous *write* operation, in our case, the one concerned with persisting the latest aggregated model parameters, is successfully acknowledged by at least the specified number of replicas. Section 4.5 provides details on how this approach was achieved alongside further implementation details about the integration of the *Redis* parameter server.

4.4 RedisAI

Being one of the key contributors to the achievement of the initially-stated objectives for improved performance and, thus, decreased communication cost, we briefly present the core concepts of the *redisAI* module in the following. The following description is largely based on the *redisAI* official introduction page [209].

RedisAI is as a module for Deep Learning/Machine Learning models data management, providing out-of-the box support for popular DL/ML frameworks. The list of the *redisAI* supported data structures contains the following entries:

- **Tensors**: being a standard representation for data in DL/ML workloads, the tensor data structure represents an n-dimensional array of values. Like any other data structure in *Redis* and *RedisAI*, tensors are stored and identified by keys;
- **Model**: represents a frozen computational graph by one of the supported DL/ML framework backends that is stored in the database and can be executed/run. Similarly to all the other data structures, it is identified by a specific key;
- **Script**: represents a *TorchScript*[10] program

Within the implementation, we find an application of the **Tensor** data structure, which allows us to simplify our workflow on two exact places, as already presented in Section 4.1. This happens after a successful client training to persist the results in the database as well as after a successful aggregation of the results by the aggregator function on the FL server side.

As noted, *redisAI* allows practitioners to store tensors in *Redis*. This is done by the use of the `AI.TENSORSET` command. A concrete example of storing a tensor looks this: `AI.TENSORSET fedless FLOAT 2 VALUES 2 3` , where the key of the tensor is *fedless* and its values are of type float as specifiec by the corresponding `FLOAT` argument. The `VALUES` argument specifies the shape of the tensor as a list of its corresponding dimensions. For the list of complete commands with all the data structures offered by the module as well as further examples of their use, the reader is referred to the official documentation from *Redis* as referenced at the beginning of this section.

[10] `https://pytorch.org/docs/stable/jit.html`

4.5 Implementation Details

Since this work is a continuation of a previous one conducted in the field with the implementation of the *FedLess* framework, we stick to the already defined standards, languages, and libraries used for the implementation of the existing system. Thus, the implementation as part of this research work is done with the use of the Python 3 language.

Throughout this work, we did not introduce any changes in terms of function deployment and invocation. All serverless functions need to load the Python *FedLess* dependency, which is done in a different manner depending on the serverless provider - functions on the Google Cloud Platform make use of *requirements.txt* to install and integrate *FedLess*, however, *OpenWhisk*, *OpenFaaS*, and *AWS Lambda* use a Docker image with *FedLess* included. For this, we used a public Docker hub repository containing all the images. For the function invocation, we use a blocking HTTP trigger event, just like in the experiments conducted in *FedLess*. While performing the evaluations with bigger model sizes, however, we had to allocate more memory to the functions, in order to handle the model size and conduct the FL training successfully.

4.5.1 Dependencies

As our first and foremost concern throughout this work is replacing the parameter server technology with *Redis*, we extend the system's external dependencies list by adding the corresponding *Redis* dependency. In order to be able to use *Redis* with the Python language, we integrate *redis-py*[11], the most widely-used *Redis* Python Client. For similar purposes, we add *redisai-py*[12], a Python client for RedisAI, which allows the use of the *redisAI* API in Python, and perfectly matches the context and needs of our system. Additionally, we add *ultra JSON*, in short *ujson*, to the used dependencies, in order to achieve faster dictionary serialization and deserialization times as compared to the standard approach with the *json* module.

4.5.2 RQ 1: Parameter Server

The section presents the implementation details and changes introduced throughout this work in the scope of the first research question (RQ 1) and, thus, is concerned with replacing the parameter server with *Redis*, in order to achieve better performance and scalability of the system.

MongoDB vs Redis

Before presenting the exact implementation changes, we briefly compare *MongoDB* and *Redis* and the implications that these had on the implementation of the parameter server. These two distinct technologies have many differences starting with the way how data is being handled and persisted. *MongoDB* is a disk-based database and as such, persists the data on the disk of

[11] `https://github.com/redis/redis-py`
[12] `https://github.com/RedisAI/redisai-py`

the corresponding host machine. On the contrary, *Redis* stores data in the memory, promising faster data access. However, it is limited by the size and amount of the data that could be stored.

Being a key/value store, *Redis* allows, as the description suggests, users store as well as fetch certain values by an identified key using the standard `.set('<KEY>', '<VALUE>')` and `.get('<KEY>')` commands. Thus, querying on multiple conditions is not a natural approach to how the data is being fetched in *Redis*, making *Redis* not the perfect solution for scenarios where the business requirements necessitate querying data on multiple conditions or fields. A concrete example from our code base that is making use of the multi-filed data fetch is the realization of the *load* method responsible for fetching the latest client results. The implementation with *MongoDB* looks as follows: As seen, we are able toapply a filter to the fetch with *MongoDB* and retrieve only client results that correspond to the passed `session_id` and `round_id` arguments. In fact, we did not find the need for such queries throughout the implementation phase as we remove the latest clients' results once these have been combined by the aggregator to achieve better performance. Nonetheless, it is a distinction that is worth mentioning at this point.

Another key difference between *MongoDB* and *Redis* is the way how Python dictionaries are handled. *MongoDB* allows storing documents in accordance with a predefined scheme and, thus, one could adopt the exact form of the dictionaries and persist these accordingly. This, however, is not the exact case with *Redis*. The most common data structures supported by *Redis* include the following: *strings*, *sets*, *sorted sets*, *lists*, *hashes*, *bitmaps*, etc., with hashes being the equivalent to dictionaries in Python, storing a set of field-value pairs. An important obstacle, however, is that the hash data structure supports nesting only one level deep dictionaries, conflicting with many of the objects within our code base.

In order to achieve this with *redis-py*, we use a serialization workaround, which allows us to serialize the nested dictionary as a string and persist a corresponding key/string value pair in *Redis*. The latter can be realized with different serialization techniques and third-party dependencies such as *json*, *ujson*, *msgpack*, and so on. Among these, we aim for the one adding the least serialization time overhead, in order to achieve the best performance and, therefore, reduce the total communication costs. The exact evaluation of the system's performance with all these serialization techniques is described in detail in Section 5.

Redis Pipelining

Even though the use of *redisai-py* allows practitioners to use the *redisAI* module with Python, we had to make some adjustments and look for workarounds aiming at increased system performance. A concrete example includes the need to use pipelining for any bulk action, due to the lack of appropriate API able to handle such scenarios. *Redisai-py*, unfortunately, does not provide any bulk get as well as set tensor operations that allow practitioners to persist and retrieve multiple tensors at once. The example below shows the realization of a bulk tensors set with the use of a *pipeline*, in order to achieve a better performance once persisting all the clients' results within *Redis*. As seen, the tensors of each layer of the corresponding client are set in an iterative manner with the use of the `.tensorset()` method provided by

redisai-py and executed at once afterward with the help of a previously defined pipeline.

```
redisai_pipe = self._redisai_client.pipeline(transaction=False)
for i in range(len(result.parameters)):
    redisai_pipe.tensorset(f"client_result:{client_result_id}_layer_{(i+1)}",
        np.array(result.parameters[i], dtype=np.float32))
redisai_pipe.execute()
```

Listing 4.1: Bulk tensor set realization

Database Type Selection

Adding *Redis* on top of the existing solution, we allow the *FedLess* administrator to decide between the two parameter server technologies once the training session is started. This is realized by introducing an enum type, shown in the listing below, which differentiates between the two different approaches. The administrator provides one of the two values and selects the technology that is to be used for the parameter server within the FL training session. Based on that selection, the initial configurations, as well as the intermediate and final results, are persisted in the corresponding database accordingly.

```
class DatabaseTypeEnum(str, Enum):
    REDIS = "redis"
    MONGO_DB = "mongo_db"
```

Listing 4.2: Database type selection

4.5.3 RQ 2: Replicated Parameter Servers

Synchronous vs Asynchronous Replica Synchronization

Due to the problems presented and described in detail in Section 4.1, we make use of the synchronous way of replication synchronization. Being the non-default option in *Redis*, the synchronous way of handling data replication requires some changes and configurations described in the following.

At first, in order to synchronously copy the data from the primary to the replication nodes, we use the `WAIT` command to prevent the clients from reading an older version of the updated model parameters. This happens once a specified amount of replicas have successfully acknowledged the latest parameters from the primary node. A second argument to the command specifies a timeout in milliseconds, which, when reached, unblocks subsequent read operations even if the specified number of replicas were not yet reached. These two parameters have to be specified by the *FedLess* administrator at the beginning of each FL training session. If not provided, they take default values of 0 replications that have to be reached as well as no predefined timeout threshold.

5 Evaluation

The following sections present the evaluation results of the experiments conducted throughout this study. Section 5.1 introduces the experiment setup that includes the data set, the model architecture, the hyper-parameters, and the hardware setup for the serverless functions. We conclude with comparative examples between the improved system and the older *FedLess* version in Section 5.2. For the comparisons, we use three different settings based on the model size as shown in Table 5.1 and evaluate the system's performance under the consideration various involved clients in each of the three test settings. It is to be noted, that the goal of the experiments is not to evaluate the accuracy of the different models, but rather to study how costly, in terms of execution duration, and, thus, pricing, is each global round of FL training. We compare the results with the results from *FedLess*. Moreover, we observe the dependency between the training costs and the model's size.

5.1 Experiment Setup

Data set

In the following, the underlying experimental environment used for the execution of the evaluations are described in detail starting with the data set. In order to conduct the experiments in an environment as close as possible to the FL setting, we use real data sets from the *LEAF* benchmark framework. The framework is specifically designed for the FL context with applications that, in addition, include multi-task learning, on-device learning, and others. It contains many data sets for FL systems and covers many different domains such as image classification. In particular, we use the *FEMNIST* data set, which is a modification, in accordance with the FL setting needs, of the extended *MNIST* data set, or in short - *EMNIST* [210]. In addition to the original data set, its extended version *EMNIST*, constitutes more challenging classification tasks involving letters as well as digits [210]. The *FEMNIST* data set used throughout the experiments contains over 800,000 image samples of digits and various characters partitioned by different writers. There are 3,550 writers in total, each contributing with roughly 226 images on average. Besides the original pre-processing steps in line with the official recommendations provided by the framework, we additionally sample more users, in order to scale the experiments. Firstly, we sample 25% of the users and discard all that have less than 100 samples. Then we use 10% for testing and end up having 895 partitions. We serve the data set with a *ngnix*[1] as a file server.

[1] `https://www.nginx.com/`

Model Architectures and Number of Clients

The first model that we use throughout the experiments is the same CNN as used for FEMNIST evaluations in *FedLess*. It consists of the following layers: two convolutional layers with a kernel size of 5x5, a fully connected layer with 2048 neurons, and a 10-neurons output layer. We use ReLU as an activation function and categorical-cross-entropy loss function. This gives us a model size of 25911521 Bytes or roughly 26 MB, which is what we refer to as the first test setting in our experiments.

In order to increase the model size and, thus, test the parameter server's performance with larger data transfer, we experiment with the parameters from above and artificially increase the size of the model. Doubling the neurons within the fully connected layer from 2048 to 4096 results in a model of size 51609830 Bytes or nearly 52 MB, which we refer to as the second scenario for our experiments.

Lastly, to achieve a model size of more than 200 MB, we, in addition to the doubled number of neurons within the fully-connected (FC) layer, add three FC layers - two of them containing the same, doubled amount of neurons - 4096, and one with the initial number of 2048 neurons. This results in a model of size 219934718 Bytes or roughly 220 MB. This model size comprises our third experiment setting.

Table 5.1 shows the resulting number of parameters that each of the model sizes contains. Intuitively, doubling the model size roughly doubles the number of trainable parameters within a given model.

	26 MB	**52 MB**	**220 MB**
Number of parameters	6476672	12901248	54981566

Table 5.1: Model sizes and the total number of their trainable parameters

Within each of these three different test scenarios, we use a different number of clients for an FL training round for both *MongoDB* and *Redis* and directly compare the parameter server's performance. Starting with a single client, we increase their number, in order to examine the performance changes as the amount of involved clients increases.

For the case of 26 MB and 52 MB model size, we scale the number of clients involved in an FL training round to 300 and compare the execution duration on both client-side as well as FL server-side using the two different parameter server technologies, namely the implementation from *FedLess* with *MongoDB* and our implementation with *Redis*.

In the case of 220 MB model size, we scale the number of clients per FL training round to 100, due to the increased size of the data that is being transferred. Moreover, this test setting contains results only using *Redis* as a parameter server. This is as a result of the bad *MongoDB* performance in the second test setting with a model of size 52 MB, once the number of clients within a training round exceeds 250. Thus, with a roughly 4 times bigger model, we did not expect that the system with *MongoDB* as a parameter server would end up having successful training sessions.

Hardware Setup

The FL server is running on a machine with 10 vCPUs and 45 GB RAM. On the same machine, we have also deployed a single-node Kubernetes[2] cluster. In contrast to the experiments performed within *FedKeeper* and *FedLess*, we deploy an *OpenFaas* aggregation function inside the cluster. This is primarily caused by the unstable performance of *OpenWhisk*. We encountered constant function failures and dropouts of the aggregator once invoked to combine the results of the clients involved in the training round. This was especially the case with a large number of clients involved in the FL training session, typically in test scenarios that include 100 and more client functions. Nonetheless, from a design perspective, we stick to the approach of keeping the aggregation process physically close to the parameter server and deploy it inside the cluster, even though it could be successfully deployed in the cloud using any cloud provider. This gives the benefit of minimized networking overhead and turns out to be a crucial performance advantage as studied and reported by [76]. The *OpenFaaS* aggregator function was configured with a memory limit of 12000 MB or 12 GB, whereas the *FaaS* client functions have a limit of 2048 MB, if not stated otherwise. The only exception from this setting is the model size of 220 MB in the third test setting. The functions deployed on public cloud providers are in the very same region, namely - Frankfurt, Germany.

5.2 Comparison with FedLess

As introduced at the beginning, the key focus of this study is not on achieving better values in terms of some pre-defined metric such as accuracy, for example. Therefore, factors affecting the convergence rate and, thus, the total number of global rounds is out of the scope of this study. Nevertheless, for the sake of completeness, we mention roughly the number of rounds that were required by the framework to terminate the training process, due to a successful training completion in each of the following settings.

In the following, we briefly describe the three main configurations that have to be set by the *FedLess* administrator at the beginning of each FL training session and which we use for the evaluation of the system's performance, namely the desired accuracy of the training round, the maximum number of rounds that are to be executed as well as the total number of allowed failing clients per round, also known as *stragglers*.

Throughout the experiments, we use a pre-defined accuracy threshold of 0.5. Once reached, the system terminates the FL training session successfully. An unsuccessful termination of the training process occurs once the corresponding accuracy has not been achieved after the execution of a certain number of maximum allowed training rounds. Within our experiments, we used various different values for the parameter, specifying the maximum number of allowed training rounds. Nonetheless, it is to be noted that we never trained the system for more than 15 training rounds. Surpassing the threshold of 15 training rounds typically indicates either a misconfiguration or an implementation error. For the maximum number of stragglers allowed, we accept up to 30% of the involved clients to fail within a round.

[2] `https://kubernetes.io/`

5.2.1 RQ 1: Parameter Server

In order to minimize the impact of performance differences caused by the different FaaS platforms, we rely on a single cloud provider. For the following experiments, we deploy the client functions on the Google Cloud Platform with 2048 MB of memory for the first two model sizes and double their memory size to 4096 MB for the biggest model size. As noted, we deploy the aggregator function with *OpenFaaS*. Throughout the experiments presented in the following, we investigate and compare the scalability as well as the capability of the system to handle larger models when shifting towards *Redis* as the technology for the parameter server.

In Figure 5.1 we show the running times of the client functions for the smallest model within our experiments of the size of 26 MB and a small number of clients starting with a single client setting and increasing their number to 10. We consider different approaches throughout our experiments by exploring different ways of storing the model and client configurations as well as the model parameters. We study their implication on the system's performance, including different serialization techniques.

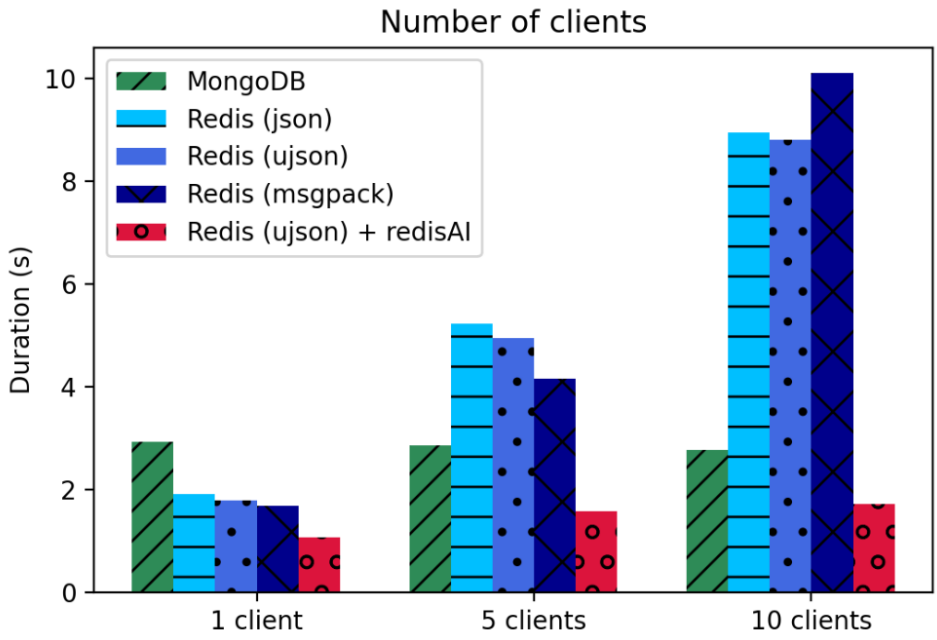


Figure 5.1: Client function duration compared between five different parameter server approaches

All but the *Redis* (with the *redisAI* module) approaches perform serialization and, respectively, deserialization of the model weights when interacting with the underlying parameter server. The model weights are serialized and stored in *BSON* format. The three different *Redis* approaches in blue color differ in the way the serialization for the model and client configurations is being achieved, considering three different serialization techniques, namely *json*, *ujson*, and *msgpack*. The need for serialization for the client configurations follows from the lack of

support in *Redis* for deeply nested Python dictionaries as already discussed in Section 4.5. Therefore, we apply a serialization to both the client as well as the model configurations and store them as strings corresponding to some key. By measuring the performance with all the three evaluated serialization techniques we study the implications as well as the additional overhead that these serialization approaches have on the systems' performance. Finally, we compare all of the above with a novel approach in the DL/ML context by enabling the *redisAI* module. *RedisAI* allows us to omit any serialization and, respectively, deserialization step within the training session.

As seen in Figure 5.1, all of the solutions with *Redis*, in which we serialize the model parameters in *BSON* format, *Redis (json)*, *Redis (ujson)*, and *Redis (msgpack)*, do not lead to a performance increase once we scale the number of clients. In fact, besides the single-client test case, these approaches perform significantly worse even than the existing *MongoDB* solution. Scaling the number of clients involved in an FL training round to 10, which we consider a relatively small number, all but the *redisAI* approaches deliver more than **3 times** slower running times. *MongoDB* requires **2.765195** seconds on average to perform a single training round in *FedLess*, whereas *Redis (json)* needs **8.932911** seconds, thus, being exactly **3.23** slower. *Redis (ujson)* and *Redis (msgpack)* need **8.808135** and **10.094108** seconds respectively. The solution using *redisAI*, in contrast, performs even better than the original one, resulting in **1.566101** seconds. It is to be noted, that all these timings exclude the actual training time of the model as we consider this to be independent on the underlying parameter server technology and serialization technique.

To support these bad scalability findings with another example, we increase the number of clients involved to 50 and perform the exact same comparisons, receiving the following results: *MongoDB* - **5.400897** seconds, *Redis (json)* - **105.632522** seconds, *Redis (ujson)* - **94.485904** seconds, *Redis (msgpack)* - **123.469976** seconds, and *Redis (ujson) + redisAI* - **2.541085** seconds. As seen, we observe similar results - the three approaches using *Redis* with *BSON* serialized model parameters perform significantly bad, independent on the exact serialization technique used for the client as well as model configurations.

We reason this with the inappropriate format in which the weights are serialized when using *Redis* as a technology for the parameter server. Based on these results, we no longer consider using *BSON* serialized model parameters with *Redis* and evaluate the system's performance only with the *Redis* approach, which involves the *redisAI* module. Thus, in the following, we compare the existing, *MongoDB* solution only to the *redisAI* solution and do not consider weights serialization in the context of *Redis* any further.

Though, we still need to perform serialization for the model and clients' configurations, due to the limitations in *Redis* related to the nested dictionaries as described above. Even though *msgpack* showed to be superior to the other two serialization methods, namely *json* and *ujson*, in test cases with smaller number of clients, *ujson* delivered better results once increasing the number of clients to 10 and 50. As one of the main goals of this study is to evaluate the performance of the system in a large-scale scenario case, considering up to 300 clients involved in an FL training, we perform the corresponding serialization with the use of *ujson*.

Test setting 1: Model size of 26 MB

Clients: Figures 5.2 and 5.3 show running time comparisons between the two parameter server technologies used with a model size of 26 MB. To achieve these results, we deploy 300 client functions solely on the Google Cloud Platform. Figure 5.2, however, does not include the training times of the clients as we expect these to be independent of the underlying parameter server technology.

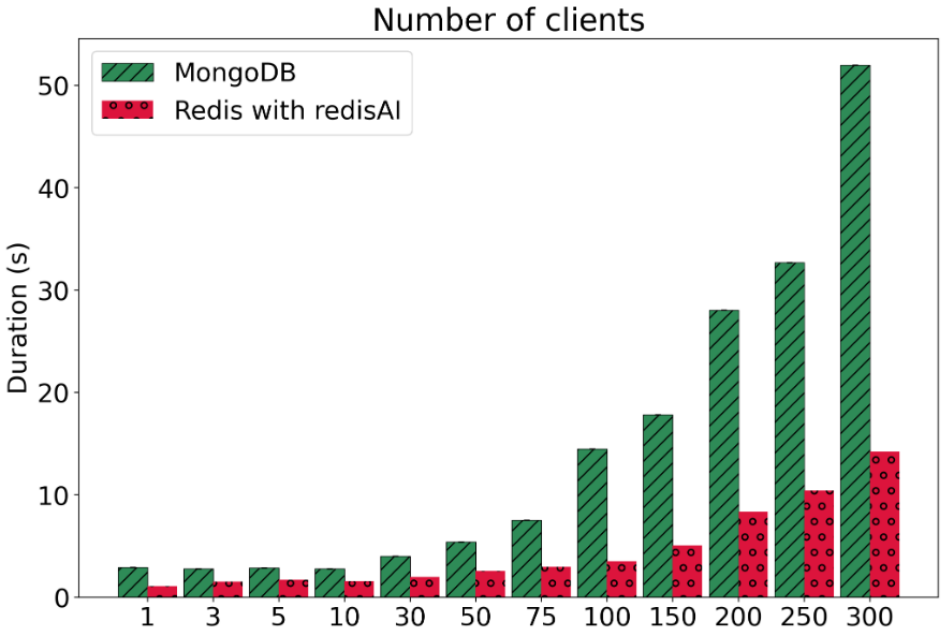


Figure 5.2: Client functions running time comparisons between *MongoDB* and *Redis* for a model size of 26 MB and varying number of clients

The events contributing to the clients' total running times seen in the figure above are the following - *Model Download*, representing the time for the client to load the latest model parameters, *Training Preparation* - the time used for parameters' deserialization (applicable only to *MongoDB*) as well as a model compilation, *Upload Preparation* - covering the time needed for the serialization of the weights (applicable only to *MongoDB*), and *Model Upload* - representing the exact duration of the write operation that persists the latest client model updates.

From the figure, we can see that by increasing the total number of clients involved in an FL training round, we observe an increase in the running times for both parameter server technologies. This, in fact, we consider as expected behaviour of the system as involving more clients within an FL training round increases the complexity and the execution duration of each and every defined event.

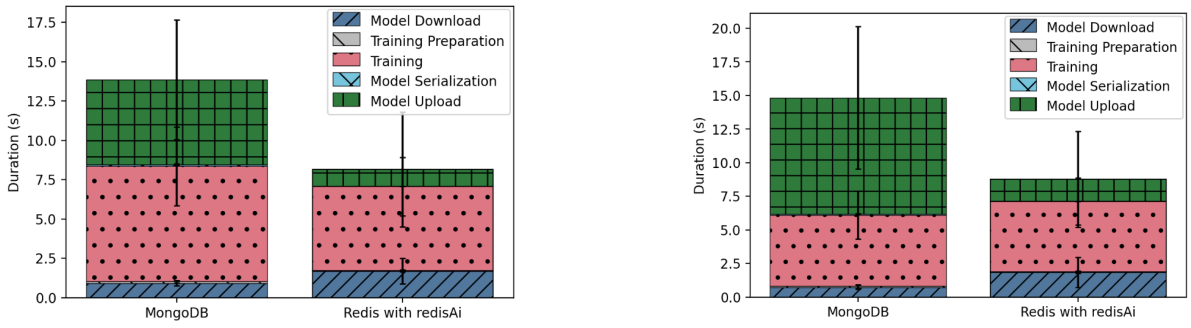
Taking a closer look at the exact comparisons, one can note that *Redis* leads to much faster running times in each and every single test case starting from a single client per round and scaling their number to 300. Training a single client gives us a total, average run time of a

round of **2.931053** seconds (excluding the actual model fit time) with *MongoDB*, whereas *Redis* gives a run time of **1.061442** seconds (again, without considering the actual training time), which is more than **2.5 times** better performance. While scaling the number of clients to 100, we observe a roughly two times performance difference in favor of the *Redis* solution.

Interestingly, having a larger number of clients involved, we see an even greater difference in terms of the system's performance, showing that *MongoDB* is unable to handle many simultaneous interactions with the participating clients, which corresponds with *FedLess*'s observations that motivate this work. Having 300 clients involved in the training process shows the better capability of the *Redis* solution to handle a large amount of participating clients. For 300 clients, for example, *Redis* results in a total duration time of under 15 seconds or exactly **14.212507**, whereas *MongoDB*, on the other side, performs significantly worse with a total duration of nearly 60 seconds or exactly **59.106725**, which is **more than 4 times** worse performance. As in the example with the single client from above, these timings do not include the exact training time as we assume it not to be influenced by the exact technology used for the underlying parameter server.

We consider these findings to happen primarily due to the much better *write* operations that *Redis* supports as compared with the solution involving *MongoDB*. *Redis* clearly outperforms *MongoDB* in the client-side *Model Upload* event by orders of magnitude, leading to a total better round execution duration. Moreover, as we are able to eliminate the need for weights serialization with the help of the *redisAI* module, allowing us to store the client results as tensors, the *Upload Preparation* event does not contribute to the total execution duration at all, which differs from the *MongoDB* solution. The combination of these two factors results in an improved running client-side time of the system with the use of *Redis*.

The two diagrams in Figure 5.3 provide a detailed view of the exact events contributing to the total round execution duration on average, comparing the two parameter server approaches. For the sake of completeness, here we include the actual *Training* event to end up with the exact total round running time required by the involved clients.



(a) Comparison with 50 clients involved in a training round

(b) Detailed comparison with 100 clients involved in a training round

Figure 5.3: Detailed client functions running time comparisons between *MongoDB* and *Redis* for a model size of 26 MB

The plot in Figure 5.3 (a) compares the two technologies under the consideration of 50 participating clients. One can observe the running time differences per individual event such as *Model Upload*, for instance. Taking a closer look to the *Model Upload* event, we can note that *MongoDB* is not able to handle many simultaneous client uploads. *MongoDB* requires roughly **5 times** more time to execute the uploads with an exact duration of **5.384037** seconds against the **1.085305** seconds needed by the *Redis* solution. Additionally, *MongoDB* requires **0.110116** seconds to serialize the weights prior to the upload event, which is not necessary with *Redis*, due to the *redisAI* module.

Similarly, Figure 5.3 (b) provides the exact same comparisons with the consideration of 100 clients involved in an FL training round. In fact, here we observe an even larger difference between the two upload events - **8.637785** seconds for *MongoDB* against **1.64687** seconds for *Redis*. Meeting our initial expectations, the training times of the two approaches are having roughly similar running times as they do not directly depend on the underlying parameter server technology.

The only noticeable drawback of the *Redis* solution relates to the *Model Download* event, which is, in fact, faster with the *MongoDB* solution, taking **0.710569** seconds against the **1.833939** seconds taken by *Redis*. This is primarily caused by the fact that enabling multi-threading for the *read* operations in *Redis* does not improve the read performance. By performing the changes related to the second research question of this study, RQ 2, however, we aim at mitigating these performance issues and decreasing the *read* operations with *Redis*.

We conclude the client-side results for the 26 MB model looking at the two parameter servers considering a larger number of involved clients, namely 200 as shown in Figure 5.4.

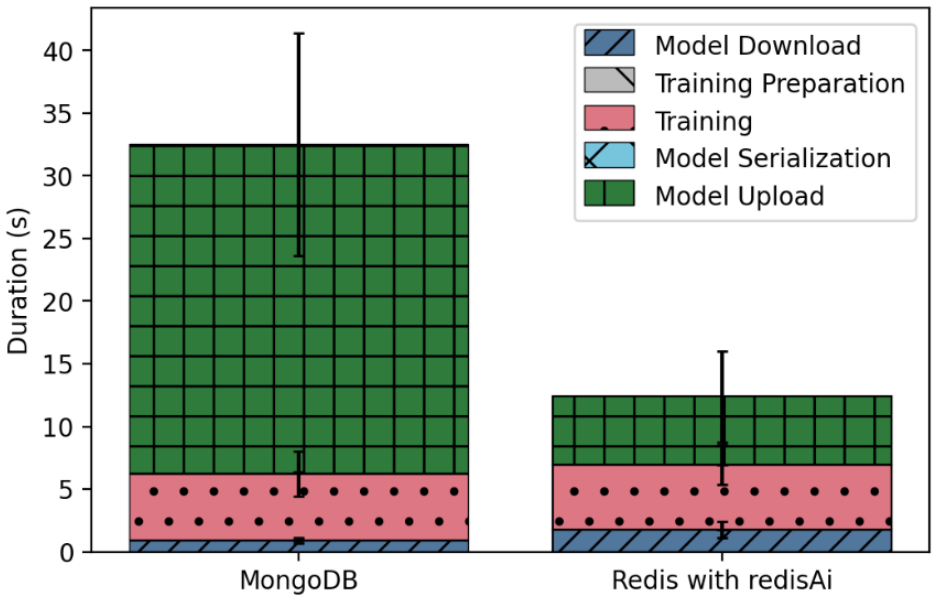


Figure 5.4: Detailed client functions running time comparisons between *MongoDB* and *Redis* for a model size of 26 MB and 200 clients

In Figure 5.4, we plot the exact running times of the individual events and show the capability of the *Redis* solution to better handle FL training, that involves a larger number of clients within a training round. Even though we can see the slightly worse running time of the *Model Download* event with the *Redis* approach - **3.504046** against **0.888442** seconds in favor of *MongoDB*, this is barely noticeable, given the timings of the other events such as the *Model Upload*, for instance. Similarly, the *Training Preparation* event can not be clearly seen in the plot as it takes a tiny portion of the total running time. However, we observe that in this event, similar to the test cases with 50 and 100 clients, *Redis* is superior to *MongoDB* taking just **0.009015** seconds against the more than **6 times** longer **0.057535** seconds of the old solution.

The *Model Serialization* event, happening once the actual training has been successfully executed and before the persistence of the corresponding model parameters, takes **0.098098** seconds of the total running time for *MongoDB* and **0** seconds with the use of the *Redis* solution, due to the convenient and elegant way of persisting the client results that the *redisAI* module provides.

The most noticeable difference, however, is the *Model Upload* event, identified by previous work on the *FedLess* framework as the weak spot of the system. We see that uploading the model weights is a large portion of the total running time throughout an FL training with the use of *MongoDB*. With a total average upload time of **65.645192** seconds, the clients are concerned with uploading their computed results to the parameter server in **89.9%** of the time total on average, which is a huge portion of the round duration. On the contrary, *Redis* requires only **9.892569** seconds to upload the client results, which is more than **6 times** faster than the *MongoDB* solution and only **73.7%** percent of the total round duration on average.

In general, considering all the different test cases that we performed throughout the experiments with a model size of 26 MB, we achieve **2.21 times** better performance with the use of the new *Redis* technology for the underlying parameter server of our system. The average round duration considering all the different scenarios with a varying number of clients is, thus, **2.21 times** faster. This reduction, furthermore, has a positive effect on the costs associated with the use of *FedLess*. The exact cost savings are presented at the end of the evaluations section.

Aggregator: In addition to measuring the performance on the client-side of the system, we evaluated the system's performance on the aggregator-side and compared it with the *FedLes* solution using *MongoDB*. Figure 5.5 shows the running times of the aggregator function, corresponding to the above-described scenario with the consideration of the different amounts of clients throughout an FL training round. We compare and plot the exact execution duration of the aggregator considering a varying number of clients that are being invoked throughout an FL training round. The events contributing to the total running times in the plot include *Clients' Model Download* - representing the download time of all the latest model updates from the latest training round; *Aggregation* - the actual aggregation time of the clients' results, which is based on some predefined aggregation scheme, as well as the *Upload* - referring to the time for the upload of the already aggregated, latest model parameters.

Similar to the client-side observations, we see that by increasing the number of clients

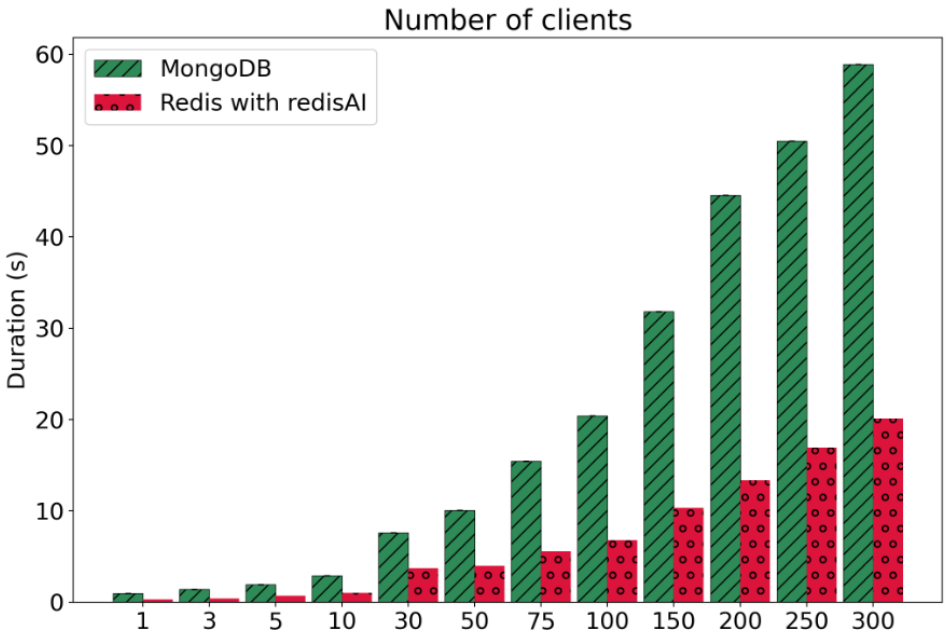


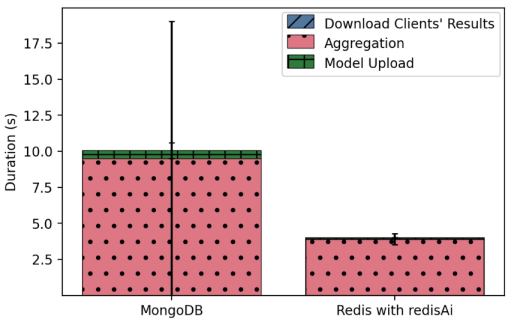
Figure 5.5: Aggregator function running time comparisons between *MongoDB* and *Redis* for a model size of 26 MB and varying number of clients

involved in an FL training round, the duration of the whole aggregation process increases as well. Moreover, we can see that the system using *Redis* as a parameter server outperforms the one with *MongoDB* with all the different numbers of clients. Interestingly, the more clients we consider throughout the training round, the bigger the difference between the two performances becomes. At the point of 300 clients, the *Redis* solution performs almost **3 times** faster than the one using *MongoDB* - with *Redis* we complete an aggregation for **20.111208** seconds on average, whereas using *MongoDB* the aggregation costs nearly a minute, namely exactly **58.906086** seconds. This gives us a significant performance improvement.

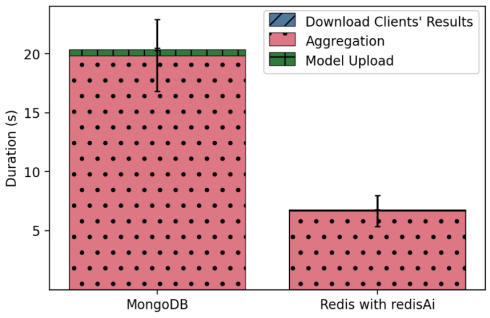
Identical to the client-side measurements, we observe a great performance increase in terms of the model uploads. Furthermore, being able to completely eliminate the weight deserialization and serialization steps, which occur before and after the aggregation process respectively, we are able to further decrease the total running time and, thus, achieve better overall performance as compared with the old *FedLess* implementation.

We plot the differences between the two approaches considering all the individual events contributing to the total round duration. Figure 5.6 compares the performances of the system with *MongoDB* and *Redis* considering 50 and 100 clients within an FL training round. We see that in the 50 client setting as well as in the 100 one, the aggregator with the *Redis* solution clearly outperforms the old implementation of *FedLess* using *MongoDB*. In the 50 clients' case, the total round aggregation time on average is roughly **2.5 faster** with *Redis* with the exact timings being **10.049266** seconds for *MongoDB* and **3.995371** for *Redis*. Comparing the individual events, we further see that both the aggregation as well as the model upload

events take comparably less amount of time. Similar to the client-side measurements, the main two reasons for these results are the better *write* operation timings of *Redis* as well as the eliminated weights serialization and deserialization.



(a) Comparison with 50 clients involved in a training round



(b) Comparison with 100 clients involved in a training round

Figure 5.6: Detailed aggregator function running time comparisons between *MongoDB* and *Redis* for a model size of 26 MB

Scaling the clients to 100, as seen in Figure 5.6 (b), results in a greater difference.*Redis* completes the aggregation within **6.764538** seconds, whereas *MongoDB* needs significantly more time - **20.390591** seconds. Looking at the largest test case possible, considering 300 clients within an FL training, we observe **58.906086** seconds aggregation execution for *MongoDB* and almost **3 times** less with *Redis* - **20.111208**. One noticeable finding, however, is the huge portion of time taken by the actual aggregation process by both of the parameter server technologies. For the case with 300 clients, the system with *MongoDB* is busy with aggregating the weights in **99.08%**. Even though we drastically improve the aggregation performance with *Redis* by removing the need for serialization and, thus, deserialization afterward, the actual aggregation is still an event that takes roughly **99%** of the total aggregation time. We consider this as a potential bottleneck on the aggregation side of the system, which could be mitigated by replacing the aggregation scheme.

In summary for the aggregation-side performance, considering all the different test cases that we performed with a model size of 26 MB, we achieve **2.97** times better performance with the use of the new *Redis* technology for the parameter server.

Throughout all the experiments conducted with a model size of 26 MB, the solution with *Redis* seemed to perform in a more stable manner. On average, we managed to achieve the corresponding predefined accuracy, in 5 to 6 training rounds. Moreover, even though we set a high number of stragglers allowed, we did not experience a training round, with more than 10% failing clients. In fact, the only scenarios in which there were any clients returning a failed function status were cases with the largest possible amount of clients involved within our setting, namely 300, however, not 10% of the total number of clients involved.

Test setting 2: Model of size 52 MB

For the doubled model size we repeat the exact same procedure, comparing the two technologies of the parameter server with various numbers of clients involved and investigating the ability of the system with *Redis* to handle larger model sizes. In contrast to the first test setting, however, here we only look at a reduced number of test cases defined by the number of clients used throughout an FL training round. We evaluate the system's performance based on 6 different settings, instead of 12 as in the preceding evaluations. The minimum, as well as the maximum number of clients involved, do not change, nonetheless, we do not consider test cases with 5, 30, 75, 150, and 250 clients throughout the evaluations of the second biggest model.

Similar to the tests performed with the smallest model, we do not aim to achieve a certain performance in terms of a predefined metric such as accuracy. Thus, we randomly pick a value of 0.5 which has to be reached in order to successfully terminate the training round. If this value is not achieved within a maximum number of 15 training rounds, the process terminates automatically. To perform the measurements, we use the exact same function configurations as used throughout the first test setting. That is, we deploy 300 Google Cloud Functions with 2048 MB memory each as well as an *OpenFaaS* aggregator function running on-premise.

Clients: Figure 5.7 shows the client running times of the two parameter server technologies in a comparative manner.

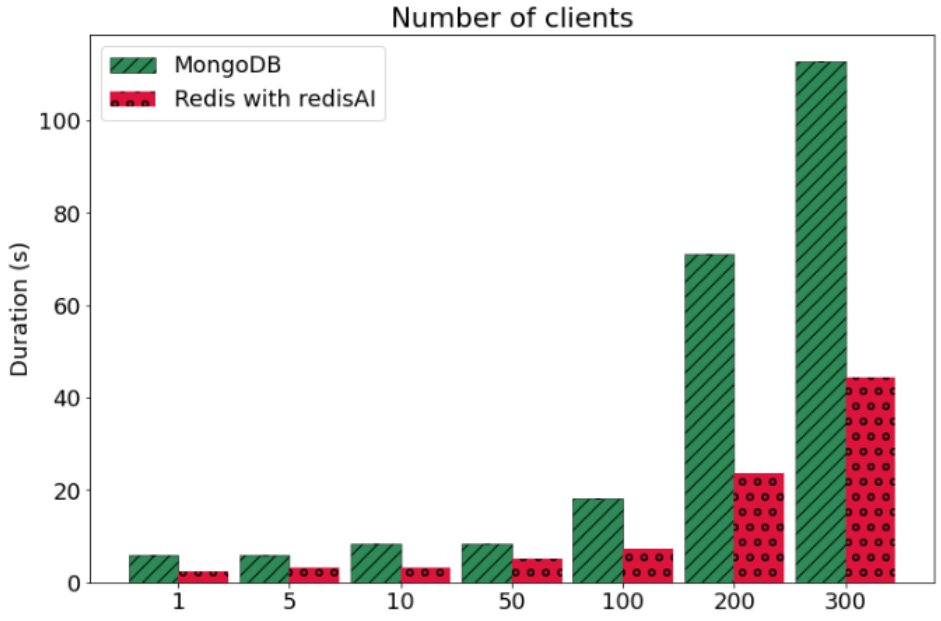
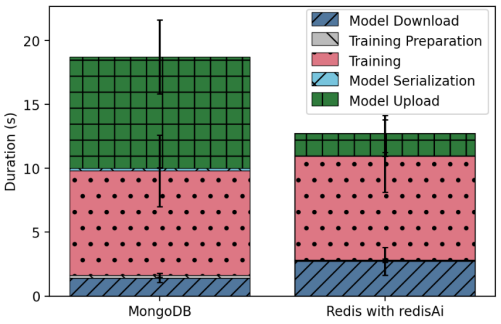


Figure 5.7: Client functions duration comparisons between *MongoDB* and *Redis* for a model size of 52 MB and varying number of clients

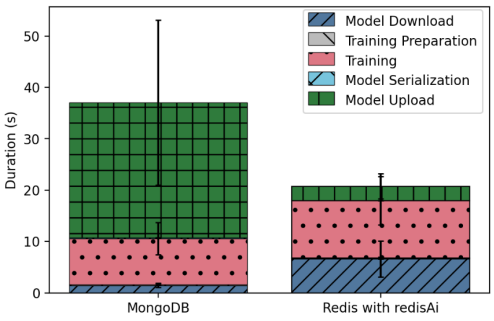
Throughout the experiments we observe the exact same pattern as with the 26 MB model - *Redis* outperforms *MongoDB* in each and every test case. Moreover, the more clients we include within a training round, the more severe the running time difference between the two technologies becomes. Like with the 26 MB size, we do not consider the actual training times in this diagram as we assume these do not depend on the underlying parameter server technology.

In the very first scenario with a single client, we see that the *Redis* solution is superior to *MongoDB*. The average round duration for *Redis* is **2.306066** seconds and the one for the solution using **MongoDB** takes **5.852588** seconds and is more than **2.5 times** slower. Evaluating the performances with a larger number of clients, for example, 200, we can see nearly the same difference in their run times - **71.02988** seconds for *MongoDB* against **26.43306** seconds for *Redis*. Looking at the even larger number of clients involved, we see an even bigger difference. Having 300 clients, *MongoDB* ends up having an average running time per round of more than 2 minutes or exactly **136.422855** seconds, which is more than **3.5 times** more as compared with the **46.189571** seconds *Redis* performance.

Figure 5.8 provides a detailed view of the performances of the two approaches considering all the individual events that occur throughout an FL training on the client-side of the system, including the actual training time - the *Training* event, unlike in the figure from above.



(a) Comparison with 50 clients involved in a training round



(b) Comparison with 100 clients involved in a training round

Figure 5.8: Detailed client functions running time comparisons between *MongoDB* and *Redis* for a model size of 52 MB

Looking at the exact running times of these events, we can validate our existing observations from the evaluations with the 26 MB model. Uploading the model using *Redis* clearly outperforms the same event when using *MongoDB* by orders of magnitude, resulting in better performance in total. Figure 5.8 (a) shows the running times with 50 clients involved. With **8.715303** seconds upload time, the existing *FedLess* solution using *MongoDB* performs worse as compared with the **1.743917** seconds by *Redis*. Moreover, the upload event takes a big portion of the whole duration - **46.52%**, whereas the running times with *Redis* of the individual events are distributed uniformly. The clients are busy uploading their results only in **13.72%** of the total time.

Similar differences can be observed in Figure 5.8 (b) showing the results with the consideration of 100 clients per FL training round. The model upload event using *Redis* is significantly faster than the one with *MongoDB*. In total, using *Redis*, we achieve nearly **4 times** faster running times - **36.630309** seconds vs. **9.52527** in favor of *Redis*. The upload time with *MongoDB* takes **71.67%** and with *Redis*, on the contrary, only **29.04%**.

By measuring the system's performance with a model size of 52 MB, we average the performance speed ups and achieve a **3.35 times** better performance with the use of *Redis* as a technology for the underlying parameter server of the system. As a result of that, we were able to reduce the costs associated with the use of *FedLess*.

Aggregator: As with the smallest model size, we look at the running times once the aggregator function has been invoked to combine the latest clients' results in more detail. The running times on the aggregator-side, as displayed in Figure 5.9, strengthen the observations and show the superiority of *Redis* over the existing *MongoDB* solution.

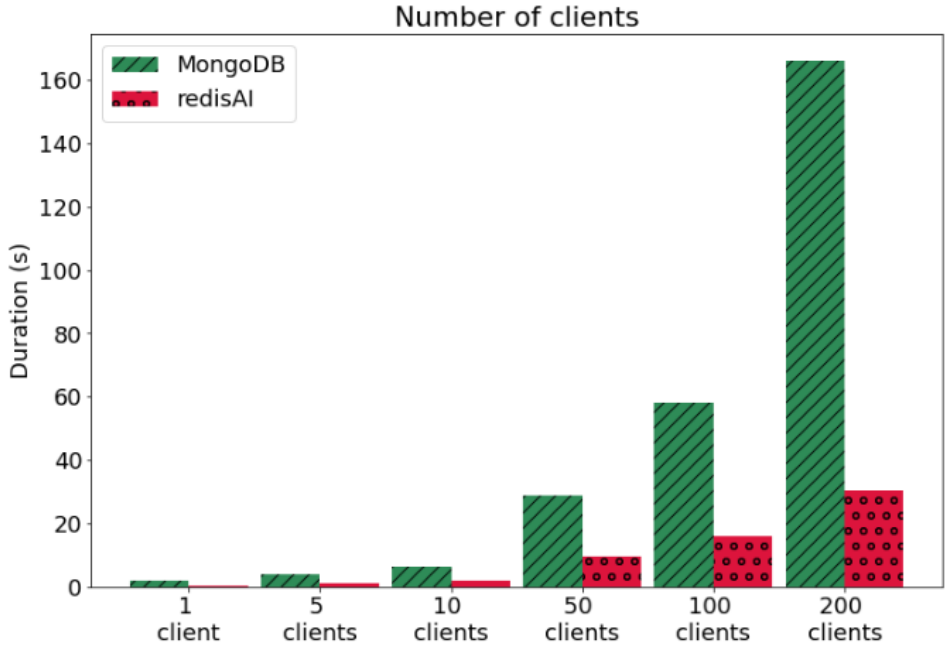


Figure 5.9: Aggregator function duration comparisons between *MongoDB* and *Redis* for a model size of 52 MB and varying number of clients

In fact, the difference in this test setting is even stronger as we achieve **4.6** times faster running times. The following values from the figure illustrate this in detail: 10 clients - **6.492808** seconds for *MongoDB* vs. **2.039934** seconds for *Redis*, which is exactly **3.1 times** slower; 50 clients - **28.961579** seconds vs. **9.798537** seconds in favor of *MongoDB*; 100 clients - **58.199336** seconds for *MongoDB* vs. **17.188075** seconds, resulting in exactly **3.3 times** slower performance. The difference increases in the case of 200 clients. We observe **5 times** better performance - **72.781905** seconds for *MongoDB* vs. **35.161906** seconds for *Redis*.

Looking at the contributions of the individual events, we see that the aggregation is the most costly operation occurring on the aggregator side. The client download events have a small contribution to the total average running time of the round. For example, with just **0.000073** seconds for the case of 50 clients involved and *MongoDB* as an underlying parameter server, the download event makes less than 0.1% of the total running time. Similarly, the upload time of storing the aggregated, latest model parameters is also relatively small as compared with the aggregation time and has a small contribution to the total running time. We have **0.971391** seconds upload time, being roughly 3.5% of the total **28.961579** seconds.

Figure 5.10 displays these detailed observations and compares *MongoDB* with *Redis* with 200 clients. As seen, the contributions of the download - *Download Clients' Results* and the upload - *Model Upload* events are relatively small in duration as compared with the aggregation event. Looking at the two aggregations of the two different parameter server approaches, we observe a great improvement with the use of *Redis*. As noted, besides the actual aggregation, we have to perform the corresponding serialization steps, when using *MongoDB*. These steps are included within the *Aggregation* event displayed below and are no longer necessary with the use of *Redis*.

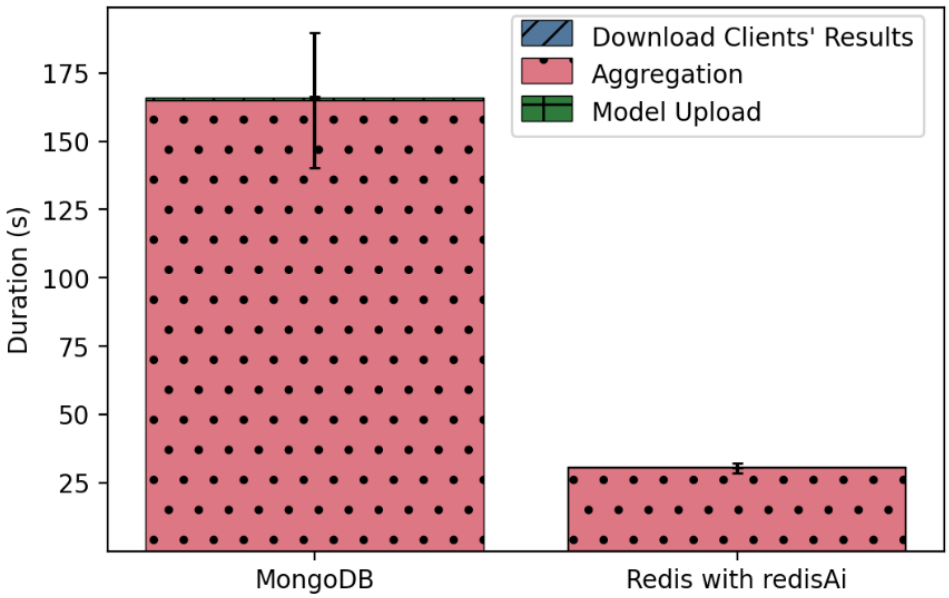


Figure 5.10: Detailed aggregator function running time comparisons between *MongoDB* and *Redis* for a model size of 52 MB and 200 clients involved per FL training round

The aggregation with *MongoDB* takes longer than **2.5 minutes**, whereas the solution with *Redis* needs a bit more than **30 seconds**, thus, being more than **5 times** faster. As with the smaller model, the aggregation event is the most costly one on the aggregator side. The download, as well as the upload events, take significantly less amount of time as compared to the actual aggregation. These are barely noticeable in the plot, contributing only a small portion of the total duration.

In general, throughout the aggregator experiments with the model of size 52 MB, we achieve **4.6 times** better running times with *Redis* as compared with the existing *MongoDB* solution. However, similar to the evaluations in the previous test setting, we see that the actual aggregation clearly dominates all the other events such as downloads and uploads of the model. Out of the total **30.630445** seconds in the 200 clients' case with *Redis*, **30.47168** are taken by the actual aggregation, which is nearly **99.5%** of the whole aggregation process.

Test setting 3: Model size of 220 MB

Within this test setting, we demonstrate the capability of the *Redis* parameter server not only to handle a large number of clients, but also to interact with models of larger size. For this, we use a model of size 220 MB. We achieve the exact model size of 220 MB by artificially adding new fully-connected layers to the network. For these experiments, we increase the memory of the functions from 2048 MB to 4096 MB. Functions with 2048 MB memory are not able to handle such big model sizes as it exceeds the pre-configured memory limit. Additionally, we evaluated the system's performance only considering the *Redis* solution for the parameter server as *MongoDB* resulted in poor performance with many unsuccessful training rounds and client functions' failures.

Clients: Figure 5.11 compares the performance of the system with *Redis* and the three different model sizes considering a varying number of clients. Due to the large size of the biggest model, we consider up to 100 clients, in contrast to the preceding test settings.

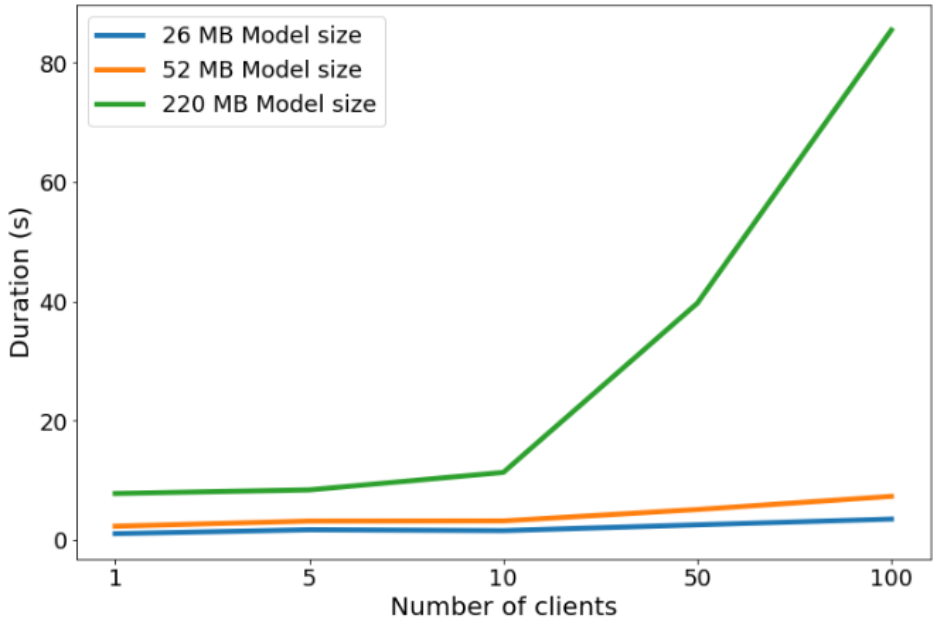


Figure 5.11: Client functions duration comparisons between models of different sizes and varying number of clients using *Redis*

The running times in the comparison plot from above, as well as the exact measures discussed in the following, do not include the actual training time. We can observe that the model size has a linear relationship with the average round duration of the clients. Looking at the single-client case, we see that the biggest model size is roughly **8 times** slower than the smallest one. The model of size 220 MB has an average running time of **7.797662** seconds and the one of size 26 MB, **1.061442** seconds. In fact, the difference in their running times is roughly equal to the difference in their exact sizes in bytes, showing the relationship between the size of the model and the average round duration of an FL training.

Looking at a larger number of clients, we see that the difference tends to increase. Considering 10 clients, the smallest model of 26 MB requires **1.566101** seconds on average, the model of size 51 MB - **3.200821** seconds, and the biggest model - **11.327892** seconds, in line with the observations from above. Training with 50 clients gives us the following results - **2.451213** seconds, **6.506457** seconds, and **39.698943** seconds for all of the three models sorted by their size respectively. In this case, with 50 as well as 100 clients, we observe an even larger running time difference between the smallest and biggest model.

In general, by averaging through all the performance differences that we evaluate within this test setting, we see that by increasing the size of the model **8 times**, we observe a roughly **10 times** slower round duration on the client-side of the system.

Aggregator: For the sake of completeness, we show the exact same system performance comparisons with the aggregator function in Figure 5.12.

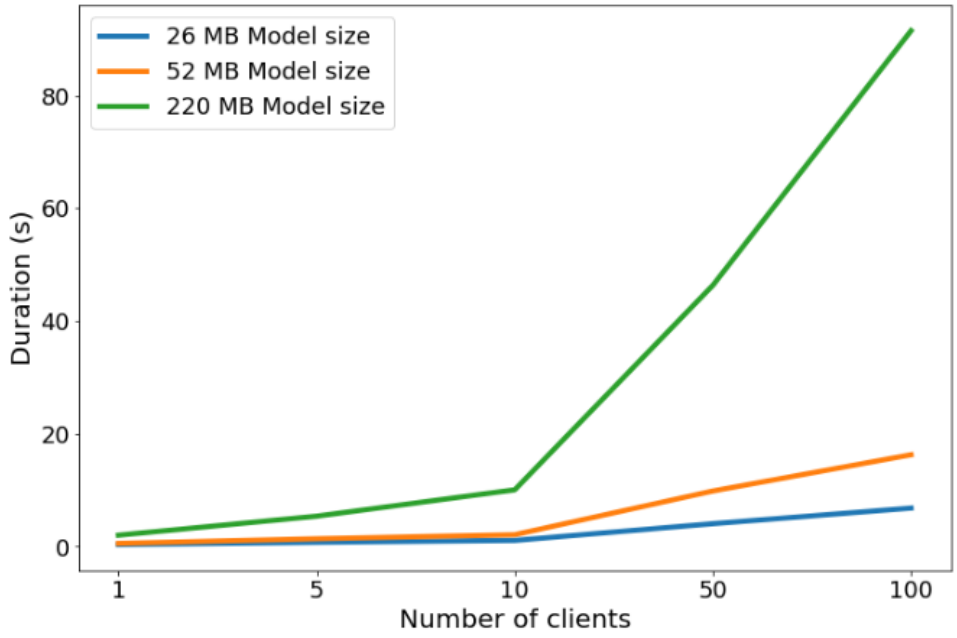


Figure 5.12: Aggregator function duration comparisons between models of different sizes and varying number of clients using *Redis*

Similar to the client-side performances, we see a relationship between the model size and the corresponding running time of the aggregator. A single client leads to the following running times: **0.274577** seconds for the smallest model, **0.494854** seconds, which is roughly **twice as much**, in line with the model size difference of the second model, and **1.952163** seconds, being roughly **8 times more** for the largest model. We observe similar differences by scaling the number of involved clients to 10 with **1.021198** seconds, **2.039934** seconds, and **10.007791** for the corresponding model sizes respectively. Looking at the diagram for 100 clients, however, similar to the observations on the client-side, the gap between the smallest and the largest models seems to increase. The model with a size of 26 MB requires a bit more than 5 seconds on average, or exactly **6.764538**, whereas the biggest model needs 1.5 minutes or exactly **90.537119** seconds to perform the aggregation of the latest model updates.

Even though we were able to increase the performance of the system on the aggregator by introducing *Redis*, we, even with the largest model size, see that the whole aggregation process, including the events measuring the interaction with the underlying parameter server such as the download and upload ones, is clearly dominated by the time for the actual aggregation. This is, however, independent of the parameter server technology. For the 100 clients case, the actual weights aggregation takes **90.537108** out of the total **91.501616** seconds, which is nearly **99.0%** of the whole aggregation process. Thus, in the future, we might consider switching to a different aggregation scheme that could eventually speed up the actual aggregation's performance.

Based on all the experiments conducted throughout the first research question, we could achieve a better performance of the system with the use of *Redis* as a technology for the underlying parameter server. We outperform *MongoDB* by orders of magnitude on both the client-side as well as the FL-server side of the system considering a various number of clients and models of different sizes. Having a model of size 26 MB, our evaluations showed **2.21 times** faster client-side running times and **2.97 times** faster performance for the aggregation function. Increasing the model to 52 MB, we end up having **3.35 times** better running time for the client functions and **4.6 times** for the aggregator.

The only event in which *MongoDB* showed to be superior to the *Redis* solution is the model download - while downloading the latest model parameters, in order to train the model in a corresponding training round. This, as discussed, is primarily caused by the single-threaded *Redis* nature. However, due to the much faster *write* operations as well as the eliminated need for serialization as well as deserialization of the model weights, the system ends up having a better average round running times in total and, thus, results in reduced costs associated with the use of the framework.

In the scope of the second research question (RQ 2), we aim to further investigate and improve this observed performance issue with *Redis*. We try to reduce the download times of the clients involved in an FL training round by integrating multiple replicated parameter servers within the system and, thus, achieve an even better performance.

5.2.2 RQ 2: Replicated parameter servers

Throughout the system's performance evaluations in the scope of this research question, we experiment with different number of replications. We use the results for *Redis* from RQ 1 as basis for the following comparisons. We compare the system's performance with no replicas involved (results from RQ 1) with two different replication configurations, using 2 and then the maximal supported number of replication nodes in *Redis*, namely 5. For this, we use models of two different sizes as in the RQ 1 - namely, 26 MB as well as 52 MB. Moreover, we perform the evaluations with the use of three different test cases defined by the number of clients involved - 30, 50, and 100 clients.

Test setting 1: Model size of 26 MB

The experiments identified several interesting observations by applying various numbers of replicas and comparing the results between the two different model sizes. Table 5.2 shows the evaluation results, in terms of the running times of the individual events as well as their total time.

No. clients	Event	MongoDB (0 replicas)	Redis (0 replicas)	Redis (2 replicas)	Redis (5replicas)
30	*Model Download*	0.707914 s.	0.842111 s.	0.732189 s.	0.610326 s.
	Training Preparation	0.102284 s.	0.077436 s.	0.035969 s.	0.06933 s.
	Weights Serialization	0.09106 s.	0 s.	0 s.	0 s.
	Model Upload	3.098602 s.	1.051363 s.	1.258174 s.	1.540627 s.
	Total	**3.99986 s.**	**1.97091 s.**	**2.026332 s.**	**2.220283 s.**
50	*Model Download*	0.77321 s.	1.37695 s.	0.704794 s.	0.658058 s.
	Training Preparation	0.081641 s.	0.076203 s.	0.033261 s.	0.026617 s.
	Weights Serialization	0.087529 s.	0 s.	0 s.	0 s.
	Model Upload	4.458517 s.	1.087932 s.	1.713158 s.	2.108837 s.
	Total	**5.400897 s.**	**2.541085 s.**	**2.451213 s.**	**2.793512 s.**
100	*Model Download*	0.845638 s.	3.565815 s.	0.711076 s.	0.798763 s.
	Training Preparation	0.149125 s.	0.074946 s.	0.008399 s.	6.247293 s.
	Weights Serialization	0.099693 s.	0 s.	0 s.	0 s.
	Model Upload	13.3744 s.	2.307927 s.	5.227238 s.	10.908693 s.
	Total	**14.468856 s.**	**5.948688 s.**	**5.946713**	**17.954749**

Table 5.2: Detailed clients' running times for *Redis* with different number of replications nodes involved and a model size of 26 MB

We identify that the use of a different number of replicas does influence the individual events. The total running time, however, seems to be in a similar range with the only exclusion being the 100 clients with the use of 5 replicas. For example, we see a *Model Download* time of the system without the use of any replication nodes to be **0.842111** seconds, whereas by

adding two replication nodes, we are able to decrease it to **0.732189** seconds. Increasing the number of replicas even further to the maximum allowed number of 5 replicas, we again reduce the download time of the model even further and achieve a download running time of **0.610326** seconds. Thus, we observe that increasing the number of replication nodes has a positive effect on the read operations in *Redis*, and on the corresponding *Model Download* event, respectively. This is true when looking at the system's performance with an increased number of involved clients as well. For 30 clients, for instance, we are able to decrease the download time from **1.37695** seconds to **0.658058** seconds by scaling the number of replication nodes used from 0 to 5.

Nonetheless, introducing replication nodes and decreasing the running time of the *Model Download* event seems to have the exact opposite effect on the fast *write* operations and the *Model Upload* event, respectively. We see that in the 30 clients' case, the best performance we can achieve in terms of model upload is without the consideration of any replicas involved. Starting with an upload time of **1.97091** seconds with no replication nodes, the running time of the event increases to **2.026332** seconds when we enable 2 replication nodes, and even to **2.220283** seconds, once the number of replicas reaches the maximum allowed, namely 5. The same holds true for the case of 50 clients, we get constantly increasing running times of **1.087932** seconds, **1.713158** seconds, and **2.220283** seconds, while increasing the number of replication nodes within the system. An identical dependency is to be observed once 100 clients are involved in an FL training round.

The observations from above show a certain trade-off with the use of replications. The more replica nodes we use within the system, the better the *read* running times of the system seem to be and, thus, the less costly the *Model Download* event. On the other hand, however, the increased number of replicas leads to worse *write* operation running times and, thus, more costly *Model Upload* times. In total, these two events seem to compensate for each other, which results in similar overall performances of the system. Even though the total execution duration seems to slightly increase, they do not differ by much. The overall duration with 30 clients for 0 replication nodes is **1.9709** seconds against the **2.026332** seconds for 2 replicas and **2.220283** seconds for 5 replicas. This relation can be observed in any of the three cases described in Table 5.2. The 50 clients' results, for instance, have a difference of less than 0.2 seconds being exactly **2.541085** and **2.793512** seconds for the two replication extremes within our setting.

Test setting 2: Model of size 52 MB

For the test setting with the model of size 52 MB we follow the exact same approach and execute performance tests considering two distinct test cases including 50 and 100 involved clients. Starting with 50 clients, we scale the experiments to 100, using three different replication configurations, namely without any replica nodes involved, with 2 as well as with 5 replicas. The evaluation results are shown in Table 5.3.

Within the 30 clients' case, we have the following running times for the *Model Download* - **2.457766** seconds for 0 replicas, **1.947995** seconds for 2, and **1.305657** seconds for 5, in line with the observations from above. Thus, adding more replication nodes to the system seems

to have a positive effect on the *read* time of the system, reducing the running time for the model download. Similarly, with the 100 clients case, we observe the best performance in terms of the *Model Download* event shows the system with 5 replication nodes.

Exactly on the opposite are the times for model uploads. We get the best running time performance having no replica nodes, **2.567118** seconds for 30 clients' case, and the worst once we add 5 nodes - **8.949222** seconds for the same test case including 30 clients. Overall, we see that even though it ends up with the worst model download running times, the system with 0 replication nodes seems to deliver the best running time results.

No. clients	Event	MongoDB (0 replicas)	Redis (0 replicas)	Redis (2 replicas)	Redis (5replicas)
50	*Model Download*	1.527567 s.	2.457766 s.	1.947995 s.	1.305657 s.
	Training Preparation	0.217863 s.	0.082871 s.	0.029884 s.	0.022745 s.
	Model Serialization	0.226666 s.	0 s.	0 s.	0 s.
	Model Upload	6.498947 s.	2.567118 s.	4.528578 s.	8.949222 s.
	Total	**8.471043 s.**	**5.107755 s.**	**6.506457 s.**	**10.277624 s.**
100	*Model Download*	1.691633 s.	4.718409 s.	4.260935 s.	1.193245 s.
	Training Preparation	0.201122 s.	0.070589 s.	0.091714 s.	0.202474 s.
	Model Serialization	0.228204 s.	0 s.	0 s.	0 s.
	Model Upload	16.177124 s.	2.524331 s.	13.325055 s.	16.746699 s.
	Total	**18.298083 s.**	**7.313329 s.**	**17.677704**	**18.142418**

Table 5.3: Detailed clients' running times for *Redis* with different number of replications nodes involved and a model size of 52 MB

With the implementation throughout the second research question of this study (RQ 2), we were able to successfully address the identified issue of the system with *Redis* discovered in the scope of RQ 1, namely, the bad *read* operations. With the use of replication nodes, we were able to decrease the running times of the *Model Download* event that happens at the client-side of the system once a particular client has been selected to participate in an FL training round.

Nonetheless, we identified that this improvement comes at a certain cost related to the *write* operations. By improving the download times, we worsen the upload running times. Introducing more and more replication nodes, the *Model Upload* event shows increasingly worse performance. In fact, the more replication nodes we involve, the better *read* operation performance we achieve, however, the worse *write* operation becomes. Looking at the test cases including a large number of clients, we see that having multiple replication nodes even increases the total running times and, thus, decreases the general performance of the system.

As a result of these observations, we do not see any general improvement of the system with the use of various replications and find no benefits in introducing this approach to the system. In fact, it is just the opposite, not only doesn't it lead to a reduction in the total round duration, but it also introduced an additional configuration overhead. Thus, we could more easily classify it as a downside, rather than an actual performance improvement.

5.2.3 FedLess Costs

Based on the improvements and the observed results throughout the experiments, in the following, we provide monthly cost estimation and compare the costs related to the two parameter server approaches - *MongoDB* and *Redis*. As one of the core distinctions of *FedLess* is its ability to work with any type of *FaaS* provider, be it a public or a self-hosted one, achieving better communication costs for the whole training process is an important improvement.

We look at the three different test settings, defined by the size of the model, and estimate the monthly costs of *FedLess* with *MongoDB* and *Redis* in US dollar. As we evaluate the performance of the system with the use of the Google Cloud Platform, the costs in the following are based on the Google Cloud pricing model.

For the calculation, we use the following inputs: the memory and the CPU of the functions used, the number of function invocations as a product of the number of clients involved with the number of rounds in the FL training, the total runtime of the function as well as the model size used.

Test setting 1: Model of size 26 MB Table 5.4 shows the monthly cost estimations for the use of *FedLess* with a 26 MB model size. For the case of 50 clients, we receive a total runtime duration of **4153.73** seconds with *MongoDB* and two times smaller runtime, exactly **2037.80** seconds, with *Redis*. As a result of that, we see that given the parameters from above, *FedLess* with the use of a *Redis* as a parameter server is two times cheaper as compared with the solution with *MongoDB*. With *Redis*, *FedLess* ends up with **15.53 USD/month** against the **30.87 USD/month** with the use of *MongoDB*. In the large-scale scenario, we observe an event higher price difference, namely **572.04 USD/month** against **172.96 USD/month** in favor of *Redis*.

No. clients	MongoDB costs (USD/month)	Redis costs (USD/month)	Cost savings
50	30.87	15.53	50.3%
100	109.03	77.80	71.36%
200	277.72	83.89	30.20%
300	572.04	172.96	30.24%

Table 5.4: Monthly cost estimation comparisons between *MongoDB* and *Redis* for 26 MB model

The last column of the table displays the cost savings that we were able to achieve with the *Redis* solution in each test case. On average, we ensure a **45.52%** cost reduction with *Redis*.

Test setting 2: Model of size 52 MB Table 5.5 shows the calculated monthly cost comparisons of *FedLess* for a model of size 52 MB. Here we use the exact same parameters as in the first test setting with the smallest model.

No. clients	MongoDB costs (USD/month)	Redis costs (USD/month)	Cost savings
50	83.00	47.58	57.33%
100	268.56	84.11	31.32%
200	700.15	405.31	57.89%
300	960.84	592.23	61.64%

Table 5.5: Monthly cost estimation comparisons between *MongoDB* and *Redis* for 52 MB model

We identify roughly the same pattern as with the model with the smallest size, namely, *Redis* seems to be nearly twice as cheap as *MongoDB* on a monthly basis in US dollars. For the case of 50 clients involved in an FL training round, we receive a total duration of **5619.773967** seconds for *MongoDB* and **3176.863725** seconds for *Redis* and, thus, costs of **83.00 USD/month** and **47.58 USD/month** respectively. Increasing the number of clients involved, increases the total round duration, having the effect of increased monthly costs in accordance with the pre-defined pricing by the Google Cloud Platform.

Even though these estimations are based on the exact pricing by a single public cloud provider and, thus, are specific to Google Cloud, we expect to observe the exact same pattern and, therefore, similar results, when looking at other prominent public cloud providers such as AWS and Azure, for instance.

Similar to the setting with the model of size 26 MB, we take a closer look at the cost savings with the *Redis* approach. The last column of Table 5.5 displays the exact cost savings per test case, defined by the number of clients involved in FL training. On average, we receive a total cost reduction of **52.04%** by replacing the parameter server in *FedLess* with *Redis*.

Test setting 3: Model of size 220 MB In Table 5.6 we present the results for the biggest model of size 220 MB . The table shows the costs for *FedLess* using the biggest model of size 220 MB, even though we are not able to compare the costs between the two parameter server options as we execute only measurements with *Redis* in this test setting. For these experiments, we use two different test cases, defined by the number of clients involved, namely 50 and 100 clients.

No. clients	Redis costs (USD/month)
50	203.50
100	621.04

Table 5.6: Monthly *FedLess* cost estimation with *Redis* for 220 MB model

In this scenario, we slightly adjust the configurations as the functions require an increased amount of memory, namely 4096 MB, instead of the previously used 2048 MB as a result of the increased model size, which increases the costs for using *FedLess*. Having 50 clients gives a total running time of **11881.69** seconds and, therefore, an estimated monthly cost of **203.50 USD**. In the larger case, including 100 clients, we end up with **36442.90621** seconds of total runtime resulting in **621.04 USD** monthly costs for the execution of *FedLess*, in line with the pricing by the provider.

5.3 Discussion

In summary, we were able to evaluate the system's performance with the new technology for the parameter server and, thus, reveal several interesting findings. Firstly, within the first part of the work and in the scope of the first research question (RQ 1), we were able to clearly outperform the existing *MongoDB* parameter server solution in *FedLess*, in terms of the communication that occurs within an FL training round. By implementing it in *Redis*, the system executes multiple client uploads much faster as compared with the current system's parameter server solution. Within our experiments, we scaled the number of clients involved in an FL training round to 300 and proved the ability of the *Redis*' implementation to handle many simultaneous model uploads in a significantly reduced amount of time. We observed this improvement in various test cases including not only the above-mentioned large amount of clients, but also at a much smaller scale starting with a single client.

Moreover, by artificially increasing the model sizes, we showed that with the use of *Redis*, *FedLess* is able to handle models of much larger size in a stable manner. We demonstrate this with the use of three different model sizes 26 MB, 52 MB, and 220 MB. In addition to that, we evaluated the system with the use of replicated parameter servers, aiming at increased performance for model downloads. Even though we could achieve reduced running times for the *read* operations, we identified that this comes at the cost of having worse *write* times.

Nonetheless, as a result of the improved overall performance, we were able to significantly reduce the training costs. As the framework should support and work with client functions that could be deployed on a diverse number of cloud providers, being able to achieve a cost reduction is an important improvement.

6 Conclusion and Future Work

Initially introduced by Chadha, Jindal, and Gerndt, *FedKeeper* is a novel solution in the Federated Learning (FL) context. Applying serverless computing principles, the authors address common issues in the FL setting such as efficient FL-clients' management. By making use of *Function-as-a-Service* (FaaS) infrastructure [75], *FedKeeper* shows promising results once evaluated in an image classification task with a varying number of clients. Its successor, named *FedLess*, solves open problems identified in the initial framework, by providing additional vital features like authentication, authorization, and differential privacy. In this work, we build on top of the foundations set in the development of these two consecutive frameworks for serverless FL.

Within the scope of this thesis, we worked on evaluating and improving the performance of *FedLess* by looking at known issues and bottlenecks of the system. The latest experiments performed with *FedLess* serve as an input and motivation for this work, showing the weaknesses of the system and providing directions for potential improvements.

Firstly, we approached the underlying parameter server of the system, identified by previous evaluations as the main problem in terms of the system's performance, due to its lacking ability to efficiently handle multiple simultaneous model uploads. We replaced the parameter server technology with *Redis*, an in-memory data object store, and evaluated the system's performance by comparing the new implementation with the existing, *MongoDB* parameter server solution in detail. Within the comparisons, we considered not only the execution duration of the participating clients in an FL training round, but also looked at and measured the aggregation process, occurring once an aggregator function is invoked to combine the latest model updates of the clients involved in the training process. With the use of *Redis*, we considered two different approaches for handling the serialization of the model weights. At first, we kept the serialization technique present in the existing *MongoDB* solution, which serializes the model weights in a *BSON* format prior to their persistence, and applied it to the new *Redis* solution. This, however, did provide the expected results as we were not able to achieve better performance results of the system. In fact, just to the opposite, by using the exact same weights serialization approach, the system showed an even worse running times as compared with the existing solution.

Then, inspired by the different *Redis* modules for various industry use cases, we applied *redisAI*, a *Redis* module specifically designed for applications in the ML/DL context. *RedisAI* maximizes computation throughput and reduces latency. The module allowed us to completely ignore any serialization as well as deserialization steps throughout the FL training session by storing and retrieving client results as tensors directly. With this, we were able to further reduce the execution running times of the system, achieve a better performance, and, thus, reduce the associated costs.

Throughout the conducted experiments, we scaled the number of clients involved in FL training to up to 300, proving the better performance of the system with the newly introduced parameter server technology not only in a small test setting, involving a few participating clients, but also in the large-scale scenario. Thus, we were able to outperform the *MongoDB* parameter server solution in any single test case that we performed by looking at the clients as well as the aggregator performances.

In addition to the increased number of clients, in order to examine the ability of the system to handle models of larger size, we increased the size of the model used throughout the FL training **8 times**, ending with a model of size 220 MB. Similar to previous test cases, the *Redis* parameter server solution showed to be superior to the existing implementation with *MongoDB* in terms of the clients' as well as aggregator running times by orders of magnitude.

Nonetheless, even though we could reach a much faster and more stable model upload performance and a better performance of the system in general, we experienced a slightly worse running times when it comes to the model download or read operations as compared with the *MongoDB* solution. This is, in fact, primarily caused by the single-threaded nature of the *Redis* technology. We experimented with several different configurations in line with the official *Redis* documentation, though without success. Thus, within the scope of the second part of this work, we aimed at increasing the read running times of the system by implementing replication parameter servers in *FedLess*.

In order to better evaluate the performance of the system under the consideration of multiple replication parameter servers, throughout the experiments conducted in the second part of this work, we used several different configurations with varying numbers of replication nodes involved. We compared the running times of the system considering three different settings, defined by the number of replication nodes involved, namely 2, 5, which is the maximum number of nodes allowed as well as 0 or in other words, without any replication nodes. Similar to the existing experiments, we considered different test scenarios with varying numbers of clients, starting with a single one and scaling up to 300. In addition to that, we used different model sizes with the largest one being 220 MB.

The observations indicated a close relationship between the number of replicas involved and the corresponding read and write operations. By increasing the number of replication nodes to 5, we were able to achieve the best running time for the corresponding model downloads. This, however, has the cost of decreasing the write operations of the system and, thus, a negative effect on the model uploads, increasing the time required for their execution. Based on our evaluations, we observed a trade-off between having a better, in terms of execution duration, model downloads, on the one side, and model uploads, on the other. Thus, we can conclude the following: the more replication nodes we involve, the better download performance we reach, however, the worse the model upload time becomes and vice-versa - less number of replication nodes leads to better model upload performance and, therefore, worse model downloads.

Even though we were not able to further decrease the model download running times of the system, replacing the parameter server with *Redis* allowed us to significantly decrease the execution duration of an FL training round on the client as well as on the aggregator side by

orders of magnitude, and, thus achieve a significant cost reduction. For the model size of 26 MB, we achieved **2.21** times better clients' running times and **2.97** times better performance for the aggregator, which, according to our measurements, allows up to **45.54%** cost reduction in *FedLess* with *Redis* as a parameter server. Within the experiments with the bigger model of size 52 MB, we reached **3.35** times better clients' running times and **4.6** better aggregator performance as compared with the existing, *MongoDB* solution, resulting in **52.05%** cost reduction on a monthly basis for the use of *FedLess*. Since one of the key and distinctive characteristic of *FedLess* is that it is cloud provider-agnostic and, thus, spposed to interact with any kind of *FaaS* function, the cost reduction can be seen as an important improvement.

Despite these performance and, therefore, cost-related advancements, we identified several weaknesses of the system that could not be addressed within the scope of this work. These could be potential subject to improvement in the future. Firstly, as discussed above, the use of replication did not lead to the desired outcomes of reduced read times of the system. In fact, looking only at the model download times, the existing solution with *MongoDB* is superior to *Redis*, providing faster round training duration on average. One could eventually investigate this problem further and apply a different replication approach such as the one with *enabled* cluster mode, in order to improve the running times of the clients participating in an FL training round.

Additionally, considering the aggregator's performance, we identified that the system is concerned with the actual aggregation of the latest model parameters in nearly **99.00%** of the time of the whole aggregation process. This is, in fact, independent of the underlying parameter server technology and is influenced by factors such as the predefined aggregation scheme, for example. Thus, replacing the existing aggregation scheme with a more efficient one could be considered in the future, in order to improve the aggregation performance of the framework.

In conclusion, throughout our work, we extended and evaluated a framework of serverless FL by integrating novel techniques designed for the DL/ML context. Based on the current interest in the FL field, the fast-growing serverless applications in many different contexts as well as the technological advancements we expect to see many interesting findings in this context in the future.

List of Figures

List of Tables

Bibliography

[1] M. Anderson. *Technology device ownership: 2015*. Available at: https://www.pewresearch.org/internet/2015/10/29/technology-device-ownership-2015/. Accessed on 13/07/2022.

[2] W. Y. B. Lim, N. C. Luong, D. T. Hoang, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, and C. Miao. "Federated Learning in Mobile Edge Networks: A Comprehensive Survey". In: *IEEE Communications Surveys Tutorials* 22.3 (2020), pp. 2031–2063. DOI: 10.1109/COMST.2020.2986024.

[3] E. Ahmed, A. Gani, M. Sookhak, S. H. A. Hamid, and F. Xia. "Application optimization in mobile cloud computing: Motivation, taxonomies, and open challenges". In: *Journal of Network and Computer Applications* 52 (2015), pp. 52–68. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2015.02.003. URL: https://www.sciencedirect.com/science/article/pii/S1084804515000417.

[4] G. Marinoni, H. van't Land, and T. Jensen. "The impact of COVID-19 on higher education around the world". In: (2020). URL: https://www.iau-aiu.net/IMG/pdf/iau_covid19_and_he_survey_report_final_may_2020.pdf.

[5] O. Zawacki-Richter. "The current state and impact of Covid-19 on digital higher education in Germany". In: *Human Behavior and Emerging Technologies* 3.1 (2021), pp. 218–226. DOI: https://doi.org/10.1002/hbe2.238. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/hbe2.238. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/hbe2.238.

[6] J. Poushter. *Smartphone ownership and internet usage continues to climb in emerging economies.* Available at: https://www.pewresearch.org/global/2016/02/22/smartphone-ownership-and-internet-usage-continues-to-climb-in-emerging-economies/. Accessed on 13/07/2022. 2016.

[7] *Mobile Fact Sheet*. Available at: https://www.pewresearch.org/internet/fact-sheet/mobile/. 2021.

[8] M. Chiang and T. Zhang. "Fog and IoT: An Overview of Research Opportunities". In: *IEEE Internet of Things Journal* 3.6 (2016), pp. 854–864. DOI: 10.1109/JIOT.2016.2584538.

[9] K. L. Lueth. *State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time*. Available https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/. 2020.

[10] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. https://arxiv.org/abs/1602.05629. 2017. arXiv: 1602.05629 [cs.LG].

[11] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage. *Federated Learning for Mobile Keyboard Prediction*. 2018. arXiv: 1811.03604 [cs.CL].

[12] Y. Jing, B. Guo, Z. Wang, V. O. K. Li, J. C. K. Lam, and Z. Yu. "CrowdTracker: Optimized Urban Moving Object Tracking Using Mobile Crowd Sensing". In: *IEEE Internet of Things Journal* 5.5 (2018), pp. 3452–3463. DOI: 10.1109/JIOT.2017.2762003.

[13] H.-J. Hong, C.-L. Fan, Y.-C. Lin, and C.-H. Hsu. "Optimizing Cloud-Based Video Crowdsensing". In: *IEEE Internet of Things Journal* 3.3 (2016), pp. 299–313. DOI: 10.1109/JIOT.2016.2519502.

[14] W. He, G. Yan, and L. Xu. "Developing Vehicular Data Cloud Services in the IoT Environment". In: *Industrial Informatics, IEEE Transactions on* 10 (May 2014), pp. 1587–1595. DOI: 10.1109/TII.2014.2299233.

[15] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan. "Adaptive Federated Learning in Resource Constrained Edge Computing Systems". In: *IEEE Journal on Selected Areas in Communications* 37.6 (2019), pp. 1205–1221. DOI: 10.1109/JSAC.2019.2904348.

[16] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer. "A Survey on Distributed Machine Learning". In: *ACM Comput. Surv.* 53.2 (Mar. 2020). ISSN: 0360-0300. DOI: 10.1145/3377454. URL: https://doi.org/10.1145/3377454.

[17] R. Kelly. *Internet of Things Data to Top 1.6 Zettabytes by 2022*. Available https://campustechnology.com/articles/2015/04/15/internet-of-things-data-to-top-1-6-zettabytes-by-2020.aspx. 2016.

[18] W. Chen, K. Bhardwaj, and R. Marculescu. *FedMax: Enabling a Highly-Efficient Federated Learning Framework*. https://arxiv.org/abs/2004.03657. 2020.

[19] 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*. HPDC '20. Stockholm, Sweden: Association for Computing Machinery, 2020, pp. 125–136. ISBN: 9781450370523. DOI: 10.1145/3369583.3392686. URL: https://doi.org/10.1145/3369583.3392686.

[20] K. He, X. Zhang, S. Ren, and J. Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[21] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. *End to End Learning for Self-Driving Cars*. 2016. arXiv: 1604.07316 [cs.CV].

[22] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. Engel, L. Fan, C. Fougner, T. Han, A. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu. *Deep Speech 2: End-to-End Speech Recognition in English and Mandarin*. 2015. arXiv: 1512.02595 [cs.CL].

[23] A. E. Khandani, A. J. Kim, and A. W. Lo. "Consumer credit-risk models via machine-learning algorithms". In: *Journal of Banking and Finance* 34.11 (2010), pp. 2767–2787. ISSN: 0378-4266. DOI: https://doi.org/10.1016/j.jbankfin.2010.06.001. URL: https://www.sciencedirect.com/science/article/pii/S0378426610002372.

[24] J. "Paparrizos, R. W. White, and E. Horvitz. ""Screening for Pancreatic Adenocarcinoma Using Signals From Web Search Logs: Feasibility Study and Results"". "en". In: *"J Oncol Pract"* 12.8 (June 2016), "737–744".

[25] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays. "Applied Federated Learning: Improving Google Keyboard Query Suggestions". In: (Dec. 2018).

[26] A. Hard, K. Rao, R. Mathews, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage. "Federated Learning for Mobile Keyboard Prediction". In: (Nov. 2018).

[27] S. SASI PRIYA, S. Rajarajeshwari, K. Sowmiya, and P. Vinesha. "Road Traffic Condition Monitoring using Deep Learning". In: *2020 International Conference on Inventive Computation Technologies (ICICT)*. 2020, pp. 330–335. DOI: 10.1109/ICICT48043.2020.9112408.

[28] L. Huang, A. L. Shea, H. Qian, A. Masurkar, H. Deng, and D. Liu. "Patient clustering improves efficiency of federated machine learning to predict mortality and hospital stay time using distributed electronic medical records". In: *Journal of Biomedical Informatics* 99 (2019), p. 103291. ISSN: 1532-0464. DOI: https://doi.org/10.1016/j.jbi.2019.103291. URL: https://www.sciencedirect.com/science/article/pii/S1532046419302102.

[29] A. Sawsan, S. Abdulrahman, A. Mourad, and M. El Barachi. "An Infrastructure-Assisted Crowdsensing Approach for On-Demand Traffic Condition Estimation". In: *IEEE Access* 7 (Nov. 2019), p. 163323. DOI: 10.1109/ACCESS.2019.2953002.

[30] A. Mourad, H. Tout, O. A. Wahab, H. Otrok, and T. Dbouk. "Ad Hoc Vehicular Fog Enabling Cooperative Low-Latency Intrusion Detection". In: *IEEE Internet of Things Journal* 8.2 (2021), pp. 829–843. DOI: 10.1109/JIOT.2020.3008488.

[31] V. Mani, C. Kavitha, S. S. Band, A. Mosavi, P. Hollins, and S. Palanisamy. "A Recommendation System Based on AI for Storing Block Data in the Electronic Health Repository". In: *Frontiers in Public Health* 9 (2022). ISSN: 2296-2565. DOI: 10.3389/fpubh.2021.831404. URL: https://www.frontiersin.org/article/10.3389/fpubh.2021.831404.

[32] B. Liu, L. Wang, and M. Liu. "Lifelong Federated Reinforcement Learning: A Learning Architecture for Navigation in Cloud Robotic Systems". In: *IEEE Robotics and Automation Letters* 4.4 (Oct. 2019), pp. 4555–4562. ISSN: 2377-3774. DOI: 10.1109/lra.2019.2931179. URL: http://dx.doi.org/10.1109/LRA.2019.2931179.

[33] S. Lu, Y. Yao, and W. Shi. "Collaborative Learning on the Edges: A Case Study on Connected Vehicles". In: *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotedge19/presentation/lu.

[34] S. Russel and P. Norving. *Artificial Intelligence: A Modern Appriach*. Prentice Hall Press, One Lake Street Upper Saddle River, NJ, United States, 2010. ISBN: 978-0-13-604259-4.

[35] P. Flach. "Machine Learning: The art and science of algorithms that make sense of data". English. In: (Sept. 2012).

[36] D. E. Sorkin. "Technical and Legal Approaches to Unsolicited Electronic Mail". In: *University of San Francisco Law Review* 35 (2001). Available at SSRN: https://ssrn.com/abstract=265768, pp. 325–384.

[37] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521.7553 (May 2015), pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[38] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 0123748569.

[39] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. USA: Cambridge University Press, 2014. ISBN: 1107057132.

[40] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[41] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. *Natural Language Processing (almost) from Scratch*. 2011. arXiv: 1103.0398 [cs.LG].

[42] A. Bordes, S. Chopra, and J. Weston. *Question Answering with Subgraph Embeddings*. 2014. arXiv: 1406.3676 [cs.CL].

[43] S. Jean, K. Cho, R. Memisevic, and Y. Bengio. *On Using Very Large Target Vocabulary for Neural Machine Translation*. 2015. arXiv: 1412.2007 [cs.CL].

[44] I. Sutskever, O. Vinyals, and Q. V. Le. *Sequence to Sequence Learning with Neural Networks*. 2014. arXiv: 1409.3215 [cs.CL].

[45] S. Abdulrahman, H. Tout, H. Ould-Slimane, A. Mourad, C. Talhi, and M. Guizani. "A Survey on Federated Learning: The Journey From Centralized to Distributed On-Site Learning and Beyond". In: *IEEE Internet of Things Journal* 8.7 (2021), pp. 5476–5497. DOI: 10.1109/JIOT.2020.3030072.

[46] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. "Practical Secure Aggregation for Privacy-Preserving Machine Learning". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1175–1191. ISBN: 9781450349468. DOI: 10.1145/3133956.3133982. URL: https://doi.org/10.1145/3133956.3133982.

[47] E. Jeong, S. Oh, H. Kim, J. Park, M. Bennis, and S.-L. Kim. *Communication-Efficient On-Device Machine Learning: Federated Distillation and Augmentation under Non-IID Private Data*. 2018. arXiv: 1811.11479 [cs.LG].

[48] D. Swinhoe. "The 15 Biggest Data Breaches of the 21st Century." In: *IEEE Internet of Things Journal* 8.7 (2020). Available:https://www.csoonline.com/article/2130877/the-biggest-data-breachesof-the-21st-century.html, pp. 5476–5497.

[49] J. Xu, B. S. Glicksberg, C. Su, P. Walker, J. Bian, and F. Wang. "Federated Learning for Healthcare Informatics". In: *Journal of Healthcare Informatics Research* 5.1 (Mar. 2021), pp. 1–19. ISSN: 2509-498X. DOI: 10.1007/s41666-020-00082-4. URL: https://doi.org/10.1007/s41666-020-00082-4.

[50] "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)." In: (2016). https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[51] "California Consumer Privacy Act (CCPA)." In: (2018). Available at: https://oag.ca.gov/privacy/ccpa. Accessed on 13/07/2022.

[52] "Cybersecurity Law of the People's Republic of China." In: (2017). Available at: http://www.lawinfochina.com/display.aspx?id=22826&lib=law. Accessed on 13/07/2022.

[53] C. Tankard. "What the GDPR means for businesses". In: *Network Security* 2016.6 (2016), pp. 5–8. ISSN: 1353-4858. DOI: https://doi.org/10.1016/S1353-4858(16)30056-3. URL: https://www.sciencedirect.com/science/article/pii/S1353485816300563.

[54] B. Custers, A. M. Sears, F. Dechesne, I. Georgieva, T. Tani, and S. van der Hof. *EU Personal Data Protection in Policy and Practice*. T.M.C. Asser Press, The Hague, 2018, pp. XIX, 249. ISBN: 978-94-6265-282-8. DOI: https://doi.org/10.1007/978-94-6265-282-8.

[55] B. M. Gaff, H. E. Sussman, and J. Geetter. "Privacy and Big Data". In: *Computer* 47.6 (2014), pp. 7–9. DOI: 10.1109/MC.2014.161.

[56] J. K. O'Herrin, N. Fost, and K. A. Kudsk. "Health Insurance Portability Accountability Act (HIPAA) regulations: effect on medical record research". eng. In: *Annals of surgery* 239.6 (June 2004). 00000658-200406000-00004[PII], pp. 772–778. ISSN: 0003-4932. DOI: 10.1097/01.sla.0000128307.98274.dc. URL: https://doi.org/10.1097/01.sla.0000128307.98274.dc.

[57] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. *Federated Learning: Strategies for Improving Communication Efficiency*. 2017. arXiv: 1610.05492 [cs.LG].

[58] M. T. Beck, S. Feld, C. Linnhoff-Popien, and U. Pützschler. "Mobile Edge Computing". In: *Informatik-Spektrum* 39.2 (Apr. 2016), pp. 108–114. ISSN: 1432-122X. DOI: 10.1007/s00287-016-0957-6. URL: https://doi.org/10.1007/s00287-016-0957-6.

[59] K. Zhang, Y. Mao, S. Leng, Q. Zhao, L. Li, X. Peng, L. Pan, S. Maharjan, and Y. Zhang. "Energy-Efficient Offloading for Mobile Edge Computing in 5G Heterogeneous Networks". In: *IEEE Access* 4 (2016), pp. 5896–5907. DOI: 10.1109/ACCESS.2016.2597169.

[60] A. Ahmed and E. Ahmed. "A survey on mobile edge computing". In: *2016 10th International Conference on Intelligent Systems and Control (ISCO)*. 2016, pp. 1–8. DOI: 10.1109/ISCO.2016.7727082.

[61] P. Mach and Z. Becvar. "Mobile Edge Computing: A Survey on Architecture and Computation Offloading". In: *IEEE Communications Surveys Tutorials* 19.3 (2017), pp. 1628–1656. DOI: 10.1109/COMST.2017.2682318.

[62] T. Kuflik, J. Kay, and B. Kummerfeld. "Challenges and Solutions of Ubiquitous User Modeling". In: *Cognitive Technologies* (Apr. 2012). DOI: 10.1007/978-3-642-27663-7_2.

[63] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks". In: *Computer Vision – ECCV 2016*. Ed. by B. Leibe, J. Matas, N. Sebe, and M. Welling. Cham: Springer International Publishing, 2016, pp. 525–542. ISBN: 978-3-319-46493-0.

[64] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith. "Federated Learning: Challenges, Methods, and Future Directions". In: *IEEE Signal Processing Magazine* 37.3 (May 2020), pp. 50–60. ISSN: 1558-0792. DOI: 10.1109/msp.2020.2975749. URL: http://dx.doi.org/10.1109/MSP.2020.2975749.

[65] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief. "A Survey on Mobile Edge Computing: The Communication Perspective". In: *IEEE Communications Surveys Tutorials* 19.4 (2017), pp. 2322–2358. DOI: 10.1109/COMST.2017.2745201.

[66] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

[67] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, H. B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander. *Towards Federated Learning at Scale: System Design*. 2019. arXiv: 1902.01046 [cs.LG].

[68] J. Park, S. Samarakoon, M. Bennis, and M. Debbah. *Wireless Network Intelligence at the Edge*. 2019. arXiv: 1812.02858 [cs.IT].

[69] C. Chen, H. Xu, W. Wang, B. Li, B. Li, L. Chen, and G. Zhang. "Communication-Efficient Federated Learning with Adaptive Parameter Freezing". In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 2021, pp. 1–11. DOI: 10.1109/ICDCS51616.2021.00010.

[70] E. Diao, J. Ding, and V. Tarokh. "HeteroFL: Computation and Communication Efficient Federated Learning for Heterogeneous Clients". In: (Oct. 2020).

[71] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. "Deep Learning with Differential Privacy". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Oct. 2016). DOI: 10.1145/2976749.2978318. URL: http://dx.doi.org/10.1145/2976749.2978318.

[72] A. Pantelopoulos and N. G. Bourbakis. "A Survey on Wearable Sensor-Based Systems for Health Monitoring and Prognosis". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 40.1 (2010), pp. 1–12. DOI: 10.1109/TSMCC.2009.2032660.

[73] S. Samarakoon, M. Bennis, W. Saad, and M. Debbah. "Federated Learning for Ultra-Reliable Low-Latency V2V Communications". In: *2018 IEEE Global Communications Conference (GLOBECOM)* (Dec. 2018). DOI: 10.1109/glocom.2018.8647927. URL: http://dx.doi.org/10.1109/GLOCOM.2018.8647927.

[74] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, H. Eichner, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konečný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, M. Raykova, H. Qi, D. Ramage, R. Raskar, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao. *Advances and Open Problems in Federated Learning*. 2021. arXiv: 1912.04977 [cs.LG].

[75] 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.

[76] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang. "Towards Demystifying Serverless Machine Learning Training". In: *Proceedings of the 2021 International Conference on Management of Data* (2021).

[77] S. Allen, L. Browning, L. Calcote, A. Chaudhry, D. Davis, L. Fourie, A. Gulli, Y. Haviv, D. Krook, O. Nissan-Messing, C. Munns, K. Owens, M. Peek, C. Zhang, and C.A. "CNCF WG-Serverless Whitepaper v1.0". In: *CNCF, Tech. Rep.* (2018). [Online]. Available: https://github.com/cncf/wg-serverless/blob/master/ whitepapers/serverless-overview/cncf serverless whitepaper v1.0.pdf.

[78] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. *Serverless Computing: Current Trends and Open Problems*. 2017. arXiv: 1706.03178 [cs.DC].

[79] J. Jiang, B. Cui, C. Zhang, and L. Yu. "Heterogeneity-Aware Distributed Parameter Servers". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 463–478. ISBN: 9781450341974. DOI: 10.1145/3035918.3035933. URL: https://doi.org/10.1145/3035918.3035933.

[80] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage. *Federated Learning for Mobile Keyboard Prediction*. 2019. arXiv: 1811.03604 [cs.CL].

[81] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays. *Applied Federated Learning: Improving Google Keyboard Query Suggestions*. 2018. arXiv: 1812.02903 [cs.LG].

[82] M. Hao, H. Li, X. Luo, G. Xu, H. Yang, and S. Liu. "Efficient and Privacy-Enhanced Federated Learning for Industrial Artificial Intelligence". In: *IEEE Transactions on Industrial Informatics* 16.10 (2020), pp. 6532–6542. DOI: 10.1109/TII.2019.2945367.

[83] D. Leroy, A. Coucke, T. Lavril, T. Gisselbrecht, and J. Dureau. *Federated Learning for Keyword Spotting*. 2019. arXiv: 1810.05512 [eess.AS].

[84] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik. *Federated Optimization: Distributed Machine Learning for On-Device Intelligence*. 2016. arXiv: 1610.02527 [cs.LG].

[85] S. Caldas, J. Konečny, H. B. McMahan, and A. Talwalkar. *Expanding the Reach of Federated Learning by Reducing Client Resource Requirements*. 2019. arXiv: 1812.07210 [cs.LG].

[86] Y. Liu, S. Garg, J. Nie, Y. Zhang, Z. Xiong, J. Kang, and M. S. Hossain. "Deep Anomaly Detection for Time-Series Data in Industrial IoT: A Communication-Efficient On-Device Federated Learning Approach". In: *IEEE Internet of Things Journal* 8.8 (2021), pp. 6348–6358. DOI: 10.1109/JIOT.2020.3011726.

[87] M. Ekmefjord, A. Ait-Mlouk, S. Alawadi, M. Åkesson, P. Singh, O. Spjuth, S. Toor, and A. Hellander. *Scalable federated machine learning with FEDn*. 2021. DOI: 10.48550/ARXIV.2103.00148. URL: https://arxiv.org/abs/2103.00148.

[88] P. Courtiol, C. Maussion, M. Moarii, E. Pronier, S. Pilcer, M. Sefta, P. Manceron, S. Toldo, M. Zaslavskiy, N. Le Stang, N. Girard, O. Elemento, A. G. Nicholson, J.-Y. Blay, F. Galateau-Sallé, G. Wainrib, and T. Clozel. "Deep learning-based classification of mesothelioma improves prediction of patient outcome". In: *Nature Medicine* 25.10 (Oct. 2019), pp. 1519–1525. ISSN: 1546-170X. DOI: 10.1038/s41591-019-0583-3. URL: https://doi.org/10.1038/s41591-019-0583-3.

[89] D. Verma, S. Julier, and G. Cirincione. "Federated AI for building AI Solutions across Multiple Agencies". In: (Sept. 2018).

[90] V. Patel, S. Kanani, T. Pathak, P. Patel, M. I. Ali, and J. Breslin. "An Intelligent Doorbell Design Using Federated Deep Learning". In: *8th ACM IKDD CODS and 26th COMAD*. CODS COMAD 2021. Bangalore, India: Association for Computing Machinery, 2021, pp. 380–384. ISBN: 9781450388177. DOI: 10.1145/3430984.3430988. URL: https://doi.org/10.1145/3430984.3430988.

[91] S. Doomra, N. Kohli, and S. Athavale. "Turn Signal Prediction: A Federated Learning Case Study". In: (Dec. 2020).

[92] S. Ramaswamy, R. Mathews, K. Rao, and F. Beaufays. *Federated Learning for Emoji Prediction in a Mobile Keyboard*. 2019. arXiv: 1906.04329 [cs.CL].

[93] T. S. Brisimi, R. Chen, T. Mela, A. Olshevsky, I. C. Paschalidis, and W. Shi. "Federated learning of predictive models from federated Electronic Health Records". In: *International Journal of Medical Informatics* 112 (2018), pp. 59–67. ISSN: 1386-5056. DOI: https://doi.org/10.1016/j.ijmedinf.2018.01.007. URL: https://www.sciencedirect.com/science/article/pii/S138650561830008X.

[94] S. Silva, B. A. Gutman, E. Romero, P. M. Thompson, A. Altmann, and M. Lorenzi. "Federated Learning in Distributed Medical Databases: Meta-Analysis of Large-Scale Subcortical Brain Data". In: *2019 IEEE 16th International Symposium on Biomedical Imaging (ISBI 2019)* (Apr. 2019). DOI: 10.1109/isbi.2019.8759317. URL: http://dx.doi.org/10.1109/ISBI.2019.8759317.

[95] L. Huang, Y. Yin, Z. Fu, S. Zhang, H. Deng, and D. Liu. *LoAdaBoost: loss-based AdaBoost federated machine learning with reduced computational complexity on IID and non-IID intensive care data*. 2018. arXiv: 1811.12629 [cs.LG].

[96] J. Xu, B. S. Glicksberg, C. Su, P. Walker, J. Bian, and F. Wang. "Federated Learning for Healthcare Informatics". In: *Journal of Healthcare Informatics Research* 5.1 (Mar. 2021), pp. 1–19. ISSN: 2509-498X. DOI: 10.1007/s41666-020-00082-4. URL: https://doi.org/10.1007/s41666-020-00082-4.

[97] N. Rieke, J. Hancox, W. Li, F. Milletarì, H. R. Roth, S. Albarqouni, S. Bakas, M. N. Galtier, B. A. Landman, K. Maier-Hein, S. Ourselin, M. Sheller, R. M. Summers, A. Trask, D. Xu, M. Baust, and M. J. Cardoso. "The future of digital health with federated learning". In: *npj Digital Medicine* 3.1 (Sept. 2020), p. 119. ISSN: 2398-6352. DOI: 10.1038/s41746-020-00323-1. URL: https://doi.org/10.1038/s41746-020-00323-1.

[98] H. R. Roth, K. Chang, P. Singh, N. Neumark, W. Li, V. Gupta, S. Gupta, L. Qu, A. Ihsani, B. C. Bizzo, Y. Wen, V. Buch, M. Shah, F. Kitamura, M. Mendonça, V. Lavor, A. Harouni, C. Compas, J. Tetreault, P. Dogra, Y. Cheng, S. Erdal, R. White, B. Hashemian, T. Schultz, M. Zhang, A. McCarthy, B. M. Yun, E. Sharaf, K. V. Hoebel, J. B. Patel, B. Chen, S. Ko, E. Leibovitz, E. D. Pisano, L. Coombs, D. Xu, K. J. Dreyer, I. Dayan, R. C. Naidu, M. Flores, D. Rubin, and J. Kalpathy-Cramer. "Federated Learning for Breast Density Classification: A Real-World Implementation". In: *Lecture Notes in Computer*

Science (2020), pp. 181–191. ISSN: 1611-3349. DOI: 10.1007/978-3-030-60548-3_18. URL: http://dx.doi.org/10.1007/978-3-030-60548-3_18.

[99] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz. "A Public Domain Dataset for Human Activity Recognition using Smartphones". In: *ESANN*. 2013.

[100] Q. Yang, Y. Liu, T. Chen, and Y. Tong. *Federated Machine Learning: Concept and Applications*. 2019. arXiv: 1902.04885 [cs.AI].

[101] M. van Dijk, N. V. Nguyen, T. N. Nguyen, L. M. Nguyen, Q. Tran-Dinh, and P. H. Nguyen. *Asynchronous Federated Learning with Reduced Number of Rounds and with Differential Privacy from Less Aggregated Gaussian Noise*. 2020. DOI: 10.48550/ARXIV.2007.09208. URL: https://arxiv.org/abs/2007.09208.

[102] S. Ioffe and C. Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. DOI: 10.48550/ARXIV.1502.03167. URL: https://arxiv.org/abs/1502.03167.

[103] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz. *Revisiting Distributed Synchronous SGD*. 2016. DOI: 10.48550/ARXIV.1604.00981. URL: https://arxiv.org/abs/1604.00981.

[104] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith. *Federated Optimization in Heterogeneous Networks*. 2018. DOI: 10.48550/ARXIV.1812.06127. URL: https://arxiv.org/abs/1812.06127.

[105] M. van Dijk, N. V. Nguyen, T. N. Nguyen, L. M. Nguyen, Q. Tran-Dinh, and P. H. Nguyen. *Asynchronous Federated Learning with Reduced Number of Rounds and with Differential Privacy from Less Aggregated Gaussian Noise*. 2020. DOI: 10.48550/ARXIV.2007.09208. URL: https://arxiv.org/abs/2007.09208.

[106] J. Langford, A. Smola, and M. Zinkevich. *Slow Learners are Fast*. 2009. DOI: 10.48550/ARXIV.0911.0491. URL: https://arxiv.org/abs/0911.0491.

[107] X. Lian, Y. Huang, Y. Li, and J. Liu. *Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization*. 2015. DOI: 10.48550/ARXIV.1506.08272. URL: https://arxiv.org/abs/1506.08272.

[108] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu. "Asynchronous Stochastic Gradient Descent with Delay Compensation". In: (2016). DOI: 10.48550/ARXIV.1609.08326. URL: https://arxiv.org/abs/1609.08326.

[109] S. Shi, Q. Wang, K. Zhao, Z. Tang, Y. Wang, X. Huang, and X. Chu. *A Distributed Synchronous SGD Algorithm with Global Top-k Sparsification for Low Bandwidth Networks*. 2019. DOI: 10.48550/ARXIV.1901.04359. URL: https://arxiv.org/abs/1901.04359.

[110] M. van Dijk, N. V. Nguyen, T. N. Nguyen, L. M. Nguyen, Q. Tran-Dinh, and P. H. Nguyen. *Asynchronous Federated Learning with Reduced Number of Rounds and with Differential Privacy from Less Aggregated Gaussian Noise*. 2020. DOI: 10.48550/ARXIV.2007.09208. URL: https://arxiv.org/abs/2007.09208.

[111] Q. Yang, Y. Liu, T. Chen, and Y. Tong. "Federated Machine Learning: Concept and Applications". In: (2019). DOI: 10.48550/ARXIV.1902.04885. URL: https://arxiv.org/abs/1902.04885.

[112] A. Gascón, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans. "Secure Linear Regression on Vertically Partitioned Datasets". In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 892.

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

[114] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Foundations and Trends in Machine Learning* 3 (Jan. 2011), pp. 1–122. DOI: 10.1561/2200000016.

[115] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. "Scaling Distributed Machine Learning with the Parameter Server". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 583–598. ISBN: 9781931971164.

[116] K. Hsieh, A. Phanishayee, O. Mutlu, and P. B. Gibbons. *The Non-IID Data Quagmire of Decentralized Machine Learning*. 2019. arXiv: 1910.00189 [cs.LG].

[117] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu. "Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 629–647. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/hsieh.

[118] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. "An In-Depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 363–374. ISBN: 9781450320566. DOI: 10.1145/2486001.2486006. URL: https://doi.org/10.1145/2486001.2486006.

[119] C. H. (van Berkel. "Multi-Core for Mobile Phones". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '09. Nice, France: European Design and Automation Association, 2009, pp. 1260–1265. ISBN: 9783981080155.

[120] 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*. 2021. arXiv: 2010.05958 [cs.DC].

[121] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, H. B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander. *Towards Federated Learning at Scale: System Design*. 2019. arXiv: 1902.01046 [cs.LG].

[122] Z. Chai, H. Fayyaz, Z. Fayyaz, A. Anwar, Y. Zhou, N. Baracaldo, H. Ludwig, and Y. Cheng. "Towards Taming the Resource and Data Heterogeneity in Federated Learning". In: *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. Santa Clara, CA: USENIX Association, May 2019, pp. 19–21. ISBN: 978-1-939133-00-7. URL: https://www.usenix.org/conference/opml19/presentation%20/chai.

[123] 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].

[124] A. Reisizadeh, I. Tziotis, H. Hassani, A. Mokhtari, and R. Pedarsani. *Straggler-Resilient Federated Learning: Leveraging the Interplay Between Statistical Accuracy and System Heterogeneity*. 2020. arXiv: 2012.14453 [cs.LG].

[125] V. Smith, C.-K. Chiang, M. Sanjabi, and A. Talwalkar. *Federated Multi-Task Learning*. 2018. arXiv: 1705.10467 [cs.LG].

[126] T. Wang, J.-Y. Zhu, A. Torralba, and A. A. Efros. *Dataset Distillation*. 2020. arXiv: 1811.10959 [cs.LG].

[127] J. C. Duchi, M. I. Jordan, and M. J. Wainwright. *Privacy Aware Learning*. 2012. DOI: 10.48550/ARXIV.1210.2085. URL: https://arxiv.org/abs/1210.2085.

[128] N. Carlini, C. Liu, Ú. Erlingsson, J. Kos, and D. Song. *The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks*. 2018. DOI: 10.48550/ARXIV.1802.08232. URL: https://arxiv.org/abs/1802.08232.

[129] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang. *Learning Differentially Private Recurrent Language Models*. 2017. DOI: 10.48550/ARXIV.1710.06963. URL: https://arxiv.org/abs/1710.06963.

[130] X. Li, K. Huang, W. Yang, S. Wang, and Z. Zhang. *On the Convergence of FedAvg on Non-IID Data*. 2020. arXiv: 1907.02189 [stat.ML].

[131] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek. *Robust and Communication-Efficient Federated Learning from Non-IID Data*. 2019. arXiv: 1903.02891 [cs.LG].

[132] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra. *Federated Learning with Non-IID Data*. 2018. arXiv: 1806.00582 [cs.LG].

[133] A. Reisizadeh, A. Mokhtari, H. Hassani, A. Jadbabaie, and R. Pedarsani. *FedPAQ: A Communication-Efficient Federated Learning Method with Periodic Averaging and Quantization*. 2020. arXiv: 1909.13014 [cs.LG].

[134] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. S. Rabkin, I. Stoica, and M. A. Zaharia. "Above the Clouds: A Berkeley View of Cloud Computing". In: *Science* 53 (2009), pp. 07–013.

[135] *AWS re:Invenr 2014: Getting started with AWS Lmabda*. Available: https://www.youtube.com/watch?v=UFj27laTWQA. 2014.

[136] W. MJ. "Hype Cycle for Emerging Technologies". In: Gartner, 2017. URL: https://www.gartner.com/en/documents/3768572.

[137] *Emerging Technology Analysis: Serverless Computing and Function Platform as a Service*. 2016.

[138] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. *Serverless Computing: One Step Forward, Two Steps Back*. 2018. DOI: 10.48550/ARXIV.1812.03651. URL: https://arxiv.org/abs/1812.03651.

[139] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski. "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research". In: *CoRR* abs/1708.08028 (2017). arXiv: 1708.08028. URL: http://arxiv.org/abs/1708.08028.

[140] W. Ye, A. Khan, and E. Kendall. "Distributed network file storage for a serverless (P2P) network". In: *The 11th IEEE International Conference on Networks, 2003. ICON2003.* 2003, pp. 343–347. DOI: 10.1109/ICON.2003.1266214.

[141] D. Bryan, B. Lowekamp, and C. Jennings. "SOSIMPLE: A Serverless, Standards-based, P2P SIP Communication System". In: *First International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications (AAA-IDEA'05)*. 2005, pp. 42–49. DOI: 10.1109/AAA-IDEA.2005.15.

[142] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. 2019. arXiv: 1902.03383 [cs.OS].

[143] M. Roberts. "Serverless Architecture". In: Available: https://martinfowler.com/articles/serverless.html. 2016, pp. 2658–2659.

[144] *Amazon. [n.d.]. AWS Cognito.* Available: https://aws.amazon.com/cognito/.

[145] *Firebase*. Available: https://firebase.google.com/.

[146] *Amazon. [n.d.]. AWS DynamoDB.* Available: https://aws.amazon.com/dynamodb/.

[147] M. Yan, P. Castro, P. Cheng, and V. Isahagian. "Building a Chatbot with Serverless Computing". In: Dec. 2016, pp. 1–4. DOI: 10.1145/3007203.3007217.

[148] V. Ishakian, V. Muthusamy, and A. Slominski. "Serving deep learning models in a serverless platform". In: *CoRR* abs/1710.08460 (2017). arXiv: 1710.08460. URL: http://arxiv.org/abs/1710.08460.

[149] P. McDonald. *Introducing Google App Engine*. https://googleappengine.blogspot.com/2008/04/introducing-google-app-engine-our-new.html. 2008.

[150] *Amazon Web Services. AWS Elastic Beanstalk - PaaS Application Managemen*. https://aws.amazon.com/de/elasticbeanstalk/.

[151] *Software Engineering in der Industriellen Praxis (SEIP) from msg group for Technische Universität München (TUM), Computer Science faculty, chair Software Engineering for Business Information Systems (sebis)*. Available https://seip.msg.group/repo/seip/public.

[152] H. Lee, K. Satyam, and G. Fox. "Evaluation of Production Serverless Computing Environments". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 442–450. DOI: 10.1109/CLOUD.2018.00062.

[153] *CNCF Serverless WG*. Cncf wg-serverless whitepaper v1. 0. https://gw.alipayobjects.com/os/basement_prod/24ec4498-71d4-4a60-b785-fa530456c65b.pdf. 2018.

[154] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. "Serverless Programming (Function as a Service)". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 2658–2659. DOI: 10.1109/ICDCS.2017.305.

[155] R. Chard, K. Chard, J. Alt, D. Y. Parkinson, S. Tuecke, and I. Foster. "Ripple: Home Automation for Research Data Management". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2017, pp. 389–394. DOI: 10.1109/ICDCSW.2017.30.

[156] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and P. Suter. "Cloud-Native, Event-Based Programming for Mobile Applications". In: *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 2016, pp. 287–288. DOI: 10.1109/MobileSoft.2016.063.

[157] N. Bila, P. Dettori, A. Kanso, Y. Watanabe, and A. Youssef. "Leveraging the Serverless Architecture for Securing Linux Containers". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2017, pp. 401–404. DOI: 10.1109/ICDCSW.2017.66.

[158] P. Castro, V. Isahagian, V. Muthusamy, and A. Slominski. "The rise of serverless computing". In: *Communications of the ACM* 62 (Nov. 2019), pp. 44–54. DOI: 10.1145/3368454.

[159] *The Netflix Cosmos Platform*. https://netflixtechblog.com/the-netflix-cosmos-platform-35c14d9351ad.

[160] M. Kiener, M. Chadha, and M. Gerndt. "Towards Demystifying Intra-Function Parallelism in Serverless Computing". In: *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*. WoSC '21. Virtual Event, Canada, 2021, pp. 42–49. DOI: 10.1145/3493651.3493672. URL: https://doi.org/10.1145/3493651.3493672.

[161] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. "Cirrus: A Serverless Framework for End-to-End ML Workflows". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 13–24. ISBN: 9781450369732. DOI: 10.1145/3357223.3362711. URL: https://doi.org/10.1145/3357223.3362711.

[162] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. *Serverless Computing: One Step Forward, Two Steps Back*. 2018. arXiv: 1812.03651 [cs.DC].

[163] G. Safaryan, A. Jindal, M. Chadha, and M. Gerndt. "SLAM: SLO-Aware Memory Optimization for Serverless Applications". In: *arXiv preprint arXiv:2207.06183* (2022).

[164] 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.

[165] 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.

[166] A. Jindal, M. Chadha, S. Benedict, and M. Gerndt. "Estimating the Capacities of Function-as-a-Service Functions". In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. UCC '21 Companion. Leicester, United Kingdom: Association for Computing Machinery, 2021. ISBN: 978-1-4503-9163-4/21/12. DOI: 10.1145/3492323.3495628. URL: https://doi.org/10.1145/3492323.3495628.

[167] M. Chadha, A. Jindal, and M. Gerndt. "Architecture-Specific Performance Optimization of Compute-Intensive FaaS Functions". In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 2021, pp. 478–483. DOI: 10.1109/CLOUD53861.2021.00062.

[168] 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. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2966. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2966.

[169] A. Shehabi, S. J. Smith, E. Masanet, and J. Koomey. "Data center growth in the United States: decoupling the demand for services from electricity use". In: *Environmental Research Letters* 13.12 (Dec. 2018), p. 124030. DOI: 10.1088/1748-9326/aaec9c. URL: https://doi.org/10.1088/1748-9326/aaec9c.

[170] S. Newman. *Building Microservices*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1491950358.

[171] *Designing for the Serverless Age*. https://gojko.net/2017/10/05/serverless-design-gotocph.html. 2017.

[172] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. "Peeking Behind the Curtains of Serverless Platforms". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 133–146. ISBN: ISBN 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/wang-liang.

[173] *Amazon AWS Lambda.* Available at: https://aws.amazon.com/lambda/. Accessed on 13/07/2022.

[174] *Microsoft Azure Functions.* Available at: https://azure.microsoft.com/en-us/services/functions/. Accessed on 13/07/2022.

[175] *IBM Functions.* Available at: https://azure.microsoft.com/en-us/services/functions/. Accessed on 13/07/2022.

[176] *Google Cloud Functions.* Available at: https://cloud.google.com/functions/. Accessed on 13/07/2022.

[177] *OpenFaas.* Available at: https://www.openfaas.com/. Access on 13/07/2022.

[178] *Apache OpenWhisk.* Available at: https://openwhisk.apache.org/. Accessed on 13/07/2022.

[179] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley. *numpywren: serverless linear algebra*. 2018. arXiv: 1810.09679 [cs.DC].

[180] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. *Occupy the Cloud: Distributed Computing for the 99%*. 2017. arXiv: 1702.04024 [cs.DC].

[181] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. "SAND: Towards High-Performance Serverless Computing". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 923–935. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/akkus.

[182] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "Serverless Computation with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, June 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[183] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/klimovic.

[184] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar. "Towards a Serverless Platform for Edge AI". In: *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotedge19/presentation/rausch.

[185] S. Mallya, "Seamlessly scale predictions with aws lambda and mxnet," 2017, [Online]. Available: https://aws.amazon.com/blogs/compute/seamlessly-scale-predictions-with-aws-lambda-and-mxnet/.

[186] Google Cloud, "Building a Serverless ML Model." [Online]. Available: https://cloud.google.com/solutions/building-a-serverless-ml-model.

[187] M. J. Sheller, G. A. Reina, B. Edwards, J. Martin, and S. Bakas. *Multi-Institutional Deep Learning Modeling Without Sharing Patient Data: A Feasibility Study on Brain Tumor Segmentation*. 2018. DOI: 10.48550/ARXIV.1810.04304. URL: https://arxiv.org/abs/1810.04304.

[188] Y. Liu, Z. Ma, X. Liu, Z. Wang, S. Ma, and K. Ren. *Revocable Federated Learning: A Benchmark of Federated Forest*. 2019. DOI: 10.48550/ARXIV.1911.03242. URL: https://arxiv.org/abs/1911.03242.

[189] *TensorFlow Federated: Machine Learning on Decentralized Data*. https://www.tensorflow.org/federated. Mountain View, CA, USA, 2019.

[190] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach. *A generic framework for privacy preserving deep learning*. 2018. DOI: 10.48550/ARXIV.1811.04017. URL: https://arxiv.org/abs/1811.04017.

[191] Y. LeCun, C. Cortes, and C. Brurges. *MNIST Handwritten Digit*. Available : http://yann.lecun.com/exdb/mnist. NJ, USA, 2010.

[192] 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].

[193] WeBank. (2018) Fate: An industrial grade federated learning framework. [Online]. Available: Available:https://fate.fedai.org/.

[194] C. He, S. Li, J. So, X. Zeng, M. Zhang, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, X. Zhu, J. Wang, L. Shen, P. Zhao, Y. Kang, Y. Liu, R. Raskar, Q. Yang, M. Annavaram, and S. Avestimehr. *FedML: A Research Library and Benchmark for Federated Machine Learning*. 2020. DOI: 10.48550/ARXIV.2007.13518. URL: https://arxiv.org/abs/2007.13518.

[195] G. McGrath and P. R. Brenner. "Serverless Computing: Design, Implementation, and Performance". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2017, pp. 405–410. DOI: 10.1109/ICDCSW.2017.36.

[196] J. Carreira. "A Case for Serverless Machine Learning". In: 2018.

[197] L. Feng, P. Kudva, D. Da Silva, and J. Hu. "Exploring Serverless Computing for Neural Network Training". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 334–341. DOI: 10.1109/CLOUD.2018.00049.

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

[199] A. Bhattacharjee, Y. Barve, S. Khare, S. Bao, A. Gokhale, and T. Damiano. "Stratum: A Serverless Framework for the Lifecycle Management of Machine Learning-based Data Analytics Tasks". In: *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. Santa Clara, CA: USENIX Association, May 2019, pp. 59–61. ISBN: 978-1-939133-00-7. URL: https://www.usenix.org/conference/opml19/presentation/bhattacharjee.

[200] V. Gupta, S. Kadhe, T. Courtade, M. W. Mahoney, and K. Ramchandran. *OverSketched Newton: Fast Convex Optimization for Serverless Systems*. 2019. DOI: 10.48550/ARXIV.1903.08857. URL: https://arxiv.org/abs/1903.08857.

[201] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 363–376. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/p%20resentation/fouladi.

[202] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 475–488. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/fouladi.

[203] 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.

[204] 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. DOI: 10.1109/BigData52589.2021.9672067.

[205] M. B. Jones, J. Bradley, and N. Sakimura. "JSON Web Token (JWT)". In: *RFC* 7519 (2015), pp. 1–30.

[206] J. v. Brocke, A. Simons, B. Niehaves, K. Riemer, R. Plattfaut, and A. Cleven. "Reconstructing the Giant: On the Importance of Rigour in Documenting the Literature Search Process". In: June 2009.

[207] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo. "Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis". In: *2016 IEEE Trustcom/BigDataSE/ISPA*. 2016, pp. 1660–1667. DOI: 10.1109/TrustCom.2016.0255.

[208] H. Kimm, Z. Li, and H. Kimm. "SCADIS: Supporting Reliable Scalability in Redis Replication on Demand". In: *2017 IEEE International Conference on Smart Cloud (SmartCloud)*. 2017, pp. 7–12. DOI: 10.1109/SmartCloud.2017.9.

[209] *redisAI*. Available at: https://oss.redis.com/redisai/. Accessed on 13/07/2022.

[210] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik. *EMNIST: an extension of MNIST to handwritten letters*. 2017. DOI: 10.48550/ARXIV.1702.05373. URL: https://arxiv.org/abs/1702.05373.