Adaptive Aggregation For Federated Learning

K. R. Jayaram
IBM Research, USA
Vinod Muthusamy
IBM Research, USA
Gegi Thomas
IBM Research, USA
Ashish Verma
IBM Research, USA
Mark Purcell
IBM Research, Ireland

Abstract—In this paper, we present a new scalable and adaptive architecture for FL aggregation. First, we demonstrate how traditional tree overlay based aggregation techniques (from P2P, publish-subscribe and stream processing research) can help FL aggregation scale, but are ineffective from a resource utilization and cost standpoint. Next, we present the design and implementation of AdaFed, which uses serverless/cloud functions to adaptively scale aggregation in a resource efficient and fault tolerant manner. We describe how AdaFed enables FL aggregation to be dynamically deployed only when necessary, elastically scaled to handle participant joins/leaves and is fault tolerant with minimal effort required on the (aggregation) programmer side. We also demonstrate that our prototype based on Ray [1] scales to thousands of participants, and is able to achieve a > 90% reduction in resource requirements and cost, with minimal impact on aggregation latency.

Index Terms—federated learning, serverless, adaptive, aggregation

I. INTRODUCTION

Federated Learning (FL) [2], [3] is a mechanism in which multiple parties collaborate to build and train a joint machine learning model typically under the coordination/supervision of a central server or service provider (definition by Kairouz et al. [2], [3]). This central server is also called an aggregator. FL is private by design, because parties retain their data within their private devices/servers; never sharing said data with either the aggregator or other parties. An FL job involves parties performing local training on their data, sharing the weights/gradadients of their model (also called a model update) with the aggregator, which aggregates the model updates of all parties using a fusion algorithm. The use of centralized aggregation is common in FL because of the ease in which various machine learning models (neural networks, decision trees, etc.) and optimization algorithms can be expressed.

FL is typically deployed in two scenarios: cross-device and cross-silo. In the cross-silo scenario, the number of parties is small, but each party has extensive compute capabilities (with stable access to electric power and/or equipped with hardware accelerators) and large amounts of data. The parties have reliable participation throughout the entire federated learning training life-cycle, but are more susceptible to sensitive data leakage. Examples include multiple hospitals collaborating to train a tumor/COVID detection model on radiographs [4], multiple banks collaborating to train a credit card fraud detection model, etc. The cross-device scenario involves a large number of parties (> 100), but each party has a small number of data items, constrained compute capability, and limited energy reserve (e.g., mobile phones or IoT devices). They are highly unreliable/asyncrhonous and are expected to drop and join frequently. Examples include a large organization learning from data stored on employees’ devices and a device manufacturer training a model from private data located on millions of its devices (e.g., Google Gboard [5]).

Increasing adoption of FL has, in turn, increased the need for FL-as-a-service offerings by public cloud providers, which serve as a nexus for parties in an FL job and aggregate/fuse model updates. Such FL aggregation services have to effectively support multiple concurrent FL jobs, with each job potentially having tens to thousands of heterogeneous participants (mobile phones, tablets, sensors, servers) from different organizations and administrative domains.

Performance: Most existing FL platforms (IBM FL [6], Webank FATE [7], NVIDIA NVFLARE [8]) are based on a client-server model with a single aggregator per FL job deployed (as a virtual machine or container) in datacenters waiting for model updates. Such platforms are able to easily support multiple concurrent FL jobs, but performance drops as the number of parties increases, especially in cross-device settings. This is because aggregation throughput is limited by the computational capacity of the largest VM or container (memory and compute, and to a lesser extent, network bandwidth).

Resource Efficiency/Cost: While operating IBM FL and from publicly available FL benchmarks like LEAF [9] and Tensorflow Federated [10], we have observed that training at the party takes much longer compared to model update fusion/aggregation, resulting in under-utilization and wastage of computing resources dedicated to aggregation. This is a significant problem even in cross-silo settings – active participation is not guaranteed even in cross-silo settings due to competition from other higher priority workloads and variations in data availability. It is further compounded in “cross-device” deployments, where parties are highly intermittent and do not have dedicated resources for training. In these scenarios, the aggregator expects to hear from the parties eventually (typically over a several hours or maybe once a day). Large-scale FL jobs almost always involve intermittent parties – as the number of parties increases, it is extremely hard to expect that all of them participate at the same pace. This results in aggregators having to wait for long periods of time for parties to finish local training and send model updates.

Contributions: The core technical contribution of this paper is the design, implementation and evaluation of a flexible
parameter aggregation mechanism for FL – AdaFed, which has the following novel features:

- **AdaFed** reduces state in aggregators and treats aggregators as serverless functions. In many existing FL jobs, every aggregator instance typically acts on a sequence of inputs and produces a single output. State, if present, is not local to the aggregator instance and may be shared by all aggregators. Such state is best left in an external store, and consequently aggregators can be completely stateless and hence, serverless. AdaFed is therefore scalable both with respect to participants – effective for cross-silo and cross-device deployments, and with respect to geography – single/hybrid cloud or multicloud.

- **AdaFed** leverages serverless technologies to deploy and tear down aggregator instances dynamically in response to participant model updates, thereby supporting both intermittent and active participants effectively. There is no reason to keep aggregators deployed all the time and simply “awaiting input”.

- **AdaFed** is efficient, in terms of resource utilization with support for automatic elastic scaling, and in terms of aggregation latency.

- **AdaFed** is reasonably expressive for programmers to easily implement scalable aggregation algorithms. AdaFed is implemented using the popular Ray [1] distributed computing platform, and can run arbitrary Python code in aggregation functions, and use GPU accelerators if necessary.

II. **AdaFed : Design and Implementation**

A. **Associativity → Tree-based Aggregation**

Since the number of participants typically varies between FL jobs, and within a job (over time) as participants join and leave, horizontal scalability of FL aggregation software is vital. **Horizontally scalable** aggregation is only feasible if the aggregation operation is associative – assuming \( \oplus \) denotes the aggregation of model updates (e.g., gradients) \( U_i \). \( \oplus \) is associative if \( U_1 \oplus U_2 \oplus U_3 \oplus U_4 \equiv (U_1 \oplus U_2) \oplus (U_3 \oplus U_4) \).

Associativity is the property that enables us to exploit data parallelism to partition participants among aggregator instances, with each instance responsible for handling updates from a subset of participants. The outputs of these instances must be further aggregated.

A tree topology connects the aggregator instances. The output of each aggregator goes to its parent for further aggregation. We have determined that it is possible to split any associative FL aggregation operation into leaf and intermediate aggregators as illustrated by Figure 1. A leaf aggregator implements logic to fuse raw model weight updates \( U_i \) from a group of \( k \) parties to generate a partially aggregated model update \( U_k \). An intermediate aggregator implements logic to further aggregate partially aggregated model updates \( (U_k) \), in stages, to produce the final aggregated model update \( U_F \).

Establishing a tree-based aggregation topology as in Figure 1 starts by identifying the number of parties that can be comfortably handled by an aggregator instance. This is dependent on (i) size/hardware capability (CPU/RAM/GPU) of the instance (server or VM or container) and its network bandwidth, and (ii) the size of the model, which directly determines the size of the model update and the memory/compute capabilities needed for aggregation. Assuming that each instance can handle \( k \) participants, a complete and balanced \( k \)-ary tree can be used. \( \lceil \frac{n}{k} \rceil \) leaf aggregators are needed to handle \( n \) participants; the tree will have \( O(\lceil \frac{n}{k} \rceil) \) nodes.

A serious problem with tree-based aggregation overlays is that aggregator instances are “always on” waiting for updates, and this is extremely wasteful in terms of resource utilization and monetary cost. To handle FL jobs across thousands of parties, aggregation services including AdaFed must support intermittent parties effectively. Given that, for every round, parties may send model updates over an extended time period (hours), aggregators spend the bulk of their time waiting.

![Fig. 1. Hierarchical/Tree-based Aggregation](image)

**AdaFed** System Architecture. Aggregators are executed as serverless functions.

B. **Using Serverless Functions**

**AdaFed** takes associativity one step further. AdaFed mitigates issues with aggregation overlays by avoiding the construction of actual/physical tree topology. Instead, **AdaFed** uses serverless functions chained together with message queues to realize a logical tree topology. AdaFed executes both leaf and intermediate aggregation operations as serverless/cloud functions. These functions are executed in containers on a cluster managed by Kubernetes, which multiplexes multiple workloads and enables the cluster to be shared by
multiple FL jobs and/or other workloads. Also, since there is no static topology, more (or less) aggregator functions can be spawned depending on the number of parties (model updates), thereby handling party joins/leaves effectively. The challenge in executing aggregation as serverless functions, which are ephemeral and have no stable storage, is to manage state – that of each aggregation entity, intermediate aggregation outputs, inter-aggregator communications and party-aggregator communications. We also note that splitting aggregation into leaf and intermediate functions makes the logic simpler. It is also possible to have a single serverless function that can operate on both raw updates and partially fused updates; doing that will increase the complexity of the function.

C. End-to-End Illustration

As illustrated in Figure 3, a set of parties decide to start an FL job through existing private communication channels. “Matchmaking” or inducing parties to join an FL job is out of scope of this paper and AdaFed. We assume that this set of parties is convinced of the benefits of FL and want to collaborate. While forming a group, they also decide things like model architecture, model interchange format and hyperparameters (initial model weights, batch size and learning rate schedule, number of rounds, target accuracy and model update frequency). AdaFed then assigns a JobID to this job, creates metadata pertaining to the job (including party identities and hyperparameters), updates its internal data structures, instantiates two Kafka queues – JobID-Agg and JobID-Parties. A serverless function is triggered to publish the initial model architecture and weights on JobID-Agg. The FL job also specifies the triggering function. Then the first round of training starts at the parties’ local infrastructure using the model downloaded/received from JobID-Agg.

Once local training is complete, parties send model updates to JobID-Parties. The trigger (serverless) function executes, and if it determines that aggregation has to be initiated, triggers a leaf or intermediate aggregator. They pull inputs from JobID-Parties and publish their outputs to the same. This process continues as model updates arrive. When an aggregator function determines that all parties have sent their updates, the round is finished and the updated model published to JobID-Agg. Then the next round starts.

Job termination criteria may be different depending on the type of the FL job, as discussed earlier. A time-based or a quorum-based completion criterion may be also used.

D. Implementation and Elastic Scaling

We implement AdaFed using the popular Ray [1] distributed computing platform. Ray provides several abstractions, including powerful serverless functions (Ray remote functions). Ray’s internal message queue could have been used in lieu of Kafka, but we found Kafka to be more robust.

Aggregation triggers are implemented using Ray, and support typical conditions on JobID-Parties (receipt of a certain number of messages, etc.), but are flexible enough to execute user functions that return booleans, corresponding to whether aggregation should be triggered or not.

Our implementation using Ray executes on the Kubernetes cluster manager. Ray’s elastic scaler can request additional Kubernetes pods to execute serverless functions, depending on how frequently aggregation is triggered. It is also aggressive about releasing unused pods when there are no model updates pending. When aggregation is triggered, groups of model updates are assigned to serverless function invocations. Each invocation is assigned 2 vCPUs and 4GB RAM (this is configurable). If there are insufficient pods to support all these invocations, Ray autoscales to request more Kubernetes pods.

This also enables AdaFed to handle large scale party dropouts and joins effectively. Only the exact amount of compute required for aggregation is deployed – overheads to spawn tens of thousands of serverless functions). Ray’s internal message queue could have been used in lieu of Kafka, but we found Kafka to be more robust.

Aggregation latency (s) – time taken for aggregation to finish after the last model update is available

In this section, we evaluate the efficacy of AdaFed, by first comparing AdaFed against a centralized aggregator setup common in several FL frameworks like IBM FL [6], FATE [7]...
and NVFLARE [8]. We demonstrate how such single aggregator setups when scaling beyond 100 participants. We then demonstrate how a static hierarchical (tree) overlay of aggregator instances can help with the scalability issue, but is ineffective from a resource consumption, utilization, cost and elasticity perspectives.

### A. Metrics & Experimental Setup

Given that aggregation depends on whether the expected number of model updates are available, we define aggregation latency as the time elapsed between the reception of the last model update and the availability of the aggregated/fused model. We evaluate resource efficiency, by measuring resource consumption (in terms of the number and duration of containers used for aggregation), resource (CPU and memory) utilization and projected total cost. We execute both hierarchical aggregation and AdaFed using containers on Kubernetes pods in our datacenter, and measure the number of container seconds used by an FL job from start to finish. Container seconds is calculated by multiplying the number of containers used with the time that each container was used/alive. This includes all the resources used by the ancillary services, including MongoDB (for metadata), Kafka and Cloud Object Store.

Aggregation was executed on a Kubernetes cluster on CPUs, using Docker containers. For IBM FL, the container used for the single aggregator was run on a dedicated server with 16 CPU cores (2.2 Ghz, Intel Xeon 4210) and 32GB of RAM. Each container for hierarchical or serverless aggregation was equipped with 2 vCPUs (2.2 Ghz, Intel Xeon 4210) and 4 GB RAM. For hierarchical/tree aggregation, each instance was encapsulated using the Kubernetes service abstraction. Parties were emulated, and distributed over four datacenters (different from the aggregation datacenter) to emulate geographic distribution. Each party was also executed inside Docker containers (2 vCPUs and 4 GB RAM) on Kubernetes, and these containers had dedicated resources. We actually had parties running training to emulate realistic federated learning, as opposed to using, e.g., Tensorflow Federated simulator.

We select three real-world federated learning jobs – two image classification tasks from the Tensorflow Federated (TFF) [10] benchmark and one popular document classification task. From TFF [10], we select (i) CIFAR100 dataset which can be distributed over 10-10000 parties, with classification performed using the EfficientNet-B7 model and the FedProx [11] aggregation algorithm. Thus, we consider two types of images and two models of varying sizes. We do not consider other workloads from TFF because they involve less than 1000 parties. For additional diversity, we consider a third workload using the VGG16 [12] model and FedSGD [5] aggregation algorithm on RVL-CDIP [13] document classification dataset. Each job was executed for 50 synchronization rounds, with model fusion happening after every local epoch. For all scenarios, the datasets were partitioned in a realistic non-IID manner. Due to space constraints, only (i) is presented in this paper, while our tech report [14] contains all the results and findings.

### B. Aggregation Latency and Scalability

First, we consider a scenario where the number of parties remains constant throughout the FL job, for all synchronization rounds, i.e., once the job starts, no parties join or leave. From Figure 4, we observe that a centralized single aggregator setting does not scale to a large number of parties, as average aggregation latency increases significantly – almost linearly. This is because of both constrained compute/memory capacity at the single aggregator and constrained network bandwidth needed to transfer/load model updates for aggregation. Figure 4 also illustrates that the increase in aggregation latency is much more gradual for both static tree overlays and AdaFed (which uses serverless functions), enabling these architectures to scale to larger FL settings. In fact, for both static tree and AdaFed, latency increases only by \( \approx 4 \times \) when the number of parties increases 1000×. This trend is due to the data parallelism inherent in both the static tree and AdaFed.

From an efficiency standpoint, we observe that the aggregation latency is similar between static tree and AdaFed, within 4% of each other, with aggregation latency of AdaFed being slightly higher than that of the static tree overlay. This is because using serverless functions does not reduce the number of aggregation steps; it merely avoids having to keep the aggregators provisioned and alive when they are not needed. We used runtime profiling to determine that the slight (up to 4%) increase in aggregation latency over the static tree is primarily due to cold starts when functions are started; the other minor factor is the latency due to the aggregation trigger. Thus, we observe that the runtime overhead of using and triggering serverless functions is minimal.

### C. Adaptivity/Elastic Scaling for Party Joins

Next, we illustrate how AdaFed can handle parties joining in the middle of the job with minimal impact on aggregation latency. For this, we consider a single synchronization round, and increase the number of parties by 20%. Figure 5 illustrates the aggregation latency when 20% more parties send model updates during the synchronization round. For these experiments, we only illustrate static tree based overlays and AdaFed. This is because Section III-B has already demonstrated that centralized aggregators do not scale to handle large numbers of parties; the effect of party joins is similar.
### Fig. 6. EfficientNet-B7 on CIFAR100 using FedProx aggregation algorithm. Active Participants. Resource usage and projected cost, using container cost/s of 0.0002692 US$ (source Microsoft Azure [15]).

| Parties | Static Tree | AdaFed |
|---------|-------------|--------|
| 10      | 1723        | 228    |
| 100     | 2653        | 351    |
| 1000    | 22340       | 2951   |
| 10000   | 298900      | 40849  |

| Proj. Total cost US$ | Cost Savings % | Avg. CPU Util. (%) | Avg. Memory Util. (%) |
|----------------------|-----------------|--------------------|-----------------------|
| Static Tree | AdaFed | Static Tree | AdaFed | Static Tree | AdaFed | Static Tree | AdaFed |
| 0.46       | 0.09   | 86.96%  | 12.31%  | 82.95%  | 46.54%  | 73.35%  |
| 6.01       | 0.79   | 86.86%  | 19.09%  | 83.25%  | 20.89%  | 72.87%  |
| 80.46     | 11    | 86.33%  | 10.61%  | 84.27%  | 18.66%  | 75.39%  |

---

- aggregation latency increases almost linearly w.r.t number of parties joining. Serverless aggregation in AdaFed needs no overlay reconfiguration, while static tree aggregation needs to add more aggregator instances and reconfigure the tree. This manifests as a significant increase in aggregation latency (2.47× to 4.62×). This is due to the fact that the number of serverless function invocations depends on the aggregation workload, and partially aggregated updates can be stored in message queues. However, with a tree overlay, new aggregator nodes have to be instantiated and the topology changed. Thus, although both static tree and serverless aggregation methods are elastic, using serverless functions provides significantly better outcomes.

### D. Resource Consumption & Cost

We compare AdaFed with static tree aggregation in terms of resource usage. Although the single aggregator deployment (e.g., using IBM FL) has much lower resource requirements when compared to AdaFed, it has significantly higher latency and does not scale. So, we do not consider it in the experiments in this section. We first illustrate the resource consumption of experiments where parties participate actively. Figure 6 tabulates the resource usage for the three workloads, in terms of container seconds and CPU/memory utilization. This data illustrates the real benefits of using serverless aggregation, with > 85% resource and cost savings for the EfficientNet-B7/CIFAR100/FedProx job. These savings are significant and are a direct result of the adaptivity of AdaFed, by deploying aggregator functions only when needed. Resource wastage due to static tree can also be observed from the CPU/memory utilization figures, which are consistently low for static tree because aggregator instances are idle for long periods. We also observe that, while compute resources needed for aggregation increase with the number of participants for both static tree and serverless aggregation, the amount of resource and cost savings remains fairly consistent. We use Microsoft Azure’s container pricing for illustrative purposes only; pricing is similar for other cloud providers. Figure 7 demonstrates that resource and cost savings are huge (> 99%) when response timeout is set to a modest 10 minutes per aggregation round in the case of intermittent participants. Real world FL jobs typically use higher response timeouts and will thus reap enormous benefits. Thus, our experiments reinforce our confidence that serverless aggregation can lead to significant resource and cost savings with minimal overhead.

### Fig. 7. EfficientNet-B7 on CIFAR100 using FedProx aggregation algorithm. Intermittent participants updating over a 10 minute interval for every synchronization round. Resource usage and projected cost using Container cost/s of 0.0002693 US$ (source Microsoft Azure [15]).

| Num. Parties | Static Tree | AdaFed |
|--------------|-------------|--------|
| 10           | 634         | 272    |
| 100          | 576         | 385    |
| 1000         | 10516       | 1113   |
| 10000        | 105021      | 18741  |

| Proj. Total cost US$ | Cost Savings % | Avg. CPU Util. (%) | Avg. Memory Util. (%) |
|----------------------|-----------------|--------------------|-----------------------|
| Static Tree | AdaFed | Static Tree | AdaFed | Static Tree | AdaFed | Static Tree | AdaFed |
| 0.17       | 0.07   | 99.28%  | 10.58%  | 81.3%  | 42.67%  | 75.26%  |
| 0.16       | 0.1    | 98.89%  | 11.97%  | 79.77%  | 12.17%  | 74.77%  |
| 2.83       | 0.3    | 99.82%  | 11.41%  | 81.06%  | 11.05%  | 74.15%  |
| 28.27      | 5.05   | 99.7%   | 10.25%  | 81.09%  | 10.29%  | 74.71%  |

---

IV. RELATED WORK

Parallelizing FL aggregation using a hierarchical topology has been explored by [5], though the design pattern was introduced by and early work on datacenter parallel computing [16]. While [5] uses hierarchical aggregation, its programming model is different from AdaFed. Its primary goal is scalability and consequently, it deploys long lived actors instead of serverless functions. AdaFed aims to make FL aggregation resource efficient, elastic in addition to being scalable. A closely related concurrent work is FedLess [17], which predominantly uses serverless functions for the training side (party side) of FL. It also has the ability to run a single aggregator as a cloud function, but does not have the ability to parallelize aggregation, and does not seem to scale beyond 200 parties (with 25 parties updating per FL round, per [17]). Our work in AdaFed has the primary goal of parallelizing and scaling FL aggregation. Fedless [17] also does not adapt aggregation based on party behavior, and it is unclear whether parties on the edge (phones/tablets) can train using FedLess.

A number of ML frameworks – Siren [18], Cirrus [19] and the work by LambdaML [20] use serverless functions for centralized (not federated) ML and DL training. Siren [18] allows users to train models (ML, DL and RL) in the cloud using serverless functions with the goal to reduce programmer burden involved in using traditional ML frameworks and cluster management technologies for large scale ML jobs. It also contains optimization algorithms to tune training performance and reduce training cost using serverless functions. Cirrus [19]
goes further, supporting end-to-end centralized ML training workflows and hyperparameter tuning using serverless functions. LambdaML [20] analyzes the cost-performance trade-offs between IaaS and serverless for datacenter/cloud hosted centralized ML training. LambdaML can execute purely using a hybrid serverless/IaaS strategy. AdaFed differs from Siren, Cirrus and LambdaML in significant ways – Distributed ML (in Siren, Cirrus and LambdaML) is different from FL. Distributed ML involves centralizing data at a data center or cloud service and performing training at a central location. In contrast, with FL, data never leaves a participant. FL’s privacy guarantees are much stronger and trust requirements much lower than that of distributed ML.

The term “serverless” has also been used to refer to peer-to-peer (P2P) federated learning, as in [21]–[23]. In such systems, aggregation happens over a WAN overlay and not in a datacenter. [23] does not construct overlays but uses gossip-based broadcast algorithms to deliver and aggregate model updates in a decentralized manner. While these techniques are scalable and (in the case of gossip algorithms) fault tolerant, they do require either (i) that the model be revealed to more entities during routing, or (ii) homomorphic encryption [24] which can be challenging both from a key agreement and model size explosion standpoints, or (iii) differential privacy [25] which reduces model accuracy in the absence of careful hyperparameter tuning.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented AdaFed, a system for adaptive serverless aggregation in federated learning. We have described the predominant way of parallelizing aggregation using a tree topology and examined its shortcomings. We have demonstrated how serverless/cloud functions can be used to effectively parallelize and scale aggregation while eliminating resource wastage and significantly reducing costs. Our experiments using three different model architectures, datasets and two FL aggregation algorithms demonstrate that the overhead of using serverless functions for aggregation is minimal, but resource and cost savings are substantial. We also demonstrate that serverless aggregation can effectively adapt to handle changes in the number of participants in the FL job.

We are currently working to extend this work in two directions: (i) increasing the dependability and integrity of aggregation using trusted execution environments (TEEs) and (ii) effectively supporting multi-cloud environments by using service mesh (like Istio) to find the best aggregator function to route a model update.

REFERENCES

[1] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elizbol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, “Ray: A distributed framework for emerging AI applications,” in OSDI '18. USENIX Association, 2018.

[2] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings et al., “Advances and open problems in federated learning,” arXiv preprint arXiv:1912.04977, 2019.

[3] J. W. et. al., “A field guide to federated optimization,” 2021.

[4] I. e. a. Dayan, “Federated learning for predicting clinical outcomes in patients with covid-19,” Nature Medicine, 2021.

[5] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, H. B. McMahan et al., “Towards federated learning at scale: System design,” arXiv preprint arXiv:1902.01046, 2019.

[6] H. Ludwig, N. Baracaldo, G. Thomas, Y. Zhou, A. Anwar, S. Rajamoni, Y. Wu, N. Radhakrishnan, A. Verma, M. Sinn, M. Purcell, A. Rawat, T. Minh, N. Holohan, S. Chakraborty, S. Whitherspoon, D. Steuer, L. Wynter, H. Hassan, S. Laguna, M. Yurochkin, M. Agarwal, E. Chuba, and A. Abay, “Ibm federated learning: an enterprise framework white paper v0.1,” arXiv preprint arXiv:2007.10987, 2020.

[7] Y. Liu, T. Fan, T. Chen, Q. Xu, and Q. Yang, “Fate: An industrial grade platform for collaborative learning with data protection,” Journal of Machine Learning Research, vol. 22, no. 226, pp. 1–6, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-815.html

[8] NVIDIA, “Nvidia federated learning application runtime environment,” https://github.com/NVIDIA/NVFLare, 2021.

[9] 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.

[10] T. Project, “Using tff for federated learning research,” https://www.tensorflow.org/federated/tff_for_research, 2022.

[11] T. Li, A. K. Sahu, M. Záheer, M. Sanjabi, A. Talwalkar, and V. Smith, “Federated optimization in heterogeneous networks,” in ML-Sys, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., 2020, pp. 429–450.

[12] A. Das, S. Roy, U. Bhattacharya, and S. K. Parui, “Document image classification with intra-domain transfer learning and stacked generalization of deep convolutional neural networks,” 2018.

[13] A. W. Harley, A. Ufkes, and K. G. Derpanis, “Evaluation of deep convolutional nets for document image classification and retrieval,” in International Conference on Document Analysis and Recognition. IEEE, 2015, pp. 991–995.

[14] K. R. Jayaram, V. Muthusamy, G. Thomas, A. Verma, and M. Purcell, “Adaptive aggregation for federated learning,” 2022. [Online]. Available: https://arxiv.org/abs/2203.12163

[15] Microsoft, “Azure Container Instances pricing,” https://azure.microsoft.com/en-us/pricing/details/container-instances/, 2021.

[16] V. Kumar, Introduction to Parallel Computing, 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[17] A. Grafberger, M. Chadha, A. Jindal, J. Gu, and M. Gerndt, “Fedless: Secure and scalable federated learning using serverless computing,” in IEEE BigData, 2021.

[18] 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.

[19] J. Carreira, P. Fosseca, A. Tumanov, A. Zhang, and R. Katz, “Cirrus: A serverless framework for end-to-end ml workflows,” in SoCC ’19, New York, NY, USA: ACM, 2019, p. 13–24. [Online]. Available: https://doi.org/10.1145/3357223.3362711

[20] 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 ACM SIGMOD, 2021.

[21] M. Chadha, A. Jindal, and M. Gerndt, “Towards federated learning using faas fabric,” in Proceedings of the 2020 Sixth International Workshop on Serverless Computing, ser. WoSC'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 49–54. [Online]. Available: https://doi.org/10.1145/3429880.3430100

[22] C. Niu, F. Wu, S. Tang, L. Hua, R. Jia, C. Lv, Z. Wu, and G. Chen, “Billion-scale federated learning on mobile clients: A submodel design with tunable privacy,” in ACM MobiCom ’20, 2020.

[23] I. Hegedus, G. Danner, and M. Jelassi, “Gossip learning as a decentralized alternative to federated learning,” in DAIM, 2019.

[24] K. R. Jayaram, A. Verma, A. Verma, G. Thomas, and C. Butcher-Sheard, “Mystiko: Cloud-mediated, private, federated gradient descent,” in 2020 IEEE 13th International Conference on Cloud Computing (CLOUD). IEEE, 2020, pp. 201–210.

[25] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, “Deep learning with differential privacy,” in CCS ’16, 2016.