FLSTORE: EFFICIENT FEDERATED LEARNING STORAGE FOR NON-TRAINING WORKLOADS

ABSTRACT

Federated Learning (FL) is an approach for privacy-preserving Machine Learning (ML), enabling model training across multiple clients without centralized data collection. With an aggregator server coordinating training, aggregating model updates, and storing metadata across rounds. In addition to training, a substantial part of FL systems are the non-training workloads such as scheduling, personalization, clustering, debugging, and incentivization. Most existing systems rely on the aggregator to handle non-training workloads and use cloud services for data storage. This results in high latency and increased costs as non-training workloads rely on large volumes of metadata, including weight parameters from client updates, hyperparameters, and aggregated updates across rounds, making the situation even worse. We propose FLStore, a serverless framework for efficient FL non-training workloads and storage. FLStore unifies the data and compute planes on a serverless cache, enabling locality-aware execution via tailored caching policies to reduce latency and costs. Per our evaluations, compared to cloud object store based aggregator server FLStore reduces per request average latency by 71% and costs by 92.45%, with peak improvements of 99.7% and 98.8%, respectively. Compared to an in-memory cloud cache based aggregator server, FLStore reduces average latency by 64.6% and costs by 98.83%, with peak improvements of 98.8% and 99.6%, respectively. FLStore integrates seamlessly with existing FL frameworks with minimal modifications, while also being fault-tolerant and highly scalable.

1 INTRODUCTION

Federated Learning (FL) (McMahan et al., 2017) is as a privacy-aware solution for ML training across numerous clients without data centralization. The FL process also encompasses a broad range of non-training workloads. *Non-training workloads* refer to tasks such as scheduling (Lai et al., 2021b; Abdelmoniem et al., 2023), personalization (Ghosh et al., 2020; Tan et al., 2022), clustering (Liu et al., 2022), debugging (Gill et al., 2023), and incentivization (Han et al., 2022; Hu et al., 2022), etc. that are necessary for the success and efficiency of the FL process. The growing interest in Explainable AI (Gade et al., 2019; Mohseni et al., 2021), has led to several Explainable FL (XFL) systems that depend on non-training workloads including debugging (Duan et al., 2023; Gill et al., 2023), accountability (Balta et al., 2021; Baracaldo et al., 2022; Yang et al., 2022a), transparency (Han et al., 2022), and reproducibility (Desai et al., 2021; Gill et al., 2023).

Challenges Existing research concentrates only on training efficiency (Reisizadeh et al., 2020; Shlezinger et al., 2021; Yu et al., 2023; Kairouz et al., 2019; Yang et al., 2019; Lai et al., 2021b; Tan et al., 2023a). However, non-training workloads constitute a significant and equally important part of the latency and cost in the FL process (Kairouz et al., 2019). Figure 1 shows a single non-training application can comprise up to 60% of the total latency of the FL job, and several non-training applications are often executed in the same FL process (Baracaldo et al., 2022) with latency several times more than training (§ 2.1). Non-training workloads are highly data intensive and require tracking, storage, and processing of data, including model parameters, training outcomes, hyperparameters, and datasets reaching thousands of TBs across just 100 FL jobs (§ 2.2).

In current state-of-the-art FL frameworks (Qi et al., 2024; Bonawitz et al., 2019; Beutel et al., 2020; He et al., 2020; IBM, 2020; FederatedAI, 2024), cloud-based aggregators handle the non-training workloads and utilize a separate cloud object store for data storage (Amazon Web Services, 2024b) as shown in Figure 2. Consequently, aggregators are ill-equipped to store and process large volumes of FL metadata efficiently and cost-effectively.

This raises several challenges regarding costs and latency. First, utilizing cloud object stores for data storage separates the data and compute planes. As shown in Figure 2, this results in extra round trips of storing and fetching the data into the aggregator server's memory, leading to high latency and costs. Even when augmented with more expensive cloud-based caches (Amazon Web Services, 2024a) the communication bottleneck remains a challenge (Liu et al., 2023). Second, non-training workloads in FL have diverse data storage and processing requirements. For instance, tracing the provenance of specific clients necessitates access to client model updates from previous training rounds (Baracaldo et al., 2022), while identifying issues in malicious clients requires the model updates of all clients for a specific training round (Gill et al., 2023). Thus, any caching solution

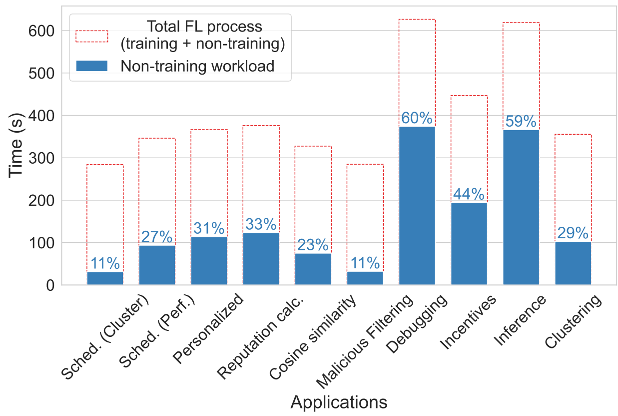

Figure 1. Non-training portion in total FL process with 200 clients, EfficientNet model (Tan & Le, 2021), 1000 training rounds, and CIFAR10 Dataset (Krizhevsky, 2009).

for non-training workloads with traditional caching policies that do not consider these unique data requirements will result in sub-optimal performance.

Third, relying on dedicated servers for executing these workloads becomes a significant issue since the demand for non-training tasks such as debugging and auditing could extend beyond the training phase, necessitating continuous operation of the servers and cache (Baracaldo et al., 2022).

Our Solution To address these challenges, we make three key observations. First, unifying the compute and data planes can significantly reduce communication bottlenecks. Second, the iterative nature of FL leads to non-training workloads having sequential and predictable data access patterns; for example, tracking a client's model updates across training rounds will require repeated access to the same client's data across rounds. Third, because non-training workloads, such as debugging, may be required long after training has concluded, a scalable and on-demand solution is essential.

We present FLStore, a caching framework that unifies the data and compute planes with a cache built on serverless functions. FLStore utilizes the co-located compute available on those functions for locality-aware execution of non-training workloads. FLStore uniquely leverages the iterative nature of FL and its sequential data access patterns to implement tailored caching policies optimized for FL. To develop these policies, we classify non-training workloads in FL applications into a comprehensive taxonomy, categorizing them by their distinct data needs and access patterns. FLStore then customizes its caching policies to the specific type of non-training request encountered.

Contributions Our contributions in this work are as follows: 1) To the best of our knowledge, we present the first comprehensive study of storage and execution requirements of non-training workloads in FL, analyzing their impact on cost and efficiency. 2) Based on the insights from this

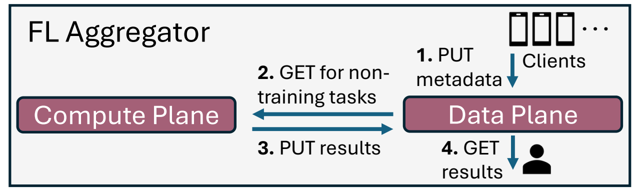

Figure 2. Data flow of serving non-training requests in conventional FL aggregators

study, we identify iterative data access patterns in FL, which we leverage to develop FLStore, a novel caching framework with tailored caching policies that use prefetching for locality-aware execution of FL workloads. FLStore is the first FL framework that unifies the data and compute planes and has native support for non-training FL workloads; 3) FLStore provides a highly scalable solution with its serverless functionality (Wang et al., 2020) to meet the demands of serving up to millions of clients in FL (Khan et al., 2023; Kairouz et al., 2019); It has a modular design (Abadi et al., 2016; Ludwig et al., 2020; Abdelmoniem et al., 2023) and can be integrated into any FL framework with minor modifications. 4) Compared to state-of-the-art FL frameworks (IBM, 2022; FederatedAI, 2024; Beutel et al., 2020) that are based on cloud services (Amazon Web Services, 2024a;b; Amazon Web Services, Inc., 2024b; Google Cloud, 2024), FLStore reduces the average per-request latency by 50.8% and up to 99.7%, and the average costs by 88.2% and up to 98.8%.

2 Background and Motivation

2.1 Non-training workloads in FL

XFL aims to improve FL by addressing issues such as clients submitting flawed models due to data quality problems or sabotage (Han et al., 2022; Gill et al., 2023). It also emphasizes auditing and regulatory compliance, especially in collaborations involving diverse entities (Balta et al., 2021; Yang et al., 2022a; Baracaldo et al., 2022). FLDebugger (Li et al., 2021) assesses the influence of each client's data on global model loss, identifying and correcting harmful clients. FedDebug (Gill et al., 2023) improves reproducibility by enabling the FL process to pause or rewind to specific breakpoints and helps detect malicious clients through differential neuron activation testing.

Other FL applications Due to the distributed nature of FL, many applications involve non-training tasks like clustering (Liu et al., 2022; Duan et al., 2021), personalization (Ruan & Joe-Wong, 2022; Tang et al., 2021), and asynchronous learning (Nguyen et al., 2021), which are essential for managing and optimizing the FL process. For example, clustering evaluates client models based on factors like training duration, networks, or energy use (Liu et al., 2022; Chai et al., 2021), while personalization groups clients by model parameters, efficiency, or accuracy on held-out data (Ruan &

Joe-Wong, 2022; Tang et al., 2021). Incentive mechanisms assess client contributions and reputations via accuracy or Shapley Values (Sun et al., 2023; Hu et al., 2022), and intelligent client selection relies on analyzing client availability, participation, and performance (Abdelmoniem et al., 2023; Lai et al., 2021b). Non-training tasks like debugging and hyperparameter tracking are also crucial for optimizing FL (Gill et al., 2023; Duan et al., 2023).

Non-training tasks can make up to 60% of the FL workflow, as shown in Figure 1. Typically, the FL process incorporates numerous non-training tasks. In this scenario involving multiple tasks such as filtering, scheduling, reputation calculation, incentive distribution, debugging, and personalization, non-training tasks account for 86% of total FL time, lasting 6× longer than training. Therefore, improving the efficiency and cost-effectiveness of non-training workloads is critical for enhancing FL applications.

2.2 Shortcomings of popular FL frameworks

State-of-the-art FL frameworks, as depicted in Figure 2, generally utilize an aggregator server on a stateful (Lai et al., 2021a; Beutel et al., 2020; IBM, 2020; He et al., 2020) or serverless compute plane (Qi et al., 2024; Jiang et al., 2021; Grafberger et al., 2021). The serverless model (Jonas et al., 2019) allows cloud providers (Amazon Web Services, Inc., 2024a; Jiang et al., 2021) to manage scaling and maintenance by executing functions on demand, with costs based on usage. However, FL data demands can escalate rapidly, reaching over 1500 TB for 100 training sessions with 10 clients each on the CIFAR10 dataset (Krizhevsky, 2009). To manage this, the compute plane is connected to a separate data plane using cloud caches like ElastiCache (Amazon Web Services, 2024a) or object stores like AWS S3 (Amazon Web Services, 2024b) and Google Cloud Storage (Google Cloud, 2024). This separation increases communication steps for non-training tasks, involving multiple rounds from receiving requests to fetching and processing data, and then storing results back, which, along with dedicated cloud services, leads to ongoing costs even when non-training requests are dormant. Compared to these FL frameworks (Lai et al., 2021a; Beutel et al., 2020; IBM, 2020; He et al., 2020), FLStore serves as a one-stop solution that processes non-training requests directly from the serverless cache, asynchronously fetching missing data from persistent storage when needed. It also utilizes tailored caching policies based on a classification of non-training workloads. The design of FLStore is discussed in detail later (§ 4).

2.3 Serverless Cache for Non-Training Apps

To build an in-memory locality-aware cache for non-training FL workloads, we must first answer two important questions: 1) *Can the models utilized in cross-device FL be stored in cloud functions' memory?* 2) *Does the execution latency of non-training workloads fall within the cloud functions lifetime thresholds?* To answer these questions, we first analyze 23 popular models used in cross-device FL settings from various works in FL (Caldas et al., 2018; Chen et al., 2022; Lai et al., 2021b; Kairouz et al., 2019). Analyzing the memory footprint of these models, the average size of these models is approximately 161 MB as discussed in detail in the Appendix D. These model sizes are perfect for storage in the in-memory cache of cloud functions, as the memory of these functions goes up to 10 GB. We also analyze the typical latency of different non-training workloads. Figure 3 shows the latencies of executing five different workloads across three different models (EfficientNetV2 Small (Tan & Le, 2021), Resnet18 (He et al., 2016), and MobileNet V3 Small (TorchVision Contributors, 2024)) and same setup as Figure 1 on a serverless cloud function (Amazon Web Services, Inc., 2024a) while fetching data from a cloud object store (Amazon Web Services, 2024b). It can be observed that the average computation latency across workloads is approximately 2.8 seconds, which is perfect for cloud functions due to their short lifetimes. The small size of the models and the short execution time of non-training tasks for cross-device FL make the memory and compute resources in serverless functions ideal for processing non-training tasks. However, the major bottleneck comes from the 31× higher average communication latency (89 sec). Thus, unifying the compute and data planes can ease this bottleneck, enabling efficient, cost-effective serving of non-training requests.

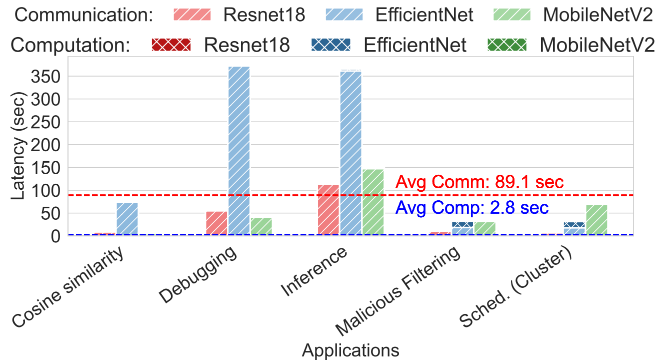

Figure 3. Average workload latencies computation and communication of non-training FL workloads.

3 Related Work

To our knowledge, no existing FL framework efficiently and cost-effectively processes non-training requests.

Generic cloud-based frameworks: General-purpose XAI cloud solutions like AWS SageMaker (Amazon Web Services, Inc., 2024b) use dedicated instances such as AWS EC2 with storage options like AWS S3 (Amazon Web Services, 2024b) or ElastiCache (Amazon Web Services, 2024a). This setup leads to high costs and decreased efficiency due to separated data storage and compute resources (Khan et al., 2023), also lacking tailored caching policies suited for FL's iterative nature.

FL frameworks: Existing State-of-the-art FL frameworks (IBM, 2020; Beutel et al., 2020; He et al., 2020; Caldas et al., 2018; FederatedAI, 2024) follow a similar architecture, where cloud-hosted aggregator servers with separate persistent storage execute non-training tasks (Khan et al., 2023; Bonawitz et al., 2019; Baracaldo et al., 2022), resulting in increased latency and costs.

Serverless aggregators: Another line of work focuses only on aggregation via serverless functions (Qi et al., 2024; Khan et al., 2023; Grafberger et al., 2021). FLStore can easily incorporate aggregation as one of the application workloads, however, FLStore is more generic and also includes additional non-training workloads for FL. Furthermore, non-training workloads such as debugging and incentivization often extend beyond the training phase, requiring aggregators beyond the training phase increasing costs (Gill et al., 2023; Khan et al., 2023; Bonawitz et al., 2019).

Serverless Storage: Serverless storage approaches utilize memory available on serverless functions at no additional cost, such as InfiniStore (Zhang et al., 2023b), a cloud storage service, and InfiniCache (Wang et al., 2020), an object caching system using ephemeral functions. These solutions primarily address storage, often underutilizing the computing resources of serverless functions.

4 FLStore

In this section, we present the detailed design for FLStore derived from the following insights we gather from our preliminary analysis (§ 2):

- I_1**:** Communication latency is the major bottleneck for non-training workloads brought by separate compute and data planes in extant solutions (§ 2.1 & 2.2).
- I_2**:** Non-training workloads show iterative data access patterns which can be classified, and leveraged to improve performance via a caching solution (§ 2.1).
- I_3**:** Memory footprint of models typically used in cross-device FL and the average latency of non-training workloads are suitable for the inexpensive on-demand Serverless functions (§ 2.3).

4.1 Unification of Compute and Data Planes

Aims. Our first design goal, guided by insight (I_1), is to integrate compute and data planes by using serverless function memories for a distributed cache with co-located compute resources like InfiniCache (Wang et al., 2020). However, InfiniCache does not use the compute capabilities of serverless functions or offer specialized caching policies (§ 2.2). This limitation presents a unique opportunity to also utilize free serverless computing for executing non-training workloads (I_3).

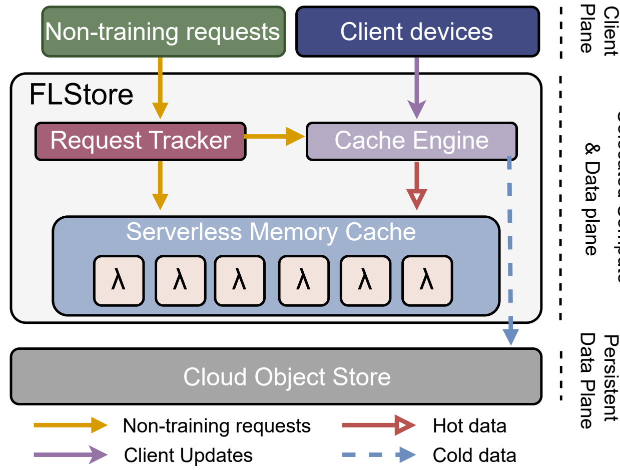

Figure 4. FLStore architecture design.

Challenges. Creating such a framework presents non-trivial challenges, which we address one by one in the following sections. First, we must track data storage, removal, and updates across multiple function memories (§ 4.2). Second, non-training requests need to be routed to the appropriate functions with the relevant data (§ 4.3). Third, it is crucial to identify which metadata should be cached, as storing all metadata would be costly and unsustainable (§ 4.4). Lastly, the solution must be scalable, fault-tolerant, and ensure data persistence (§ 4.5). We begin by introducing the main components of our solution (FLStore) that resolve the first challenge of tracking data across functions.

4.2 Tracking Data in Serverless Functions

FLStore consists of three components, a Request tracker, the Cache Engine, and a Serverless Cache as shown in Figure 4. For the Serverless cache, FLStore uses disaggregated serverless function memories similar to (Wang et al., 2020); FLStore extends this design to utilize the serverless compute resources of those functions to process non-training requests. The Cache Engine and the Request tracker can be run in the cloud or collocated with a client. The Cache Engine uses a hash table to store the location of data in disaggregated functions, tracking specific metadata to the functions where it is cached. The CacheEngine dictionary format is as follows:

$$Tuple(Client : str, Round : int) \rightarrow FunctionID : str$$

As shown by the data-flow in Figure 5, the Cache Engine receives incoming data from client training devices (Step ①) and fetches the current and incoming non-training request information from the Request tracker (Steps ② & ③). Based on the request types, it utilizes the appropriate caching policy to filter hot data from cold data (Step ④) and puts models in Serverless Cache and Persistent Store, respectively (Step ⑤). The data is cached at the granularity of client models such

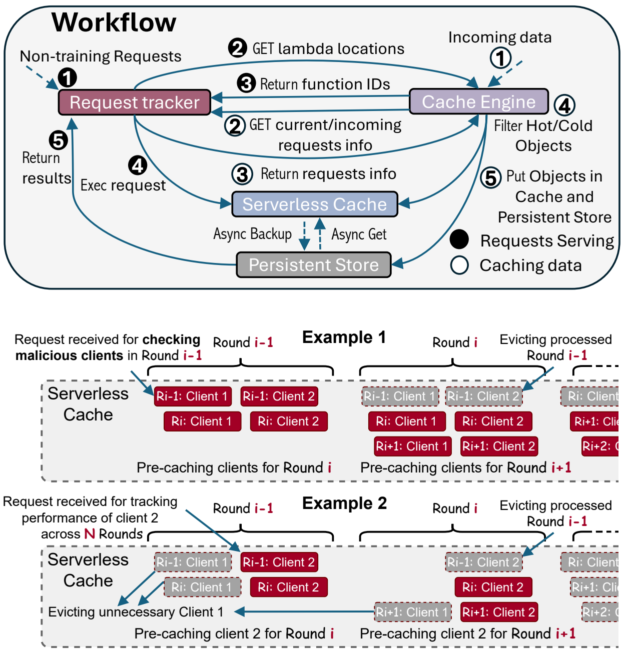

Figure 5. FLStore workflow (top) and examples (bottom).

that each function holds at least one client model. This level of granularity is practical as a single function provides up to 10 GB of memory (Amazon Web Services, Inc., 2024a). Unlike conventional cloud caching systems like ElastiCache, FLStore's serverless cache also provides compute resources for non-training tasks, ensuring that cached data is close to the compute needed to execute requests. So next we discuss how to resolve the second challenge of routing the requests to the appropriate functions containing the relevant data for locality-aware execution.

4.3 Locality-Aware Request Routing

One of FLStore's key contributions is to effectively leverage local compute resources to process data, enhancing the overall efficiency of resource utilization. Using these compute resources requires the non-training requests to be routed to the functions with data relevant to the request. The Request tracker, as shown in Figure 4, is responsible for receiving requests from clients, forwarding the request to the appropriate functions, and keeping track of the progress. The tracking data is stored in a dictionary where request IDs serve as keys, and the corresponding values include the list of function IDs to which the request was routed and the progress made by each function in executing the request. The Request Tracker dictionary is formatted as follows:

$$RequestID : str \rightarrow Tuple(List[FunctionID : str, ...], \\ Status : bool)$$

Figure 5 describes the workflow. Upon receiving the request in (Step ❶), the Request tracker fetches the function IDs from the Cache Engine where the data required for the non-training request is cached (Steps ❷ and ❸). Then, it issues the requests to those function IDs and keeps track of their progress (Step ❹), reporting the results as soon as they are returned to the client daemon (Step ❺). Next, we discuss how to determine which data is important for caching.

Table 1. Taxonomy of Non-Training Applications and Mapping of Workloads in FLStore

ID	Caching Policy	Applications and Mapped Workloads
P1	Individual Client Updates	Evaluates individual model's accuracy and fairness (Li et al., 2020; Yu et al., 2020; Ezzeldin et al., 2023).
P2	All Updates in a Round	Used in Personalization (Tan et al., 2022), Clustering (Ghosh et al., 2020), Scheduling (Chai et al., 2020), Contribution calculation (Sun et al., 2023), Filtering malicious clients (Han et al., 2022), Cosine Similarity (Liu et al., 2022).
P3	Updates Across Rounds	Facilitates debugging (Gill et al., 2023; Duan et al., 2023), fault tolerance (Balta et al., 2021; Yang et al., 2022a), reproducibility, transparency, data provenance, and lineage (Baracaldo et al., 2022).
P4	Metadata & Hyperparameters	Hyperparameter tuning (Zhou et al., 2023), tracking client resources for scheduling, clustering client priorities (Liu et al., 2022), clustering performance, client incentives, and client dropouts, monitoring payouts (Hu et al., 2022), and optimizing communication through pruning and quantization (Khan et al., 2024; Sun et al., 2023).

4.4 Workload Characterization and Caching

Based on our insight (I_2) from studying existing works (Lai et al., 2021b; Beutel et al., 2020; Gill et al., 2023; Kairouz et al., 2019), we recognize that FL follows an iterative process with sequential data access patterns, which can inform tailored caching policies. We first analyze the data processing needs of popular FL applications to develop a taxonomy of their non-training workloads (Gill et al., 2023; Duan et al., 2023; Baracaldo et al., 2022; Balta et al., 2021; Han et al., 2022) as shown in Table 1. Leveraging the insights gained from this study, we propose tailored caching policies, which also allow FLStore to be easily extended to new applications.

While a Serverless cache is scalable enough to store all metadata (Zhang et al., 2023a), FL metadata can reach several thousand Tera Bytes (TBs), so using tailored caching policies significantly reduces resource consumption and costs. For example, an FL job with 1000 clients and 1000 training rounds using the EfficientNet model (Tan & Le, 2021) would require 79 TBs of memory across 10098 Lambda functions, costing \$10.2 per hour or \$7357.8 per month. With FLStore's tailored policies, only 1.2 GB is consumed from just two Lambda functions, reducing costs to \$0.001 per hour or \$0.7 per month.

Table 1 also outlines the corresponding policies for each workload type in the taxonomy. Based on the chosen caching policy, FLStore distinguishes *hot data* from *cold data*, caching the former in serverless memory and asynchronously storing the latter in the persistent store. Next, we discuss each caching policy in detail:

P1: Single Client or Aggregated Model. This policy applies to tasks such as serving and testing a fully trained model (Li et al., 2020; Yu et al., 2020; Ezzeldin et al., 2023), and requires access to individual model updates for fine-tuning (Tang et al., 2022) or the final aggregated model (Hu et al., 2023). As previously explained (§ 2), the final aggregated model created by combining updates from participating clients after the FL training concludes is a model ready for deployment to consumers. To support these workloads, this policy requires caching the aggregated model for serving and inference. Additionally, any updates to this model are cached for workloads that involve comparative analysis or tracking of the aggregated model.

P2: All Client Model Updates per Round. Applications such as filtering malicious clients (Han et al., 2022), calculating clients' relative contributions (Sun et al., 2023), debugging (Gill et al., 2023; Duan et al., 2023), personalization (Tan et al., 2023b), and fault tolerance (Balta et al., 2021) fall under this category because they require iterative access to all client updates for specific rounds. When a request in this category is made for a particular client in a training round, we pre-cache all client updates for that round and the next, as these workloads require iterative access to clients' metadata from the requested round and possibly the next round. Metadata from previous rounds is unnecessary since these applications operate separately and incrementally for each round. Additionally, we keep the latest round cached, as workloads like scheduling, contribution calculation, and malicious client filtering run for each new round, requiring all client updates from that round.

Figure 5 illustrates two example workloads handled by FLStore. The first corresponds to this policy (P2), where a malicious filtering application is executed per round. In this example, data from round $Ri-1$ is old data that was required for a prior request, while round Ri was pre-cached during the execution of that prior request. As the current request for round Ri executes, FLStore evicts past data and pre-caches round $Ri+1$ for future requests, demonstrating how iterative non-training workloads in FL such as incentive distribution, scheduling, etc. have predictable data needs.

P3: Client Model Updates Across Rounds. Applications like reproducibility, checkpointing, transparency, data provenance, and lineage require access to a single client's model updates across consecutive rounds (Baracaldo et al., 2022). To support these, we cache the client's model update for the requested round and pre-cache that client's metadata from the previous and subsequent rounds. This is necessary because these workloads track performance, costs, or other metrics for a client over time or training rounds.

The second example in Figure 5 demonstrates a workflow for this policy (P3). In the example, the system is handling a request to track the improvement of client 2. Since tracking improvement is an iterative, round-based workload, the cache holds data from round $Ri-1$ (from a past request), while the current request is for round Ri, and the next is expected to be for round $Ri+1$. As FLStore processes the current request for round Ri, it evicts data from round $Ri-1$ and pre-caches client 2's updates for round $Ri+1$.

P4: Metadata and Hyperparameters. This includes applications such as hyperparameter tuning (Zhou et al., 2023), assessing data shift impacts on performance (Tan et al., 2023c), tracking client resource availability for scheduling, clustering by client priorities, and monitoring client payouts in FL. Communication optimization techniques like pruning, quantization, and contribution tracking for incentive distribution also require monitoring client optimization and contributions (Khan et al., 2024; Sun et al., 2023).

For these applications, we cache configuration and performance metadata, including hyperparameters, for the most recent R rounds, where R is tunable (default is 10). This ensures that up-to-date data is available for configuration and tuning, as older data may not be reliable. For instance, when scheduling client devices for training, current resource information is critical, as outdated data could cause clients to miss training deadlines.

Choice of policy. Since non-training workloads are iterative with predictable data needs (Baracaldo et al., 2022; Gill et al., 2023; Kairouz et al., 2019), we use the mappings in Table 1 to select the appropriate caching policy. While we continue to add new workloads, most fit into existing caching policies due to the iterative nature of FL. Future work includes incorporating a Reinforcement Learning with Human Feedback (RLHF) agent (Khan et al., 2024) to adapt policies for outlier workloads. Additional discussion on improving caching policy selection is in the Appendix D.

4.5 Data Persistence and Fault Tolerance

In this section, we discuss how FLStore ensures data persistence and fault tolerance against reclaimed serverless functions. The persistent store serves as a *cold data* repository for all data as protection against data loss and allows users to revisit data from past rounds. This data is crucial for post-training analysis, such as distributing incentives or visualizing convergence and loss trends. In the rare event that

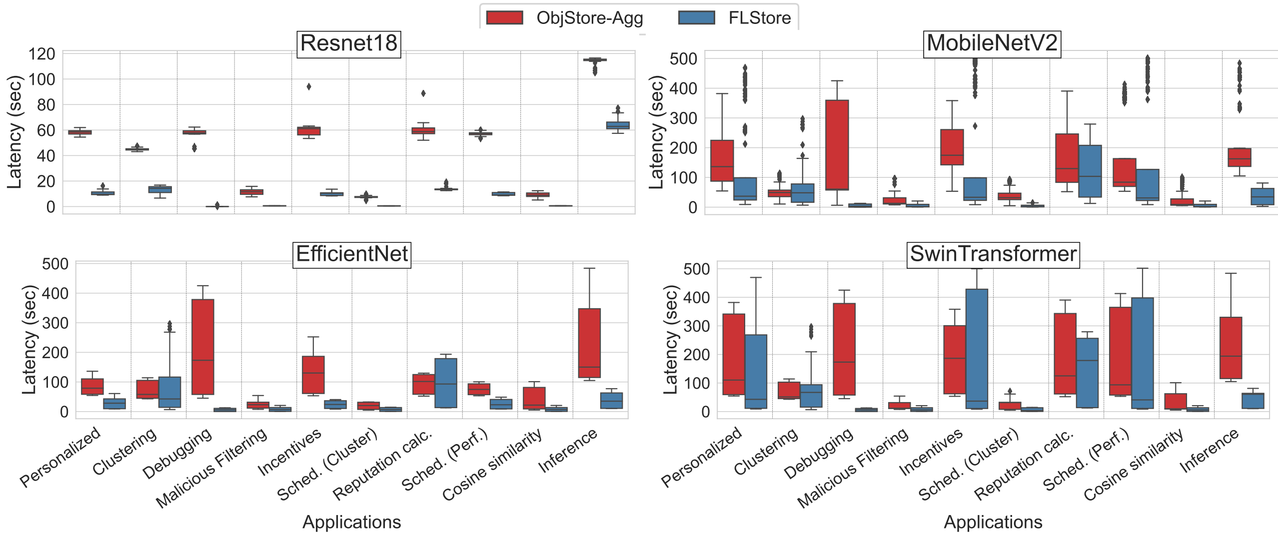

Figure 6. FLStore vs. Baseline **per request latency** comparison over 50 hours.

all cached functions fail, FLStore retrieves the necessary data from the persistent store, similar to state-of-the-art FL frameworks (FederatedAI, 2024; Beutel et al., 2020; IBM, 2020), ensuring comparable performance.

FLStore addresses fault tolerance through prevention and mitigation. We regularly ping cached functions to check their liveness, leveraging cloud platforms' default behavior (Zhang et al., 2023a). Cloud providers like AWS (Amazon Web Services, Inc., 2024a) cache functions at no cost, as long as they are regularly invoked (Zhang et al., 2023a). Pinging a function every minute, as recommended by InfiniStore (Zhang et al., 2023a), incurs a minimal monthly cost of \$0.0087 per instance and \$0.00000016 per million requests. Additionally, FLStore replicates functions to enhance reliability. Each primary function has k secondary copies to prevent stragglers and recovery delays. If the primary function fails to acknowledge a request or respond within a set time, the Request Tracker reroutes the request to a secondary instance. For added reliability, we recommend scaling function instances linearly with the number of requests, which minimizes cost and latency while preventing data re-fetching and cold starts.

Scalability over Serverless Functions FLStore's cache has two scalable facets: the cache size and handling more concurrent requests. To increase cache size, new serverless functions can be spawned to store additional data. For concurrent requests, new functions can be spawned which are simply copies of existing ones. Since serverless functions are highly scalable (Wang et al., 2020), scaling FLStore's cache is straightforward—new function instances are created as needed. FLStore can also spawn multiple instances to enhance scalability and performance.

5 Evaluation

5.1 Evaluation Setup

This section presents a proof-of-concept analysis to demonstrate the potential improvements brought by FLStore in latency and cost for non-training FL workloads. We show the effectiveness of FLStore by answering the questions:

- How well does FLStore reduce the latency of non-training workloads compared to state-of-the-art FL frameworks? (§ 5.2)
- How is the performance of FLStore's tailored caching policies compared to traditional ones? (§ 5.4)
- What is the overhead of FLStore components? (§ 5.5)
- How well does FLStore scale for parallel FL jobs? (§ A.1)
- How well does FLStore cope with faults? (§ A.2)

Baselines: We utilize baselines derived from the architectures of popular FL frameworks (Qi et al., 2024; He et al., 2020; IBM, 2020; Beutel et al., 2020), as depicted in Figure 2. Specifically, we deploy the cloud aggregator server on the $ml.m5.4xlarge$ instance of AWS SageMaker (Amazon Web Services, Inc., 2024b), a widely-used AWS service for managing non-training workloads such as inference and debugging (Liberty et al., 2020; Perrone et al., 2021; Das et al., 2020). AWS SageMaker connects with data storage options such as AWS S3 (Amazon Web Services, 2024b) for cloud object storage or AWS ElastiCache (Amazon Web Services, 2024a) for in-memory caching. Thus, our baselines are structured as follows: the first features an aggregator server on AWS SageMaker linked with AWS S3 (ObjStore-Agg), and the second connects AWS SageMaker with ElastiCache (Cache-Agg). In both setups, the data plane stores all FL metadata, while AWS SageMaker, forming the compute plane, processes non-training requests.

Workloads: We evaluate ten common non-training workloads, integral to many FL applications as shown in Table 1, across four models: EfficientNetV2 Small (Tan & Le, 2021), Resnet18 (He et al., 2016), MobileNet V3 Small (TorchVision Contributors, 2024), and SwinTransformerV2 tiny (Liu et al., 2021). Each model underwent FL training with 10 clients per round, selected from a pool of 250, across 1000 rounds or until convergence, following standard cross-device FL protocols in related studies (Lai et al., 2021b; Kairouz et al., 2019).

Metrics: Since throughput can be effectively managed through scaling, we focus on evaluating the latency and cost associated with communication and computation. We assess these metrics per request and their aggregated total for multiple requests over a period of several days, encompassing various non-training workload applications and models.

Implementation of FLStore: FLStore is implemented using the OpenFaas serverless framework (Ellis & Contributors, 2024). Function sizes are automatically adjusted to accommodate the varying model sizes, with larger function allocations (2 CPU cores and 4 GB of memory) configured for SwinTransformer and EfficientNet models and 1 CPU core and 2 GB of memory for Resnet 18 and MobileNet models. For both the baseline and FLStore setups, we use MinIO (MinIO, Inc., 2024) as our persistent data store, which is compatible with Amazon S3 (Amazon Web Services, 2024b). The MinIO configuration involves a 3-node cluster, with each node hosting six IronWolf 10TB HDDs (7200 RPM) and running default MinIO settings.

5.2 Latency Analysis

5.2.1 *FLStore vs Cloud Object Store*

We compare the latency and cost of baseline (ObjStore-Agg) and FLStore for ten workloads over 50 hours. Unlike ObjStore-Agg, FLStore co-locates the compute and data planes and utilizes tailored caching policies to cache relevant data in memory to reduce latency. However, in ObjStore-Agg the required data is fetched from the persistent store (data plane). Figure 6 shows the latency for communication and computation per request. FLStore shows significant improvements in latency with its locality-aware computation and caching policies. On average, FLStore decreases the latency by 55.14 seconds (50.75%) per request, with up to 363.5 seconds of maximum decrease (99.94%) in latency per request. It can be observed in Figure 6 that for some applications such as Incentives and Sched. (Perf.), SwinTransformer has a large distribution in the third quartile compared to ObjStore-Agg. However, FLStore still exhibits a lower median response time for these worklaods.

In distributed deep learning applications like FL, the main bottleneck is the increased communication time (Hashemi et al., 2019; Tang et al., 2023). Thus, we analyze total latency (computation vs. communication) for the baseline (ObjStore-Agg) and our solution (FLStore) over 50 hours and 3000 non-training requests across 10 workloads. ObjStore-Agg is heavily communication-bound, with communication latency accounting for an average of 98.9% of the total latency. FLStore mitigates this communication bottleneck improving the latency performance. With FLstore, we observe an average of 82.04% (35.50 second) decrease in latency for Resnet18, 47.33% (75.99 second) for MobileNet, 50.44% (100.18 second) for EfficientNet, and 20.45% (4.42 second) decrease in latency for SwinTransformer compared to ObjStore-Agg. *Due to space constraints, detailed results are provided in the Appendix.*

5.2.2 *FLStore vs In-Memory Cache*

We also compare FLStore with other popular in-memory caching solutions available by cloud frameworks. Classic caching solutions like Redis and Memcached included in AWS ElastiCache allow for such in-memory caching (Amazon Web Services, 2024a). Figure 8, shows the result of the comparison between FLStore and AWS ElastiCache with AWS SageMaker baseline (Cache-Agg) per request. It can be observed that per-request FLStore shows a 64.66% on average and a maximum of 84.41% reduction in latency when compared with Cache-Agg. This reduction in latency is brought by co-located compute and data planes and locality-aware request processing in FLStore.

For the total latency breakup analysis over 50 hours and across 3000 non-training requests, FLStore shows a decrease in the total time by 37.77% to 84.45%, amounting to a reduction of 191.65 accumulated hours for all requests. *When comparing both Cache-Agg and ObjStore-Agg on the same workloads, FLStore shows an average decrease in latency of* 71% *with ObjStore-Agg and* 64.66% *with Cache-Agg. The larger reduction with ObjStore-Agg is due to cloud object stores being slower than cloud caches.*

5.3 Cost Analysis

5.3.1 *FLStore vs Cloud Object Store*

In addition, we performed a per-request cost comparison across the ten selected workloads and 50 total hours. Figure 7 shows significant cost reduction with FLStore compared to ObjStore-Agg. The majority of this cost reduction stems from the reduced latency due to low data movement and the overall low computation cost of serverless functions for computation-light workloads. FLStore has an average

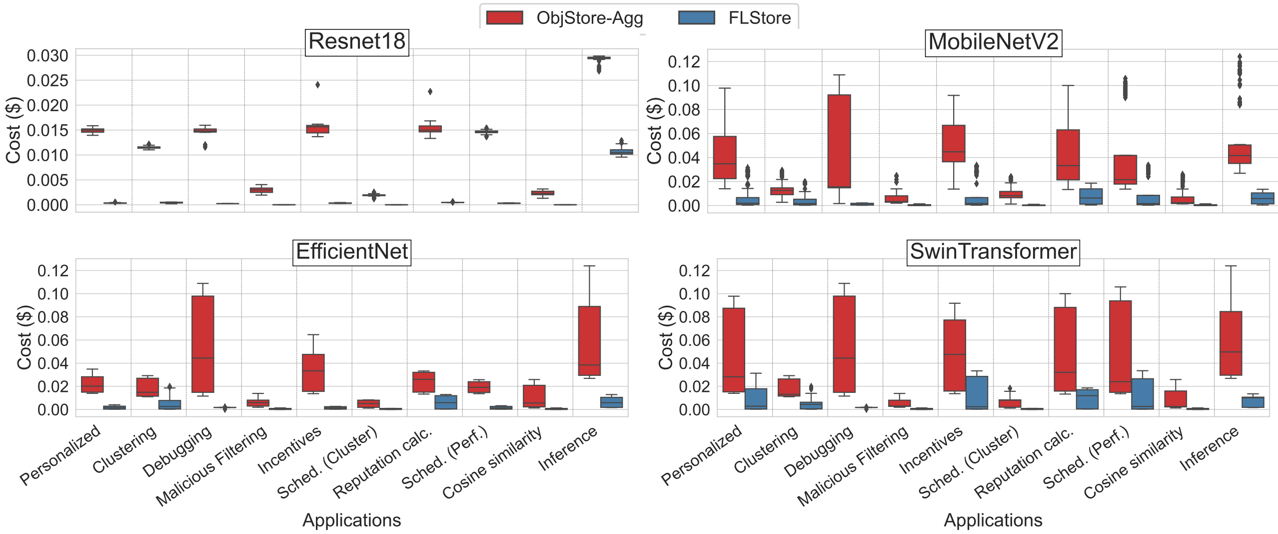

Figure 7. FLStore vs. ObjStore-Agg **per request cost** comparison over 50 hours.

cost decrease of 0.025 cents per request with a maximum decrease of 0.094 cents. On average, the cost of these applications in FLStore is 88.23% less than the cost of ObjStore-Agg baseline, with one application (Client Scheduling with Cosine Similarity for MobileNetV2) showing a 99.78% decrease in per request cost.

We also performed the total cost breakup analysis over 50 hours, 3000 total non-training requests, and 10 workloads, calculating both the communication and computation costs for ObjStore-Agg and FLStore. We observe that the majority of the cost for ObjStore-Agg stems from the communication bottleneck. Resnet18, EfficientNet, SwinTransformer, and MobileNet spend 87.46%, 76.96%, 53.32%, and 85.80% of their total latency respectively in communication. For the same settings, FLStore shows an average decrease of 94.73%, 92.72%, 77.83%, and 86.81% in costs for Resnet18, MobileNet, SwinTransformer, and EfficientNet models respectively. Thus, FLStore significantly reduces the data transfer costs by unifying the compute and data planes. *Due to space constraints, Figures for these results are provided in the Appendix.*

5.3.2 FLStore vs In-Memory Cache

We can observe in Figure 8 that keeping data in an in-memory cache such as ElastiCache is more costly in comparison to FLStore. FLStore shows an average decrease of 98.83% and a maximum decrease of 99.65% in cost per request compared to Cache-Agg. This stems from the increased communication latency and costs because Cache-Agg does not have co-located computational resources for processing the cached data so the data still needs to be transferred to another cloud service such as AWS SageMaker (Amazon Web Services, Inc., 2024b).

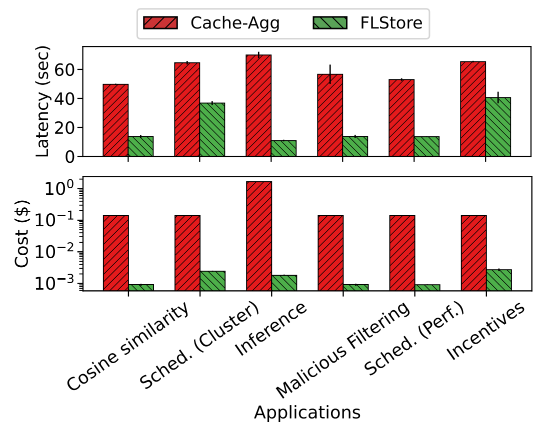

Figure 8. Cache-Agg baseline vs. FLStore variants: **Per request latency** (top) and **cost** (bottom) over 50 hours.

For the cost breakup analysis over 50 hours and across 3000 non-training requests, FLStore shows a reduction of 98.12% to 99.89%, resulting in accumulated savings of \$7047.16. Cloud caches tend to be more expensive than cloud object stores, which is why FLStore demonstrates an average cost decrease of 98.83% when compared to Cache-Agg, and a 92.45% decrease in cost when compared to ObjStore-Agg. *The total time and total cost breakup analysis for both ObjStore-Agg and Cache-Agg is provided in the Appendix B.*

5.4 FLStore vs Traditional Caching Policies

We introduce traditional caching strategies like Least Recently Used (LRU) and First In First Out (FIFO) in FLStore, alongside our tailored workload-specific policies derived from a developed taxonomy. We evaluate these against FLStore and its variant, FLStore-limited which depicts a limited storage availability scenario having half the stor-

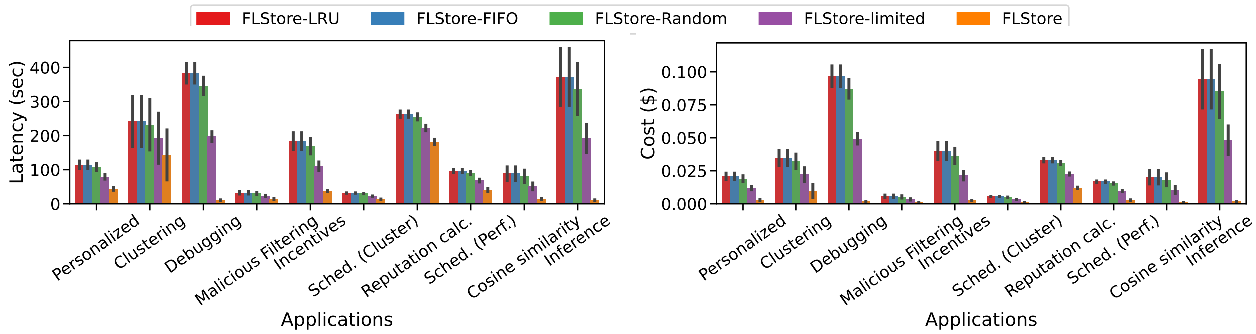

Figure 9. **Per request latency** (left) and **cost** (right) comparison of various caching policies in FLStore over 50 hours.

Table 2. Cache Policy Performance Across Workloads

Applications	Cache Policy	Hits	Misses	Total	Hit %
(Lai et al., 2021b), (Liu et al., 2022), (Tan et al., 2023b), (Sun et al., 2023)	FLStore (P2)	19999	1	20000	0.99
	FIFO	0	20000	20000	0
	LFU	0	20000	20000	0
	LRU	0	20000	20000	0
(Gill et al., 2023), (Baracaldo et al., 2022), (Han et al., 2022), (Duan et al., 2023)	FLStore (P3)	63	1	64	0.98
	FIFO	0	64	64	0
	LFU	0	64	64	0
	LRU	0	64	64	0
(Khan et al., 2024), (Khodak et al., 2021), (Balta et al., 2021), (Lai et al., 2021b)	FLStore (P4)	20000	0	20000	1
	FIFO	0	20000	20000	0
	LFU	0	20000	20000	0
	LRU	0	20000	20000	0

age capacity of FLStore. As depicted in Figure 9, both FLStore-LRU and FLStore-FIFO show similar performance due to their generic nature, unlike the taxonomy-driven policies of FLStore and FLStore-limited, which preemptively cache relevant data for imminent requests, thereby markedly reducing latency and costs. For instance, the debugging workload in Table 1 mandates the P2 caching policy, directing FLStore to cache the current training round's metadata rather than outdated information, leading to a significant reduction in debugging latency by 97.15% (380 seconds) and cost savings of \$0.1 per request. Notably, even with limited capacity, FLStore-limited surpasses traditional policies. *These improvements are substantial, especially given that the non-training requests can range from thousands to hundreds of thousands.*

We evaluated FLStore's performance against traditional caching policies like LFU, LRU, and FIFO using a simulated trace for non-training FL requests, crafted from FL jobs for 10 clients each round from a pool of 250 over 2000 rounds on popular FL frameworks like Oort (Lai et al., 2021b), FedDebug (Gill et al., 2023), REFL (Abdelmoniem et al., 2023), and others (Tan et al., 2023b; Baracaldo et al., 2022; Khodak et al., 2021) that utilize non-training applications. As shown in Table 2, FLStore's caching policy achieves a 99% hit rate for Clustering (Liu et al., 2022) and Personalized FL (Tan et al., 2023b) under the P2 caching policy and 98% hit rate for tasks under the P3 caching policy (Gill et al., 2023; Duan et al., 2023; Baracaldo et al., 2022) with similar results observed for the P4 policy workloads (Khan et al., 2024; Khodak et al., 2021; Balta et al., 2021). In contrast, traditional policies consistently register a 0% hit rate across all tested scenarios.

Ablation study. We also evaluated FLStore variants without tailored caching policies: FLStore-Random and FLStore-Static. FLStore-Random, using random caching policy selection regardless of workload, shows lower latency in some cases, as depicted in Figure 9. However, for critical workloads like Scheduling and Incentivization, its performance aligns with FLStore-FIFO and FLStore-LRU. Comparison with FLStore-Static is detailed in Appendix C.

5.5 Overhead of FLStore's components

The Cache Engine and Request Tracker can run co-located with the aggregator service or locally, with minimal overhead. We measure the overhead for 1000 concurrent non-training requests. The Request Tracker uses less than 0.19 MB of memory, and the Cache Engine uses 0.6 MB. Scaling to 100000 requests increases memory usage to 20.3 MB and 63.2 MB, respectively. In both cases, the time to retrieve, use, or remove data from these services is **under one millisecond**. The minimal overhead of the Cache Engine and Request Tracker allows them to be run locally, on the aggregator server, or even on a serverless function.

6 Conclusion

This paper introduces FLStore, an efficient and cost-effective storage solution with locality-aware processing for FL's communication-heavy non-training workloads. Our experiments demonstrate that FLStore is efficient and cost-effective compared to other caching and cloud storage solutions. FLStore is scalable and robust and can incorporate new workloads by adding a new caching policy.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

Abdelmoniem, A. M., Sahu, A. N., Canini, M., and Fahmy, S. A. Refl: Resource-efficient federated learning. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, pp. 215–232, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394871. doi: 10.1145/3552326.3567485. URL https://doi.org/10.1145/3552326.3567485.

Amazon Web Services. Amazon elasticache, 2024a. URL https://aws.amazon.com/elasticache/. Accessed: 2024-05-18.

Amazon Web Services. Amazon simple storage service api reference, 2024b. URL https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html. Accessed: 2024-05-17.

Amazon Web Services, Inc. Aws lambda: Serverless compute. https://aws.amazon.com/lambda/, 2024a. Accessed: 2024-03-22.

Amazon Web Services, Inc. Amazon SageMaker. https://aws.amazon.com/sagemaker/, 2024b. Accessed: yyyy-mm-dd.

Balta, D., Sellami, M., Kuhn, P., Schöpp, U., Buchinger, M., Baracaldo, N., Anwar, A., Ludwig, H., Sinn, M., Purcell, M., and Altakrouri, B. Accountable federated machine learning in government: Engineering and management insights. In *Electronic Participation: 13th IFIP WG 8.5 International Conference, EPart 2021, Granada, Spain, September 7–9, 2021, Proceedings*, pp. 125–138, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-82823-3. doi: 10.1007/978-3-030-82824-0_10. URL https://doi.org/10.1007/978-3-030-82824-0_10.

Baracaldo, N., Anwar, A., Purcell, M., Rawat, A., Sinn, M., Altakrouri, B., Balta, D., Sellami, M., Kuhn, P., Schopp, U., and Buchinger, M. Towards an accountable and reproducible federated learning: A factsheets approach, 2022.

Beutel, D. J., Topal, T., Mathur, A., Qiu, X., Fernandez-Marques, J., Gao, Y., Sani, L., Kwing, H. L., Parcollet, T., Gusmão, P. P. d., and Lane, N. D. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390*, 2020.

Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C., Konečný, J., Mazzocchi, S., McMahan, B., et al. Towards federated learning at scale: System design. *Proceedings of Machine Learning and Systems*, 1:374–388, 2019.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Caldas, S., Duddu, S. M. K., Wu, P., Li, T., Konečný, J., McMahan, H. B., Smith, V., and Talwalkar, A. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097*, 2018.

Chai, Z., Ali, A., Zawad, S., Truex, S., Anwar, A., Baracaldo, N., Zhou, Y., Ludwig, H., Yan, F., and Cheng, Y. Tifl: A tier-based federated learning system. *To appear in ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2020.

Chai, Z., Chen, Y., Anwar, A., Zhao, L., Cheng, Y., and Rangwala, H. Fedat: a high-performance and communication-efficient federated learning system with asynchronous tiers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–16, 2021.

Chen, D., Gao, D., Kuang, W., Li, Y., and Ding, B. pfl-bench: A comprehensive benchmark for personalized federated learning, 2022.

Das, P., Ivkin, N., Bansal, T., Rouesnel, L., Gautier, P., Karnin, Z., Dirac, L., Ramakrishnan, L., Perunicic, A., Shcherbatyi, I., Wu, W., Zolic, A., Shen, H., Ahmed, A., Winkelmolen, F., Miladinovic, M., Archembeau, C., Tang, A., Dutt, B., Grao, P., and Venkateswar, K. Amazon sagemaker autopilot: a white box automl solution at scale. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, DEEM '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380232. doi: 10.1145/3399579.3399870. URL https://doi.org/10.1145/3399579.3399870.

Desai, H. B., Ozdayi, M. S., and Kantarcioglu, M. Blockfla: Accountable federated learning via hybrid blockchain architecture. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, CODASPY '21, pp. 101–112, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450381437. doi: 10.1145/3422337.3447837. URL https://doi.org/10.1145/3422337.3447837.

Duan, M., Liu, D., Ji, X., Liu, R., Liang, L., Chen, X., and Tan, Y. Fedgroup: Accurate federated learning via decomposed similarity-based clustering. 2021.

Duan, S., Liu, C., Han, P., Jin, X., Zhang, X., Xiang, X., Pan, H., and Yan, X. Fed-dnn-debugger: Automatically debugging deep neural network models in federated learning. 2023, jan 2023. ISSN 1939-0114. doi: 10.1155/2023/5968168. URL https://doi.org/10.1155/2023/5968168.

Ellis, A. and Contributors, O. Openfaas. https://github.com/openfaas/faas, 2024. Accessed: 2024-05-20.

Ezzeldin, Y. H., Yan, S., He, C., Ferrara, E., and Avestimehr, A. S. Fairfed: Enabling group fairness in federated learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 7494–7502, 2023.

Face, H. Tinyllama-1.1b-step-50k-105b. https://huggingface.co/TinyLlama/TinyLlama-1.1B-step-50K-105b, 2024. Accessed: 2024-05-21.

FederatedAI. Fate: An industrial grade federated learning framework, 2024. URL https://github.com/FederatedAI/FATE. GitHub repository.

Gade, K., Geyik, S. C., Kenthapadi, K., Mithal, V., and Taly, A. Explainable ai in industry. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, pp. 3203–3204, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362016. doi: 10.1145/3292500.3332281. URL https://doi.org/10.1145/3292500.3332281.

Ghosh, A., Chung, J., Yin, D., and Ramchandran, K. An efficient framework for clustered federated learning. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 19586–19597. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/e32cc80bf07915058ce90722ee17bb71-Paper.pdf.

Gill, W., Anwar, A., and Gulzar, M. A. Feddebug: Systematic debugging for federated learning applications. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 512–523, 2023. doi: 10.1109/ICSE48619.2023.00053.

Google Cloud. *Google Cloud Storage Documentation*, 2024. URL https://cloud.google.com/storage/docs. Accessed: 2024-07-24.

Grafberger, A., Chadha, M., Jindal, A., Gu, J., and Gerndt, M. Fedless: Secure and scalable federated learning using serverless computing. In *2021 IEEE International Conference on Big Data (Big Data)*, pp. 164–173, 2021. doi: 10.1109/BigData52589.2021.9672067.

Han, J., Khan, A. F., Zawad, S., Anwar, A., Angel, N. B., Zhou, Y., Yan, F., and Butt, A. R. Tiff: Tokenized incentive for federated learning. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pp. 407–416, 2022. doi: 10.1109/CLOUD55607.2022.00064.

Hashemi, S. H., Abdu Jyothi, S., and Campbell, R. Tictac: Accelerating distributed deep learning with communication scheduling. In Talwalkar, A., Smith, V., and Zaharia, M. (eds.), *Proceedings of Machine Learning and Systems*, volume 1, pp. 418–430, 2019. URL https://proceedings.mlsys.org/paper_files/paper/2019/file/94cb28874a503f34b3c4a41bddcea2bd-Paper.pdf.

He, C., Li, S., So, J., Zhang, M., Wang, H., Wang, X., Vepakomma, P., Singh, A., Qiu, H., Shen, L., Zhao, P., Kang, Y., Liu, Y., Raskar, R., Yang, Q., Annavaram, M., and Avestimehr, S. Fedml: A research library and benchmark for federated machine learning. *CoRR*, abs/2007.13518, 2020. URL https://arxiv.org/abs/2007.13518.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hu, C., Liang, H. H., Han, X. M., Liu, B. A., Cheng, D. Z., and Wang, D. Spread: Decentralized model aggregation for scalable federated learning. In *Proceedings of the 51st International Conference on Parallel Processing*, ICPP '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450397339. doi: 10.1145/3545008.3545030. URL https://doi.org/10.1145/3545008.3545030.

Hu, M., Wu, D., Zhou, Y., Chen, X., and Chen, M. Incentive-aware autonomous client participation in federated learning. *IEEE Transactions on Parallel and Distributed Systems*, 33(10):2612–2627, 2022. doi: 10.1109/TPDS.2022.3148113.

IBM. IBM Federated Learning Framework. https://github.com/IBM/federated-learning-lib, 2020. [Online; accessed 05-June-2022].

IBM. IBM Federated Learning Framework Contributors. https://github.com/IBM/federated-learning-lib/graphs/contributors, 2022. [Online; accessed 05-July-2022].

Jiang, J., Gan, S., Liu, Y., Wang, F., Alonso, G., Klimovic, A., Singla, A., Wu, W., and Zhang, C. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pp. 857–871, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3459240. URL https://doi.org/10.1145/3448016.3459240.

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I., and Patterson, D. A. Cloud programming simplified: A berkeley view on serverless computing, 2019. URL https://arxiv.org/abs/1902.03383.

Kairouz, P., McMahan, H. B., Avent, A., Bellet, A., Bennis, M., Bhagoji, A. N., Bonawitz, K., Charles, C., Cormode, G., Cummings, R., et al. Advances and open problems in federated learning. *Foundations and Trends in Machine Learning*, 12(3-4):1–357, 2019.

Khan, A. F., Li, Y., Wang, X., Haroon, S., Ali, H., Cheng, Y., Butt, A. R., and Anwar, A. Towards cost-effective and resource-aware aggregation at edge for federated learning. In *2023 IEEE International Conference on Big Data (BigData)*, pp. 690–699, 2023. doi: 10.1109/BigData59044.2023.10386691.

Khan, A. F., Khan, A. A., Abdelmoniem, A. M., Fountain, S., Butt, A. R., and Anwar, A. FLOAT: Federated learning optimizations with automated tuning. In *Nineteenth European Conference on Computer Systems (EuroSys '24)*, pp. 19, New York, NY, USA, 2024. ACM. doi: 10.1145/3627703.3650081. URL https://doi.org/10.1145/3627703.3650081.

Khodak, M., Tu, R., Li, T., Li, L., Balcan, M.-F. F., Smith, V., and Talwalkar, A. Federated hyperparameter tuning: Challenges, baselines, and connections to weight-sharing. In Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 19184–19197. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/a0205b87490c847182672e8d371e9948-Paper.pdf.

Krizhevsky, A. Learning multiple layers of features from tiny images. Technical Report TR-2009, University of Toronto, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Lai, F., Dai, Y., Zhu, X., Madhyastha, H. V., and Chowdhury, M. Fedscale: Benchmarking model and system performance of federated learning. In *Proceedings of the First Workshop on Systems Challenges in Reliable and Secure Federated Learning*, pp. 1–3, 2021a.

Lai, F., Zhu, X., Madhyastha, H. V., and Chowdhury, M. Oort: Efficient federated learning via guided participant selection. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pp. 19–35, 2021b.

Li, A., Zhang, L., Wang, J., Tan, J., Han, F., Qin, Y., Freris, N. M., and Li, X.-Y. Efficient federated-learning model debugging. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 372–383, 2021. doi: 10.1109/ICDE51399.2021.00039.

Li, T., Sahu, A. K., Talwalkar, A., and Smith, V. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020. doi: 10.1109/MSP.2020.2975749.

Liberty, E., Karnin, Z., Xiang, B., Rouesnel, L., Coskun, B., Nallapati, R., Delgado, J., Sadoughi, A., Astashonok, Y., Das, P., Balioglu, C., Chakravarty, S., Jha, M., Gautier, P., Arpin, D., Januschowski, T., Flunkert, V., Wang, Y., Gasthaus, J., Stella, L., Rangapuram, S., Salinas, D., Schelter, S., and Smola, A. Elastic machine learning algorithms in amazon sagemaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pp. 731–737, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3386126. URL https://doi.org/10.1145/3318464.3386126.

Liu, J., Lai, F., Dai, Y., Akella, A., Madhyastha, H., and Chowdhury, M. Auxo: Heterogeneity-mitigating federated learning via scalable client clustering. *arXiv preprint arXiv:2210.16656*, 2022.

Liu, Y., Su, L., Joe-Wong, C., Ioannidis, S., Yeh, E., and Siew, M. Cache-enabled federated learning systems. In

Proceedings of the Twenty-Fourth International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing, MobiHoc '23, pp. 1–11, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399265. doi: 10.1145/3565287.3610264. URL https://doi.org/10.1145/3565287.3610264.

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., and Guo, B. Swin transformer v2: Scaling up capacity and resolution. *arXiv preprint arXiv:2111.09883*, 2021.

Ludwig, H., Baracaldo, N., Thomas, G., Zhou, Y., Anwar, A., Rajamoni, S., Ong, Y., Radhakrishnan, J., Verma, A., Sinn, M., et al. Ibm federated learning: an enterprise framework white paper v0. 1. *arXiv preprint arXiv:2007.10987*, 2020.

McMahan, B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pp. 1273–1282. PMLR, 2017.

MinIO, Inc. Minio: High performance, kubernetes native object storage. https://min.io/, 2024. Accessed: 2024-03-22.

Mohseni, S., Zarei, N., and Ragan, E. D. A multidisciplinary survey and framework for design and evaluation of explainable ai systems. 11(3–4), 2021. ISSN 2160-6455. doi: 10.1145/3387166. URL https://doi.org/10.1145/3387166.

Nguyen, J., Malik, K., Zhan, H., Yousefpour, A., Rabbat, M. G., Malek, M., and Huba, D. Federated learning with buffered asynchronous aggregation. *CoRR*, abs/2106.06639, 2021. URL https://arxiv.org/abs/2106.06639.

NinjaOne. Double data rate memory: A generational overview of ram, 2024. URL https://www.ninjaone.com/blog/double-data-rate-memory/. Accessed: 2024-08-30.

Perrone, V., Shen, H., Zolic, A., Shcherbatyi, I., Ahmed, A., Bansal, T., Donini, M., Winkelmolen, F., Jenatton, R., Faddoul, J. B., Pogorzelska, B., Miladinovic, M., Kenthapadi, K., Seeger, M., and Archambeau, C. Amazon sagemaker automatic model tuning: Scalable gradient-free optimization. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, pp. 3463–3471, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383325. doi: 10.1145/3447548.3467098. URL https://doi.org/10.1145/3447548.3467098.

Qi, S., Ramakrishnan, K. K., and Lee, M. Lifl: A lightweight, event-driven serverless platform for federated learning, 2024. URL https://arxiv.org/abs/2405.10968.

Reisizadeh, A., Mokhtari, A., Hassani, H., Jadbabaie, A., and Pedarsani, R. Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization. In Chiappa, S. and Calandra, R. (eds.), *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pp. 2021–2031. PMLR, 26–28 Aug 2020. URL https://proceedings.mlr.press/v108/reisizadeh20a.html.

Ruan, Y. and Joe-Wong, C. Fedsoft: Soft clustered federated learning with proximal local updating. In *AAAI*, 2022.

Shlezinger, N., Chen, M., Eldar, Y. C., Poor, H. V., and Cui, S. Uveqfed: Universal vector quantization for federated learning. *IEEE Transactions on Signal Processing*, 69: 500–514, 2021. doi: 10.1109/TSP.2020.3046971.

Sun, Q., Li, X., Zhang, J., Xiong, L., Liu, W., Liu, J., Qin, Z., and Ren, K. Shapleyfl: Robust federated learning based on shapley value. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, pp. 2096–2108, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701030. doi: 10.1145/3580305.3599500. URL https://doi.org/10.1145/3580305.3599500.

Tan, A. Z., Yu, H., Cui, L., and Yang, Q. Towards personalized federated learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

Tan, A. Z., Yu, H., Cui, L., and Yang, Q. Towards personalized federated learning. *IEEE Transactions on Neural Networks and Learning Systems*, 34(12):9587–9603, 2023a. doi: 10.1109/TNNLS.2022.3160699.

Tan, A. Z., Yu, H., Cui, L., and Yang, Q. Towards personalized federated learning. *IEEE Transactions on Neural Networks and Learning Systems*, 34(12):9587–9603, 2023b. doi: 10.1109/TNNLS.2022.3160699.

Tan, M. and Le, Q. Efficientnetv2: Smaller models and faster training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

Tan, Y., Chen, C., Zhuang, W., Dong, X., Lyu, L., and Long, G. Is heterogeneity notorious? taming heterogeneity to handle test-time shift in federated learning. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 27167–27180. Curran Associates, Inc., 2023c. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/565f995643da6329cec701f26f8579f5-Paper-Conference.pdf.

Tang, X., Guo, S., and Guo, J. Personalized federated learning with clustered generalization. *ArXiv*, abs/2106.13044, 2021.

Tang, X., Guo, S., and Guo, J. Personalized federated learning with contextualized generalization. In Raedt, L. D. (ed.), *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pp. 2241–2247. International Joint Conferences on Artificial Intelligence Organization, 7 2022. doi: 10.24963/ijcai.2022/311. URL https://doi.org/10.24963/ijcai.2022/311. Main Track.

Tang, Z., Shi, S., Wang, W., Li, B., and Chu, X. Communication-efficient distributed deep learning: A comprehensive survey, 2023. URL https://arxiv.org/abs/2003.06307.

TorchVision Contributors. Mobilenet v3 small. https://pytorch.org/vision/main/models/generated/torchvision.models.mobilenet_v3_small.html, 2024. Accessed: 2024-05-20.

Wang, A., Zhang, J., Ma, X., Anwar, A., Rupprecht, L., Skourtis, D., Tarasov, V., Yan, F., and Cheng, Y. InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pp. 267–281, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/wang-ao.

Yang, Q., Liu, Y., Chen, T., and Tong, Y. Federated machine learning: Concept and applications. *ACM Trans. Intell. Syst. Technol.*, 10(2), jan 2019. ISSN 2157-6904. doi: 10.1145/3298981. URL https://doi.org/10.1145/3298981.

Yang, X., Zhao, Y., Chen, Q., Yu, Y., Du, X., and Guizani, M. Accountable and verifiable secure aggregation for federated learning in iot networks. *IEEE Network*, 36(5): 173–179, 2022a. doi: 10.1109/MNET.001.2200214.

Yang, Y., Zhao, L., Li, Y., Zhang, H., Li, J., Zhao, M., Chen, X., and Li, K. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pp. 768–781, New York, NY, USA, 2022b. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507709. URL https://doi.org/10.1145/3503222.3507709.

Yu, H., Liu, Z., Liu, Y., Chen, T., Cong, M., Weng, X., Niyato, D., and Yang, Q. A fairness-aware incentive scheme for federated learning. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, pp. 393–399, 2020.

Yu, S., Nguyen, P., Anwar, A., and Jannesari, A. Heterogeneous federated learning using dynamic model pruning and adaptive gradient, 2023.

Zhang, J., Wang, A., Ma, X., Carver, B., Newman, N. J., Anwar, A., Rupprecht, L., Skourtis, D., Tarasov, V., Yan, F., and Cheng, Y. Infinistore: Elastic serverless cloud storage, 2023a.

Zhang, J., Wang, A., Ma, X., Carver, B., Newman, N. J., Anwar, A., Rupprecht, L., Tarasov, V., Skourtis, D., Yan, F., and Cheng, Y. Infinistore: Elastic serverless cloud storage. *Proc. VLDB Endow.*, 16(7):1629–1642, mar 2023b. ISSN 2150-8097. doi: 10.14778/3587136.3587139. URL https://doi.org/10.14778/3587136.3587139.

Zhou, Y., Ram, P., Salonidis, T., Baracaldo, N., Samulowitz, H., and Ludwig, H. Single-shot general hyper-parameter optimization for federated learning. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=3RhuF8foyPW.

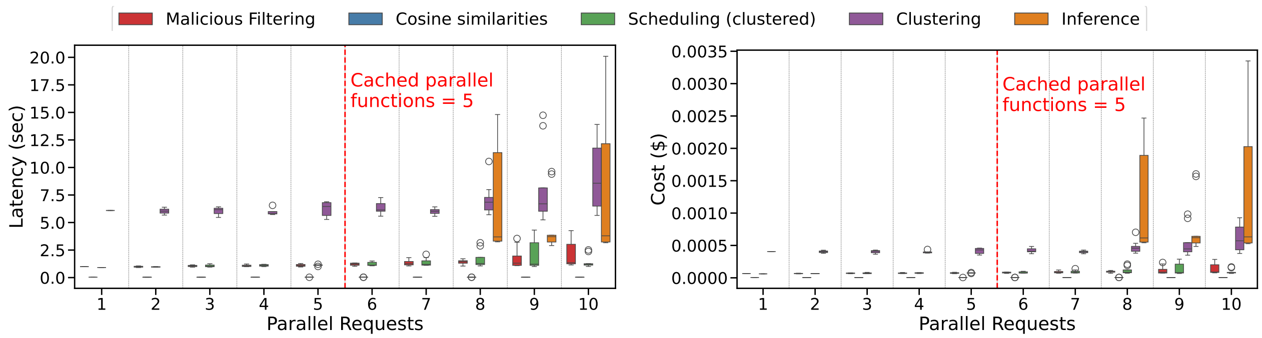

Figure 10. FLStore **scalability** for iteratively increasing parallel requests and 5 parallel cached functions.

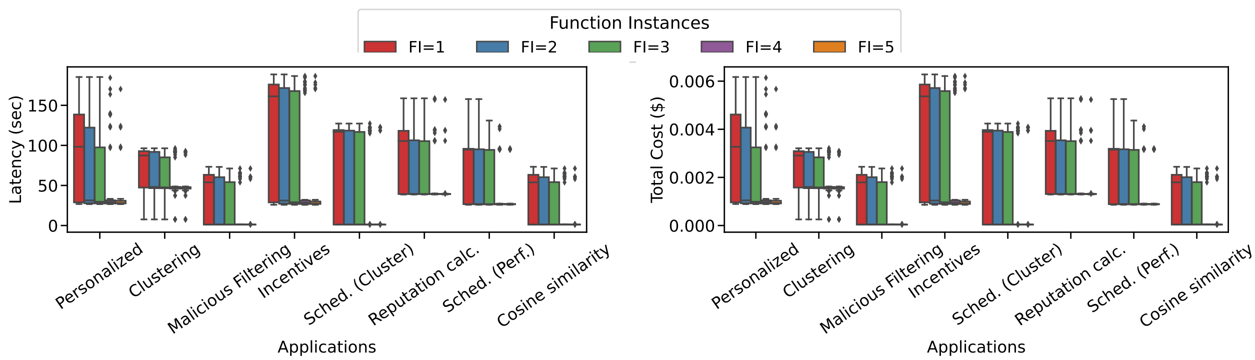

Figure 11. FLStore latency and cost per request over 50 hours with varying function instances (FI) for **fault tolerance**.

A Supplementary Features of FLStore

Modular design FLStore's modular design enables seamless integration with existing FL frameworks without modifying clients or aggregators. Training can proceed unchanged, while client updates and metadata received by the aggregator are asynchronously relayed to FLStore's cache. FLStore then serves as a scalable and efficient storage solution, handling non-training tasks.

Multi-tenancy The serverless computing paradigm inherently provides isolation (Amazon Web Services, Inc., 2024a; Ellis & Contributors, 2024), allowing each user to create an isolated cache on the same FLStore instance. This enables customized caching policies per non-training workload/application, allowing FLStore to handle requests from multiple users simultaneously.

A.1 Scalability of FLStore

To demonstrate FLStore's scalability, we simulated increasing concurrent non-training requests, with FLStore maintaining 5 cached function instances (red line, Figure 10). We varied the number of concurrent client requests from 1 to 10 across six representative non-training workloads using the EfficientNet model. As shown in Figure 10, latency and cost remain nearly constant when concurrent requests are equal to or fewer than the cached functions. For 1 to 5 requests, the average latencies were 1.05 seconds for Malicious Filtering, 0.031 seconds for Cosine Similarities, 1.039 seconds for Scheduling (clustered), and 6.067 seconds for Clustering. Even with 6 and 7 requests, there was minimal increase in latency or cost. For 8 to 10 requests, latencies start increasing. However, this can be easily mitigated by scaling cached functions (creating copies of already cached functions) linearly with the number of requests, which incurs minimal additional cost, as discussed next.

A.2 Fault Tolerance

We evaluated FLStore's fault tolerance by testing ten different workloads using the EfficientNet model and sending 3000 requests over 50 hours. Faults (function reclamations) were generated based on the Zipfian distribution, observed in measurement studies on AWS Lambda (Wang et al., 2020). Figure 11 shows that with only 1 function instance, latency and cost are highest, with improvement as the number of replicas increases. With 3 to 5 function instances, latency and cost remain nearly constant, despite faults. In particular, 3 instances reduce latency by 50-150 seconds per request compared to a single instance in the face of faults.

Interestingly the cost of maintaining function replicas is neg-

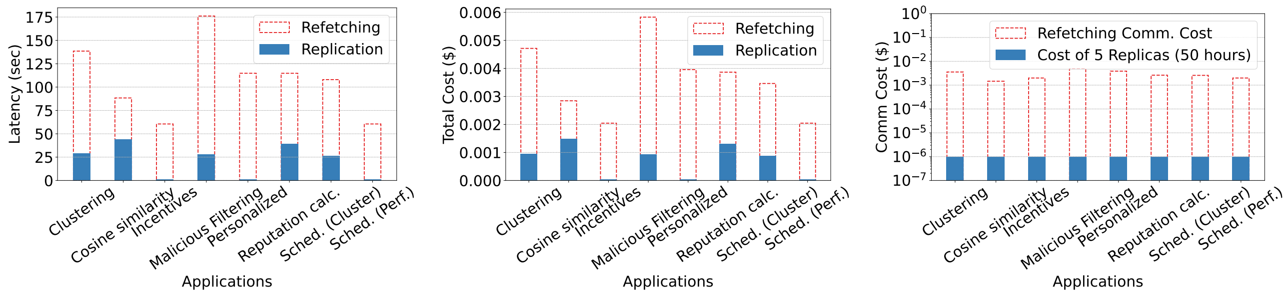

Figure 12. Overall latency and cost comparison of replication vs. re-fetching (first and second from left), and communication cost comparison (rightmost).

ligible compared to the overhead and cost of re-computation and communication due to faults. For 50 hours and 3000 requests, maintaining 5 replicas costs just \$0.003, or \$0.000001 per non-training request served (Figure 12). In contrast, fewer instances lead to higher overhead and costs while maintaining more replicas reduces these costs by up to 3000×. Notably, we did not evaluate the impact of regular pinging, as this has already been explored in prior works (Zhang et al., 2023a; Wang et al., 2020).

B Latency and Cost Performance Breakup

To identify the bottleneck, we broke up the accumulated latency between communication and computation time over 50 hours of experiments for the 10 different workloads.

B.1 FLStore vs ObjStore-Agg

Figure 13 shows the results with both communication and computation time for the ObjStore-Agg and only computation time for FLStore because communication time for FLStore is negligible in comparison due to co-located data and compute planes. The major bottleneck in ObjStore-Agg is Communication, in comparison the I/O time from memory to CPU is negligible (NinjaOne, 2024). For some workloads such as Inference, Debugging, and Scheduling, the difference between computation and communication times is significant. During inference communication consumes an average of 98.9% of time. This shows that current methodologies for computing non-training workloads for distributed learning techniques such as FL are significantly communication-bound. Thus, the reduction in communication times as brought by FLStore significantly improves the efficiency performance, which can be observed in Figure 13. We can also observe that FLStore provides significant improvements for smaller models, which is why FLStore is suitable in cross-device FL settings (Kairouz et al., 2019; Abdelmoniem et al., 2023). Across 50 hours and 3000 total requests we see Resnet18 with an average 82.04% (35.50 second) decrease in latency, MobileNet has an average 47.33% (75.99 second) decrease in latency, EfficientNet has an average 50.44% (100.18 second) decrease in latency, and Swin has an average 20.45% (4.42 second) decrease in latency. Thus, FLStore can significantly improve non-training tasks in FL with reduced latency. We next observe the reduction in total cost with FLStore.

We perform the same breakup analysis on the costs in Figure 14, showing both the communication and computation costs for ObjStore-Agg and computation costs for FLStore where communication costs are negligible. We can observe that the majority of the cost stems from the I/O (including communication) of data relevant to the non-training workloads. Resnet18, EfficientNet, and MobileNet spend 87.46%, 76.96%, and 85.80% of their total time respectively in I/O, and SwinTransformer spends 53.32% percent of its total time in I/O. Thus, by reducing the I/O time and data transfer costs FLStore provides a cost-effective solution for offloading the non-training workloads in FL. Across 50 hours and 3000 total requests we see that Resnet18, MobileNet, and EfficientNet show a 94.73%, 92.72%, and 86.81% average decrease in cost respectively, and SwinTransformer has an average 77.83% reduction in cost.

B.2 FLStore vs In-Memory Cache

We also perform the total cost breakup analysis over 50 hours, 3000 total non-training requests, and 10 workloads, calculating both the communication and computation costs for Cache-Agg and FLStore. Results for this analysis are shown in Figure 15 FLStore decreases the total time by $37.77\% - 84.45\%$ amounting to 191.65 accumulated hours reduced for all requests and a $98.12\% - 99.89\%$ decrease in total cost resulting in a reduction of \$7047.16 accumulated dollar costs for all 3000 requests across 50 hours. To compare both (Cache-Agg and ObjStore-Agg) on the same workloads tested with Cache-Agg, FLStore shows an average decrease in latency of 71% with ObjStore-Agg and 64% with Cache-Agg, the decreases with ObjStore-Agg is larger as cloud object stores are slower than cloud caches. However, in terms of costs cloud caches are more expensive than cloud object stores, which is why for the workloads

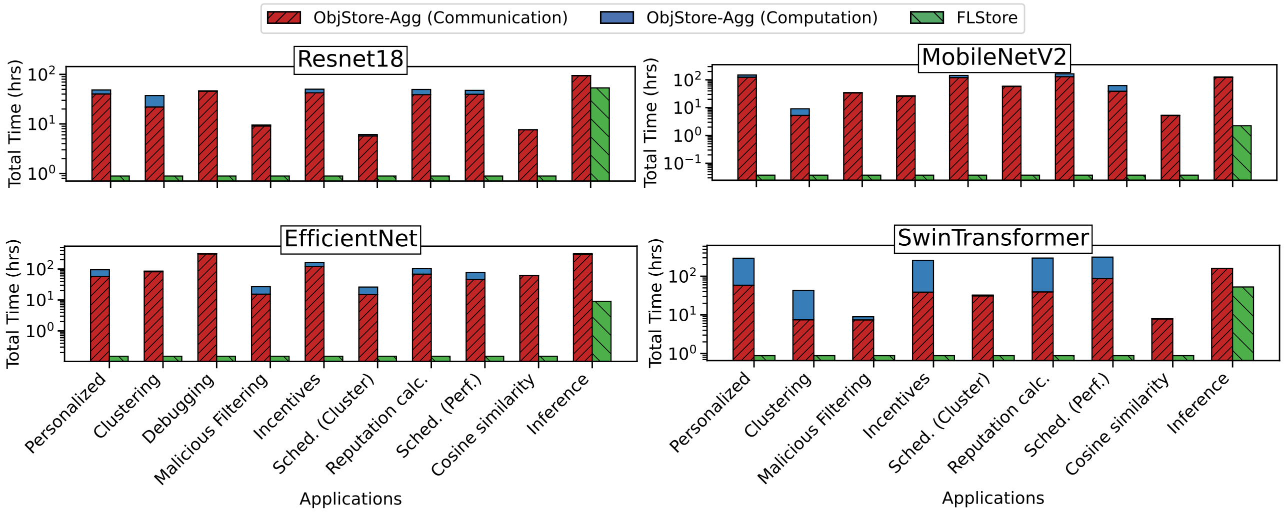

Figure 13. FLStore vs. ObjStore-Agg **total time** breakup comparison over 50 hours.

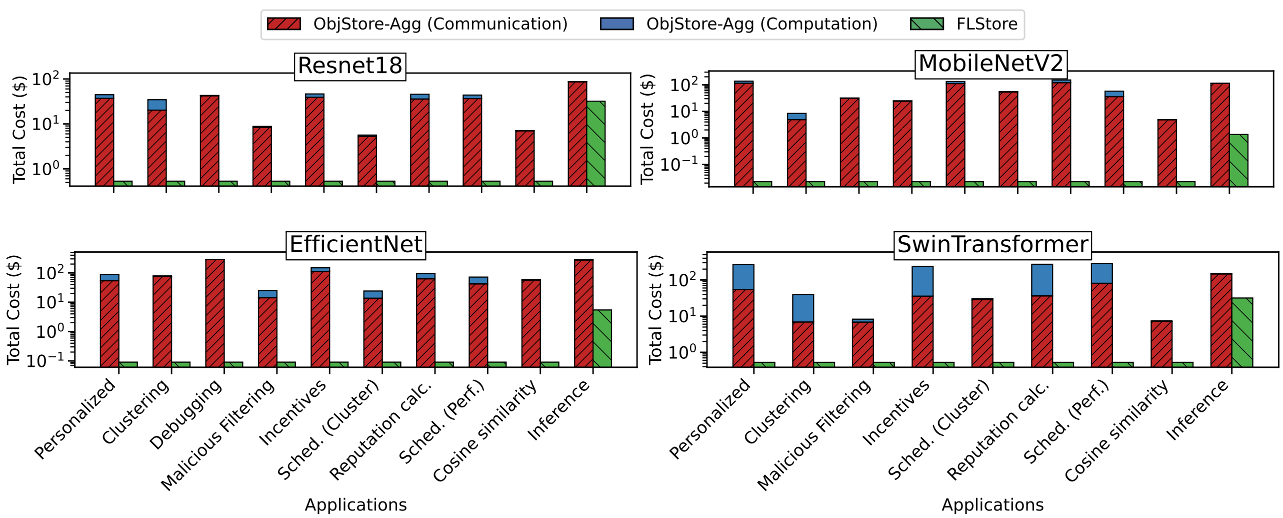

Figure 14. FLStore vs. ObjStore-Agg **total cost** breakup comparison over 50 hours.

tested with Cache-Agg, FLStore shows an average decrease in costs of 98.83% compared to Cache-Agg and 92.45% decrease compared to ObjStore-Agg.

C FLSTORE STATIC: ABLATION STUDY

For comparison with FLStore-Static, we consider a scenario where the workload changes from *model inference* to *malicious filtering*. Caching policy of FLStore-Static remains static (Individual Client Updates) which was for model inference workload while FLStore changes its caching policy to *All Client Updates* based on the new workload (malicious filtering). Results in Figure 16 show that FLStore reduces per-request average latency by 99% (8 seconds) and costs by approximately 3×. This analysis highlights the importance of designing caching policies tailored for non-training FL workloads.

D DISCUSSION: LIMITATIONS AND FUTURE WORK

Support for Foundation Models Foundation Models are a class of models that have undergone training with a broad and general data set. Users can then fine-tune foundational models for specific use cases without training a model from scratch. We have added and evaluated several foundation models from Figure 17 in FLStore and continue to add more such models. We also add model inference as an application for FLStore with the aim of providing a cost-effective alternative for serving models efficiently compared

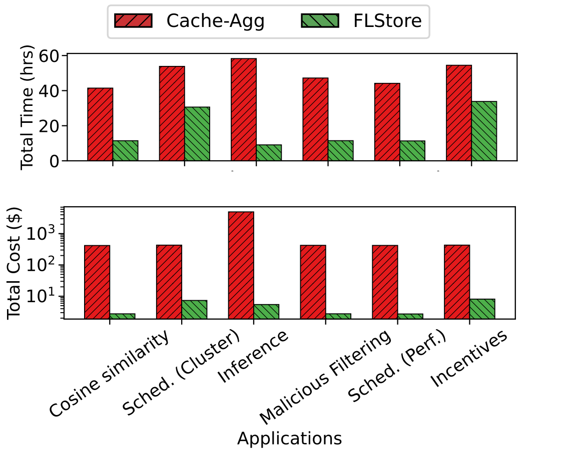

Figure 15. **Total time** (top) and **total cost** (bottom) comparison of Cache-Agg baseline vs. FLStore over 50 hours and 3000 total requests.

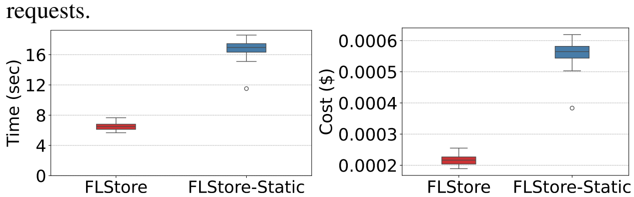

Figure 16. FLStore vs. FLStore-Static: **Per request latency** (left) and **cost** (right) while filtering malicious clients.

to other cloud solutions such as AWS SageMaker (Amazon Web Services, Inc., 2024b) which incurs high latency and costs as shown by our analysis in Figures 13 and 14.

FLStore Integration Due to the modular nature of FLStore, it can be easily integrated into any existing FL framework. We have successfully integrated FLStore with IBMFL (IBM, 2020), and FLOWER (Beutel et al., 2020), two popular FL frameworks used in industry and research.

Adaptive Caching Policies Our ongoing efforts include designing agents based on Reinforcement Learning with Human Feedback (RLHF) that can understand the characteristics of non-training workloads and create new caching policies for those workloads using our existing caching policies as a base. RLHF has successfully been deployed for hyperparameter and optimization configuration in FL (Khan et al., 2024) and the configuration of caching policies is a similar challenge that we hope to resolve by employing this technique.

Function Memory Limitations Serverless functions are limited in memory resources having a maximum of 10 GB memory (Amazon Web Services, Inc., 2024a). This is more than sufficient for handling non-training workloads for cross-device FL even for small transformer models such as (Face, 2024). As shown in Figure 17, the average size of popular models used in cross-device FL is just 161 MB approximately. However, for even larger foundational models such as Large Language Models (LLMs) (Brown et al., 2020), we are working on utilizing pipeline parallel processing where function groups can be assigned for each workload and each function in that group can perform computations in a pipeline parallel manner (Jiang et al., 2021; Yang et al., 2022b).

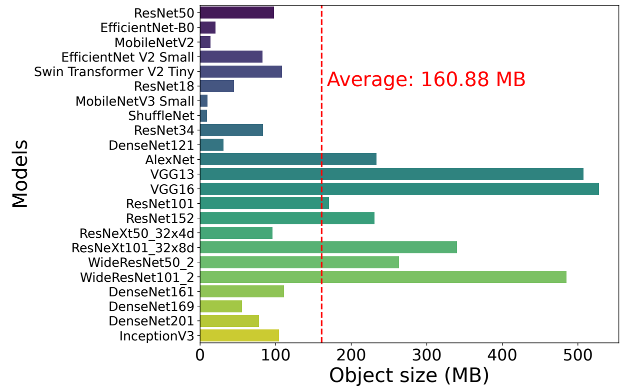

Figure 17. Memory footprint of commonly used models in FL.