LIFL: A LIGHTWEIGHT, EVENT-DRIVEN SERVERLESS PLATFORM FOR FEDERATED LEARNING

Shixiong Qi [1] **K. K. Ramakrishnan** [1] **Myungjin Lee** [2]

ABSTRACT

Federated Learning (FL) typically involves a large-scale, distributed system with individual user devices/servers training models locally and then aggregating their model updates on a trusted central server. Existing systems for FL often use an always-on server for model aggregation, which can be inefficient in terms of resource utilization. They may also be inelastic in their resource management. This is particularly exacerbated when aggregating model updates at scale in a highly dynamic environment with varying numbers of heterogeneous user devices/servers.

We present LIFL, a lightweight and elastic serverless cloud platform with fine-grained resource management for efficient FL aggregation at scale. LIFL is enhanced by a streamlined, event-driven serverless design that eliminates the individual heavy-weight message broker and replaces inefficient container-based sidecars with lightweight eBPF-based proxies. We leverage shared memory processing to achieve high-performance communication for hierarchical aggregation, which is commonly adopted to speed up FL aggregation at scale. We further introduce locality-aware placement in LIFL to maximize the benefits of shared memory processing. LIFL precisely scales and carefully reuses the resources for hierarchical aggregation to achieve the highest degree of parallelism while minimizing the aggregation time and resource consumption. Our experimental results show that LIFL achieves significant improvement in resource efficiency and aggregation speed for supporting FL at scale, compared to existing serverful and serverless FL systems.

1 INTRODUCTION

Federated Learning (FL (McMahan et al., 2017)) enables collaborative model training across a network of decentralized devices/machines while keeping individual user data secure and private. In FL, instead of sending raw data to a central server, models are trained on individual devices/machines using local data, and only the model updates are shared and aggregated to create a global model.

To support FL at scale, *hierarchical aggregation* is often adopted to increase the service capacity for model aggregation (Bonawitz et al., 2019; Jayaram et al., 2022b). This can accommodate a large number of clients and handle a substantial volume of model updates, avoiding potential slow-down of the aggregation process. In the process, each level performs intermediate aggregation, combining the updates from lower-level aggregators or clients.

Existing FL frameworks (*e.g.,* Google's FL stack (Bonawitz et al., 2019), Meta's PAPAYA (Huba et al., 2022)) adopt a static, always-on[1] deployment to support model aggregation. However, in a dynamic FL environment, it's difficult to have a one-size-fits-all service capacity for model aggregation. System heterogeneity (*i.e.,* different hardware capabilities) and a dynamically varying number of participating clients in each round require frequent adjustments of the capacity so that the aggregation service effectively uses resources on demand and avoids significant resource wastage.

Serverless computing promises to provide an event-driven, resource-efficient cloud computing environment, enabling services to use resources on demand (Shahrad et al., 2020a). Running FL model aggregation service as serverless functions can right-size the provisioned resources and reduce resource waste compared to an always-on aggregation server implementation. In addition, stateless processing by serverless functions makes it easy to support continual updates to the aggregation hierarchy. By increasing the capacity of aggregation through a hierarchy of serverless aggregators, model aggregation in FL can be executed in parallel, responding to increasing loads from trainer model updates.

However, the excessive overhead in current serverless frameworks, caused by the loose coupling of data plane components (Qi et al., 2022), is a barrier to achieving efficient and

[1]University of California, Riverside [2]Cisco Research. Correspondence to: Shixiong Qi <sqi009@ucr.edu>, K. K. Ramakrishnan <kk@cs.ucr.edu>, Myungjin Lee <myungjle@cisco.com>.

Proceedings of the 7th MLSys Conference, Santa Clara, CA, USA, 2024. Copyright 2024 by the author(s).

[1]meaning that aggregators are up all the time within a round.

timely aggregation, compared to a monolithic serverful design. Further, the use of individual, constantly-running components (e.g., container-based sidecars) in current serverless frameworks is inefficient and sacrifices much of the benefit of serverless computing. This prompts us to create a more streamlined, responsive serverless framework that is tailored to achieve just-in-time FL aggregation on demand.

We introduce LIFL, a lightweight serverless platform for FL that uses hierarchical aggregation to achieve parallelism in aggregation and exploits intra-node shared memory processing to reduce data plane overheads. LIFL also utilizes a locality-aware placement policy to maximize the benefits of the intra-node shared memory data plane. Unlike typical serverless platforms that use a heavyweight sidecar implemented as a separate container, LIFL seeks to eliminate this wasteful overhead by taking advantage of eBPF-based event-driven processing. This ensures that resource usage is truly load-proportional. Instead of depending on inaccurate, threshold-based autoscaling, LIFL uses hierarchy-aware autoscaling to precisely adjust the capacity of model aggregation to match the incoming load. We also use a policy of reusing runtimes to sidestep the impact of startup delay on the model convergence time, while also improving resource efficiency of aggregation. LIFL favors eager aggregation to enable timely aggregation, reducing the queuing time for model updates. By harnessing the capabilities of LIFL, FL systems can achieve efficient resource utilization and reduced aggregation time. LIFL is available at (fla, 2024).

We highlight the contributions of LIFL below:

(1) LIFL's enhanced data plane achieves 3× (compared to serverful) and 5.8× (compared to serverless) latency reduction on transferring a relatively heavyweight ResNet-152 model update within the aggregation hierarchy (intra-node).

(2) LIFL's locality-aware placement can maximize shared memory processing, achieving up to 2.1× additional latency reduction on aggregating a batch of updates in a round (details in §6). After applying hierarchy-planning, aggregator reuse, and eager aggregation, LIFL can further obtain 1.5× latency reduction. The enhanced orchestration also helps improve efficiency, saving up to 2× CPU consumption compared to simply using the enhanced data plane.

(3) Our evaluation with a real FL workload using ResNet-18 and 120 simultaneous active clients (the total number of clients used is 2,800) shows that the combination of LIFL's enhanced data and control planes achieve **5**× and **1.8**× less CPU cost and reduces **2.7**× and **1.6**× on time-to-accuracy (70% accuracy level), compared to existing serverless and even serverful FL systems. We also train a relatively heavyweight ResNet-152 model. LIFL spends 1.68× less time to reach 70% accuracy than existing serverless FL systems, while using 4.23× fewer CPU cycles.

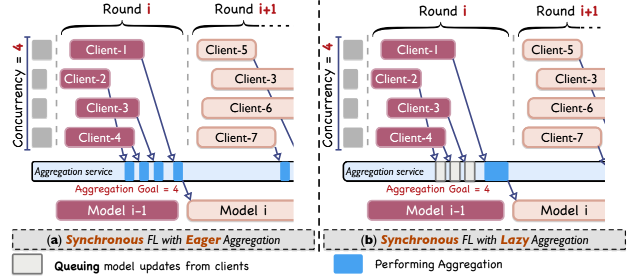


Figure 1. Synchronous FL with different aggregation timing ("Eager" and "Lazy") (Bonawitz et al., 2019; Jayaram et al., 2022c).

2 Background and Challenges

2.1 Basics of Federated Learning

FL aggregation: Aggregation in FL is a process of building a global model from individually trained model updates. The *aggregation goal*, n specifies the expected number of model updates to be received before the global model is updated to a new version. Thus, it dictates the number of selected clients for training. The aggregation process is abstracted as:

$$w_i = f(\{(w_i^k, \mathcal{A}_i^k) \mid 1 \leq k \leq n\}). \qquad (1)$$

Here $f(\cdot)$ is an aggregation function, w_i^k is k-th local model update for global model version i, and $\mathcal{A}_i^k$ is auxiliary information for aggregation. For the *FedAvg* algorithm (McMahan et al., 2017), $f(\cdot) = \sum_{k=1}^{n} w_i^k c_i^k / T_i$. $T_i = \sum_{k=1}^{n} c_i^k$ and $\mathcal{A}_i^k$ is c_i^k (the number of data samples).

Eager aggregation and Lazy aggregation: Based on the timing to trigger the aggregation, we can classify the model aggregation to be "*eager*" or "*lazy*" (Jayaram et al., 2022c): *Eager* aggregation allows aggregation to happen whenever an update is received, leading to more flexible and dynamic timing of the aggregation process. *Lazy* aggregation operates on a delayed schedule, where model updates that arrive early are queued without being aggregated immediately. Fig. 1 shows the two different aggregation methods for synchronous FL. For instance, the *eager* method is feasible for FedAvg with cumulative averaging.

2.2 Anatomy of Systems for Federated Learning

Designing a system to support FL at a ***large scale*** is essential, as a larger number of participants means a more diverse and representative dataset. It improves the model's ability to capture complex patterns and unseen relationships in the data. These benefits help the model generalize in real-world deployments, *e.g.,* Google's FL stack has been used to serve $\sim 10M$ devices daily and $\sim 10K$ devices participate in FL training simultaneously (Bonawitz et al., 2019).

Fig. 2 depicts key architectural components that are needed to ensure the success of FL at scale.[2] These components work together to enable the collaborative and decentralized training process in FL. In addition to the **aggregator** and the

[2]We adopt the terminology of FL system components from (Bonawitz et al., 2019) and (Huba et al., 2022).

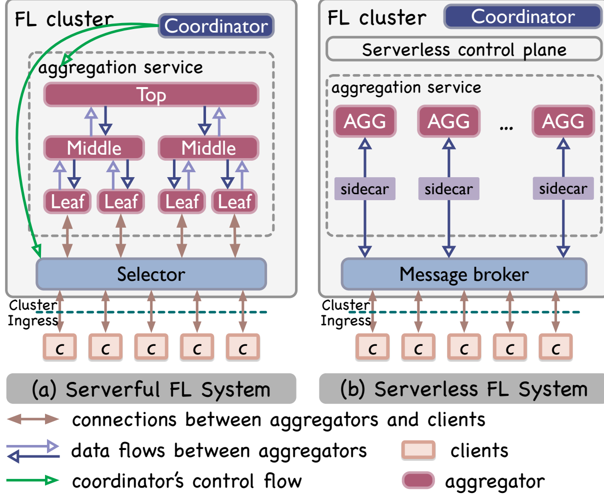


Figure 2. Generic architectures for FL systems: (a) Serverful FL systems (Bonawitz et al., 2019; Huba et al., 2022); (b) Serverless FL systems (Jayaram et al., 2022a;b; Chadha et al., 2021). Note that for simplicity, we skip the hierarchy in the diagram (b).

client, the **coordinator** oversees the flow of FL operations. It acts as an orchestrator that facilitates seamless interactions among aggregators, selectors, and clients by applying the client selection scheme and instructing the selector to map the selected clients to backend aggregators (Bonawitz et al., 2019). The **selector** plays two roles. First, it ensures that a diverse set of clients participate in the FL process to capture a representative sample of the distributed data. Second, it acts as a gateway that mediates communication (*i.e.,* queuing, load balancing) between (leaf) aggregators and clients (Bonawitz et al., 2019; Huba et al., 2022).

Need for hierarchical aggregation: The growing number of participating clients in FL requires the system to be scalable to accommodate the computational requirements of aggregating model updates from a large number of distributed clients. This primarily motivates the use of *hierarchical aggregation* potentially involving multiple levels of aggregation in the FL process (Bonawitz et al., 2019; Jayaram et al., 2022b), as depicted in Fig. 2 (a). Essentially, hierarchical aggregation is structured as a single-rooted tree. Each level in the tree includes multiple parallel aggregation tasks that are executed by one of potentially multiple aggregators. The communication during the hierarchical aggregation task takes place across multiple levels: The model updates from smaller subgroups of clients are aggregated by the lower-level aggregators (*i.e.,* leaf) and passed onto higher-level aggregators (*i.e.,* top), until a global model is obtained. This parallel aggregation at the lower levels can provide speedup and reduce queueing of model updates.

2.3 Motivation and Challenges for Serverless FL

State-of-the-art FL systems (Bonawitz et al., 2019; Huba et al., 2022) rely on a "serverful" design that relies on a fixed pool of dedicated resources (*e.g.,* CPU and memory), using a pool of provisioned VMs. Resizing the pool often takes a long time (e.g., 6 to 45 minutes on AWS (Scheller, 2023)). Serverless computing, on the other hand, brings fine-grained resource elasticity by provisioning functions (typically as containers) dynamically based on demand, ensuring that the right amount of resources is allocated only when needed.

In FL, serverless computing can be used to provide efficient model aggregation, adapting to varying numbers of clients. It eliminates the need to maintain dedicated resource pools for the aggregation service, thereby improving overall efficiency compared to the current "serverful" deployments.

Prior Work on Serverless FL. A number of FL system designs have been proposed using serverless computing (Jayaram et al., 2022a;b; Grafberger et al., 2021). A common abstract architecture of a serverless FL system and its key components is shown in Fig. 2 (b). But, prior approaches still face the following challenges:

Indirect networking: Unlike a "serverful" design (Fig. 2 (a)), a serverless FL system executes aggregators as serverless functions. Serverless function chaining can support hierarchical aggregation as well as communication between aggregators. However, because serverless functions are ephemeral and stateless (and thus unable to retain stateful information like routes), these chains typically only support *indirect* networking between functions. This raises the need for a stateful, persistent networking component (Fig. 2 (b)), such as a message broker or external storage services,[3] to maintain routes and exchange messages (Qi et al., 2022). However, having such a networking component in the internal datapath between serverless functions adds unnecessary overhead (20% added delay as in Fig. 7(a)).

Inefficient message queuing: In addition to supporting function chaining, the message broker (Fig. 2 (b)) also acts as a message queue to buffer incoming model updates from clients while aggregators are being spawned by the serverless control plane (Jayaram et al., 2022a;b). However, the message broker and dedicated queues add overhead and delay to the aggregation service.

Heavyweight sidecar: Scheduling serverless functions typically requires metrics collection, often using a sidecar. This container-based sidecar introduces additional network processing in the datapath, requiring the interception and forwarding of model updates. This leads to complex data pipelines (involving extra communication hops between aggregators) and increased communication overheads due to the reliance on kernel-based networking (Qi et al., 2022).

Application-agnostic, simple, autoscaling: Current serverless autoscaler designs typically rely on a simplistic threshold based on user input (*e.g.,* request per second, concurrency) for scaling decisions (aut, 2023b;a), often being un-

[3] For consistency, we use the generic term "message broker" to denote such a networking component throughout this paper.

Figure 3. The overall architecture of LIFL.

aware of application needs. This design, agnostic of the hierarchical structure of FL aggregation, is limited in its ability to optimize the system to maximize parallelism, *i.e.,* the number of levels and the number of aggregators at each level. Looking at Fig. 2 (a), as we go up the levels in the hierarchy, fewer aggregators are needed. This can be leveraged to potentially reuse the lower-level aggregators as we proceed up the hierarchy. Further, since hierarchical aggregation uses function chaining, current "reactive" autoscaling designs lead to a cascading effect (Park et al., 2021b) of the cold-start delays when scaling a function chain.

Locality-agnostic placement: Intra-node communication can be faster than *inter-node* communication by avoiding a lot of networking overheads (Qi et al., 2021) and using state-of-the-art serverless data plane designs with *shared memory* (Shillaker & Pietzuch, 2020; Qi et al., 2022; Yu et al., 2023). However, leveraging the benefits of shared memory effectively can be challenging when dealing with a large hierarchy of aggregators that cannot be accommodated within a single node. This requires careful function placement by taking into account the impact of communication between aggregators. Inter-node communication typically still uses kernel-based networking.

3 LIFL Overview

We aim to address the aforementioned limitations (§2.3) and develop LIFL—a high-performance, lightweight, and elastic serverless platform for FL, utilizing hierarchical aggregation. We focus on the following innovations of LIFL:

(1) High-performance intra-node dataplane: LIFL incorporates shared memory processing to provide a *zero-copy* communication channel between FL aggregators placed on the same node (§4.1). This avoids heavyweight kernel networking overheads, especially data copies (Cai et al., 2021), as model updates are often large, *e.g.,* a model update from ResNet-152 (He et al., 2016) is ∼230 MBytes. Shared memory can also eliminate other overheads such as protocol processing, serialization/de-serialization, kernel/userspace boundary crossing, and interrupts.

(2) In-place message queuing: We extensively leverage shared memory in LIFL to offer "in-place" message queuing (§4.2). Messages (*i.e.,* model updates) from selected clients are directly buffered in shared memory and can be instantly accessed by the aggregators when they are ready. This eliminates dedicated message queues and their associated queuing delays.

(3) Lightweight eBPF-based sidecar: We incorporate the extended Berkeley Packet Filter (eBPF (ebp, 2023a)) into LIFL to build a lightweight sidecar (§4.3) to provide important functionality, *e.g.,* metrics collection. Unlike a container-based sidecar, LIFL's sidecar runs as eBPF code attached at in-kernel hooks, avoiding the need for dedicated resources. We further utilize the eBPF-based sidecar to support *direct* networking between aggregators (§4.4), completely replacing the message broker.

(4) A cost-effective orchestration heuristic: LIFL orchestrates the model aggregation to fully exploit the improved serverless dataplane by employing several strategies: (1) locality-aware placement that partitions levels with large traffic into node-affinity groups to make the best use of shared memory processing (§5.1); (2) hierarchy-aware scaling that dynamically adjusts the configuration of hierarchical aggregation (§5.2), and (3) opportunistic reuse of the aggregator runtime from a lower level (§5.3).

Architectural overview of LIFL: Fig. 3 shows the overall architecture of LIFL. LIFL maintains a shared memory object store on each worker node to enable zero-copy communication between aggregators. To support in-place message queuing, LIFL introduces a gateway on each worker node that receives model updates from remote clients. The gateway performs a consolidated, one-time payload processing to queue the received model updates to shared memory. Each aggregator in LIFL has attached to it an eBPF-based sidecar for lightweight metrics collection. Aggregators in LIFL are stateless, so new ones start without state synchronization upon an aggregator failure. LIFL detects client failures with keep-alive heartbeats and enhances resilience by over-provisioning the number of clients. In the control plane, a LIFL agent is deployed on each worker node to manage the lifecycle (*e.g.,* creation, termination) of aggregators, following instructions from the LIFL control plane. The LIFL coordinator, a cluster-wide control plane component, is used for interactions between the FL job designer (ML engineer) and the serverless control plane (*e.g.,* autoscaler, placement engine).[4] It works with the serverless control plane to execute LIFL's orchestration flow (§5).

[4]Note: Even though the serverless control plane has serverful, always-on components (*e.g.,* autoscaler, placement engine), are shared and their overheads are amortized across multiple workloads, especially at scale.

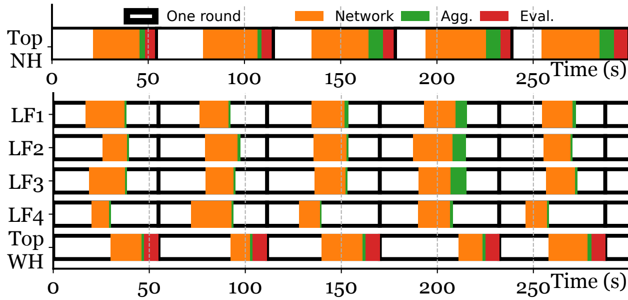


Figure 4. Impact of data plane performance on hierarchical aggregation. (upper fig.:) No hierarchy(NH); (lower fig.:) With hierarchy(WH). Top: top aggregator; LF: leaf aggregator. "Network" denotes the data transfer tasks of model updates; "Agg." denotes the aggregation tasks; "Eval." denotes the evaluation tasks.

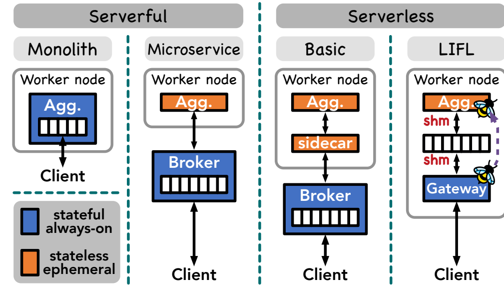


Figure 5. Message queuing solutions.

4 Optimizing the Serverless Data-Plane in LIFL

4.1 Shared Memory for Hierarchical Aggregation

Assessing data plane with hierarchical aggregation: We now assess the importance of a high-performance data plane to truly deliver on the promise of hierarchical aggregation. We consider a baseline (denoted **`NH`**) with a single aggregator without hierarchy. We evaluate the hierarchical aggregation service that has one top aggregator and four leaf aggregators (denoted **`WH`**). All aggregators are placed on the same node. We consider eight trainers to train a ResNet-152 model using FEMNIST dataset. Note that we always deploy trainers on separate nodes, to both be realistic (trainers are remote) and to avoid contention for resources on the node.

Fig. 4 shows the execution times for the representative FL stages under different settings. Note that we only show the receiving part of the networking task ("Network" in Fig. 4) to simplify the figure. Compared to the baseline (**`NH`**), **`WH`** does not exhibit a significant improvement overall, though it uses hierarchical aggregation. The average completion time per round with **`WH`** is 57 seconds, while for **`NH`** is 59.8 seconds. This is mainly because of the contention for network processing between leaf aggregators when they send/receive intermediate model updates to/from the top aggregator. This highlights the critical need for a high-performance and streamlined data plane for hierarchical aggregation. LIFL incorporates shared memory processing when the serverless aggregator functions are co-located on the same node. This enables fast and efficient communication, mitigating the impact of networking on hierarchical aggregation (demonstrated in Fig. 7). Working jointly with our locality-aware placement scheme (§5.1), LIFL can minimize the need for inter-node model update transfers. Consequently, LIFL maximizes the advantages of our efficient intra-node shared memory data plane that substantially reduces communication overheads.

Shared memory object store: The LIFL agent is responsible for the allocation/recycling/destruction of the shared memory buffer in the object store. In addition, LIFL only allows immutable (read-only) objects to guarantee the safe sharing of model updates, eliminating the need for locks. The agent periodically checkpoints the model parameters to an external persistent storage service (more details in Appendix-B).

4.2 In-place Message Queuing

Representative message queuing solutions: Fig. 5 enumerates message queuing solutions for various serverful and serverless alternatives. In the *monolithic* serverful setup (used in (Huba et al., 2022)), the model update is directly buffered into an in-memory queue residing in the aggregator, deployed as a persistent and stateful application. Another serverful setup, used in (Bonawitz et al., 2019), deploys aggregators as ephemeral, stateless *microservices*, requiring an additional persistent, stateful message broker to buffer model updates from clients before being consumed by the stateless aggregator. Switching to the *basic* serverless setup (used in (Jayaram et al., 2022b)), model updates are also buffered at a message broker, as the aggregator is now deployed as an ephemeral, stateless serverless function. Before being consumed by the aggregator, the model update has to pass through the sidecar. Finally, in LIFL, the gateway buffers the model update directly into the shared memory, which can then be seamlessly accessed by the aggregator. The distinct data pipelines between the client, message queue, and aggregator impose varying degrees of overheads. Our evaluation (details in Appendix-F) shows that LIFL's in-place message queuing achieves the best efficiency and performance (equivalent to a monolithic, serverful design) among all alternatives in Fig. 5.

Message queuing pipeline in LIFL: The gateway at each worker node is addressable/accessible by FL clients. It receives model updates from clients or from the gateway on another worker node, and performs necessary network processing (*e.g.,* protocol processing, serialization, deserialization, data type conversion, *etc*) before writing the model updates into shared memory. This avoids duplicate processing when local aggregators access model updates in shared memory. A step-by-step explanation of the processing flow of the message queuing in LIFL is given in Appendix-C.

We apply vertical scaling of the gateway by dynamically

adjusting the number of assigned CPU cores based on the load level. This avoids the gateway becoming the dataplane bottleneck and impacting the aggregation speed.

4.3 eBPF-based Sidecar

LIFL's eBPF-based sidecar is built with a set of eBPF programs attached to each aggregator's socket interface, using its in-kernel SKMSG hook (Red Hat, Inc., 2022). The execution of the eBPF-based sidecar is triggered by the invocation of the *send()* system call, which is captured by the SKMSG hook as an eBPF event. This ensures that the eBPF-based sidecar is strictly event-driven and consumes *no* CPU resources when idle. We use the eBPF-based sidecar to collect necessary metrics (*e.g.,* execution time of the aggregation task) to facilitate the orchestration in LIFL (§5).

Metrics collection: Upon invocation, the eBPF-based sidecar collects and stores metrics to an eBPF map (*metrics map*) on the local worker node. The eBPF map is an in-kernel, configurable key-value table that can be accessed by the eBPF program during execution (ebp, 2023c). The LIFL agent, on the other hand, periodically retrieves the latest metrics from the *metrics map* and feeds the metrics back to the metrics server (Fig. 3) in the serverless control plane.

4.4 Direct Routing with Hierarchical Aggregation

Direct networking between functions is not allowed in existing serverless environments because serverless functions are considered to be stateless and ephemeral. This implies that there are no long-lived, direct connections between a pair of function instances. As a result, they use an intermediate networking component (*e.g.,* message broker) to act as a stateful, persistent component to manage state, *i.e.,* routes between functions. However, the main drawback is that it adds unnecessary overhead by involving the additional networking component(s) in the datapath, making indirect networking between functions heavyweight.

LIFL improves serverless networking within hierarchical aggregation by allowing direct routing between aggregators, both within a node and between nodes. The key is to offload the stateful processing to eBPF, using the *sockmap* (Red Hat, Inc., 2022) to support flexible intra-node routing exploiting shared memory, and inter-node routing with the help of the per-node gateway, as depicted in Fig. 12. The *sockmap* is a special eBPF map (`BPF_MAP_TYPE_SOCKMAP` (Red Hat, Inc., 2022)) that maintains references to the registered socket interfaces. We take the approach from (Qi et al., 2022) to implement intra-node direct routing in LIFL. For details of intra-/inter-node routing in LIFL, refer to Appendix-A.

5 LIFL's Control Plane Design

5.1 Locality-aware Placement and Load Balancing

The placement of aggregators can lead to different routing behaviors: When aggregators with cross-level data dependencies are placed on the same node, the shared memory processing and eBPF-based sidecar can facilitate intra-node routing. When these aggregators are placed across different nodes, the gateway has to perform inter-node routing. To minimize the transfer of model updates in LIFL, we take a data-centric strategy like (Yu et al., 2023) that is aware of the locality of model updates and places the aggregator close to the model updates. As such, the in-place message queuing (§4.2), which is, in fact, the result of load balancing (clients to worker node mapping), directly affects the effectiveness of the locality-aware placement of the aggregators.

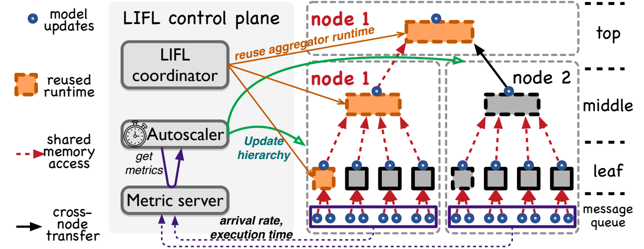


Figure 6. Control plane orchestration in LIFL: The autoscaler periodically re-plans the hierarchy based on the arrival rate of each worker node. The LIFL coordinator applies reusing of aggregators.

Our objective of load balancing involves two crucial criteria: **(1)** Minimizing inter-node communication while maximizing the utilization of shared memory within each node. **(2)** Ensuring the residual service capacity of the worker node meets the demand; the residual service capacity ($RC_{i,t}$) of worker node i at time t is determined by $RC_{i,t} = MC_i - (k_{i,t} \times E_{i,t})$. Here, MC_i represents the maximum service capacity[5], denoting the maximum number of model updates that can be aggregated simultaneously on worker node i. The value of $k_{i,t}$ is the arrival rate of model updates directed to worker node i at time t, and $E_{i,t}$ is the average execution time required to aggregate a model update on node i. We can also get a coarse-grained estimate on the queue length ($Q_{i,t} = k_{i,t} \times E_{i,t}$) of node i at time t.

We approach the load balancing task as a bin-packing problem, aiming to allocate model updates from clients to a minimal number of worker nodes, while ensuring that the residual service capacity of each worker node is not exceeded. This naturally reduces the inter-node communication as much as possible, since the communication between a particular pair of worker nodes only happens once. We use *BestFit* for the bin-packing, as it concentrates load onto the fewest nodes possible, to reduce inter-node traffic and maximize shared memory use. In contrast, *WorstFit* spreads the load across more nodes, similar to the "Least Connection" policy in Knative (§6.1). Furthermore, *FirstFit* focuses on reducing search complexity without being locality-aware.

[5]We compute MC_i offline; for details, refer to Appendix-E.

5.2 Planning the Hierarchy for Aggregation

The goal of hierarchy-aware autoscaling is to maximize the parallelism of aggregation at each level, given the number of model updates to be aggregated. This can minimize the completion time of each level and thus minimize the aggregation completion time (ACT) for hierarchical aggregation. We plan a hierarchical aggregation structure within each node, tailored to the number of pending model updates ($Q_{i,t}$) in the message queue. Every node produces an intermediate model update that is dispatched to the node chosen to have the top aggregator that updates the global model. This approach significantly reduces the need for cross-node transfers for intermediate model updates.

LIFL periodically adjusts (*i.e.,* scales) the hierarchy on node i, guided by our estimates of $Q_{i,t}$. To prevent excess resource allocation due to short-term spikes in $Q_{i,t}$, we employ the Exponentially Weighted Moving Average (EWMA) to smooth $Q_{i,t}$: $Q_{i,t} = \alpha \times Q_{i,t-1} + (1-\alpha) \times Q_{i,t}$, where α is the EWMA coefficient. We set $\alpha = 0.7$ based on it yielding the best results in our experiments. Our current implementation supports a two-level k-ary tree hierarchy on each node, comprising a "central" middle aggregator responsible for aggregating model updates from $Q_{i,t}/I$ leaf aggregators, where I is the number of model updates of clients per leaf aggregator. Given that the steps within a LIFL aggregator (Fig. 14) are executed sequentially, we want to maximize the parallelism by having a limited I to be small (*e.g.,* at 2), ensuring that a leaf aggregator experiences minimal waiting time after receiving the initial update from the first client.

LIFL re-plans the hierarchy on each worker node periodically. This involves estimating $Q_{i,t}$ across the worker nodes and creates/terminates aggregators accordingly. The LIFL control plane updates the routes between aggregators based on the renewed hierarchy (details in Appendix-A).

5.3 Opportunistic Reuse of Aggregator Instances

The scaling policy in LIFL incorporates an opportunistic "reuse" scheme to maximize the utilization of warm aggregator instances since aggregators in LIFL use homogenized runtimes (Fig. 14) with the same code and libs. This sidesteps the cascading effect (Park et al., 2021b) when starting up a hierarchy of aggregators (in fact function chains).

Given a hierarchy of aggregators selected on the node, LIFL picks a leaf aggregator that has already completed its aggregation task and is idle. LIFL converts its role to a middle aggregator on that node. No further change is required as LIFL's aggregator runtime is stateless. LIFL selects the first middle aggregator that completes its local aggregation task and converts it to be the top aggregator responsible for updating the global model. This minimizes the need to start up new instances for higher-level aggregators, and avoids additional startup delays.

5.4 Eager aggregation in LIFL

LIFL employs eager aggregation (Fig. 1) leveraging its more flexible and dynamic timing of the aggregation process. Eager aggregation performs timely aggregation as model updates arrive, even if it triggers the cold start of an aggregator (when no idle-but-warm aggregator is available). This takes advantage of the *overlap* between the start-up delay and transfers of model updates, allowing eager aggregation to mask cold starts up until the last model update. It also mitigates congestion that can occur when trying to aggregate all model updates simultaneously. In contrast, lazy aggregation aggregates all model updates in a batch when the aggregation goal is reached. But, the arrival of local model updates from trainers can be spread over a relatively long duration. Our evaluation shows eager aggregation achieves a 20% reduction on ACT (Fig. 8(a)). We implement eager aggregation in LIFL following the step-based processing model described in Appendix-G. LIFL updates the version of the global model whenever the aggregation goal is achieved.

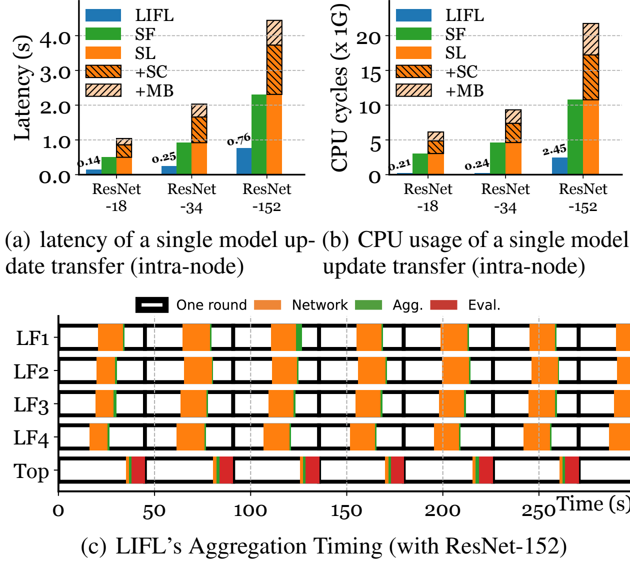
6 Evaluation & Analysis

We quantify the performance gain and resource savings by using LIFL, starting with analyzing a set of microbenchmarks to understand the different design considerations of LIFL, including shared memory processing, the effectiveness and overheads of LIFL's orchestration scheme. We then demonstrate the benefits of LIFL from a system-level perspective using real FL workloads.

Baseline Systems: We implement several baseline FL systems for LIFL to compare against. **(1) "Serverful system" (SF):** The "serverful system" is implemented following the design described in (Bonawitz et al., 2019) and (Huba et al., 2022). Both of them adopt the architecture depicted in Fig. 2 (a). **(2) "Serverless system" (SL):** The baseline "serverless system" is implemented following the design described in FedKeeper (Chadha et al., 2021) and AdaFed (Jayaram et al., 2022b) that uses the architecture depicted in Fig. 2 (b). We choose Knative (kna, 2023) as the serverless framework to build these alternatives. We utilize the open-source Flame platform (fla, 2024) to provide necessary FL components, *e.g.,* coordinator, selector, aggregator, and client.

Implementation of LIFL: We implement LIFL based on SPRIGHT (Qi et al., 2022), a lightweight, high-performance serverless framework. LIFL includes object store support, model checkpoints, and routing support for hierarchical aggregation. LIFL uses Python's multiprocessing package to implement the shared memory pool instead of the DPDK-based shared memory pool used in the original implementation of SPRIGHT. The current implementation of LIFL only supports synchronous FL. Supporting asynchronous FL is part of our future work.

Testbed setup: We leverage the NSF Cloudlab (Duplyakin

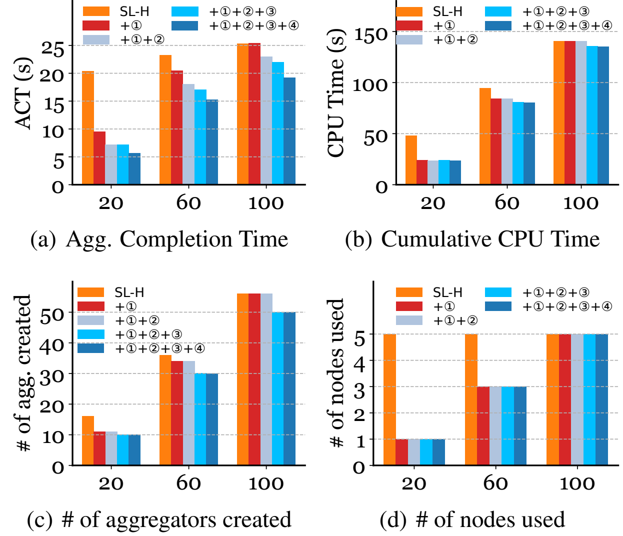


(a) latency of a single model update transfer (intra-node)

(b) CPU usage of a single model update transfer (intra-node)

(c) LIFL's Aggregation Timing (with ResNet-152)

Figure 7. Data plane improvement for hierarchical aggregation: Serverful (**SF**), Serverless (**SL**), and LIFL. **SL**'s latency includes contributions of **+SC** (sidecar) and **+MB** (message broker).



(a) Agg. Completion Time

(b) Cumulative CPU Time

(c) # of aggregators created

(d) # of nodes used

Figure 8. Improvement with LIFL's orchestration, with ① being additions to baseline LIFL; *x-axis is the number of model updates arriving at the aggregation service concurrently.*

et al., 2019). The nodes we used have a 64-core Intel Cascade Lake CPU@2.8 GHz, 192GB memory, and a 10Gb NIC. We use Ubuntu 20.04 with kernel version 5.16.

6.1 Microbenchmark Analysis

Data plane improvement for hierarchical aggregation: To understand the improvements in data plane performance of hierarchical aggregation with LIFL's shared memory processing, we use the same aggregation hierarchy as in §4.1, comprising one top aggregator and four leaf aggregators. All aggregators are placed on the same node.

We consider the following serverful and serverless alternatives: (1) The serverful setup (**SF**) establishes direct networking channels (based on gRPC) between leaf aggregators and the top aggregator; (2) the serverless setup (**SL**) uses indirect networking to connect leaf aggregators and the top aggregator, through a message broker on the same node. Each aggregator has a container-based sidecar to mediate inbound and outbound traffic; (3) the LIFL setup uses shared memory for communication between aggregators. We consider three ML models with distinct sizes: ResNet-18 (∼44MB), ResNet-34 (∼83MB), and ResNet-152 (∼232MB).

Fig. 7(a) shows the latency breakdown of a single model update transfer between the leaf aggregator and top aggregator for different model sizes. We specially mark the share of sidecar (**+SC**) and message broker (**+MB**) for the serverless setup. **SL** consistently results in 2× and 6× higher latency than **SF** and LIFL, respectively. The significant CPU usage of **SL** (Fig. 7(b)) clearly shows the poor efficiency and performance of the indirect networking used in the serverless setup, caused by its use of the message broker and heavyweight sidecar. We see that LIFL is considerably better than **SF** and **SL** in terms of both CPU usage and latency.

Fig. 7(c) shows the timing of various FL processing tasks during hierarchical aggregation when using LIFL's data plane. It is clear that LIFL's shared memory processing helps reduce the overhead and improve the performance of the data plane with hierarchical aggregation. LIFL completes each round in just 44.9 seconds compared to 57 seconds on average even for the serverful setup in Fig. 4. Further, through careful placement, aggregators in LIFL can fully exploit the high-speed intra-node data plane over shared memory, as discussed next.

Improved orchestration in LIFL: We now quantify the benefits of LIFL's orchestration in improving hierarchical aggregation. We demonstrate the effectiveness of LIFL by applying: ① locality-aware placement (§5.1), ② hierarchy-planning (§5.2), ③ aggregator reuse (§5.3), and ④ eager aggregation (§5.4) step-by-step. We use five nodes for this experiment. The maximum service capacity (MC_i) of each node in our testbed is 20.[6] We focus on two aspects: resource consumption and Aggregation Completion Time (ACT) to aggregate a given number of model updates. In this experiment, we assume the estimated $Q_{i,t}$ is equal to the actual queue length on each active node. We focus on the importance of having warm aggregators based on the pre-planned hierarchy, to avoid the cold start penalty.

We compare LIFL against a baseline serverless control plane using hierarchical aggregation (**SL-H** in Fig. 8). **SL-H** employs LIFL's shared memory data plane (so both have the same data plane) with Knative's "Least Connection" load balancing strategy (Mittal et al., 2021) that assigns newly arrived model updates to the node with the smallest queue

[6] Our testbed nodes are homogeneous, hence all MC_i are the same. With heterogeneous nodes, MC_i may vary.

length. The aggregators in **SL-H** use lazy aggregation by default. The ML model used is ResNet-152. Note that the latency to transmit a single model update of ResNet-152 across nodes (on the current testbed) is ∼4.2 seconds.

By using locality-aware placement, LIFL also achieves 2.1× and 1.13× ACT reduction than **SL-H** (for 20 and 60 model updates in Fig. 8(a)). This improvement is attributed to LIFL's bin-packing strategy, which effectively consolidates aggregators onto the same node to fully exploit shared memory processing. Applying hierarchy-planning and reusing warm aggregator instances (+①+②+③) further reduce ∼1.22× ACT of LIFL, as keeping aggregators warm mitigates the cold start delay that exists in both **SL-H** and (+①). Further, after enabling eager aggregation (+①+②+③+④), LIFL allows higher-level aggregators to consume and aggregate the model updates in a timely manner, effectively avoiding the intermediate model updates (produced by the lower-level aggregators) being queued up at the higher-level aggregators. This saves ∼1.2× in ACT compared to (+①+②+③) that uses lazy aggregation.

While being effective in reducing ACT, LIFL also helps to reduce costs. Just using locality-aware placement (+① in Fig. 8(b)), LIFL can save considerable CPU overhead by reducing inter-node data transfers (with 20 and 60 model updates). Enabling aggregator reuse saves additional CPU cycles, as it avoids having the CPU initialize new aggregators. For 100 model updates though, the service capacity of all five nodes would be maxed out, reaching the limit of the benefit of LIFL's orchestration. However, the data plane improvement of LIFL can still make it outperform the basic serverful and serverless setups, as demonstrated in Fig. 7.

As shown in Fig. 8(c), LIFL reduces the number of aggregators created, by packing more aggregators into fewer nodes. After we apply locality-aware placement to LIFL (+①), LIFL can also reduce the number of nodes used considerably (see Fig. 8(d)): Given 20, 60, and 100 model updates, LIFL's locality-aware placement efficiently packs them into 1, 3, and 5 nodes, respectively. This avoids repeatedly creating a middle aggregator on each of the 5 nodes (except when the service capacity of all 5 nodes is fully consumed). On the other hand, **SL-H** uses all 5 nodes throughout, uniformly distributing model updates across all 5 available nodes. This will lead to additional cross-node data transfers, regardless of available model updates. Note that the service capacity of all 5 nodes is fully consumed for 100 model updates.

Orchestration overhead of LIFL: We evaluate the orchestration overhead of LIFL, given a different number of clients. The time for completing the locality-aware placement in LIFL is less than 17 milliseconds, even with 10K clients, which is the maximum number of client settings observed in Google's production FL stack (Bonawitz et al., 2019). Compared to the ACT, which takes several tens of seconds with a large amount of clients, this overhead for locality-aware placement is negligible. The EWMA estimator for hierarchy-planning takes 0.2 milliseconds per estimate, which is also negligible compared to the 2-minute cycle time used by LIFL to re-plan the hierarchy on each worker node. The aggregator reuse and eager aggregation incur almost no overhead, as they do not require active involvement of the LIFL control plane.

6.2 FL Workloads Setup

Our aim is to demonstrate the generality of LIFL in improving performance and reducing the cost of FL from a system-level perspective. We consider synchronous FL (using FedAvg (McMahan et al., 2017)) to justify LIFL's design. We use Stochastic Gradient Descent on the client. Clients are configured with a batch size of 32 in a local training epoch, with the learning rate set to 0.01.

Benchmark selection: We consider image classification, training ResNet (He et al., 2016) models with the FEMNIST dataset (Yang et al., 2021). We use non-IID datasets from FedScale (Lai et al., 2022) (with its real client-data mapping) to keep the setting realistic with different data distributions across the client population.

Configuration of clients: We consider two distinct client setups: (***ResNet-18*** setup) We use the client in this setup to train a ResNet-18 model. Clients are considered to be mobile devices with limited computing capacity, available only when each has battery power and is connected to a data (*e.g.,* WiFi) network. This results in high variability in the number of mobile devices available to perform training tasks. As such, we let each client hibernate for a random interval within [0, 60] seconds to emulate the dynamic availability of typical mobile device behavior. This generates varying loads over time, as shown in Fig. 10(a), justifying the need for scaling with a serverless framework as well as LIFL. (***ResNet-152*** setup) The client in this setup trains the relatively heavyweight ResNet-152 model. The client is considered to be a server with substantial computing capacity and is highly available. As such, we keep clients in this setup always-on. This results in a more stable arrival pattern of model updates, as shown in Fig. 10(d).

We use a total of 20 physical nodes with 5 nodes used to run aggregators. We use 4 nodes as leaf/middle aggregators and dedicate one node to be the top aggregator. To deliver the benefits of a "serverful system" (**SF**), we always maximize the resource allocation to the aggregators and keep them warm throughout the experiment. For the serverless setup (**SL** and LIFL), we create aggregators on demand.

We use the remaining 15 physical nodes to run the clients. In the ***ResNet-18*** setup, since we consider clients to be compute-constrained mobile devices, we run eight clients on the same physical node, so each client only gets a small

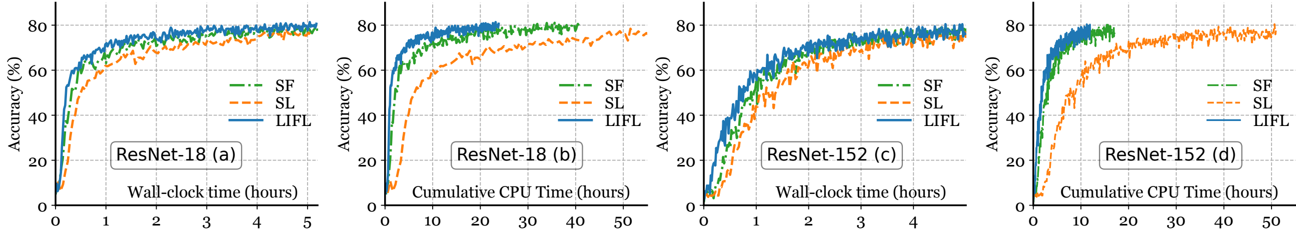
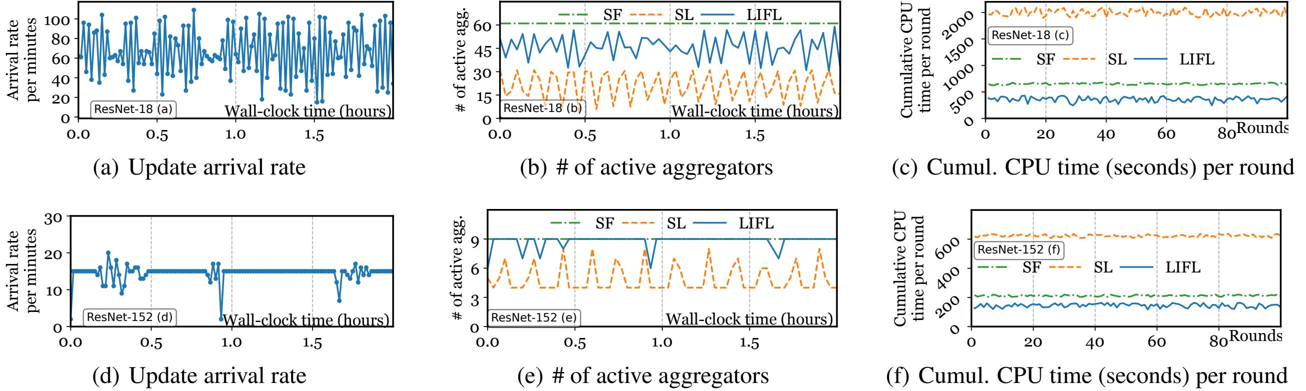


Figure 9. **ResNet-18:** (a) Time-to-accuracy and (b) Cost-to-accuracy; **ResNet-152:** (c) Time-to-accuracy and (d) Cost-to-accuracy.



(a) Update arrival rate

(b) # of active aggregators

(c) Cumul. CPU time (seconds) per round

(d) Update arrival rate

(e) # of active aggregators

(f) Cumul. CPU time (seconds) per round

Figure 10. **ResNet-18 (a, b, c), ResNet-152 (d, e, f):** Time series of arrival rate, number of active aggregators, and cumulative CPU time (seconds) per round.

share of the compute capacity of the physical node. Therefore, in the ***ResNet-18*** setup, we can keep 120 simultaneously active clients in each round. In the ***ResNet-152*** setup, we treat the client as a server node, so we dedicate a physical node for a ResNet-152 client. In this ***ResNet-152*** setup, we keep 15 simultaneously active clients in each round. The active clients are selected from a total of 2,800 real clients provided by FedScale (Lai et al., 2022).

6.3 Putting It All Together

(***ResNet-18***) **Time to Accuracy:** We compare the time-to-accuracy of LIFL against **SL** and **SF**. To reach 70% accuracy of ResNet-18 (Fig. 9 (a)), LIFL takes only 0.9 hours (wall clock time), which is 1.6× faster than **SF** (1.4 hours). Compared to **SL** which takes 2.4 hours, LIFL is 2.7× faster. The improvement with LIFL can be attributed to the shared memory data plane and the improved orchestration to effectively utilize resources, thereby reducing ACT (see §6.1).

The time spent by the **SL** aggregation service increases due to a combination of factors including sidecar overhead, function chaining, and simplistic orchestration. Frequent start-up of the aggregators in **SL** (Fig. 10(b)) adds delays to the aggregation (for the first arrival update in a round). This increased aggregation time of **SL** eventually hurts the time-to-accuracy (70%), making it even slower than **SF**.

(***ResNet-18***) **Cost savings with LIFL:** LIFL achieves significant cost savings compared to **SF** and **SL**. We focus on the cumulative costs (CPU time) consumed by the aggregation service to achieve a certain model accuracy. To reach the 70% accuracy level of ResNet-18 (Fig. 9 (b)), LIFL consumes 4.5 CPU hours, which is 1.8× less than **SF** (8 CPU hours). Further, **SF**, with its simplistic fixed resource allocation, keeps aggregators "always-on", constantly occupying its CPU allocation (Fig. 10(b)). LIFL adapts well to the arrival rate of model updates and re-plans (scales) the hierarchy accordingly, using resources to match demand. Also note that the LIFL's aggregator, when deployed as a Kubernetes pod or container, is also cheaper (smaller resource allocation) than **SF**, as LIFL requires less CPU to complete the same amount of aggregation tasks (Fig. 10(c)).

In contrast, **SL** consumes much more CPU (26 CPU hours) to achieve the 70% accuracy level of ResNet-18 (Fig. 9 (b)) compared to LIFL (4.5 CPU hours). Although **SL** has relatively fewer active aggregators over time (Fig. 10(b)), its data plane and sidecar overheads, and the CPU consumed for start-up results in **SL** having more than 5× the CPU consumption of LIFL. This higher CPU time cost per round (for the same amount of aggregation work completed) requires the cloud service provider to allocate far more resources to the aggregator (*e.g.,* as a pod), making a single aggregator in **SL** much more expensive than both **SF** and LIFL.

(***ResNet-152***) **Time to Accuracy:** Fig. 9 (c) shows the time-to-accuracy of the different alternatives for RestNet-152. To reach 70% accuracy, LIFL takes 1.9 hours (wall clock time), which is 1.15× faster than **SF** (2.2 hours). Comparatively, **SL** takes 3.2 hours. LIFL is 1.68× faster than **SL**. The heavy-weight sidecar, slow function chaining, function startup delays, and simplistic orchestration, are responsible for the larger time-to-accuracy of **SL** for ResNet-

152, just as we saw with the ResNet-18 workload, as well as with the microbenchmark analysis.

(***ResNet-152***) **Cost savings with LIFL:** As Fig. 9 (d) shows, LIFL again achieves significant cost savings (on cumulative CPU time) compared to **`SF`** and **`SL`**. To reach the 70% accuracy level of the ResNet-152 model, LIFL consumes 4.76 CPU hours, which is 1.43× less than **`SF`** (6.81 CPU hours). In contrast, **`SL`** consumes much more CPU (20.4 CPU hours) to achieve the same 70% accuracy level compared to LIFL. This again is consistent with what we observed from the ResNet-18 workload, highlighting the advantage of LIFL.

Summary: LIFL takes advantage of the fine-grained elasticity of serverless to scale the aggregation service based on load changes, saving CPU consumption compared to serverful alternatives. When comparing LIFL with **`SL`**, LIFL is even more compelling, with far lower CPU consumption because of LIFL's orchestration scheme and lightweight data plane (as we saw from the microbenchmark analysis). Thus, LIFL shows that it truly leverages the elasticity promise of the serverless computing paradigm.

7 Related work

We have discussed the pros and cons of prior work on serverful (Bonawitz et al., 2019; Huba et al., 2022) and serverless (Jayaram et al., 2022b;a; Chadha et al., 2021) FL systems in §2. LIFL goes beyond these prior designs with an optimized serverless infrastructure and efficient orchestration to truly realize the promise of serverless computing. We now discuss work related to LIFL from other perspectives.

Federated Learning: As a fast-evolving ML technology, a large body of work has been proposed for FL; the proposals in (McMahan et al., 2017; Li et al., 2020a; Nguyen et al., 2022; Li et al., 2020b; Reddi et al., 2020) focus on FL algorithms while others investigate how to select FL clients or datasets more intelligently (Lai et al., 2021; Liu et al., 2023a; Abdelmoniem et al., 2023; Jiang et al., 2022; Nishio & Yonetani, 2019; Shin et al., 2022; Guo et al., 2022; Lalitha et al., 2019; Elzohairy et al., 2022). (Liu et al., 2023b) seeks to schedule FL jobs across a shared set of FL clients with less contention and reduce job scheduling delays. These efforts are orthogonal to LIFL because LIFL focuses on system-level optimization of model aggregation of FL. This makes LIFL a good complement to these efforts by providing an efficient and high-performance FL system to bring various FL approaches to the ground.

Several open-source FL platforms, *e.g.,* Flame (fla, 2024), FATE (fat, 2023), OpenFL (ope, 2023), FedML (He et al., 2020), IBM federated learning (Ludwig et al., 2020) have been launched to facilitate the promotion and adoption of FL in both research and applications. These platforms assume themselves to be a serverful design with static, inflexible deployment, which makes them unprepared for large-scale FL. LIFL can be used as a representative case to guide the future development of these platforms.

Serverless computing optimization: Recent advances in serverless computing have triggered extensive research endeavors dedicated to optimizing its system design. Significantly, a prominent amount of investigation revolves around the enhancement of resource provisioning, function deployment, load balancing (Singhvi et al., 2021; Mittal et al., 2021; Tariq et al., 2020; Bhasi et al., 2021; Park et al., 2021a; Kaffes et al., 2022; Jin et al., 2023), runtime overhead reduction (Agache et al., 2020; Akkus et al., 2018; Shillaker & Pietzuch, 2020; Gadepalli et al., 2020; Oakes et al., 2018), and mitigation of function startup delay (Fu et al., 2020; Shahrad et al., 2020b; Lin & Glikson, 2019; Fuerst & Sharma, 2021; Schall et al., 2022; Ustiugov et al., 2021; Wang et al., 2021) within serverless platforms. Furthermore, substantial efforts have been directed towards addressing the data plane overheads inherent in serverless architectures (Qi et al., 2022; Jia & Witchel, 2021; Shillaker & Pietzuch, 2020; Yu et al., 2023), characterized by heavyweight function chaining and sidecar proxy.

Our work, combines the advantages of data plane optimization (*i.e.,* shared memory for hierarchical aggregation, in-place message queuing, event-driven sidecars, etc), to unlock the full potential of serverless computing, facilitating efficient and cost-effective FL in the cloud.

8 Conclusion

LIFL is an optimized serverless FL system aimed at making FL more efficient and significantly lowering its operational cost. LIFL adopts hierarchical aggregation to support FL at scale. Its serverless infrastructure leverages shared memory processing to offer high-speed yet efficient intra-node data plane and event-driven sidecar functionality to facilitate communication within hierarchical aggregation. LIFL's orchestration scheme adjusts the aggregation hierarchy based on load and, maximizes the utilization of shared memory through intelligent placement and reuse of aggregation function instances, thus saving the cost. Our evaluation shows that LIFL's optimized data and control planes improve the resource efficiency of the aggregation service by more than **5**×, compared to existing serverless FL systems, with **2.7**× reduction on time-to-accuracy for ResNet-18. LIFL also achieves **1.8**× better efficiency and **1.6**× speedup on time-to-accuracy than a serverful system. In training ResNet-152 to reach 70% accuracy, LIFL is **1.68**× faster than an existing serverless FL system, while reducing CPU costs by **4.23**×.

Acknowledgments

We thank the US NSF for their generous support through grants CRI-1823270, CNS-1818971, and Cisco for their generous gift and support.

References

Autoscaling - Knative. https://knative.dev/docs/serving/autoscaling/, 2023a. [ONLINE].

Autoscaling - OpenFaaS. https://docs.openfaas.com/architecture/autoscaling/, 2023b. [ONLINE].

extended Berkeley Packet Filter. https://ebpf.io/, 2023a. [ONLINE].

BPF-HELPERS - list of eBPF helper functions. https://manpages.ubuntu.com/manpages/focal/en/man7/bpf-helpers.7.html, 2023b. [ONLINE].

BPF maps. https://docs.kernel.org/bpf/maps.html, 2023c. [ONLINE].

Fate: An Industrial Grade Federated Learning Framework. https://fate.fedai.org/, 2023. [ONLINE].

Knative. https://knative.dev, 2023. [ONLINE].

Open Federated Learning (OpenFL) - An Open-Source Framework For Federated Learning. https://github.com/intel/openfl, 2023. [ONLINE].

Flame: a federated learning system for the edge. https://github.com/cisco-open/flame, 2024. [ONLINE].

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.

Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 419–434, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/agache.

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

Bhasi, V. M., Gunasekaran, J. R., Thinakaran, P., Mishra, C. S., Kandemir, M. T., and Das, C. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pp. 153–167, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386388. doi: 10.1145/3472883.3486992. URL https://doi.org/10.1145/3472883.3486992.

Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C., Konečný, J., Mazzocchi, S., McMahan, B., Van Overveldt, T., Petrou, D., Ramage, D., and Roselander, J. Towards federated learning at scale: System design. In *Proceedings of Machine Learning and Systems*, volume 1, pp. 374–388, 2019. URL https://proceedings.mlsys.org/paper/2019/file/bd686fd640be98efaae0091fa301e613-Paper.pdf.

Cai, Q., Chaudhary, S., Vuppalapati, M., Hwang, J., and Agarwal, R. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pp. 65–77, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383837. doi: 10.1145/3452296.3472888. URL https://doi.org/10.1145/3452296.3472888.

Chadha, M., Jindal, A., and Gerndt, M. Towards federated learning using faas fabric. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, pp. 49–54, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382045. doi: 10.1145/3429880.3430100. URL https://doi.org/10.1145/3429880.3430100.

Daga, H., Shin, J., Garg, D., Gavrilovska, A., Lee, M., and Kompella, R. R. Flame: Simplifying topology extension in federated learning. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, 2023.

Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., and Mishra, P. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 1–14, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/duplyakin.

Elzohairy, M., Chadha, M., Jindal, A., Grafberger, A., Gu, J., Gerndt, M., and Abboud, O. Fedlesscan: Mitigating stragglers in serverless federated learning, 2022. URL https://arxiv.org/abs/2211.05739.

Fu, S., Mittal, R., Zhang, L., and Ratnasamy, S. Fast and efficient container startup at the edge via dependency scheduling. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020. URL https://www.usenix.org/conference/hotedge20/presentation/fu.

Fuerst, A. and Sharma, P. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pp. 386–400, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446757. URL https://doi.org/10.1145/3445814.3446757.

Gadepalli, P. K., McBride, S., Peach, G., Cherkasova, L., and Parmer, G. Sledge: A serverless-first, light-weight wasm runtime for the edge. Middleware '20, pp. 265–279, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381536. doi: 10.1145/3423211.3425680. URL https://doi.org/10.1145/3423211.3425680.

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.

Guo, Y., Sun, Y., Hu, R., and Gong, Y. Hybrid local sgd for federated learning with heterogeneous communications. In *International Conference on Learning Representations*, 2022.

He, C., Li, S., So, J., Zeng, X., Zhang, M., Wang, H., Wang, X., Vepakomma, P., Singh, A., Qiu, H., Zhu, X., Wang, J., 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, 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.

Huba, D., Nguyen, J., Malik, K., Zhu, R., Rabbat, M., Yousefpour, A., Wu, C.-J., Zhan, H., Ustinov, P., Srinivas, H., Wang, K., Shoumikhin, A., Min, J., and Malek, M. Papaya: Practical, private, and scalable federated learning. In *Proceedings of Machine Learning and Systems*, volume 4, pp. 814–832, 2022. URL https://proceedings.mlsys.org/paper/2022/file/f340f1b1f65b6df5b5e3f94d95b11daf-Paper.pdf.

Jayaram, K., Muthusamy, V., Thomas, G., Verma, A., and Purcell, M. Lambda fl: Serverless aggregation for federated learning. In *International Workshop on Trustable, Verifiable and Auditable Federated Learning*, pp. 9, 2022a.

Jayaram, K. R., Muthusamy, V., Thomas, G., Verma, A., and Purcell, M. Adaptive aggregation for federated learning. In *2022 IEEE International Conference on Big Data (Big Data)*, pp. 180–185, 2022b. doi: 10.1109/BigData55660.2022.10021119.

Jayaram, K. R., Verma, A., Thomas, G., and Muthusamy, V. Just-in-time aggregation for federated learning. In *2022 30th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1–8, 2022c. doi: 10.1109/MASCOTS56607.2022.00009.

Jia, Z. and Witchel, E. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pp. 152–166, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446701. URL https://doi.org/10.1145/3445814.3446701.

Jiang, Z., Wang, W., Li, B., and Li, B. Pisces: Efficient federated learning via guided asynchronous training. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pp. 370–385, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394147. doi: 10.1145/3542929.3563463. URL https://doi.org/10.1145/3542929.3563463.

Jin, C., Zhang, Z., Xiang, X., Zou, S., Huang, G., Liu, X., and Jin, X. Ditto: Efficient serverless analytics with elastic parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, pp. 406–419, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702365. doi: 10.1145/3603269.3604816. URL https://doi.org/10.1145/3603269.3604816.

Kaffes, K., Yadwadkar, N. J., and Kozyrakis, C. Hermod: Principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pp. 289–305, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394147. doi: 10.1145/3542929.356346

8. URL https://doi.org/10.1145/3542929.3563468.

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

Lai, F., Dai, Y., Singapuram, S., Liu, J., Zhu, X., Madhyastha, H., and Chowdhury, M. Fedscale: Benchmarking model and system performance of federated learning at scale. In *International Conference on Machine Learning*, pp. 11814–11827. PMLR, 2022.

Lalitha, A., Kilinc, O. C., Javidi, T., and Koushanfar, F. Peer-to-peer federated learning on graphs, 2019. URL https://arxiv.org/abs/1901.11173.

Li, T., Sahu, A. K., Zaheer, M., Sanjabi, M., Talwalkar, A., and Smith, V. Federated optimization in heterogeneous networks. *Proceedings of Machine Learning and Systems*, 2:429–450, 2020a.

Li, T., Sanjabi, M., Beirami, A., and Smith, V. Fair resource allocation in federated learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020b. URL https://openreview.net/forum?id=ByexElSYDr.

Lin, P.-M. and Glikson, A. Mitigating cold starts in serverless platforms: A pool-based approach, 2019. URL https://arxiv.org/abs/1903.12221.

Liu, J., Lai, F., Dai, Y., Akella, A., Madhyastha, H. V., and Chowdhury, M. Auxo: Efficient federated learning via scalable client clustering. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, pp. 125–141, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9798400703874. doi: 10.1145/3620678.3624651. URL https://doi.org/10.1145/3620678.3624651.

Liu, J., Lai, F., Ding, D., Zhang, Y., and Chowdhury, M. Venn: Resource management across federated learning jobs, 2023b. URL https://arxiv.org/abs/2312.08298.

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. URL https://arxiv.org/abs/2007.10987.

McMahan, B., Moore, E., Ramage, D., Hampson, S., and Arcas, B. A. y. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pp. 1273–1282. PMLR, 20–22 Apr 2017. URL https://proceedings.mlr.press/v54/mcmahan17a.html.

Mittal, V., Qi, S., Bhattacharya, R., Lyu, X., Li, J., Kulkarni, S. G., Li, D., Hwang, J., Ramakrishnan, K. K., and Wood, T. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pp. 168–181, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386388. doi: 10.1145/3472883.3487014. URL https://doi.org/10.1145/3472883.3487014.

Nguyen, J., Malik, K., Zhan, H., Yousefpour, A., Rabbat, M., Malek, M., and Huba, D. Federated learning with buffered asynchronous aggregation. In *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*, volume 151 of *Proceedings of Machine Learning Research*, pp. 3581–3607. PMLR, 28–30 Mar 2022. URL https://proceedings.mlr.press/v151/nguyen22b.html.

Nishio, T. and Yonetani, R. Client selection for federated learning with heterogeneous resources in mobile edge. In *ICC 2019-2019 IEEE international conference on communications (ICC)*, pp. 1–7. IEEE, 2019.

Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 57–70, Boston, MA, July 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/oakes.

Park, J., Choi, B., Lee, C., and Han, D. Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, pp. 154–167, New York, NY, USA, 2021a. Association for Computing Machinery. ISBN 9781450390989. doi: 10.1145/3485983.3494866. URL https://doi.org/10.1145/3485983.3494866.

Park, J., Choi, B., Lee, C., and Han, D. Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21,

pp. 154–167, New York, NY, USA, 2021b. Association for Computing Machinery. ISBN 9781450390989. doi: 10.1145/3485983.3494866. URL https://doi.org/10.1145/3485983.3494866.

Qi, S., Kulkarni, S. G., and Ramakrishnan, K. K. Assessing container network interface plugins: Functionality, performance, and scalability. *IEEE Transactions on Network and Service Management*, 18(1):656–671, 2021. doi: 10.1109/TNSM.2020.3047545.

Qi, S., Monis, L., Zeng, Z., Wang, I.-c., and Ramakrishnan, K. K. Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pp. 780–794, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394208. doi: 10.1145/3544216.3544259. URL https://doi.org/10.1145/3544216.3544259.

Red Hat, Inc. Understanding the eBPF networking features in RHEL. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/assembly_understanding-the-ebpf-features-in-rhel-8_configuring-and-managing-networking, 2022. [ONLINE].

Reddi, S., Charles, Z., Zaheer, M., Garrett, Z., Rush, K., Konečný, J., Kumar, S., and McMahan, H. B. Adaptive federated optimization, 2020. URL https://arxiv.org/abs/2003.00295.

Schall, D., Margaritov, A., Ustiugov, D., Sandberg, A., and Grot, B. Lukewarm serverless functions: Characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, pp. 757–770, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450386104. doi: 10.1145/3470496.3527390. URL https://doi.org/10.1145/3470496.3527390.

Scheller, B. Best practices for resizing and automatic scaling in Amazon EMR. https://aws.amazon.com/blogs/big-data/best-practices-for-resizing-and-automatic-scaling-in-amazon-emr/, 2023. [ONLINE].

Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 205–218. USENIX Association, July 2020a. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/shahrad.

Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 205–218. USENIX Association, July 2020b. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/shahrad.

Shillaker, S. and Pietzuch, P. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 419–433. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/shillaker.

Shin, J., Li, Y., Liu, Y., and Lee, S.-J. Fedbalancer: Data and pace control for efficient federated learning on heterogeneous clients. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, pp. 436–449, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391856. doi: 10.1145/3498361.3538917. URL https://doi.org/10.1145/3498361.3538917.

Singhvi, A., Balasubramanian, A., Houck, K., Shaikh, M. D., Venkataraman, S., and Akella, A. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pp. 138–152, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386388. doi: 10.1145/3472883.3486981. URL https://doi.org/10.1145/3472883.3486981.

Tariq, A., Pahl, A., Nimmagadda, S., Rozner, E., and Lanka, S. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pp. 311–327, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421306. URL https://doi.org/10.1145/3419111.3421306.

Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., and Grot, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pp. 559–572, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172.

doi: 10.1145/3445814.3446714. URL https://doi.org/10.1145/3445814.3446714.

Wang, A., Chang, S., Tian, H., Wang, H., Yang, H., Li, H., Du, R., and Cheng, Y. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 443–457. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/wang-ao.

Yang, J., Shi, R., and Ni, B. Medmnist classification decathlon: A lightweight automl benchmark for medical image analysis. In *IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pp. 191–195, 2021.

Yu, M., Cao, T., Wang, W., and Chen, R. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 1489–1504, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/yu.

A MESSAGE FLOW OF INTRA-NODE AND INTER-NODE ROUTING

Intra-node routing: LIFL makes full use of its shared memory support to facilitate *zero-copy* exchange of model updates between aggregators. The shared memory object in LIFL is addressed by the object key, which is a 16 byte string randomly generated by the shared memory manager when it initializes shared memory objects. We also assign each aggregator a unique ID. The zero-copy data exchange between aggregators depends on delivering the object key, as the data is kept in place in shared memory.

LIFL utilizes eBPF's SKMSG (integrated in the eBPF-based sidecar), combined with eBPF's sockmap (Red Hat, Inc., 2022), to pass the object key between aggregators on the same node. Upon receiving the object key, the SKMSG program uses the ID of the source aggregator as the key to look up the *sockmap* to find the socket interface of the destination aggregator so that the object key may be delivered to it for access of the shared memory object.

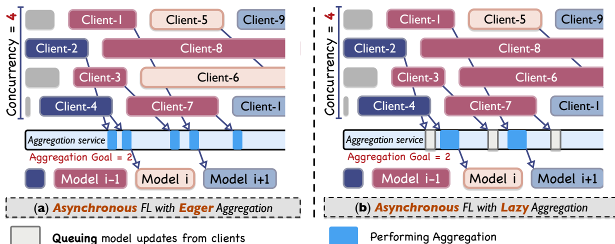


Figure 11. Asynchronous FL (Huba et al., 2022) with different aggregation timing ("Eager" and "Lazy").

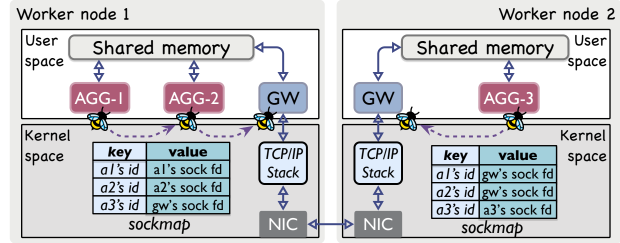


Figure 12. Intra-/inter-node direct routing within hierarchical aggregation.

Inter-node routing: When the source aggregator communicates with a destination aggregator on a different node, it sends the object key to the local gateway first. The local gateway uses the object key to retrieve the model update from shared memory and performs the necessary payload transformation. It then uses the source aggregator ID to look up the inter-node routing table to obtain the destination aggregator ID and the IP address of the remote node hosting the destination aggregator. The model update is sent through the remote node's gateway to the destination aggregator. The remote gateway stores the received model update in shared memory and uses SKMSG to notify the destination aggregator, along with the local object key.

Online hierarchy update: LIFL re-configures intra-/inter-node routes each time the hierarchy is updated. The routing manager in the LIFL agent takes the DAG input (generated by the TAG, §D) from the control plane that describes the connectivity between aggregators, and correspondingly updates routes into the inter-node routing table in the gateway and in-kernel *sockmap*, using the userspace eBPF helper, `bpf_map_update_elem()` (ebp, 2023b). The TAG describes the cross-level data dependency between aggregators.

B MODEL CHECKPOINTS

We support model checkpoints, where the model parameters are periodically saved to an external storage service to ensure data persistence and potential recovery in case of failures. The checkpointing occurs after the aggregator completes the aggregation of specified model updates, where the aggregator submits a request to the LIFL agent to perform model checkpoints asynchronously in the background. This prevents checkpoint delays from being added to the aggregation completion time.

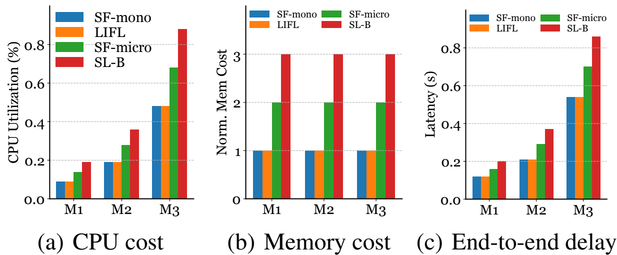
C Message queueing flow in LIFL: Receive (RX) and Transmit (TX)

On the receive (RX) path, protocol processing by the kernel TCP/IP stack is first performed. The gateway running in userspace receives the raw L7 payload from the kernel and then extracts the model updates (encoded as `tensor` data type), depending on the adopted L7 protocol (*e.g.,* gRPC, MQTT). We convert the model update from `tensor` data type to `NumpyArray` before writing it to shared memory, as Python's `multiprocessing` module does not support manipulation of the `tensor` data type. On the transmit (TX) path, the reverse payload processing is done.

D Abstraction for fine-grained control

To facilitate fine-grained control of LIFL's orchestration, we treat an aggregator process within a sandboxed runtime (*e.g.,* container) as the atomic unit for management. The control plane needs a generic means to describe connectivity between components and placement affinity. We make use of Topology Abstraction Graph (*TAG*) in Flame (Daga et al., 2023) to describe the aggregator-to-aggregator connectivity and aggregator-client connectivity. Each node in such a graph is associated with a "*role*" metadata, denoted as either aggregator or client. A "*channel*" metadata denotes the underlying communication mechanism (*e.g.,* intra-node shared memory, inter-node kernel networking) used for connectivity.

We configure the placement-affinity to facilitate locality-aware placement through the *groupBy* attribute in the channel abstraction, which accepts a string as a label to specify a group. Therefore, keeping the same label in the attribute allows us to cluster roles into a group. The LIFL coordinator enables necessary orchestration decisions, *e.g.,* runtime reuse and locality-aware placement, through manipulation of these abstractions (role and channel).



(a) CPU cost (b) Memory cost (c) End-to-end delay

Figure 13. Message queuing overheads.

E Maximum Service Capacity of Worker Nodes

LIFL actively monitors both $E_{i,t}$ and the arrival rate $k_{i,t}$ using the sidecar in §4.3. We determine the value of MC_i offline. We incrementally increase the arrival rate k_i to node i. Let k_i' and E_t' denote the arrival rate and average execution time at the point we observe a significant increase in E_i. This indicates that node i is becoming overloaded and we estimate MC_i as $k_i' \times E_i'$.

F In-place message queueing benefit

We examine LIFL's in-place message queuing through a comparison with the serverful and serverless alternatives depicted in Fig. 5, including the *monolithic* serverful setup (denoted as **`SF-mono`**), the *microservice*-based serverful setup (denoted as **`SF-micro`**), and the *basic* serverless setup (denoted as **`SL-B`**). We quantify the overheads of message queuing for a single model update transfer between the client to the aggregator. We consider three metrics: (1) the total memory consumed for queuing the model update along the data pipeline; (2) the CPU cycles spent in the data pipeline; and (3) the end-to-end networking delay from the client to the aggregator. Note that we exclude the overhead on the client-side. We consider three ML models with distinct sizes: (**`M1`**) ResNet-18 (∼44MB), (**`M2`**) ResNet-34 (∼83MB), and (**`M3`**) ResNet-152 (∼232MB).

Fig. 13 shows the results of CPU, memory cost and end-to-end networking delay. The memory consumption in **`SF-mono`** is mainly from the in-memory queue inside the aggregator. For LIFL it is primarily consumed by the shared memory used to buffer the model update. But, **`SL-B`** consumes 3× more memory than **`SF-mono`** and LIFL. The extra memory consumption of **`SL-B`** comes from the use of sidecar and message broker, both of which need to locally buffer the model update. **`SF-micro`**, on the other hand, saves one queuing stage at the sidecar, but still incurs the queuing at the message broker and consuming extra memory. LIFL's in-place message queuing totally eliminates these unnecessary queuing stages.

Looking at the CPU consumption, LIFL is ∼1.5× and ∼1.9× less than **`SL-B`** and **`SF-micro`**, respectively. In terms of the end-to-end networking delay (client to aggregator), LIFL is ∼1.3× and ∼1.7× less than **`SL-B`** and **`SF-micro`**, respectively. LIFL's improvement in CPU cost and networking delay, compared to **`SL-B`** and **`SF-micro`**, are also a result of the elimination of the sidecar and message broker from the data pipeline, and the message queuing if far more efficient. This illustrates the benefits of LIFL's in-place message queuing, achieving the equivalent efficiency and performance of a monolithic, serverful design (with far less resource consumption as we see for typical FL applications).

F.1 Stateful “tax” in LIFL

The per-node gateway is a key component that enables a number of data plane functionalities in LIFL, including in-place message queuing and inter-node data transfer. Unlike stateless aggregators, the gateway is deployed as a stateful, persistent component on every LIFL worker node. This raises the concern about the stateful “tax”, *i.e.,* the CPU/memory cost of having stateful components in the FL system.

On the other hand, a stateful “tax” of some form commonly exists in serverful and serverless alternatives, as shown in Fig. 5. The stateful component in a monolithic serverful setup is the aggregator itself, running as an “always-on” monolith. In the microservice-based serverful setup, the message broker is the stateful component, as is the case for the basic serverless setup. We quantitatively compare the stateful “tax” of LIFL’s gateway with serverful and serverless alternatives in Fig. 5. The result in §4.2 shows that stateful “tax” in LIFL is the lowest.

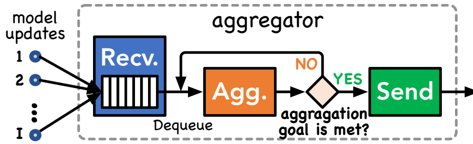


Figure 14. Step-based processing model.

G Step-based Processing Model

The basic processing model of an LIFL aggregator can be abstracted as a multiple-producer, single-consumer pattern, as shown in Fig. 14. Multiple upstream producers (clients or aggregators) are mapped to a single consumer (aggregator only). The single consumer gathers model updates from assigned producers and computes the aggregated model update.

Looking deeper into the aggregator, LIFL adopts a step-based processing model. At the core of this design is a processing pipeline of three steps: **(1)** `Recv`**:** Receive model updates from all assigned producers. The received model update is enqueued in a FIFO queue. In LIFL, the object key of the model update is enqueued as the actual model update resides in shared memory; **(2)** `Agg`**:** Aggregator dequeues a model update from the FIFO queue in `Recv` and then aggregates it. The `Agg` step checks if the aggregation goal is met after the dequeued update is aggregated. If the aggregation goal is not met, `Agg` is repeated until the aggregation goal is met, before moving to `Send`; and **(3)** `Send`**:** sends the final model update to the designated consumer. The execution of `Recv` and `Agg` overlaps to enable eager aggregation, *i.e.,* once the `Recv` step receives a model update, it immediately passes the model update to `Agg` step for aggregation.