ARCHITECTURES AND PROTOCOLS FOR ORCHESTRATION OF DISTRIBUTED LEARNING SYSTEMS

Andrea Pinto, B.S., M.S.

A Dissertation Presented to the Faculty of the Graduate School
of Saint Louis University in Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

2026

© Copyright by
Andrea Pinto, B.S., M.S.
ALL RIGHTS RESERVED

2026

COMMITTEE IN CHARGE OF CANDIDACY:

Associate Professor Flavio Esposito,
Chairperson and Advisor

Assistant Professor Nan Cen

Assistant Professor Emilio Paolini

Associate Professor Reza Tourani

Dr. Yuefeng Wang

TABLE OF CONTENTS

List of Tables

List of Figures

CHAPTER 1: Introduction

As Machine Learning (ML) models scale to unprecedented sizes, distributing these systems effectively is paramount. However, the paradigm of scaling has fundamentally changed; it is no longer sufficient to simply increase GPU compute. Instead, system bottlenecks have shifted away from processing power and into the broader operating ecosystem, exposing severe limitations in memory capacity and network I/O. This reality complicates the dual challenge of modern ML architectures: managing massive centralization of computation [**1**] alongside the increasing decentralization of data generation [**2**]. Whether training in a highly controlled datacenter or in a federated edge environment, communication overhead and hardware constraints severely compromise efficiency [**3**, **4**]. To address this, we propose unified optimization techniques for distributed system orchestration, ensuring performance stability across both centralized training in Data Centers and Distributed Edge Federated Learning workloads.

1. Data Center and Edge Training

Federated Learning (FL) [**2**] and Datacenter (DC) [**5**] training represent fundamentally opposing paradigms in machine learning orchestration [**6**], primarily distinguished by their approach to data [**2**] and resource coupling [**1**]. In traditional DC training, the system is designed around the "move-data-to-compute" principle: datasets are aggregated into a Cloud GPU Cluster, where high-bandwidth interconnects (e.g., Ethernet, NVLink, InfiniBand) guarantee synchronous, large-batch optimization on typically i.i.d. (Independent and Identically Distributed) data [**7**, **8**]. Optimization techniques introduce varying layers of complexity; several orchestration architectures have been proposed that focus solely on controlling hardware resources, without requiring a deep understanding of underlying communication patterns [**5**, **9**–**14**]. Other approaches focus on

optimizing network utilization at scale. These include restricting synchronization to specific networking patterns [**15**, **16**], leveraging programmable network resources at the cost of expensive switches and complex in-network computation logic [**17**], or prioritizing physically proximate resources to minimize network load spreading [**5**, **18**]. Consequently, there is a clear need for an orchestration architecture that improves allocation performance while actively mitigating networking bottlenecks. In contrast, FL operates on a "move-compute-to-data" basis, necessitated by privacy constraints and the impracticality of transmitting raw edge data [**2**]. Here, the training process is decoupled across thousands to millions of heterogeneous, unreliable edge devices (clients) that perform local SGD on non-I.I.D. data [**19**]. This paradigm shift moves the primary system bottleneck from compute throughput (in DC) to communication latency and bandwidth (in FL). The system must orchestrate global convergence across unstable Wide Area Networks (WANs) while managing extreme heterogeneity in computation, networking, and data distribution (e.g., stragglers) that cannot be tightly controlled as in a datacenter. Therefore, optimizing these systems requires addressing resource bottlenecks on both the constrained edge devices handling the data [**20**–**23**] and the frequently congested network paths that inflate training times. To this end, recent approaches aim to select optimal clients based on computing capacity, network conditions, and data quality to accelerate convergence in accuracy over time [**24**–**26**]. In the next section, we show how we close these gaps and give an overview of the main contributions of this thesis.

2. Challenges of Existing Distributed Training Systems

The rapid evolution of Machine Learning (ML) models, particularly Large Language Models (LLMs), has driven an unprecedented demand for computational resources. Traditionally, training these models relies on centralized Cloud Data Centers and distributed Edge environments. However, these paradigms face several critical challenges across different layers.

In Cloud environments, efficient orchestration is critical, yet standard operating systems and orchestrators, such as Linux and Kubernetes, lack intrinsic, holistic visibility

into cluster-wide resource utilization without relying on continuous monitoring. Furthermore, the exponential growth of ML model sizes necessitates massive parallel training, introducing significant network contention. While specialized interconnects like NVLink offer high-bandwidth GPU-to-GPU communication, most data centers remain constrained by standard Ethernet infrastructure, typically limited to 20–100 Gbps. In these environments, the backbone often risks saturation during synchronization steps, creating a scalability ceiling where adding compute resources yields diminishing returns.

At the Near Edge layer, there is a critical bottleneck regarding network load during Federated Learning (FL) sessions, exacerbated by the surge in device connections via 5G. The necessity for selected clients to communicate with a Cloud-based Parameter Server (PS) at every training round creates significant uplink overhead, delaying progress.

At the Far Edge layer, participating clients are often resource-constrained. Such limitations preclude the use of valuable local data, as these devices lack the computational capacity to execute the full model locally without exposing raw data.

Finally, a persistent challenge in FL is predicting which clients are most valuable to include in a training session. This requires complex algorithms that must balance computational speed and data quality to minimize time-to-accuracy under fluctuating network conditions.

3. Mechanisms in Distributed Training

To address these challenges, this dissertation proposes a multi-tier architecture that decouples computational and networking requirements across three distinct layers: the Cloud Data Center, the Near Edge, and the Far Edge. As illustrated in Figure 1, the proposed training mechanisms are distributed across this hierarchy.

We present this architecture using a top-down approach, transitioning from centralized, high-performance Data Centers down to distributed Edge computing environments, and ultimately to resource-constrained end devices. The remainder of this work follows this structural flow. Furthermore, please note that throughout this dissertation, terms

such as “Edge Training,” “Distributed Training,” and “Cloud Training” may be used adaptively, reflecting the specific environment and the hierarchical layer in which the operations take place.

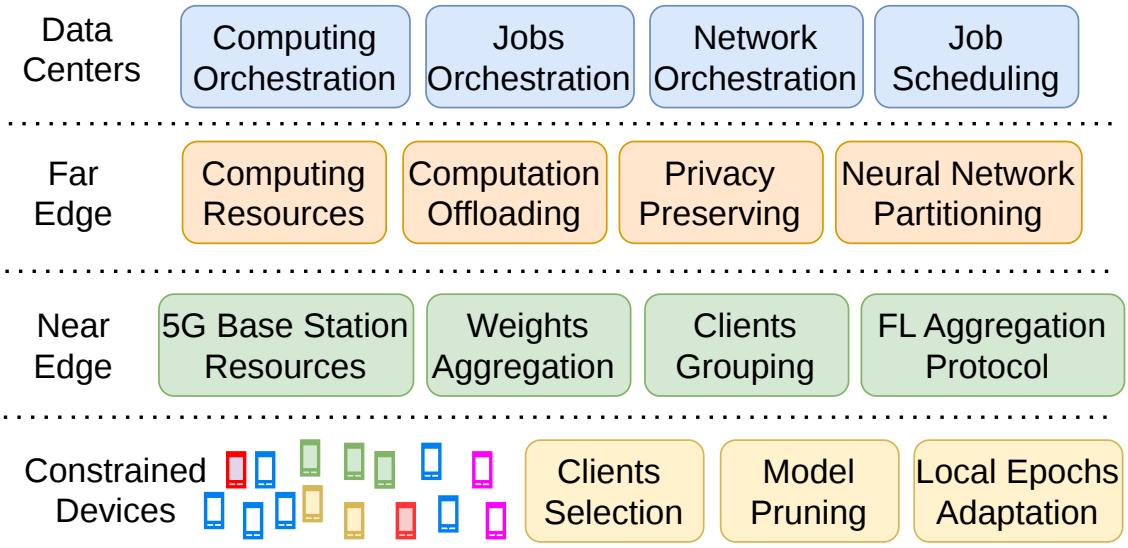


FIGURE 1. We propose a multi-layer architecture spanning data center resource orchestration, edge computing, and device management. At the Data Center layer, we introduce technologies designed to mitigate networking bottlenecks within data centers (Chap. 3). At the Far Edge, we leverage localized computation to offload tasks from resource-constrained devices (Chap. 4). Furthermore, we utilize the Near Edge, specifically 5G base stations, to manage client traffic and reduce network congestion (Chap. 5). Finally, we present an advanced client selection algorithm incorporating model pruning and epoch adaptation, which reduces federated learning costs by optimizing the process via LLMs. (Chap. 6)

Within Data Centers, we deploy a Distributed Asynchronous Orchestration mechanism. By having each node track the global system load and exchange utility values, this architecture assigns ML job requests to available resources. This mechanism maximizes computational utilization and effectively masks network overhead, reducing synchronization delays.

At the Near Edge Layer, we exploit the programmability of the 5G infrastructure. We introduce a mechanism for in-network aggregation directly within the 5G stack. This accommodates a larger number of concurrent users and drastically reduces uplink traffic to the Parameter Server, shortening communication times.

At the Far Edge Layer, we employ the Federated Split Learning (FSL) paradigm. This mechanism leverages client data by executing only a fraction of the neural network locally while offloading the remaining computation to the network edge. It incorporates privacy-preserving mechanisms for parameter exchange and relies on a neural network optimization problem to determine the ideal partitioning ratio.

Finally, we introduce an overarching intelligent mechanism utilizing Large Language Models (LLMs) to optimize client selection. By feeding the system state space into a modified LLM data processing flow, combined with Model Pruning and Epoch Adaptation mechanisms, we dynamically reduce the amount of data sent and the number of local epochs each client runs.

4. Thesis Contributions

This dissertation systematically addresses the aforementioned challenges by connecting specific architectural mechanisms to the proposed layers. The core contributions are structured as follows.

Chapter 2: Background and Related Work provides the foundational context on distributed learning paradigms, Cloud orchestration limitations, and Edge computing constraints. Chapter 3: Improving Cloud Orchestration Training Efficiency contributes to the Cloud Data Center layer. It details Plebiscito, the distributed asynchronous orchestration architecture that tracks system load via utility values to mask Ethernet network bottlenecks and improve overall cluster throughput. Chapter 5: Federated Learning with In-Network Aggregation addresses the Near Edge mechanisms. It introduces FLAG, which implements in-network aggregation within the programmable 5G stack to reduce uplink traffic to the Parameter Server and mitigate the straggler problem. Chapter 4: Distributed Edge Federated Resources focuses on Far Edge mechanisms. It formulates the Federated Split Learning (FSL) optimization problem to balance privacy and training performance, offloading computation so severely resource-constrained devices can participate safely. Chapter 6: Intelligent Client Selection via LLMs introduces the advanced client selection mechanism. It demonstrates how LLMs, combined with model pruning

and epoch adaptation, can dynamically manage federated sessions based on real-time state spaces to achieve faster time-to-accuracy. Chapter 7: Conclusion and Future Work summarizes the core contributions of this multi-layer architecture and discusses future research directions.

Next, we present a literature review to identify the problems relative to the state of the art and to provide a deeper understanding of the main trade-offs.

CHAPTER 2: Background and Related Work

In this chapter, we analyze related work to highlight the current state of the art in both data center and edge environments. Specifically, we explore the challenges associated with Cloud computing resources and demonstrate how data center networks can become a bottleneck for distributed machine learning (ML) jobs, especially when the number of required resources increases and network traffic slows down the training process. Next, we introduce the challenges of machine learning architectures at the edge, highlighting how resource and network constraints can limit training performance. Finally, we discuss the performance of the clients participating in the training sessions, exploring how to optimize client selection and improve overall training efficiency in terms of time-to-accuracy.

1. Data Center Machine Learning Networking Bottlenecks

Efficient orchestration of both compute and network resources is essential for modern distributed ML, yet most prior systems lack bandwidth awareness.

Production systems. Deep Learning training jobs in multitenant production clusters are orchestrated by infrastructures such as Kubernetes [**9**] or YARN [**10**], where jobs are allocated on dedicated GPUs, leading to low performance and utilization [**27**]. In addition to these systems, various optimizers, such as Ray [**28**] and Clipper [**12**], offer flexible task management and model serving, which speeds up training while deferring placement to underlying schedulers without built-in bandwidth-aware logic. Plebiscito can be used to augment the placement directly on such architectures to augment their placement capabilities. The term Machine Learning as a Service (MLaaS) has been coined as users tend to train ML/DL models on Cloud resources. MLaaS clusters share resources orchestrated by a centralized scheduler that aims to reduce the JCT and job

makespan [**5**, **18**, **29**]. For example, PAI [**5**] allocations worsen JCT because their solutils are not flexible and are oriented towards distributed training jobs.

Fairness-Driven Allocation. Here is a polished version of your text.

The primary issues in the original draft were the grammatical structure of the Tiresias sentence and a repetitive conclusion. You had two consecutive sentences starting with ”However” that basically made the same point about network resources and contention.

I have smoothed out the transitions and combined the critique of the existing systems to lead naturally into the introduction of Plebiscito.

DRF [**14**] provides fair resource allocation in systems with multiple resource types by generalizing max-min fairness and seeking to maximize the minimum dominant share across all users. THEMIS [**13**] addresses the unfairness of placement-sensitive characteristics in DL jobs by proposing a long-term fairness objective. Meanwhile, the Least Attained Service algorithm in Tiresias [**18**] attempts to balance resource allocation, but it does so without considering actual resource contention or optimizing bandwidth. While these technologies successfully enhance the fairness of computational resources, they typically overlook the network. Specifically, they ignore the temporal contention on network links that frequently causes training stalls. As our results demonstrate, Plebiscito addresses this gap by enhancing its placement strategies with network awareness.

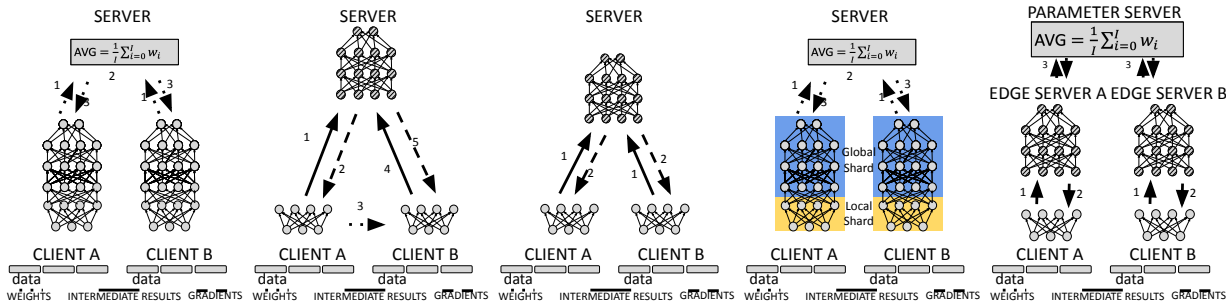
Communication Scheduling vs. Placement. While Plebiscito addresses the *spatial* allocation of jobs, a complementary class of schedulers optimizes the *temporal* usage of resources. Systems like Muri [**16**] and Cassini [**15**] exploit the iterative nature of DL training to interleave computation and communication, thereby smoothing traffic bursts on established links. Although recent work such as Crux [**30**] notes that precise interleaving is difficult to scale, these approaches operate at a different layer of the stack than Plebiscito. Flow schedulers manage congestion on fixed paths, whereas Plebiscito optimizes the topology itself before execution begins. Consequently, Plebiscito is orthogonal to these solutions; it provides a superior initial placement with high residual bandwidth, upon which fine-grained schedulers like Crux can be layered to further optimize tail latencies.

Parameter Server Oriented. The PS architecture [**31**] has been shown to be competitive with ring all-reduce, managing to reduce bandwidth overhead while delivering comparable, if not improved, performance, as demonstrated by BytePS [**32**]. Dynamic orchestrators, such as Optimus [**33**], address this challenge by predicting the optimal design and placement of the PS architecture, dynamically adjusting the number and location of PS instances based on workload conditions. Such strategies can complement Plebiscito, which provides a bandwidth-aware allocation mechanism that can operate underneath these dynamic policies. In our current implementation, we focus on the allocation of both PS-based and ring all-reduce jobs, demonstrating that Plebiscito can serve as a foundational mechanism for bandwidth-efficient scheduling across architectures. By embedding up-to-date link utilization into a decentralized, asynchronous max-consensus auction, Plebiscito distinguishes itself from these prior works by maintaining a global resource view without centralized bottlenecks. This allows it to avoid network hotspots before they form, delivering provable $(1 - \frac{1}{e})$ approximation and convergence guarantees.

2. Edge Distributed Learning Paradigms

The main difference between data centers and edge training is in the data management process. In data centers, we have full control over the data and how to share it across different machines, whereas in FL settings, data is kept private. We introduce FL distributed learning paradigms and highlight the specific limitations our proposed Federated Split Learning (FSL) architecture aims to address.

Federated Learning (FL) [**34**–**38**] is a decentralized deep learning technique where data sources "owned" by multiple clients are used to train a local model (Figure 1a). A logically centralized parameter server orchestrates weight sharing. Initially, the server sends randomly initialized weights to each client. Clients train these local models until a specific accuracy threshold is reached, at which point the parameter server retrieves, averages, and overwrites the client weights (Figure 1a – steps 1 to 3). While FL preserves privacy by keeping data local, it suffers from a significant drawback: *it requires*



(A) Federated Learning (FL) (B) Split Learning (SL) (C) Parallel Split Learning (PSL) (D) Federated Reconstruction (FRC) (E) Federated Split Learning (FSL)

FIGURE 1. Distributed NN Training Architectures: *(a) Federated Learning.* The NN is in the client. The parameter server calculates the average weights among clients and overrides the local weights. *(b) Split Learning.* The server partition sequentially trains with each of the clients. Client weights are shared with the next training client. *(c) Parallel Split Learning.* The server trains clients' outputs in batches in parallel, but the clients' weights are kept private. *(d) Federated Reconstruction.* Multiple clients train local and global shards of weights alternately in parallel, and only the global shards are preserved and averaged in between epochs. *(e) Federated Split Learning.* Multiple Edge Server and Client pairs train simultaneously. The Edge Servers' weights are averaged by a Parameter Server. The clients' weights are kept private.

each client to have sufficient resources to train the full neural network. In contrast, our FSL architecture addresses this limitation by partitioning the model, allowing resource-constrained devices to offload the majority of the training workload to an edge server while maintaining data privacy.

Split Learning (SL) [**20**] addresses the resource constraints of FL by computationally splitting the neural network into two partitions. The client trains only a small fraction of the model (e.g., the first few layers), while the server trains the rest. During forward propagation, the client sends the output of its last layer (hidden variables) to the server (Figure 1b – step 1). The server completes the forward propagation, calculates the loss, and initiates backward propagation. Gradients are sent back to the client to update the local partition (step 2). However, standard SL is inherently sequential; the server partition pairs with one client at a time (steps 3–5), leading to long convergence times and

poor scalability. Our contribution differs fundamentally here: unlike standard SL, FSL enables multiple client-server pairs to train in parallel, effectively solving the high latency and scalability issues inherent to sequential Split Learning.

Hybrid and Parallel Approaches. Recent works such as Parallel Split Learning (PSL) [**23**] and SplitFed [**39**] (Figure 1c) have attempted to parallelize SL. In these systems, multiple clients compute forward propagation in parallel, and their outputs are sent to a single server. The server aggregates these outputs (often as a large batch) and computes gradients. However, PSL creates a synchronization bottleneck: the central server acts as a barrier, often waiting for all clients to report before performing a global gradient update. Furthermore, sending all intermediate data to a single node can overwhelm the bandwidth. Our FSL architecture improves upon this by decoupling the training process. Instead of a single server aggregating gradients at every step, FSL utilizes independent Edge Servers that train alongside clients. A Parameter Server then averages the weights of these Edge Servers periodically (similar to FedAvg), rather than synchronizing gradients at every iteration. This design provides superior robustness, reduces bottlenecks, and enables asynchronous scalability that PSL cannot achieve.

Optimization Techniques. We note that several remedies exist to mitigate transmission delays in split architectures, such as Knowledge Distillation [**40**], BottleNeck layers [**41**, **42**], or Early Exits [**43**]. While these methods optimize transmission, they do not address the architectural bottlenecks of sequential processing (in SL) or synchronized central aggregation (in PSL). FSL integrates the benefits of FL's parallel aggregation with SL's resource efficiency, offering a generalized architecture in which these optimization techniques can be applied as complementary enhancements (Chapter 4).

We next analyze networking bottlenecks and how the FL edge training architecture strains the underlying network.

3. Networking Overhead in Edge Learning

Our work builds upon previous efforts to scale Federated Learning (FL), which can be broadly categorized into architectural modifications and client-centric optimizations.

Architectural Approaches to Scaling Federated Learning. Our design is inspired by the concept of in-network aggregation introduced in SwitchML [**44**], which demonstrated that programmable network hardware can significantly accelerate gradient aggregation in data center training. Unlike SwitchML, however, FLAG operates in wireless and federated settings, embedding aggregation within the 5G user-plane (SDAP) to cope with heterogeneous, asynchronous, and lossy client updates. Rather than achieving exact packet-level reduction in a lossless fabric, FLAG performs deadline-driven partial aggregation and bias correction at the gNB, co-optimizing communication efficiency and convergence under realistic network dynamics.

In addition to in-network aggregation solutions, other approaches to reduce communication load have been proposed. For example, Hierarchical FL (HFL), introduced intermediate aggregators between the clients and the central PS [**45**]. While HFL effectively lowers the number of direct transmissions to the PS, it typically relies on application-layer edge servers, which can increase end-to-end latency and require careful tuning without leveraging the line-rate processing capabilities of the underlying network fabric [**46**]. Recent wireless FL studies have also explored tighter integration between learning and radio processing. For example, FL-aided dual-side channel estimation has been proposed to reduce pilot overhead and enable collaborative model training at the base station [**47**]. While effective for PHY-layer learning tasks, these approaches focus on model sharing and estimation accuracy, and do not address the systemic communication bottlenecks arising from repeated model exchanges across the user plane and backhaul, which are the primary target of FLAG.

Other works have explored fully decentralized, Peer-to-Peer (P2P) architectures, such as Totoro [**26**], to eliminate the central Parameter Server entirely. However, P2P communication can be highly inefficient in volatile mobile environments, often aggravating communication overheads and leading to slower convergence.

Client-Centric and Algorithmic Optimizations. Another category of related solutions places the optimization burden on the clients and the learning algorithm. Advanced client selection is a key technique in this area, with systems like Oort [**25**], and PyramidFL [**24**]

which intelligently choose clients based on data quality and resource availability. While this mitigates the impact of stragglers, it fundamentally assumes the network is a passive bottleneck and does not address the backhaul congestion for the selected clients. These algorithmic approaches are complementary to our work and could be combined with our system.

Asynchronous FL mechanisms have also been proposed to mitigate stragglers and improve training robustness, such as AiFed [**48**], which dynamically adapts aggregation to heterogeneous client availability. These solutions operate at the algorithmic level and assume a conventional communication pipeline, whereas FLAG complements them by restructuring how model updates are transported and aggregated within the network itself.

In contrast to these approaches, we propose Federated Learning in-network AGgregation (FLAG), a systems-level, network-centric solution (Chapter 5). By embedding aggregation directly into the 5G gNB's data plane, FLAG addresses the systemic congestion that architectural solutions like HFL only partially mitigate and that client-centric approaches like PyramidFL ignore. This makes FLAG a fundamentally different approach for scaling FL in real-world mobile networks.

3.1. Edge Distributed Training Optimization Challenges. Improving the efficiency of FL through intelligent client selection has received significant attention. Early FL systems relied on random participant sampling, which often leads to suboptimal convergence due to heterogeneity in both data and system capabilities [**49**, **50**].

To address this, recent studies have proposed utility-aware selection mechanisms that jointly consider statistical contribution and system efficiency. Oort [**51**] introduces guided participant selection by explicitly modeling a utility function that combines (i) statistical utility, derived from the magnitude of recent training loss and gradient contributions, and (ii) system utility, based on computation speed and communication latency. Oort formulates client selection as an exploration–exploitation problem and demonstrates significant improvements over random selection. In large-scale experiments across

real-world datasets (e.g., OpenImage [**52**], Reddit [**53**], StackOverflow [**54**]), Oort improves time-to-accuracy by $1.2\times$–$14.1\times$ and final model accuracy by 1.3%–9.8% compared to existing selection strategies. PyramidFL [**24**] further refines this direction by proposing a fine-grained client selection architecture that exploits heterogeneity not only across selected and non-selected clients, but also within the selected set. PyramidFL adapts local training iterations and parameter update sparsification based on ranking-based utility profiling. On the FedScale benchmark with OpenImage and MobileNet/ShuffleNet models, PyramidFL reports $2.71\times$–$13.66\times$ speedup in time-to-accuracy and 3.68%–7.33% improvements in final model accuracy compared to Oort. AUCTION [**55**] formulates client selection as a RL problem under budget constraints. It encodes client attributes—including data size, data quality (e.g., mislabel rate, non-IID skew), and claimed cost—into an encoder–decoder policy network trained via policy gradient methods. AUCTION demonstrates significant robustness in scenarios with low-quality or mislabeled data.

More broadly, recent work has investigated the use of Large Language Models (LLMs) as general-purpose decision engines for networking systems. NetLLM [**56**] demonstrates that pre-trained LLMs can be adapted to networking tasks such as viewport prediction, adaptive bitrate streaming, and cluster job scheduling through multimodal encoders and low-rank adaptation, achieving 10–36% performance gains over task-specific neural models. These results highlight the potential of foundation models as reusable backbones for networking control problems.

In parallel, recent advances in preference-based optimization have proposed alternatives to RL for aligning LLM with desired behaviors. DPO [**57**] reformulates the Reinforcement Learning from Human Feedback (RLHF) objective by leveraging a closed-form mapping between reward functions and optimal policies under a KL-constrained formulation, enabling preference alignment through a simple classification loss without explicit reward modeling or policy rollouts. FLLLM adopts this optimization principle in a networking context, using DPO to learn a client-selection policy from pairwise

comparisons between candidate client subsets, thereby avoiding explicit reward modeling and environment interaction during training. While existing FL approaches rely either on carefully engineered utility formulations (e.g., Oort [**25**], PyramidFL [**24**]) or explicit RL pipelines (e.g., AUCTION [**55**]), our approach treats client subset selection as a preference-driven policy learning problem. Utility-based methods require task-specific balancing of statistical and system factors and may need re-tuning as deployment conditions evolve, whereas RL-based approaches depend on policy-gradient optimization and reward design, introducing additional training complexity. In contrast, by leveraging a pre-trained LLM as a general-purpose policy backbone and optimizing it through preference comparisons between candidate subsets, FLLLM enables stable adaptation to heterogeneous and non-stationary FL environments without task-specific reward engineering. Next, we will detail how similar problems apply to Data Centers, specifically in terms of resource orchestration and networking bottlenecks.

We present our findings through a top-down analytical framework. First, we examine network optimization and resource orchestration within the centralized data center. We then transition to Federated Learning optimization at the network edge, detailing existing distributed training capabilities and identifying how current networking architectures introduce severe latency bottlenecks. Building upon this, we explore how harnessing far-edge computing resources can accelerate training velocity, specifically demonstrating the capacity of 5G-based networks to mitigate these communication delays. Finally, we establish the need for dynamically adaptable orchestration in volatile environments. To address this, we introduce the application of Large Language Models to autonomously infer optimal client selection policies and dynamically adapt system hyperparameters, entirely eliminating the need for manual human intervention.

CHAPTER 3: Improving Cloud Orchestration Training Efficiency

ML-as-a-Service (MLaaS) [**5**] represents a paradigm shift that enables the use of provider resources to train machine learning jobs. ML jobs require distinct levels of orchestration compared to traditional computing jobs for several reasons. First, these jobs typically require GPU-based computation. Furthermore, due to high dataset dimensionality, resources must often be scaled out to minimize the Job Completion Time (JCT). However, this necessitates managing network and resource orchestration patterns, which are often prolonged and exhibit learnable compute and traffic characteristics.

The challenge highlighted here is that a provider's infrastructure must host many simultaneous jobs from different peers, all of which must be allocated to available resources. Consequently, when resources are reserved, traditional allocators such as Linux, Slurm, and Kubernetes merely check for availability, leading to suboptimal orchestration and poor performance management.

As highlighted in Chapter 1, there are efforts to align the resources orchestration and to reduce the collisions by optimizing the JCT. We find that the proposed solutions are either of a synchronization nature or do not fully account for the effects of incorrect placement on both the network and resource availability. To solve this, we introduce Plebiscito to remove the centralized controller bottleneck in Cloud orchestration environments, while introducing a fully distributed asynchronous protocol to efficiently use resources. Next, we introduce the background and the motivations behind this choice.

1. Distributed ML Orchestration: Background and Challenges

In a distributed environment, once the GPUs finish their calculations, the primary bottleneck in machine learning training becomes the speed and efficiency of the network. To motivate our work, we first review the communication patterns that create network

pressure. We then show why job placement is the critical factor in this communication-bound regime and conclude by enumerating the core challenges any effective orchestration solution must address.

1.1. Communication Patterns in Data-Parallel Training. The predominant method for scaling ML training is data-parallelism, where a model is replicated across multiple worker nodes and the training dataset is partitioned among them. The process is iterative, typically using a variant of Stochastic Gradient Descent (SGD). In each iteration t, every worker i locally computes a model update, $\Delta(x^t, D_i^t)$, based on the current model parameters x^t and its local data minibatch D_i^t. These updates are then aggregated across all n workers to produce the global model for the next iteration, x^{t+1}:

$$x^{t+1} = x^t + \sum_{i=1}^{n} \Delta(x^t, D_i^t)$$

The aggregation step ($\sum$) requires an intensive, synchronous communication phase that stalls computation. Given its heavy size, this phase often dominates the iteration time. This communication is handled by one of two topology architectures: the *parameter servers (PS)* approach [**31**], where workers send updates to central servers, or the *All-Reduce* [**58**] approach, where workers communicate in a peer-to-peer fashion. Although all-reduce can deliver high efficiency and excellent bandwidth utilization, recent work shows that PS architectures are often more flexible and adaptive in heterogeneous, resource-shared data-center clusters [**32**, **33**].

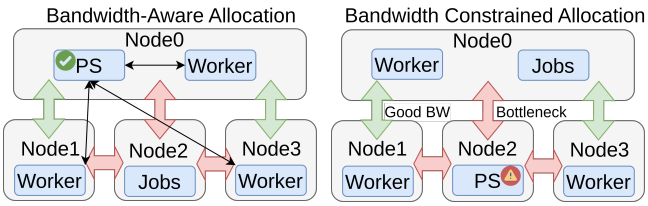
Our assumption. In a data center constrained by compute and bandwidth, our goal is to find a network-aware placement that maximizes training speed. Each job arrives with a specified topology (PS or ring all-reduce). Jobs arrive over time, and we seek to free resources as quickly as possible, alleviating the bandwidth contentions. We illustrate the problem with the following example.

1.2. The Placement-Sensitive Bottleneck. Because GPU performance has increased far faster than data-center network bandwidth [**17**], and model sizes keep growing, the communication-to-computation ratio is rising. Consequently, the physical placement of a job's components becomes a first-order concern.

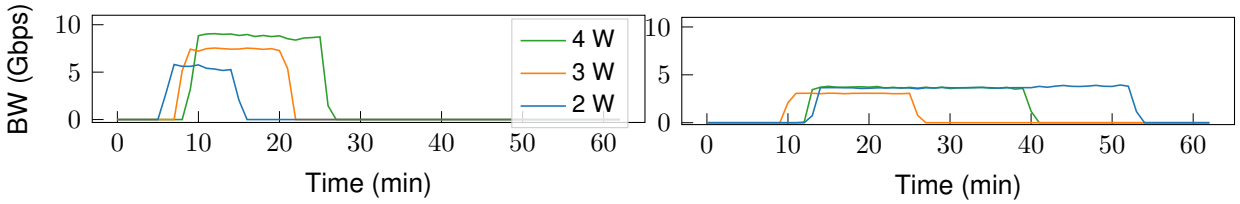
Let us consider a system under load where we need to allocate a job with one PS and three workers for a traditional OpenImage [**59**] and MobileNet [**60**] training task. The system is already running several jobs, which are saturating some network links. As shown in Figure 1a, the red links are saturated, while the green links have sufficient bandwidth for the new job. A *network-agnostic* scheduler might place a PS and its workers on nodes that satisfy the computational requirements but route their traffic across a congested network link (Figure 1a, right). This creates a severe bottleneck, leaving expensive GPUs idle while waiting for data. In contrast, a *bandwidth-aware* placement co-locates communicating components to leverage high-speed, local links, avoiding congestion and improving throughput (Figure 1a, left).

Consequently, we measure the communication time at the PS for a single training round in our prototype. As shown in Figure 1b and Figure 1c, the network-aware placement is up to 3x faster than the bandwidth-unaware placement for jobs with 2 to 4 workers. Assigning jobs to nodes based solely on computational resources is therefore ineffective. An effective placement must consider bandwidth availability, the job's bandwidth requirements, and the communication topology. This placement-sensitive challenge is the central problem we aim to solve. We continue by describing the orchestration challenges for Data Centers.

1.3. Training Overhead: Model and Tradeoff Considerations. When we need to place a PS architecture, each of the N workers computes a gradient on a local batch of size m and sends it to one of the S parameter servers; these servers then aggregate all N updates and broadcast the averaged parameters back to the workers. Instead, under a ring all-reduce architecture, the same N workers are arranged in a logical cycle and exchange partial gradients with their two neighbors, performing a fully decentralized reduction without any central coordinator.



(A) Green links are non-bottleneck, red links are potential bottlenecks.



(B) PS Throughput (BW-Aware)

(C) PS Throughput (BW-Unaware)

FIGURE 1. Motivation: Bandwidth-aware allocation avoids the congested red link (a-left), while an unaware one does not (a-right). The measured bottleneck bandwidth at the PS (b-c), changing the number of workers from 2 to 4, highlights that BW-aware placement (b) achieves higher throughput and reduced transfer times, while the bw-unaware (c) has higher contention on the network.

The first (computation) component, which we denote as Θ_{train}, captures the local computation performed by each worker. Let us assume that each sample in the mini-batch training phase requires τ_{fw} time for the forward propagation and τ_{bw} time for the backward propagation. For a batch of size m, the total computation cost can be hence denoted as $m\tau_{\text{fw}} + \tau_{\text{bw}}$.

The second (communication) component, Θ_{comm}, models the communication overhead that arises from gradient exchanges among distributed workers. Let $\mathcal{E}$ denote the set of communication links, that is, server–worker links in the parameter-server architecture and ring edges in the all-reduce setting. For each link $e \in \mathcal{E}$, we define B_e as the total volume of data exchanged (two transfers per iteration, one send and one receive) and C_e as the link capacity. The overall communication delay is governed by the slowest link in the network and can thus be expressed as $\max_{e\in\mathcal{E}} \frac{2B_e}{C_e}$. Finally, the *update* latency Θ_{update} accounts for the time required to apply gradients after aggregation. Here, we denote with τ_{up} the time to process a single update, with n'_e the number of worker updates traversing link e, and with q_e the number of parameter servers sharing that link. Note that

in a parameter-server architecture $q_e = S$, while in a ring all-reduce $q_e = 1$, since each edge carries the entire update without further splitting. By combining these three contributions, the total training time per iteration can be expressed as:

$$\Theta = \underbrace{m\,\tau_{\text{fw}} + \tau_{\text{bw}}}_{\Theta_{\text{train}}} + \underbrace{\max_{e\in\mathcal{E}} \frac{2\,B_e}{C_e}}_{\Theta_{\text{comm}}} + \underbrace{\max_{e\in\mathcal{E}} \tau_{\text{up}} \frac{n'_e}{q_e}}_{\Theta_{\text{update}}}. \tag{1}$$

Figure 1a shows a single training iteration from the perspective of a worker node in PS topology settings, over different loads, breaking down the components of Equation (1). As bandwidth availability decreases, the overhead of transferring the entire model becomes a significant bottleneck, increasing the total training time. Furthermore, due to the distributed nature of the training, each model transfer competes for shared network links with other concurrent flows, potentially resulting in substantial transfer delays, especially as the communication paths become increasingly congested.

Parameter Servers vs. Ring All-Reduce Let M be the total gradient payload per iteration (bytes). In a ring all-reduce with N workers, the payload is split into N chunks ($\approx M/N$) and each worker exchanges $\frac{2(N-1)}{N}M$ bytes per all-reduce, with performance gated by the slowest link in the ring. In a PS setup with S model parameters, disjoint subsets called shards, each worker exchanges $2M$ bytes per iteration, and each server handles $\frac{2N}{S}M$, creating many-to-one hotspots near servers. This contrast of central aggregation vs. pipelined peer exchange directly affects Θ_{comm} and makes link-aware placement essential. The choice between these architectures presents a fundamental trade-off in network overhead. In the PS approach, each of the N workers sends and receives the complete model update, resulting in a minimal communication cost per worker of $2|U|$, where $|U|$ is the size of the update. However, this creates a massive ingress-and-egress bottleneck at the PSs themselves, which must handle traffic proportional to $N \cdot |U|$. Conversely, a ring all-reduce total data moved per worker is higher, approaching $2|U|\frac{N-1}{N}$ [**17**], and no single node or link becomes a central hotspot.

Next, we formalize our problem definition using optimization theory.

1.4. A Network Utility Maximization Problem. We formulate our resource allocation problem as a network utility maximization task, modeled as a mixed-integer linear program (MILP). Not all MILPs are NP-hard [**61**]. It is easy to see, however, that our specific formulation is NP-hard due to its combinatorial structure and the complexity of the infrastructure constraints, which motivates the need for a (distributed) approximation algorithm. A reduction can be built from the *Generalized Assignment Problem (GAP)*, which is known to be NP-hard [**62**].

Consider an overlay network of N nodes, indexed by $n \in \mathcal{N}$, where $\mathcal{N} = \{1, \ldots, N\}$. Each node offers computing resources such as CPUs, GPUs, and memory to run time-consuming training jobs. The resources available at each node are represented by ρ_{rn}, where $r \in \mathcal{R}$ indexes the required computing resources (i.e. GPU, CPU, memory), with $\mathcal{R} = \{1, \ldots, R\}$. Each edge $e = (i, j)$ can be connected by $e_{n_i n_j} \in E$, where E is the adjacency matrix [1] weighted by the physical link capacity. $E = e_{i,j} \in \{0, \text{bw}\}^{N \times N}$. Each element $e_{i,j}$ of E has a value of 0 if the two nodes are not adjacent (disconnected); otherwise, it represents the available (or residual) bandwidth between the node pairs.

The primary design objective of *Plebiscito* is to orchestrate job allocations while dynamically managing network link capacities, thereby ensuring efficient distributed training and achieving near-optimal bandwidth utilization with respect to the adjacency matrix E. A job j has a set of $\Lambda_j \in \{1, \ldots, \Lambda_j\}$ components, PS and workers, with each $\lambda \in \Lambda_j$ requiring specific computing resources $\varrho_{rj\lambda}$ and a given network capacity $\Theta_{comm_{j\lambda}}$. Under these notations, our infrastructure needs to allocate training jobs by solving a network utility maximization problem.

We denote by $U \in \mathbb{R}_+^{|N|}$ the objective function to be maximized, where U quantifies the utility that each hosting node n assigns to a component λ during the allocation process. The allocation variable $\boldsymbol{x}_j \in \{0, 1\}^{N \times \Lambda_j}$ indicates the assignment of each component λ of job j to node n. Similarly, the allocation variable $\boldsymbol{y}_j \in \{0, 1\}^{E \times \Lambda_j}$ indicates the

[1] We assume that node-to-node bandwidths are obtained by an underlying network telemetry or monitoring system, or as we describe in Section 4. However, given the distributed nature of the Plebiscito protocol, every node is aware of the job placements Section 2.2; thus, we can approximate the network matrix.

usage of link e by a demand λ of job j. (3) enforces that the total demand for resources $\varrho_{rj\lambda}$ assigned to each node n does not exceed its available capacity ρ_{rn} for each type of resource r. Constraints (4) address the limits of network capacity, stipulating that the cumulative data transmission requirements $\Theta_{comm_{j\lambda}}$ for all components λ of a job j across a link e must not exceed the link capacity $e \in E$, so that there is a direct connection between the pair of nodes. This prevents network congestion and ensures that data flows do not exceed the physical bandwidth of links, leading to long training times. Constraint (5) enforces a conflict-free assignment where any λ of a job j has to be assigned, while constraint (6) is an existential constraint.

$$\max_{\boldsymbol{x},\boldsymbol{y}} \quad \sum_{n\in N_n}\sum_{j\in J_j}\sum_{\lambda\in\Lambda_j} U_{nj\lambda}(\boldsymbol{x}_j, \boldsymbol{y}_j) \tag{2}$$

$$\sum_{j\in J_j}\sum_{\lambda\in\Lambda_j} \varrho_{rj\lambda} x_{nj\lambda} \leq \rho_{rn}, \qquad \forall n \in \mathcal{N}, \forall r \in \mathcal{R} \tag{3}$$

$$\sum_{j\in J_j}\sum_{\lambda\in\Lambda_j} \Theta_{comm_{j\lambda}} y_{ej\lambda} \leq e, \qquad \forall e \neq 0 \in E, y_{ej\lambda} \neq 0 \tag{4}$$

$$\sum_{n\in N_n} x_{nj\lambda} = \Lambda_j, \qquad \forall j \in \mathcal{J}, \forall \lambda \in \Lambda_j \tag{5}$$

$$x_{nj\lambda}, y_{ej\lambda} \in \{0,1\} \qquad \forall n \in \mathcal{N}, \forall e \in E, \quad \forall j \in \mathcal{J}, \forall \lambda \in \Lambda_j. \tag{6}$$

1.5. Utility Function Library for Orchestration. The utility function $U(\cdot)$ is a policy that can be tailored to satisfy specific application-level or infrastructure-oriented objectives. Although $U(\cdot)$ defines a policy that may remain private, in Theorem 1 we show that, under the assumptions of positivity, monotonicity, and submodularity, it is possible to construct an approximation algorithm with a provable optimality bound. We formalize this with the following definition.

Definition 1 (Utility Function). *Given a job j to be allocated over an overlay network of N nodes using a max-consensus auction algorithm, we define the utility function of each bidder (or voter) i as $U_i \in \mathbb{R}_+^{|N|}$, where $\mathbb{R}_+^{|N|}$ denotes the vector space of positive real numbers of dimension $|N|$.*

This function quantifies the utility that hosting node i assigns to each job component λ during the allocation phase. In *Plebiscito*, we decompose this utility into two principal components to maximize: a compute-related term, called the *resource-fraction*, which captures the availability of local computational resources, and a networking term, called the *network-fraction*, which reflects the residual bandwidth along the communication paths.

Definition 2 (Residual Capacity Function). We define the residual capacity of resource r at node n as:

$$\Delta_{rn} = \rho_{rn} - \sum_{j' \in \mathcal{J}} \sum_{\lambda' \in \Lambda_{j'}} \varrho_{rj'\lambda'} \, x_{nj'\lambda'}, \tag{7}$$

where ρ_{rn} denotes the total capacity of resource r on node n, and the summation accounts for all resource allocations currently assigned to that node. Δ_{rn} therefore represents the residual (unallocated) capacity of resource r relative to its total available capacity.

We then define the *resources-fraction* as:

$$f_i^{\text{comp}} = \sum_{r \in \{C,G,M\}} w_r \, \frac{\Delta_{rn}}{\varrho_{rj\lambda}}, \tag{8}$$

where each weight $w_r \geq 0$ reflects the relative importance of CPU (C), GPU (G), or memory (M) resources in the allocation policy.

Bandwidth-Aware Utility. In our MILP formulation, constraint (4) enforces that the aggregate transfer demand on any link does not exceed its physical capacity. To satisfy this condition in a fully decentralized setting, each *Plebiscito* agent maintains a local view of the overlay graph $G = (V, E)$, where each edge $e \in E$ is characterized by its

capacity C_e and the current set of active flows $\mathcal{F}$. Every flow $\phi \in \mathcal{F}$ consumes B_ϕ bandwidth units along its routing path; therefore, the residual bandwidth available on link e is given by

$$\Delta_e = C_e - \sum_{\phi \in \mathcal{F}:\, e \in \mathrm{path}(\phi)} B_\phi, \tag{9}$$

which quantifies the remaining bandwidth on link e that can be allocated to new traffic.

When the consensus is reached, each node estimates the residual bandwidth, updating the local graph view. When an agent considers placing the component λ of the job j on the node i, it first identifies the unique sequence of links $\mathrm{path}(i, d_{j,\lambda})$ from i to the destination of the component $d_{j,\lambda}$.

Along this path, the bottleneck link is identified as the one with the smallest ratio $\Delta_e / B_{j,\lambda}$. We capture such constraints through the *network-fraction* function, defined as

$$f_i^{\mathrm{net}} = \min_{e \in \mathrm{path}(i, d_{j,\lambda})} \frac{\Delta_e}{B_{j,\lambda}}\,, \tag{10}$$

which takes values in the range $[0, 1]$ and quantifies the fraction of the requested transfer volume that the network can maintain end to end without violating the constraint (4). This measure directly determines the achievable communication efficiency Θ_{comm} for the given allocation.

Plebiscito Utility. By combining f_i^{net} with the compute-fraction f_i^{comp} in (11), Plebiscito's max-consensus auction naturally favors hosts that are sufficiently provisioned both in terms of local resources and available network capacity, yielding allocations that respect all resource constraints while balancing compute-network trade-offs.

$$U_{i,j,\lambda} \;=\; \alpha \cdot f_i^{\mathrm{comp}} + (1 - \alpha) \cdot f_i^{\mathrm{net}}, \quad 0 \;\leq\; \alpha \;\leq\; 1 \tag{11}$$

By adjusting α, one smoothly trades off emphasis between compute and network, rewarding hosts that provide ample capacity on both dimensions. Plebiscito max-consensus auction with this utility formulation ensures that winners not only avoid CPU/GPU/memory

TABLE 1. The Utility function policy supported by Plebiscito subsume related solutions.

Utility	Note
Smallest GPU First (SGF) [**5**]	$f_i^{\text{comp,SGF}} = -(\Delta_{\text{GPU},i} - \varrho_{\text{GPU},j})$ packs jobs, with $\Delta_{\text{GPU},i}$ the free GPU slots on node i and $\varrho_{\text{GPU},j}$ job j's GPU demand.
Largest GPU First (LGF) [**5**]	$f_i^{\text{comp,LGF}} = \Delta_{\text{GPU},i} - \varrho_{\text{GPU},j}$ favors large GPU requests on well-provisioned nodes.
Apache YARN [**10**]	$f_i^{\text{comp,YARN}} = \min_{r\in\mathcal{R}}\big((\Delta_{r,i} - \varrho_{r,j})/\rho_{r,i}\big)$ maximizes the minimum normalized resource headroom after placement.
Tetris [**63**]	$f_i^{\text{comp,Tetris}} = \sum_{r\in\mathcal{R}}(\Delta_{r,i}/\rho_{r,i})\,\varrho_{r,j}$, where $\Delta_{r,i}$ is the residual amount of resource r on node i (i.e. total capacity $\rho_{r,i}$ minus currently allocated), and $\varrho_{r,j}$ the peak demand of job j.
Tiresias [**18**]	$f_j^{\text{comp,Tiresias}} = W_j\, t_j$, where W_j is a job-specific weight (e.g. priority) and t_j its expected runtime.
DRF [**14**]	$f_i^{\text{comp,DRF}} = -\max_{r\in\mathcal{R}}\big((u_{i,r} + \varrho_{r,j})/\rho_{r,n}\big)$, with $u_{i,r}$ the user's current resource allocation. This minimizes the user's dominant share.
Themis [**13**]	We capture co-location interference via $f_i^{\text{comp,Themis}}(\mathbf{G}_i) = T_i^{\text{sh}}(\mathbf{G}_i)/T_i^{\text{ind}}$, where T_i^{sh} is the runtime when job i shares hosts under allocation $\mathbf{G}_i$, and T_i^{ind} its isolated runtime.

exhaustion (when feasible) but also route their gradient and parameter traffic over links that can sustain the required bandwidth.

Remark: Note that the function in Eq. 11 is monotone, and since each factor exhibits diminishing returns, our submodularity assumptions still hold (Definition 2).

Utility Function Library. *Plebiscito* provides a configurable *resources-fraction* policy library that allows reproducing or subsuming various placement strategies by substituting the previously defined f_i^{comp} with one of the supported formulations (see Table 1). Each policy captures a distinct allocation behavior, such as prioritizing packing efficiency, fairness, or resource balance, and can be selected according to the desired orchestration objective. The impact of these different policies is evaluated in Section 5.

2. Plebiscito: System and Algorithmic Design

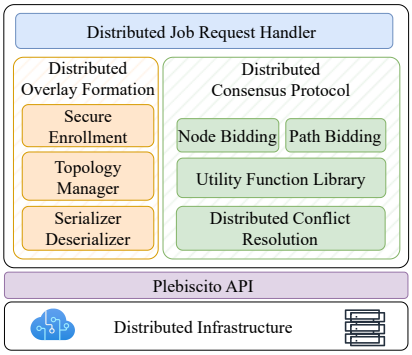
To solve the NP-hard problem defined in the previous section, we designed Plebiscito, a fully decentralized architecture based on a distributed max-consensus auction. At the core of *Plebiscito* lies an asynchronous distributed *max-consensus* mechanism that enables nodes to reach agreement on job allocation without centralized coordination. Each node maintains local state variables: bids, assignments, and timestamps for every job component. Through iterative message exchanges with its neighbors, nodes propagate the most recent and highest-valued information, ensuring that all participants converge toward a consistent global allocation. This mechanism guarantees that, after a finite number of communication steps, all nodes agree on the winning bids and assignments, providing a scalable and fault-tolerant foundation for decentralized resource orchestration.

This section details the core components of our solution: the utility function that drives bidding and the protocol that facilitates agreement.

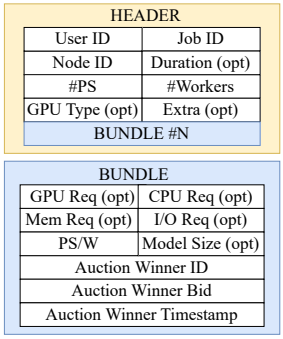
2.1. Policy-Based Architecture. The architecture shown in Figure 2a is divided into two distinct, yet cooperative mechanisms that orchestrate the creation, management, and execution of computing tasks across the nodes in the architecture. Each node executes a Plebiscito agent, which can run all the mechanisms presented next.

Pre-allocation (Distributed Overlay Formation). Nodes enroll in the network overlay through the *Topology Manager* that keeps track of the available nodes serving the job requests. A secure enrollment procedure provides each node with a unique identifier (ID) for subsequent requests. Each Plebiscito node instance has a *Serializer/Deserializer* API to exchange Plebiscito protocol messages.

Asynchronous Distributed Max-Consensus Protocol. Each Plebiscito agent assigns jobs using a distributed max-consensus auction. Auction bids are generated through the Utility Function Library (Section 1.5), a customizable and extensible set of allocation functions. Different utility functions lead to different allocations and prioritize different objectives, as the bid value reflects the node's ability to host a given job and accounts for



(A) *Plebiscito Architecture*



(B) *Plebiscito Consensus Protocol*

FIGURE 2. (a) Plebiscito API and mechanisms to orchestrate distributed training with guarantees. (b) Bundles carry the states needed for the distributed max-consensus auction.

residual resources such as GPU, CPU, and network bandwidth. After the bidding phase, an asynchronous max-consensus protocol resolves conflicts to determine the NP-hard job placement. Once the protocol converges, the distributed training job starts executing.

Protocol Design. We define the protocol format in Figure 2b that nodes must use to allocate a job. The packet header contains *UserID, JobID*, and $NodeID$ indicates the forwarding node. The Topology Manager assigns IDs for $NodeID$s and $UserID$s as nodes/users join/leave the overlay. Optionally, a job may specify GPU type and duration, which have been shown to be beneficial for scheduling and billing [**18**]. Each job has a defined number of $ParameterServer$ (0 for ring-all-reduce) and $Workers$. The $Extra$ field is used for future extensions (e.g., handling job preemption and migration via rebidding on already-allocated jobs). The bundle field records the auction winner ID, Bid, $Timestamp$, and the required resources (GPU, CPU, memory, I/O Bandwidth, role (PS, Worker, Rank), model size) to inform the nodes of the specific job requirements.

2.2. Plebiscito Distributed Consensus Auction Protocol. To solve the problem described above, we designed within the Plebiscito architecture a distributed max-consensus protocol. Before an action can start, an overlay is formed among the bidders. Communication is only among first-hop neighbors. Upon submitting a job j an allocation request starts using the Plebiscito API. When a node receives the allocation request, it starts the

distributed consensus Algorithm 1 by locally initializing ($bundle(j, \tau)$, where τ is the current timestamp) for the allocation. Specifically: $b(j, \tau) \in \mathbb{R}_+^{\lambda_j}$ is the current node bid (each node places a bid for each component λ using the system-wide Utility Function Table 1). The vector $a(j, \tau) \in \mathbb{R}_+^{\lambda_j}$ *stores the identifiers of the nodes currently leading the auction for each job component*. The vector $t(j, \tau) \in \mathbb{N}_+^{\lambda_j}$ records the timestamps associated with each bid, indicating when the corresponding offer was generated. These timestamps are required to ensure correctness under asynchronous communication. Each entry in these vectors corresponds to one job component λ (either a parameter server or a worker instance), and includes a flag indicating the component's role (PS or worker). For each job to be allocated, nodes proceed asynchronously, alternating between *bidding* and *agreement* phases. They exchange the most up-to-date information by forwarding a packet compliant with Figure 2b. Each node updates the local bundles with the latest neighbor information and, if the local information changes, forwards the local bundle to $\mathcal{N}_n$. This process continues until no more packets are exchanged, indicating that all nodes have bid, with no need for rebroadcasting; however, a timeout mechanism is included in the system implementation Section 4. When the protocol converges, this field will contain the winning bid values of the hosting node n for job j.

Bidding Phase. *(Algorithm 2)* Upon receiving an allocation request, a node executes Algorithm 2 to assess the job's suitability against local policies. Each node evaluates its current capabilities (GPU, CPU, memory, network) against the job requirements specified in the Plebiscito packet, placing a bid on that specific PS or worker λ if its requirements fit on that node. Bidding calculations use a *Utility function* U, representing node local policy as described in Equation (11). In particular, the algorithm works as follows: if a bid is placed, the local bundle vectors are updated with the utility values in $b(\lambda)$, the node ID in $a(\lambda)$, and the timestamp in $t(\lambda)$ to support the distributed asynchronous max-consensus algorithm Definition 1. The timestamp helps the algorithm consider the most up-to-date information, as the updated bundle is not broadcast but forwarded to neighbors. As each node runs the max consensus algorithm, only the most up-to-date information will propagate on the node overlay.

Algorithm 1 Plebiscito Algorithm

Initialization:
Each node initializes $bundle(j, \tau(0)) \leftarrow (b(j), a(j), t(j))$
repeat
  for each node i **do**
    for packet of job j in node queue **do**
      Execute: See Algorithm 2 & Algorithm 3
    end for
    Execute: Bidding Algorithm (Algorithm 2) ▷ Final bid if possible
    if $bundle(j, \tau - 1) \neq bundle(j, \tau)$ **then**
      BROADCAST(N_n) ▷ Send to neighbors
    end if
  end for
until convergence criterion is met

Algorithm 2 Bidding Algorithm

Input: $bundle^*(j, \tau - 1)$ ▷ Local Bundle
Output: $bundle^*(j, \tau)$ ▷ Independent for each node
while $bundle^*(j, \tau - 1)$ has available λ **do**
  if resources and bandwidth available for λ **then**
    $b_j^*[\lambda] \leftarrow$ U(λ) ▷ Custom Utility policy
    $a_j^*[\lambda] \leftarrow$ GETNODEID
    $t_j^*[\lambda] \leftarrow$ CURRENTTIME
  end if
end while

Remark (Gang scheduling). The bidding mechanism in *Plebiscito* can implicitly enforce *gang scheduling* not a specific scheduling algorithm, but the practice of ensuring that all components of a job run concurrently and are placed as close as possible to one another to support synchronized execution and stable bandwidth availability [**5**,**33**,**64**]. With an appropriate utility function, this behavior emerges naturally: if a node bids on n components and wins the first (i.e., offers the highest utility among all bidders), its bids tend to remain dominant for the remaining $n-1$ components as well, leading to co-location, reduced inter-job contention, and more efficient use of network resources.

Agreement Phase. *(Algorithm 3)* The agreement (or conflict resolution) phase compares the local bundle, which contains all jobs temporarily self-assigned, with the received bid messages. Each node updates the local bundle if at least one of the received

Algorithm 3 Agreement Algorithm

Input: $bundle(j, \tau - 1)$ ▷ Received Bundle
for each node i **do**
  for each λ in b_j **do**
    ▷ Local vs received bundle
    if $b_j > b_j^*$ or $(y_{kj} = y_{ij}$ and $t_{kj} > t_{ij})$ **then**
      Update local bundle if beaten
      $bundle^*(j, \tau - 1)(\lambda) \leftarrow (b_j(\lambda), a_j(\lambda), t_j(\lambda))$
    end if
  end for
end for

bids on one of the resources outbids the currently known highest bid. If a bif update occurs, the local bundle is broadcast again to the node's first-hop neighbors, as detailed in Algorithm 1. To avoid suboptimal allocation, all jobs allocated after the outbid item are released, as they were assigned with an out-of-date residual capacity. The protocol continues until the network becomes quiet, signifying that a consensus has been reached on the final allocation.

DEFINITION 1 (Max-Consensus). *Given a job $j \in \mathcal{J}$ to allocate on a set of nodes $\mathcal{N}$, each node $n \in \mathcal{N}$ maintains, for each component $\lambda \in \Lambda_j$, a bid value $b_\lambda^n(j, \tau) \in \mathbb{R}_+$, an assignment indicator $a_\lambda^n(j, \tau) \in \{0, 1\}$, and a timestamp $t_\lambda^n(j, \tau) \in \mathbb{N}_+$ at communication instance τ. At each communication instance $\tau + 1$, node n updates its local information for each component λ based on the information from its neighbors $\mathcal{N}_n$ (including itself) as follows: $n' = \arg\max_{n' \in \mathcal{N}_n} \{t_\lambda^{n'}(j, \tau)\}$, $a_\lambda^n(j, \tau + 1) = a_\lambda^{n'}(j, \tau)$, $b_\lambda^n(j, \tau + 1) = b_\lambda^{n'}(j, \tau)$, $t_\lambda^n(j, \tau + 1) = t_\lambda^{n'}(j, \tau)$. If multiple nodes have the same maximum timestamp, ties are broken using a predetermined rule (e.g., the node with the highest bid or lowest identifier). Max-consensus among the nodes is said to be achieved with a convergence time $\overline{\tau}$ if, for all $\tau \geq \overline{\tau}$ and for all $n, n' \in \mathcal{N}$, the following holds: $a_\lambda^n(j, \tau) = a_\lambda^{n'}(j, \tau), \quad b_\lambda^n(j, \tau) = b_\lambda^{n'}(j, \tau), \quad \forall \lambda \in \Lambda_j$. This means that all nodes agree on the assignment and bid values for each component $\lambda \in \Lambda_j$. The timestamps $t_\lambda^n(j, \tau)$ are assumed to be non-decreasing with respect to τ to ensure convergence.* □

With the Max-Consensus mechanism, Plebiscito nodes maintain bid values, assignment indicators, and timestamps for every job component. Winning bids are only considered when the bid generation time-stamp is up to date; otherwise, they are ignored. Only the maximum bid propagates to seek "Max-Consensus".

3. Approximation and Convergence Guarantees

In this section, we provide the theoretical foundation for Plebiscito performance guarantees. We first establish a formal approximation guarantee for the NP-hard placement problem, and analyze the protocol convergence properties.

3.1. Plebiscito's Approximation Bound. Our main theoretical result is that the Plebiscito distributed consensus auction converges to a solution (a stable assignment) with a quality that has a bounded worst-case. Our approximation bound is a corollary of a known result on the maximization of a greedy algorithm using a submodular function [**65**].

The Role of Submodularity. The foundation of our guarantee is the mathematical property of submodularity, which can be thought of as monotonicity on a set, or a diminishing marginal gain property. In particular:

DEFINITION 2 (Submodular Function). *Given a finite set Λ_j (e.g., the set of components of job $j \in \mathcal{J}$), a set function $U : 2^{\Lambda_j} \to \mathbb{R}$ is called submodular if it satisfies the following condition: for all subsets $\lambda' \subseteq \lambda \subseteq \Lambda_j$ and for every element $\lambda'' \in \Lambda_j \setminus \lambda$:* $U(\lambda \cup \{\lambda''\}) - U(\lambda) \leq U(\lambda' \cup \{\lambda''\}) - U(\lambda')$.

This property is known as the diminishing returns property, meaning that the marginal gain from adding an element λ'' to a set λ does not increase as the set λ becomes larger.

This definition emphasizes that with submodular functions, the incremental gain from adding an element to a set decreases as the set becomes larger.

Remark. Note that the residual capacity in our resource allocation use case, defined in Equation (7) is a positive, monotone, and submodular function.

Approximation Bound. A greedy algorithm (one that iteratively picks the item with the highest marginal gain) achieves a constant-factor approximation for maximizing a monotone submodular function [**65**]. The Plebiscito auction is a distributed implementation of this greedy algorithm; the max-consensus protocol ensures that the component is assigned to the node offering the highest utility gain. Because our utility functions are either submodular or are made so by the warping function, we can state the following guarantee:

THEOREM 1. *The Plebiscito consensus algorithm achieves a* $(1 - \frac{1}{e})$*-approximation to the optimal job assignment when the utility functions are positive, monotone, and submodular.* □

PROOF. Let Λ_j be the set of components (PS / workers) of the job j, and let $S \subseteq \mathcal{N} \times \Lambda_j$ denote a partial assignment of components to the nodes. We define the set function $F(S)$ as the total utility accrued by all nodes for the assignments in S; by assumption, F is *positive*, *monotone*, and *submodular*. For any feasible pair (n, λ), let the marginal gain be

$$\Delta((n, \lambda) \mid S) \;=\; F(S \cup \{(n, \lambda)\}) - F(S).$$

Each node n bids on component λ only if the local utility $U_n(\lambda) \geq 0$, hence all advertised marginal gains are non-negative. Monotonicity of F implies $\Delta((n, \lambda) \mid S \cup T) \leq \Delta((n, \lambda) \mid S)$ for any $T \supseteq \emptyset$, i.e., marginal gains do not increase as S grows.

At every iteration, nodes compute local bids equal to their current marginal gains and exchange *(value, timestamp)* tuples with their neighbors. The asynchronous max-consensus rule ensures that the globally maximal (most recent) bid propagates and is selected cluster-wide. Thus, the chosen assignment at each step is exactly

$$(n^\star, \lambda^\star) \in \arg \max_{(n,\lambda) \text{ feasible}} \Delta((n, \lambda) \mid S),$$

i.e., the same choice a centralized greedy algorithm would make. Since F is positive, monotone, and submodular, and *Plebiscito* selects at each step the feasible pair with

maximum marginal gain, the sequence of assignments produced coincides with classical greedy. By the Nemhauser–Wolsey–Fisher theorem for maximizing a monotone submodular function under a cardinality/partition assignment constraint [**65**], greedy achieves a $(1 - \frac{1}{e})$-approximation with respect to the optimal solution. Therefore, the *Plebiscito* max-consensus algorithm inherits the same $(1 - \frac{1}{e})$ guarantee. □

Furthermore, the following result holds:

THEOREM 2. *The Plebiscito resource allocation problem cannot be approximated within a ratio better than* $(1 - \frac{1}{e})$ *in polynomial time, unless* $P = NP$. □

Proof. We establish this result by reducing the Budgeted Maximum Coverage Problem (BMCP) to our resource allocation problem. The BMCP is known to be NP-hard and hard to approximate within a factor better than $(1 - \frac{1}{e})$ [**66**]. In BMCP, we have a universe of elements $\mathcal{U}$, a collection of subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ with associated costs, and a budget L. The goal is to select subsets whose total cost does not exceed L while maximizing the number of covered elements. To perform the reduction, we map our resource allocation problem to BMCP. Each component $\lambda \in \Lambda$ corresponds to an element $u \in \mathcal{U}$ in the BMCP. Each node n_i in our problem corresponds to a subset S_i in BMCP, where S_i contains the elements corresponding to the components that node n_i can host. The resource constraints of nodes correspond to the costs of subsets, and the overall resource capacity corresponds to the budget L. The utility of assigning components to nodes corresponds to the number of elements covered by the selected subsets. An optimal solution to our resource allocation problem would yield an optimal solution to the BMCP. If we could approximate our resource allocation problem within a ratio better than $(1 - \frac{1}{e})$, we could achieve the same for BMCP, which contradicts known hardness results unless $P = NP$. Therefore, the $(1 - \frac{1}{e})$-approximation achieved by the Plebiscito algorithm is the best possible under these conditions unless $P = NP$. □

The $(1 - \frac{1}{e})$ is fundamental for Plebiscito because it sets a hard algorithmic ceiling: unless $P{=}NP$, no polynomial-time method can guarantee a factor better than $(1 - \frac{1}{e})$

for our allocation problem. Practically, this (i) justifies pursuing approximation/greedy designs instead of exact solvers at scale, (ii) validates the use of submodular-style algorithms whose best-known guarantees match this barrier, and (iii) calibrates expectations for evaluation if an algorithm consistently achieves near $(1 - \frac{1}{e})$ in experiments, it is essentially optimal up to known complexity limits. In short, the theorem defines the best attainable frontier for efficient network-aware allocation.

Handling Practical Deviations from Submodularity. Although many resource-based utility functions are naturally submodular, this assumption may be too restrictive in practice. For example, in certain allocation scenarios, the value of assigning an additional component may increase when colocated with other components on the same node, violating the diminishing-returns property required for submodularity. To guarantee convergence even in these cases, *Plebiscito* extends its analysis to a broader class of *pseudo-submodular* functions, which preserve a generally diminishing trend but are not strictly submodular.

To enforce convergence under such conditions, we introduce a lightweight *warping function* $\mathcal{F}$ that transforms each node's bid into a monotonically non-increasing function of its previous bids:

$$\mathcal{F}_{n\lambda j} = U_{n\lambda j} - \delta \cdot |\Lambda_j^{n,\mathsf{a}}|, \tag{12}$$

where $U_{n\lambda j}$ denotes the original utility of node n for component λ of job j, $\delta > 0$ is a small constant penalizing repeated allocations, and $|\Lambda_j^{n,\mathsf{a}}|$ is the number of components of job j already mapped to node n. This modification ensures that the marginal utility of assigning new components decreases as more components are allocated to the same node, restoring a diminishing-returns behavior.

Example. Consider two nodes bidding on two components, λ_1 and λ_2, whose utilities would otherwise increase with colocation. Applying the warping function prevents bids from growing as new components are added, avoiding oscillations in the distributed auction and guaranteeing that all bids appear as if they were generated by a submodular utility function.

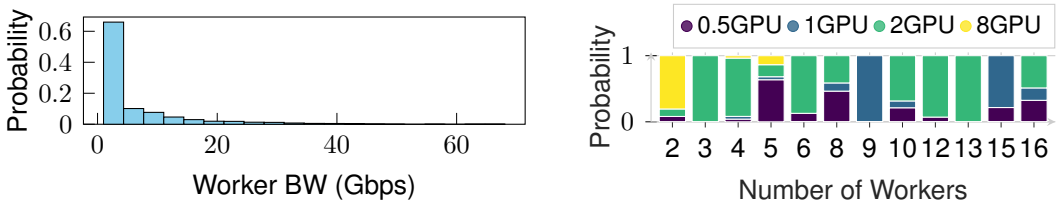
By uniformly applying $\mathcal{F}$ across all nodes and components, *Plebiscito* maintains stable convergence while preserving the $(1 - \frac{1}{e})$ optimality bound. Even when utilities deviate from strict submodularity, the algorithm effectively behaves as though they were submodular, achieving consistent and efficient decentralized resource allocation.

3.2. Convergence Guarantee. Beyond solution quality, we establish an upper bound on the protocol's convergence time. Considering a network of N nodes with diameter D, the analysis assumes synchronous communication, reliable message delivery, and submodular (or warped-to-submodular) utility functions, ensuring that the bidding process terminates within a finite number of steps.

THEOREM 3. *Given an network of N bidders with diameter D, for a job consisting of Λ components, the Plebiscito algorithm achieves consensus on the final allocation within an upper bound of $O(\Lambda \cdot N \cdot D)$ message-passing iterations.* □

PROOF. The proof proceeds by induction on the number of components, Λ. We first establish the bound for a single component ($\lambda = 1$) and then extend it.
Base Case ($\Lambda = 1$): For a single component, each of the N nodes may place a bid. In the worst-case scenario, the winning bid must be propagated across the entire network, which takes a number of iterations proportional to the network diameter, D. An adversarial situation could arise where each of the $N - 1$ other nodes sequentially submits a slightly higher bid, each requiring a new propagation to ensure global awareness. Thus, the total iterations for one component is bounded by $(N - 1) \cdot D$, which is $O(N \cdot D)$.
Inductive Step: Assume that consensus for k components is reached within $k \cdot O(N \cdot D)$ iterations. For the $(k + 1)$-th component, the bidding process is independent of the first k and, following the base case logic, will also converge within $O(N \cdot D)$ iterations. By induction, the total number of iterations required to reach consensus on all Λ components is bounded by $\Lambda \cdot O(N \cdot D)$, which simplifies to $O(\Lambda \cdot N \cdot D)$. This bound holds even in the worst case where every node is involved in the bidding for every component. □

While this theorem establishes a formal worst-case bound, it is essential to put it into context. First, our empirical evaluation in Section 5 shows that, in practical scenarios,



(A) Bandwidth requirements. (B) GPU requirements.

FIGURE 3. *Workload characterization*: (a) Bandwidth follows a Pareto distribution. (b) Workers range from 2 to 16, with varying numbers of GPUs required per worker.

the system falls well short of this theoretical upper bound, converging significantly faster. Second, this linear scalability with the number of nodes (N) is a key advantage of our decentralized design. A comparable centralized algorithm, which must collect bids from all N nodes and process potential conflicts, would face a communication and computation overhead that grows quadratically ($O(N^2)$) [**67**], making it less practical for large-scale networks.

4. Plebiscito Implementation

We developed a prototype of the Plebiscito architecture as a pluggable agent module designed to be deployed as a Kubernetes *DaemonSet*, ensuring that one agent runs on each schedulable node in the cluster, forming a distributed orchestration layer.

Agent Architecture and State Management. Each Plebiscito agent is a standalone process responsible for managing its local node's state and participating in distributed auctions. The agent maintains an in-memory database of its node's total and available resources (CPU, GPU, memory), as well as a view of the network topology and the state of ongoing auctions. This state is managed using thread-safe data structures to handle concurrent API requests and protocol messages. A dedicated API, implemented as a lightweight HTTP server, exposes endpoints for submitting jobs ('POST') and querying system state ('GET').

Communication Protocol and Overlay Management. The agents form a peer-to-peer overlay network to exchange auction messages. Node discovery is handled dynamically using the Kubernetes (K8s) API: upon startup, each agent queries the K8s API server to

find the Pod IPs of all other agents in the *DaemonSet*, thus building its list of neighbors. All inter-process communication for the max-consensus auction protocol is performed over standard UDP sessions. The protocol messages themselves, which encapsulate the bid bundles shown in Figure 2b, are serialized to JSON, as it is a well-known and widely used serializer.

Practical Consensus and Convergence Detection. A major challenge in any real-world asynchronous system is determining when consensus has been reached and handling cases where it cannot be. *Plebiscito* employs a two-tier timeout mechanism to ensure both liveness and safety. First, under normal operation, our prototype uses a *quiescence detection* mechanism, which monitors message exchanges among nodes and detects when no new updates or bids are being propagated in the network. This indicates that the system has reached a stable state and no further state changes are expected, so consensus can be safely declared without additional synchronization overhead. In our case, an agent considers that consensus for a specific job has been reached when it has not received any bids with a newer timestamp for that job for a configurable period τ_{quiet} (e.g., 500ms). This allows the auction to converge quickly in most cases. Second, to guard against network partitions, silent agent failures, or other conditions that could prevent convergence, the quiescence detection is backed by a hard timeout, $\tau_{timeout}$ (e.g., 1 seconds), managed by the agent that initiated the auction. If consensus is not reached within this period, the initiating agent takes the following explicit steps: (i) It unilaterally declares the auction for that job_id as *FAILED*. (ii) It purges all local state associated with the failed auction (e.g., current winning bids and timestamps) to prevent stale information from affecting future requests. (iii) It returns an HTTP error code (e.g., '504 Timeout') to the submitting client, indicating that the placement attempt was unsuccessful. This design places the responsibility for retries on the client, a standard pattern that prevents uncontrolled retry message sequences within the orchestration system. The client can then choose to resubmit the job immediately or after a back-off period.

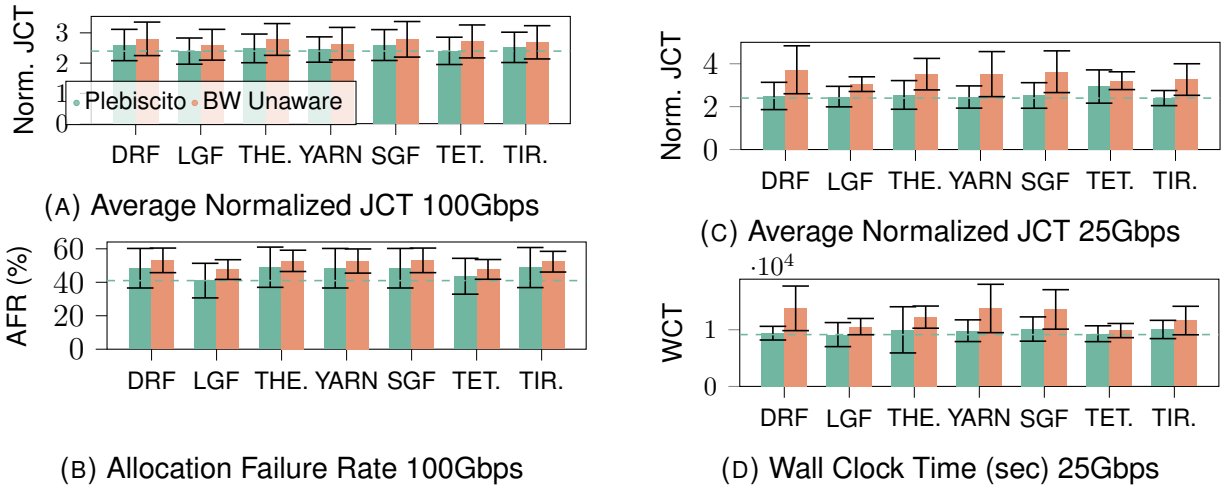
In the case of other failures, such as a transient agent crash, Plebiscito relies on the resilience of Kubernetes and the gossip protocol itself. The Kubernetes `DaemonSet` controller will automatically restart a failed agent. Upon restart, the agent rejoins the overlay with a clean state and begins participating in ongoing auctions as soon as it receives a protocol message. Although the temporary absence of an agent might delay convergence, the timestamp-based logic of the protocol ensures correctness. The hard timeout on the initiating node remains the ultimate backstop, guaranteeing that no auction can run indefinitely, even in the face of repeated agent failures.

Integration with the Kubernetes Scheduler. A key design choice has been to use Plebiscito for placement decisions while still leveraging Kubernetes for application lifecycle management. Plebiscito acts as a custom, out-of-process scheduler. Once the auction converges and a winning node is selected for each job component, the agent that initiated the request generates the necessary Kubernetes manifests (e.g., 'Deployments', 'Services', 'Pods'). Crucially, to bypass the default Kubernetes scheduler and enforce the placement decision, the agent populates the 'nodeName' field in the Pod's specification ('spec.nodeName'). Upon receiving the manifest, Kubernetes honors the `nodeName` field and schedules the Pod on the designated node, ensuring that the deployment matches the placement decided by the *Plebiscito* auction.

5. Evaluation Results

In this section, we present trace-driven results from both our discrete-event simulator and our Kubernetes implementation, highlighting Plebiscito's allocation trade-offs and the importance of bandwidth-aware placements for job performance. We also compare the utility functions in Table 1. Across large-scale data-center scenarios, bandwidth-aware placement reduces total training time by up to $1.5\times$.

Evaluation Metrics. We evaluate *Plebiscito* along multiple dimensions that jointly capture system efficiency, stability, and fairness. *Job Completion Time (JCT)* quantifies overall efficiency, measuring the elapsed time from job submission to completion and normalizing it against an ideal, contention-free baseline. *Allocation Failure Rate (AFR)*



(A) Average Normalized JCT 100Gbps

(B) Allocation Failure Rate 100Gbps

(C) Average Normalized JCT 25Gbps

(D) Wall Clock Time (sec) 25Gbps

FIGURE 4. *Plebiscito mitigates network bottlenecks, reducing JCT, WCT, and AFR* Figure 4a and Figure 4b show improvements in Average Normalized JCT and AFR under 100Gbps node bottleneck, while Figure 4c and Figure 4d highlight gains in JCT and WCT under 25Gbps node bottleneck. Plebiscito Network-aware placements lead up to $1.5\times$ JCT, WCT, and AFR improvements.

reflects the system's robustness under resource pressure by recording the fraction of jobs that cannot be placed on their first attempt due to insufficient capacity. To assess throughput and scalability, we measure the *Wall Clock Time (WCT)*, representing the total time required for the system to process a fixed batch of jobs. Fairness in resource sharing is evaluated using *Jain's index*, which captures how evenly compute and network resources are distributed across nodes. Finally, we monitor *bandwidth utilization* over time to evaluate how effectively the system leverages network capacity while avoiding congestion. All experiments assume a heterogeneous cluster, where weaker nodes are penalized through reduced utility values, ensuring that low-performance nodes are selected only when necessary. This approach maintains fairness and efficiency in decentralized scheduling decisions.

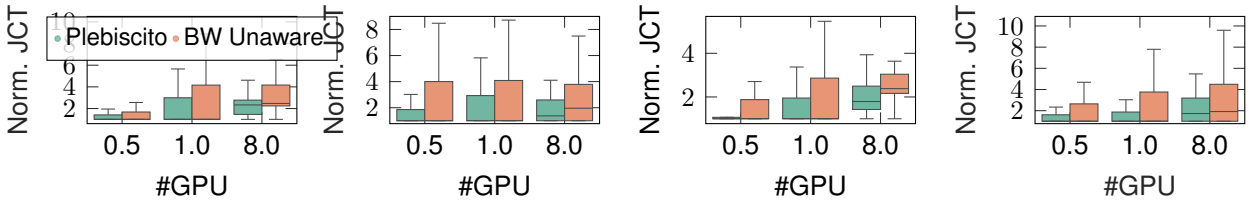
Trace-driven Simulations. Similar to prior related work [**5**], our simulator models a data center with 100 heterogeneous nodes, each with 8 GPUs and 96 CPU cores. We use publicly available traces [**68**], and fix two bandwidth scenarios with a standard node to switch 25Gbps and 100Gbps. For our hosting infrastructure, we use a leaf-spine topology (5 leaf and 2 spine switches) with an oversubscription ratio of 3:1 between leaf and

spine switches. The traces used in our simulation include details on physical node specifications, job CPU/GPU resources, network I/O, job duration, submission time, the number of PS $\Sigma \in [0, 16]$, and the number of workers $N \in [2, 16]$ **[5]**. Note that 0 PS means ring-all-reduce architecture.

Dataset. Our trace-driven simulator models a datacenter with 100 heterogeneous nodes. We use a workload derived from the public Alibaba cluster traces **[68]**, which contain detailed information on job resource requirements and duration. Figure 3 characterizes this workload, showing a Pareto-like distribution for bandwidth demand where a few jobs are responsible for the majority of network traffic.

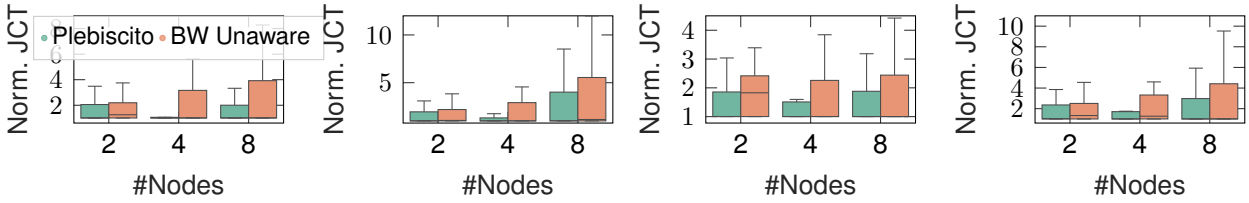
For each experimental run, we create a representative workload by sampling 250 jobs from the trace. Each job is composed of a number of distributed components (Λ), ranging from 2 (e.g., 1 PS and 1 worker, or just 2 workers for ring topologies) to as many as 32 (e.g., 16 PSs and 16 workers). Diversity in job scale enables us to test the scheduler under various distribution scenarios. To simulate a moderately high-load environment, jobs are submitted following a Poisson process with a mean inter-arrival time of 15 seconds. This arrival rate is configured to cause peak resource saturation approximately two-thirds of the way through the simulation, allowing us to observe system behavior under both contention and recovery. We run 30 such sampled workloads for each configuration to ensure statistical relevance. Since Plebiscito supports multiple allocation policies, we evaluate several of them by modifying the utility function as described in Section 1.5. Unless otherwise specified, the results shown assume a value of $\alpha = 0.5$ (Equation (11)) to equally balance the preference for compute and network resources.

Evaluation Goals. We organize the study around four guiding questions. First, *to what extent does a bandwidth-aware placement strategy improve overall cluster efficiency?* We quantify effects on JCT, AFR, and WCT across network capacities (§5.1). Second, *what is the underlying mechanism for these improvements?* We examine how Plebiscito responds to different resource footprints and how it shapes fairness across compute and network resources (§5.2, §5.3). Third, *is the decentralized protocol practical for real-world deployment?* We validate effectiveness and overhead on a physical prototype and



(A) DRF (B) YARN (C) THEMIS (D) LGF

FIGURE 5. *JCT vs. GPUs number.* Tail JCT jobs distribution shows the impact of different allocations depending on the number of required GPUs. The higher the number of GPUs, the higher the JCT, which is mitigated with Plebiscito BW aware formulation.



(A) DRF (B) YARN (C) THEMIS (D) LGF

FIGURE 6. JCT vs. allocated Nodes number. Tail JCT jobs distribution shows the impact of different allocations depending on the number of nodes used to allocate the job. The higher the number of used nodes, the higher the JCT, which is mitigated with Plebiscito BW aware formulation.

analyze decision latency and message complexity at scale (§5.5, §5.6). Finally, *how does Plebiscito perform under heavy load?* We push the system toward saturation and probe allocation race conditions to assess robustness in congested settings (§5.4).

5.1. Performance Gains from Bandwidth-Aware Orchestration. Here we test the workload over the settings described earlier to compare the JCT obtained by changing the different utility functions (Table 1), highlighting the importance of being bandwidth aware. We found that, regardless of the allocation policy, Plebiscito achieves up to $1.5\times$ lower JCT compared to bandwidth-unaware allocators.

Comparing Figure 4a (100Gbps) and Figure 4c (25Gbps), which report the normalized average JCT, it is evident that higher bandwidth leads to shorter JCT. Nevertheless, Plebiscito consistently improves performance across all tested utility functions Equation (11). In both scenarios, as the allocation across data center nodes and the network

becomes more congested, relying solely on local resource availability becomes insufficient. The worst bandwidth-unaware performances are achieved by YARN, DRF, and Tetris. Instead, SGF employs a packing strategy that can increase local leaf switch congestion, whereas LGF attempts to evenly distribute jobs across nodes, potentially leading to uniform bandwidth saturation and, consequently, higher congestion. In contrast, Themis and Tiresias introduce more placement-aware strategies. Themis balances fairness and performance using a finish-time fairness metric, while Tiresias relies on profile-based consolidation. However, these methods trade off placement quality against long-term performance and may be too slow to adapt to dynamic conditions, unlike Plebiscito, which captures trade-offs more effectively in real time. The same pattern holds for the AFR metric in Figure 4b, where Plebiscito is more successful in meeting bandwidth requirements, resulting in fewer failed allocations. A job fails on the first attempt when its resource requirements, especially bandwidth for distributed training jobs, cannot be met. In this case, bandwidth-aware scheduling leads to better bandwidth utilization, reduced AFR, higher resource usage, and minimized idle time. Finally, the WCT under 25Gbps in Figure 4d highlights that bandwidth-unaware allocations result in heavier bandwidth contention and poorer allocations, leading to higher WCT.

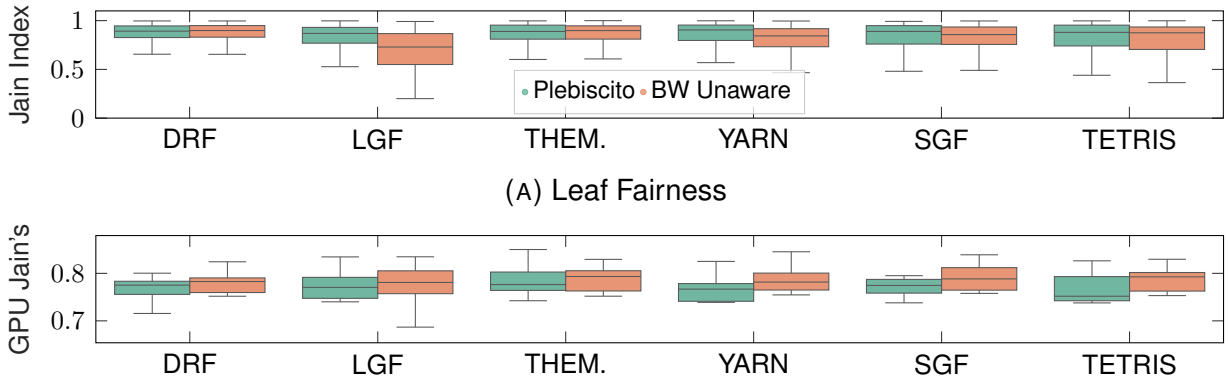
Plebiscito bandwidth awareness achieves up to $1.5\times$ improvement, enabling jobs to run on the cluster with reduced JCT, fewer bandwidth contentions, and optimal performance.

5.2. JCT Sensitivity to Resource Footprint. To better understand how resource demands and placement strategies impact the JCT, we highlight the behavior of jobs whose tail latency exceeds their nominal execution time, which can be estimated for each job based on its given duration. A higher duration is a direct consequence of bandwidth contention, which typically occurs at the peak of system utilization. Specifically, we analyze JCT as a function of two key dimensions: the number of GPUs per worker and the total number of nodes provisioned. We compare Plebiscito's JCT under the suite of utility functions introduced in our earlier experiments. This breakdown reveals clear patterns: jobs with high GPU requirements per worker suffer more when the allocation

policy is not bandwidth-aware. Similarly, jobs distributed across many nodes experience increased JCT, depending on how sensitive the utility function is to network versus compute headroom. This analysis highlights how distinct resource profiles map to JCT improvements under each placement strategy.

GPU Sensitivity. Figure 5 shows job performance grouped by GPU requirements per worker, focusing on jobs requesting 0.5, 1, or 8 GPUs. For each category, we include only jobs whose JCT exceeds their expected duration due to bandwidth bottlenecks. A general trend emerges: as GPU requirements per worker increase, baseline allocation mechanisms show greater slowdowns, especially when compared to the bandwidth-aware Plebiscito allocation. The long upper whiskers in Figure 5a, Figure 5b, and Figure 5c suggest that a small fraction of jobs endure severe contention; likely those whose workers end up straddling the data center spine. Plebiscito's bandwidth-aware bids reduce these extreme tails. The less severe JCT performance is observed in Figure 5c, where the Themis policy results in slowdowns up to 5$\times$ the target JCT. In contrast, Plebiscito mitigates these slowdowns through its bandwidth-aware allocation strategy and Gang Scheduling policy, which places workers of the same job in close proximity within the data center to reduce communication overhead, as described next.

In this experiment, we evaluate Plebiscito's Gang Scheduling performance by enforcing bandwidth-aware allocation and analyzing its impact under varying network capacities. Specifically, Figure 6 shows the JCT as a function of the number of nodes used per job (2, 4, and 8). We observe how different allocation strategies affect performance under these conditions. We further examine the spatial distribution of job components across nodes to assess whether Gang Scheduling is effectively enforced. This analysis focuses exclusively on jobs requiring one GPU per worker. As shown, a consistent trend emerges: the more nodes used per job, the higher the JCT. Notably, Themis exhibits strong performance in these cases (Figure 6c). However, it may still benefit from Plebiscito's bandwidth-aware placement, which attempts to colocate workers to minimize path length and avoid bottlenecks.



(A) Leaf Fairness

(B) GPU Fairness

FIGURE 7. *Fairness evaluation.* While the leaf switches remain heavily utilized during training, representing a major bottleneck for allocations, Plebiscito manages to achieve a fairer utilization across different placement functions.

Conversely, when a job spans fewer nodes, bandwidth consumption is more localized and results in lower JCT across all utility functions (Figures 6a, 6b and 6d). Again, Plebiscito consistently prioritizes bandwidth feasibility, reinforcing Gang Scheduling by keeping job components closer together on the node overlay.

5.3. Plebiscito Fairness. In a multi-tenant environment, unfair scheduling can allow small jobs to monopolize prime resources, starving larger jobs and increasing their JCT. Fairness ensures all users receive a reasonable share of resources, leading to predictable performance and better Quality of Service. We evaluate fairness using Jain's index for both bandwidth and GPU load, as shown in Figure 7. In bandwidth-unaware settings, policies like LGF and YARN exhibit poor fairness at the leaf-switch level, as they often route large jobs over already saturated links. Plebiscito, by contrast, proactively steers traffic to underutilized NICs, achieving a better allocation distribution that reduces contention and slightly improves overall fairness. This network-aware approach also fosters more balanced GPU allocations. When bandwidth is considered, all policies improve, but Plebiscito's utility function systematically prevents both network and compute imbalances by avoiding congested links from the start.

5.4. Performance Under High Load. A critical test for any orchestration system is its performance as the cluster approaches saturation. To evaluate Plebiscito under

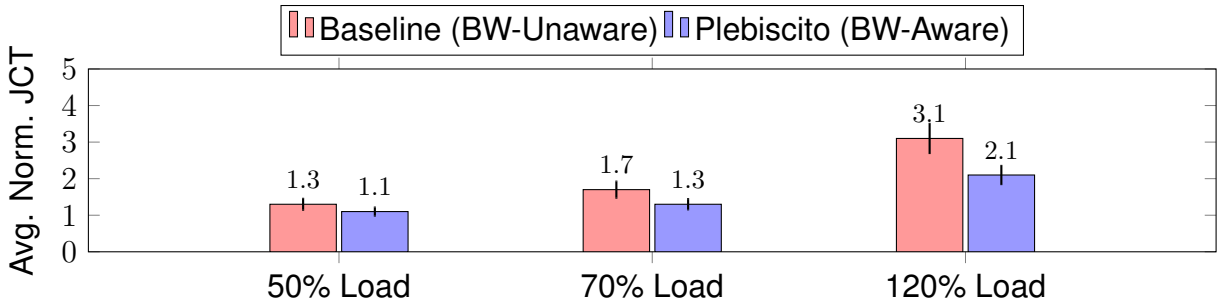
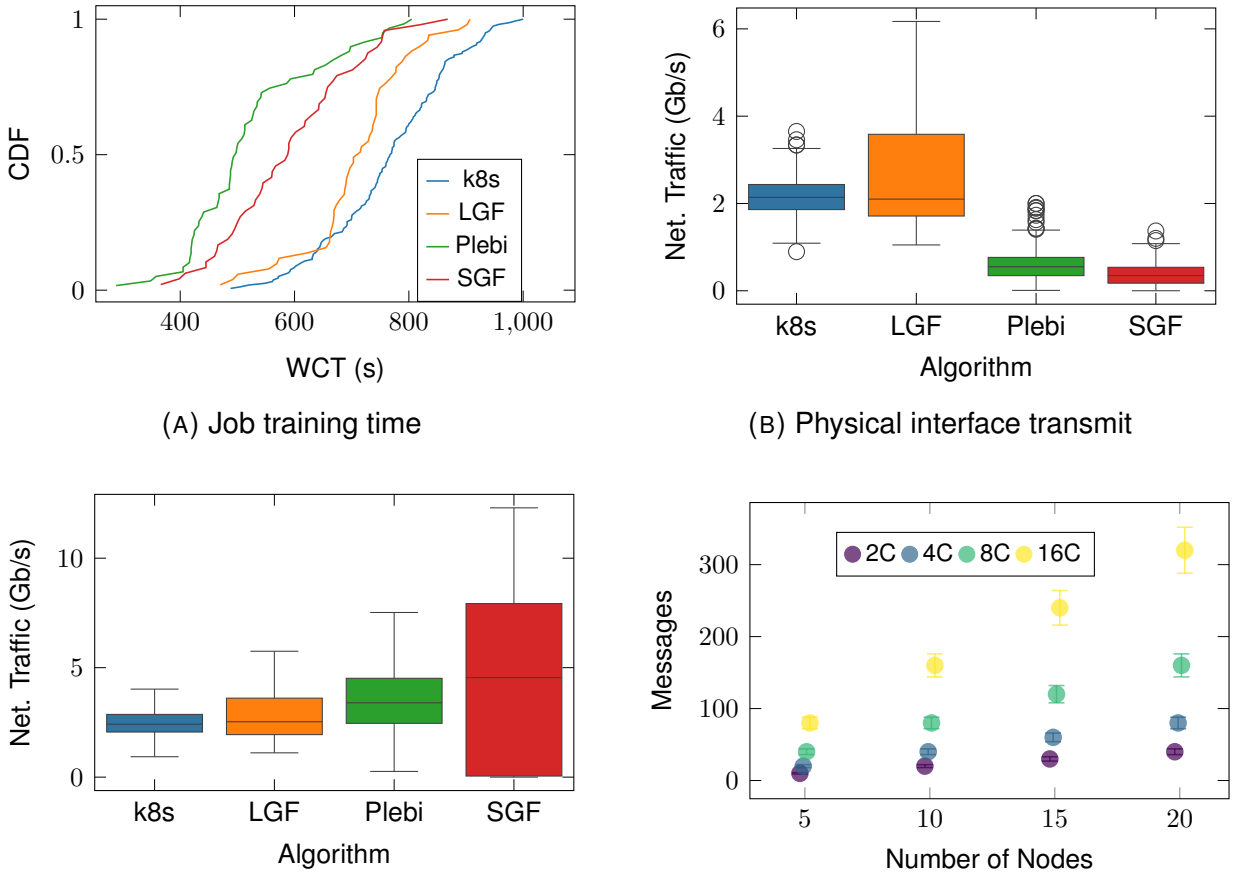


FIGURE 8. *Allocation race condition.* Comparison of average JCT as system load increases. The performance gap between Plebiscito and the baseline widens at higher loads, demonstrating Plebiscito's effectiveness in congested environments.

these conditions, we designed a stress-test scenario in our simulator, which we refer to as an *allocation race condition*, where all jobs arrive simultaneously (at time 0) into an unloaded system. We created three distinct workloads with aggregate GPU demands corresponding to 50%, 70%, and 120% of the total cluster capacity. For this experiment, we compared the average normalized JCT of Plebiscito against a bandwidth-unaware baseline. Both schedulers use a basic LGF utility function (Table 1), but Plebiscito is configured with $\alpha = 0.8$ (Equation (11)) to heavily prioritize network awareness. The results, shown in Figure 8, confirm that Plebiscito's benefits are most pronounced when resources are scarce. At 50% load, Plebiscito already provides a notable improvement. As the load increases to 70%, the baseline scheduler's performance degrades. Finally, in the overloaded 120% scenario, the baseline scheduler's JCT reaches 3.1x the ideal, while Plebiscito's JCT is only 1.9x the ideal. The performance gap widens as the load increases because network-agnostic placements are no longer viable, and intelligent, network-aware decisions become essential for avoiding bottlenecks. This demonstrates that Plebiscito is most valuable in the congested environments for which it was designed.

5.5. Prototype Evaluation. We validated Plebiscito's core functionalities on a small-scale evaluation on the FABRIC testbed [**69**]. As part of this evaluation, we analyzed the resource overhead of the Plebiscito agent to confirm its suitability for deployment on production nodes alongside training workloads. Our measurements confirm that the agent



(A) Job training time

(B) Physical interface transmit

(C) Localhost interface transmit

(D) Messages Overhead

FIGURE 9. *System Prototype*. Plebiscito strikes a better WCT compared to allocation policies more favorable to bin packing (SGF) or fairness (LGF) and the Kubernetes default allocator (Figure 9a). If possible, Plebiscito reduces inter-node bandwidth contention, as shown in (Figure 9b), while achieving higher local intra-node bandwidth utilization (Figure 9c). The message overhead to reach consensus among Plebiscito agents grows with the number of nodes and workers to place (Figure 9d).

is lightweight. The agent's computational footprint is minimal, as we observed no significant increase in CPU utilization even during active auctions. Memory consumption is also small; the agent's memory footprint, primarily used for storing auction state and the cluster topology view, increased by approximately 75 MB during our tests. Furthermore, the communication overhead is low, as protocol messages are small, averaging between 500 and 600 bytes (per Figure 2b). Given this low and predictable resource overhead, we conclude that the Plebiscito agent is a lightweight component. For the remainder of

our evaluation, we focus on the more critical performance metrics of JCT and allocation efficiency.

Test Case 1. We deploy six nodes behind a single switch, throttling one node's NIC to 5Gbps and leaving others at 10Gbps to demonstrate Plebiscito's bandwidth-aware allocation. Each node has an NVIDIA T4 GPU and 12 CPU cores. Following [**18**, **70**], we test a Deep Neural Network training use case. We used the PyTorch library [**71**] to train a ResNet-110 [**72**] model on the CIFAR-100 [**73**] dataset, running workloads of 20 training tasks with different worker configurations and batch sizes (32, 64, 128) under both PS and ring-all-reduce architectures. For each setup, nodes run the Plebiscito agent, placing bids per Equation (11), with f_{res} and f_{net} representing local resources and bandwidth. In PS placements, Plebiscito consistently computes the bandwidth-effective placement, improving throughput and reducing transfer time, as shown in Figure 1c compared to Figure 1b, with varying worker counts. Results for ring-all-reduce, omitted due to space, show similar behavior: Plebiscito avoids congestion and selects bandwidth-optimal placements.

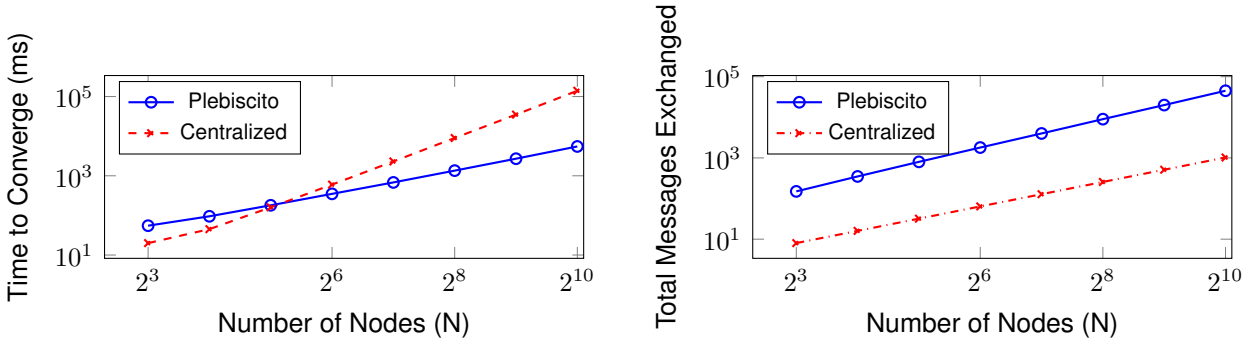
Test Case 2. To scale our evaluation beyond limited physical GPUs, we design a trace-driven emulator that replicates training durations and transfer sizes from earlier runs. While it abstracts GPU dynamics, it faithfully reproduces training round durations and send rates, enabling large-scale experiments without losing generality. We deploy the emulator on a Leaf–Spine topology with 4 Leaf and 2 Spine Fabric nodes, using FRRouting [**74**] for L4 routing. Each router interface has a 20Gbps NIC; each Leaf connects to 4 nodes with 10Gbps NICs. End nodes simulate 8 GPUs and 48 CPUs to support denser job placement. We submit 50 jobs whose communication patterns may use inter-node (NIC) or intra-node (localhost) paths depending on the allocation. Network stats are collected from */proc* per interface, with bandwidth samples averaged over 30s to avoid measurement overhead spikes that may be hidden but are discussed later. Applications also report training times to evaluate allocation efficiency.

Figure 9d presents the measured message overhead expressed as the number of exchanged messages associated with the Plebiscito allocation protocol across varying infrastructure and application sizes (i.e., 2, 4, 8, and 16 components). The results demonstrate that the overhead remains consistently low, with Plebiscito agents reaching consensus within 300 messages even in relatively large system configurations. Furthermore, the observed linear trend in message growth suggests that the proposed allocation exhibits favorable scalability characteristics. We analyze the time convergence complexity in the next Section 5.6

Focusing then on the testbed previously described, Figure 9b reflects usage on the physical NIC of the servers, while Figure 9c shows usage on the localhost interface. Kubernetes and LGF, being unaware of bandwidth constraints, result in higher physical interface utilization. In contrast, both Plebiscito and SGF reduce physical NIC usage: Plebiscito achieves this via explicit bandwidth-aware placement, while SGF implicitly reduces inter-node traffic by aggressively consolidating workloads on fewer nodes. Specifically, the Plebiscito allocation strategy takes into account the available bandwidth on the localhost interface. As a result, it tends to distribute the workload more broadly than SGF, which can lead to certain servers becoming overloaded while others remain underutilized (or idle).

Bandwidth-aware placement improves training performance, as illustrated in Figure 9a. Plebiscito performs the allocation resulting in the lowest training time; this can be achieved by carefully balancing the intra- and inter-node communication. On the other hand, SGF, while making even better use of the physical network, overloads the localhost interface of the nodes, resulting in an average training time that is approximately 20% higher. Finally, being the physical network far more critical than the localhost one, LGF and Kubernetes experience the highest training time because of the excessive use of inter-node communication.

5.6. Protocol Performance and Scalability. While our theoretical analysis provides worst-case bounds on performance, a critical question for any distributed system is its practical, real-world viability. The core challenge in large-scale resource orchestration



(A) Wall-clock time to find a placement.

(B) Protocol overhead.

FIGURE 10. Empirical analysis of Plebiscito's performance and scalability for placing a 16-component job. The y-axes are on a logarithmic scale.

is balancing the trade-off between the quality of a placement decision and the latency incurred in making it. An ideal scheduler must be fast enough for real-time operation, yet sophisticated enough to handle the complexity of a large, dynamic cluster. This section directly addresses these concerns by empirically evaluating the performance of the Plebiscito protocol.

We measure the end-to-end wall-clock time from when a job is submitted until a placement decision is reached in milliseconds. We analyze both the convergence time and communication overhead as the number of nodes grows. We compare Plebiscito's performance against the modeled performance of a traditional centralized scheduler to quantify the scalability advantage of our design. To this point, we conducted experiments in our event-driven simulator, generalizing the results from the implementation on the FABRIC datacenter, in order to scale our results up to 1024 datacenter nodes, for a 16-component job.

The results presented in Figure 10 confirm the practical benefits of our decentralized approach. As shown in Figure 10a, the decision latency for a centralized scheduler scales quadratically, quickly becoming impractical for clusters larger than 64 nodes. In contrast, Plebiscito's convergence time scales in a near-linear fashion, remaining well under a second, demonstrating its feasibility for real-time decision-making. Figure 10b explains this performance difference. While the centralized approach has a minimal

message count of $O(2N)$, it creates a serial bottleneck at the scheduler. Plebiscito's protocol generates more total messages, but because the communication and computation are parallelized across all nodes, it achieves a much lower wall-clock time. Furthermore, the empirical message count remains far below the theoretical worst-case bound. Together, these results validate that Plebiscito provides a scalable and practical solution for real-time, distributed resource orchestration.

After this discussion focusing on the data center, we will move on to describing the details of the FL edge architectures and how these can be improved with our proposed optimization. We start from the far edge of resource utilization and show how this can impact the FL training and improve the total time to accuracy.

CHAPTER 4: Distributed Edge Federated Resources

1. Privacy-Oblivious FSL

In this section, we detail our Federated Split Learning (FSL) architecture, which we originally proposed in [**75**]. FSL is a hybrid approach that combines the advantages of SL and FL. It avoids sending users' source data or sharing the complete NN parameters through the network while being scalable.

Our FSL architecture shown in Fig. 1e has three types of entities: (i) edge servers, (ii) clients, and (iii) parameter servers. To train a NN with FSL, we first set up an authentication protocol [**76**, **77**] among the entities to pair each client with one edge server, and edge servers with a parameter server. After pairs are found, the communications are sent without encryption. Then, in each client and server pair, we partition the complete NN into the client's partition and (edge) server's partition.

FSL has three training steps. In step 1, the client forward propagates with source data and transmits the intermediate data to the edge server. The server then finishes propagation and calculates loss. In step 2, the server backward propagates to the client source data. In step 3, after a few epochs, the *parameter server* averages the weights in the edge servers.

Comparison with other hybrid methods. Consequently, FSL will have multiple advantages compared to the other approaches discussed. FSL is more practical at the edge as its clients have lower computation and memory demand, i.e., fewer NN layers to train, compared with FL. Also, FSL only averages the weights in the edge servers, instead of the complete weights in FedAVG. Therefore, FSL reduces the risk of suffering model inversion attacks [**78**, **79**]. FSL also provides better scalability than SL since client and server pairs can train independently. Compared with FRC [**80**], FSL is more efficient in training time. FRC updates one weight shard per forward and backward propagation pipeline and runs multiple times to update the full model. Compared with PSL [**23**],

FSL has four potential advantages. *Scalability:* FSL allows Client-Edge Server pairs to train independently, which allows it to scale to more clients. On the other hand, the PSL server can either work with each client sequentially or enlarge the batch size accommodating the number of clients and waiting for all intermediate data. In the worst case, when n clients arrive at the same time, the lower bound on waiting time for each PSL client to begin backward propagation is $O(n)$. *Robustness:* We also observe that FSL has to maintain fewer states in each independent pair. The PSL server would temporarily store multiple batches of intermediate data simultaneously, so it needs a sophisticated logging and compaction storage system to avoid or recover from failures. Consequently, FSL is more robust than PSL. *Resilience:* In FSL, failure in one pair would not prevent other pairs from training or inference. However, the PSL design may suffer from resiliency problems when the single server failed. *Bandwidth Congestion:* For FSL, each edge server only processes intermediate data from its client, which leads to lower communication overhead. Meanwhile in PSL, since all intermediate data will be sent to and processed by the single server, bandwidth and computation resources at the server node may get overwhelmed. Our findings are presented in Section 3.

2. Privacy-Aware FSL

We have discussed the efficiency and resilience of FSL. In this section, we consider instead the privacy-preserving properties of different architectures and propose our privacy-aware FSL. In particular, we discuss how to complement general split learning-based architectures to mitigate the privacy concerns of sharing hidden variables or weights over an honest but curious network. We first give a formal definition of our privacy attacker model, and then we discuss how a Client-based Privacy Approach and certain ways of partitioning Neural Networks help to avoid such attacks.

2.1. Privacy Attacker Model and Assumptions. We assume that the attacker knows the structure of the client NN ($Client$), the *Intermediate Data* ($Client(x_{Sou})$) transmitted from the client to the edge server in plaintext and an auxiliary dataset (x_{Aux}) with features similar to the client source data (x_{Sou}). Then, following Eq. 13, the adversary can train an AutoEncoder NN with an Encoder part (Enc) and a Decoder part (Dec) [**81**]

to reconstruct x_{Aux} by minimizing a Mean Squared Error (MSE) loss comparing x_{Aux} and the AutoEncoder output $Dec \circ Enc(x_{Aux})$ where Enc has the same NN structure of $Client$.

$$\arg\min_{Dec,Enc}(MSE(x_{Aux}, Dec \circ Enc(x_{Aux}))) \tag{13}$$

And then the trained Decoder (Dec) can reconstruct the client source data (x_{Sou}) based on the intermediate data generated by client model ($Client(x_{Sou})$), as shown in Eq. 14.

$$x_{Rec} = Dec \circ Client(x_{Sou}) \tag{14}$$

Details of using this attack are discussed in Section 3.3.2.

2.2. Attack Resilience. Given the attacker model, in order to compare the level of privacy guarantee among different privacy approaches, we define an *Attack Resilience* metric (τ) as:

$$\tau = 1 - \frac{\|correct\|}{\|reconstructed\|} \tag{15}$$

It measures the misclassification rate. $\|correct\|$ counts the number of reconstructed images, which can be correctly classified by a trained classifier (Section 3.3.3). And $\|reconstructed\|$ is the total number of reconstructed images. Then we can compare the misclassification rate of the trained classifier based on the reconstructed datasets. The higher value means the reconstructed images have fewer features to be correctly recognized by the attacker's classifier, which is an indicator for higher attack resilience.

2.3. Client-Based Privacy Approach in Distributed Setting via Distance Correlation (CPA-DC). Motivated by the NoPeek approach [**82**], where they minimize Cross-Entropy while maximizing Distance Correlation (DC) in one loss function, we optimize the two loss functions in an alternately scheduling mechanism with two rounds (Equation 16): At epoch e, if $e \ mod$ F $== 0$, given *Frequency* F, the *regular round* minimizes the cross-entropy loss function $Loss(\cdot)$ with results of the inference model (server NN $g(\cdot)$ takes outputs of client NN $f(\cdot)$ based on input x) and ground truth $label$s in the (edge) server. Otherwise, the *DC round* maximizes the DC loss function comparing client source

data x and output $f(x)$ in the client node. Then by balancing the two rounds with *loss multiplier* m and *Frequency* F, we achieve high accuracy and privacy guarantee as only important features are preserved for training.

$$L = \begin{cases} Loss(g \circ f(x), label) & if \ \ e \bmod F == 0 \\ m \cdot DC(x, f(x)) & \text{otherwise,} \end{cases} \tag{16}$$

Our Client-Based Privacy Approach (CPA) can also work with other loss functions or methods teaching NNs to focus on important features. In section 3.3, we evaluated the trade-off among training time, accuracy, and attack resilience, comparing DC loss and Differential Privacy (DP-SGD).

2.4. How to partition a neural network? In this section, we discuss the problem of selecting how many layers need to be assigned for each NN partition, i.e., client NN depth. This tradeoff will tune training time (processing and transmission delay), privacy, and accuracy. Capturing the tradeoff between all these metrics is challenging. To illustrate, consider the tradeoff between processing and transmission delays. The output size of different layers can be quite different, so a few partitioning policies may lead to significant transmission overhead, increasing training time and hence diminishing the gain of the hybrid FSL compared to the original Federated Learning architecture. For example, in VGG-16 [**83**], the output size of the first convolutional layer is two times the size of the second convolutional layer. Thus, a system with a model cut after the second convolutional layer can trade off the extra processing delay at low-capacity clients while yielding a lower transmission delay. We evaluate this effect in Section 3.2.2.

Analytically, this effect is captured by solving Problem 17, where α and β are developer-specified parameters that represent positive weights for the transmission delay of intermediate data (I) and computation delay (C), the parameter d represents the depth of the client neural network, and finally, b represents the bandwidth, which is measured periodically. To efficiently solve this problem, similar to Neurosurgeon [**84**], we consider using two regression models to predict the delays I and C given the available bandwidth,

by profiling the model layers for output sizes and processing times offline using the resources for training.

$$\min_{d}(\alpha I(d \mid b) + \beta C(d)) \tag{17}$$

The solution of Problem 17 is optimal with respect to delays, however, it can be suboptimal with respect to privacy and accuracy. As Section 3.2.2 shows, the client processing and transmission delay of FSL reaches the minimum when the client NN depth is between 7 and 16. In Fig. 5a, instead, the client NN needs more than 16 layers to be above 90% attack resilience. Thus, we conclude that an optimal NN partitioning decision should balance different objectives and constraints, including transmission delay, processing time, privacy, and accuracy, as shown in our Problem formulation 18. In such a problem, W represents the model weight vector, (I', C', A, R) is the tuple representing the observations for transmission delay, computation delay, accuracy, and resilience, (γ, κ) are new user-specified positive weights, and d is the client NN depth.

$$\begin{aligned} \max_{W,d} \quad & (-\alpha I'(W, d \mid b) - \beta C'(W, d) \\ & + \gamma A(W, d) + \kappa R(W, d)) \end{aligned} \tag{18}$$

To solve Problem 18, we have to train W for each d until convergence and then find the best d. This brute force method is inefficient. A more efficient approach would rely on predicting the delays, accuracy, and privacy without the full training of the model. Extending the approach in [**84**] to go beyond profiling delays, is challenging. This is because the accuracy and attack resilience for each client and edge server pair are harder to profile and predict. Specifically, their profiling depends on the weights trained on other pairs, the distribution of source data among clients, number of clients, number of layers to average in SerAVG, and training epochs (Sec. 3.2.4 and 3.3). Another work can predict the model accuracy [**85**], but it is based on the already trained model. Therefore, for our FSL architecture with SerAVG, a prediction method for partitioning remains an open question for future work. In this paper, we experimentally demonstrate the best model partitioning that balances requirements on training time, accuracy, and privacy.

3. Evaluation Results

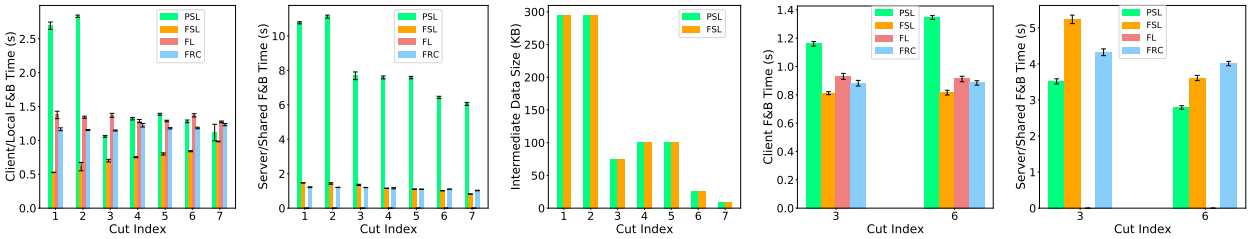
In this section, we describe the evaluation results related to our Privacy-Oblivious FSL and Privacy-Aware FSL architectures with our privacy-aware approaches (CPA in Section 2.3 and Neural Network Partitioning in Section 2.4) [1]. Our evaluation demonstrates the advantages of FSL over PSL and FRC in terms of training time, memory usage, and convergence rate. Moreover, we also show that our privacy-aware approaches can prevent the reconstruction of source images from intermediate data in the Split Learning-based systems. We first discuss our experimental setup, then present our evaluation results of Privacy-Oblivious FSL and Privacy-Aware FSL in Sections 3.2 and 3.3.

3.1. Experimental Setup. This experiment set studies the convergence for Privacy-Oblivious FSL and the privacy guarantee of Privacy-Aware FSL across different hardware and applications with different NNs and datasets.

For the hardware, we used two types of nodes on Chameleon Cloud [**86**]. One has an RTX6000 GPU, two Intel Xeon Gold 6126 CPUs and 187 GB memory. The other one has four NVIDIA V100 GPUs, two Intel Xeon Gold 6230 CPUs and 128 GB of memory. We emulated the computer network among our distributed learning entities on the localhost interface on a physical machine, and each experiment was set to use a single GPU, so that we can ignore the network bandwidth bottlenecks.

For the applications, we considered three classification tasks and implemented them with PyTorch [**71**]. Then the distributed communication among entities of the systems was handled by PySyft [**87**] and PyGrid [**88**], and no encryption was applied to the transmission. The first application runs a general image classification task with a VGG-16 [**83**] Convolutional Neural Network (CNN). The model was pre-trained using ImageNet [**59**] and then fine-tuned with the CIFAR-10 dataset [**89**]. We run this task on 5 clients running an NVIDIA V100 GPU. The second task uses a LENET [**90**] CNN to recognize hand-written numbers in the MNIST dataset [**91**] on 20 clients running an NVIDIA RTX6000 GPU. The third task classifies traffic, not images. In particular, we decomposed a one-dimensional-CNN, trained with the ISCX VPN-nonVPN (ISCX) traffic dataset [**92**], using

[1] `https://github.com/HEECMA-BU/FSL`



(A) LENET+MNIST Client Measurements

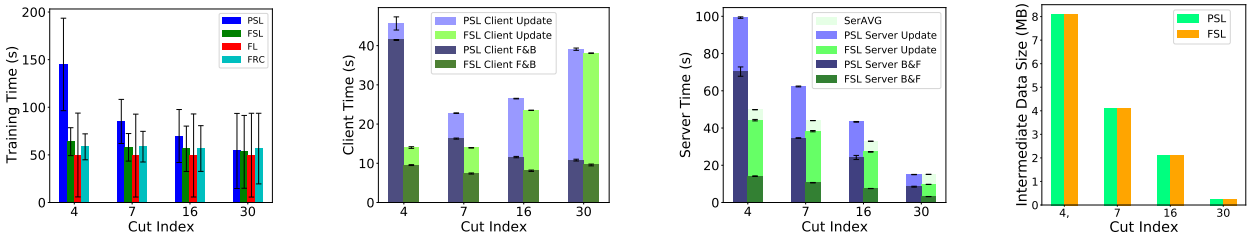
(B) LENET+MNIST Server Measurements

(C) LENET+MNIST Intermediate Data

(D) VPN Workload Client Measurements

(E) VPN Workload Server Measurements

FIGURE 1. *LeNet+MNIST*: (a) Client Time, (b) Server Time, (c) Intermediate data size. Observations: 1) Intermediate data size is correlated with the times taken by the PSL architecture, while having little correlation under FSL. Notice that since FSL and PSL do not modify the NN structure other than splitting the NN, the sizes of intermediate data generated by an FSL client and a PSL client at one cut index with the same input image are the same; 2) FRC has almost twice the overall training time as FL; 3) Plots are obtained by averaging 20 clients' results. The intermediate data size is under the batch size of 16 and each image was resized to (1,32,32). *VPN Workload*: (d) Client Time (e) Server Time. Similar considerations are valid for the VPN dataset. Tested with 5 clients with a one-dimensional NN with an input size of (1,784). Still, FSL has the shortest Client F&B time compared to the other settings.



(A) Pair Time

(B) Client Time

(C) Server Time

(D) Intermediate Data Size

FIGURE 2. VGG + CiFar10: Plots show the effect of Cut Index over (a) averaged overall training time, (b) time spent in clients (5 clients average), (c) time spent in server, (d) intermediate data size. The transmission delay caused by intermediate data size can dominate the training time (occurring during F&B propagation).

5 clients running on an RTX6000 GPU. We partitioned the dataset and assigned among different clients with Independent and Identically Distributed (IID) probabilities, and all our plots show 95% confidence intervals unless otherwise specified. Our goal is to verify that FSL can always converge with different tasks, different NNs, different devices, and

different data distributions. We verified the advantages in delay or privacy of FSL over existing solutions.

3.2. Evaluation Results for Privacy-Oblivious FSL. This section illustrates the methodology and draws observations of our experiments. Overall, our evaluations show that Privacy-Oblibious FSL has less overhead and similar accuracy compared to existing solutions. In particular, we evaluate training time (Sec. 3.2.2), memory consumption (Sec. 3.2.3), and learner accuracy (Sec. 3.2.4).

3.2.1. *Experiment Design.* Given the size of the different datasets and the number of clients, to reach at least 90% accuracy, the neural networks used for image classification needed 20 epochs. While for the traffic classification model, 80 epochs were used.

3.2.2. *Training Time Evaluation.* To evaluate training time, let us consider the experiment whose results are reported in Figures 1 and 2. The x-axis indicates the *Cut Index*, i.e., the index of the last layer running in the client/local part of the NN. When tested over the MNIST scenario, we can observe from Figure 1a and 1b that FSL has the shortest "Client Forward and Backward Propagation" (Client F&B) time among all other distributed architectures. The Client F&B time includes transmission time for gradients and computing both activations and gradients in Client NN. And Server F&B time includes transmission time for hidden variables from client to server and computation in Server NN. Notice that the weight update time is separately counted by "Client Update Time" and "Server Update Time". Moreover, we note that PSL is more vulnerable than FSL to limited bandwidth across splits. PSL is consistently the slowest due to its inefficient server design; the server has to synchronize and process all batches of intermediate data in each training epoch stacked as a big batch or sequentially.

Observing FRC and FL, we see the F&B times do not change along with the Cut Index (Figures 1a and 1b). Note also that FRC is not training time efficient. It updates its complete model with two almost full forward and backward steps [**80**]. This can be noted in the same figures: the FRC total F&B time for local and shared weights is almost doubled compared to the FL training time.

We were able to obtain similar results comparing the F&B times in another predicting scenario: the 1D CNN implemented by [**92**] (Figures 1d and 1e). Due to the limited size

of this neural network (with only two convolutional layers), we evaluated the architectures with merely two Cut Indexes: at layer 3 and layer 6 of the NN. Even in this experiment, we observe how our FSL still has the shortest Client F&B time. PSL is the worst performant at each cut, and FL keeps performing better than FRC.

FSL consistently uses less time in each training epoch than the other analyzed architectures. *We found that PSL perform worse than FSL because of the single-server architecture. PSL has similar results when comparing its client F&B time with FL and FRC. FRC is not training time efficient. Its total F&B time almost doubles compared to FL.*

When evaluating the training time on the CIFAR-10 scenario, we found a different trend (Fig. 2a): PSL had the longest training time, except for a cut index of 30. Moreover, FSL did not always perform the best. When most of the layers run within the client, FL has a shorter training time. This is because the size of intermediate data changes as the cut moves, and with smaller data to send, the overall training time can be shorter. Fig. 2b and 2c show the extra F&B time during training. And existing works for SL have discussed the similar behavior [**40**–**43**].

In particular, we show that FSL outperforms PSL as PSL F&B time is more vulnerable to intermediate data transmission. In Figure 1, Client F&B time of FSL keeps decreasing with a smaller Cut Index, while that of PSL still increases at Cut Index 6, 5, and 3, 2, although the intermediate data in this experiment is much smaller than using VGG-16. This behavior is caused by the single server bottleneck. Thus, FSL is more scalable in terms of training time. Such observation also explains why both client and server F&B times of PSL are consistently larger than FSL.

The intermediate data and gradients can cause significant network overhead. Such overhead, however, is better mitigated by our FSL than PSL. Existing work [**42**] [**43**] for Split Learning, as well as our partitioning strategies (Section 2.4) can further mitigate the communication overhead. Thus, the additional delay in FSL is not considered a severe bottleneck compared to those systems training at the edge, like FL.

3.2.3. *Memory Consumption Evaluation.* Memory usage of each entity in the edge training and inference systems limits the scope of devices that can join the system. To

compare which system is more flexible to deploy in terms of memory capacity on devices, we show the memory demands in FSL, PSL, FL, and FRC systems.

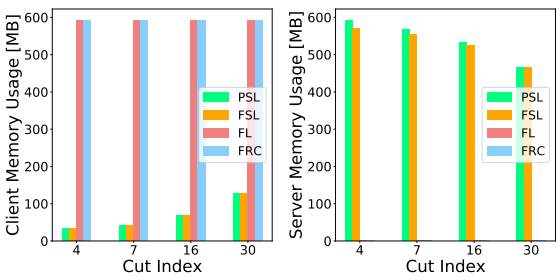
Real-world memory utilization can be highly variable as it depends on several implementation factors, such as libraries used and the Remote Procedure Calls (RPCs) implemented. However, the size of a model and its hidden variables are known. The following results show that FSL clients consume less memory compared to FL and FRC, and the FSL edge server occupies less memory than PSL.

The two plots in Figure 3 show the memory demands computed at the client and the server for each architecture. The x-axis shows the Cut Index and the y-axis represents the corresponding expected memory usage in MB. Note that FL and FRC do not split the NN, so their memory demands during training are only shown in the left plot.

As shown in Figure 3a, the sizes of each client NN's weight and hidden variables at each layer are the same in FSL and PSL. Since FRC and FL compute the full NN in the client during training, they require more than five times the memory, for Cut Index 4 to 30. Figure 3b instead shows that the server memory demand decreases with the cut index, as expected. However, notice that the PSL server needs extra memory to hold intermediate data from different clients, which leads to slightly higher memory demand than FSL.

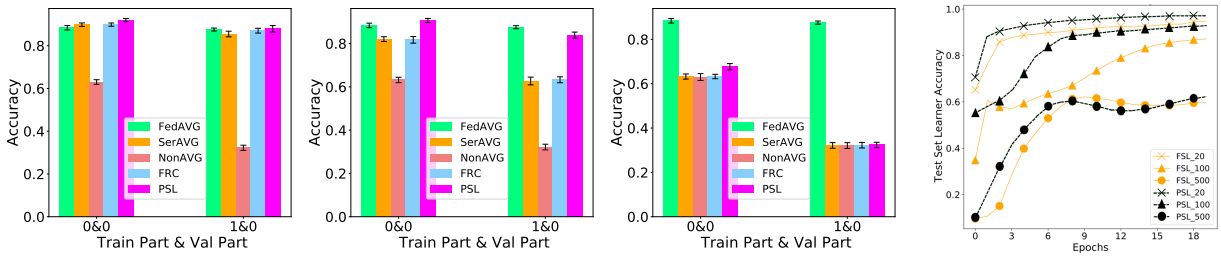
Memory usage of FSL compared to other systems. To conclude, we found that FSL's servers are lightweight compared to PSL. Thus, state management would be easier in FSL. Also, FSL's clients are lightweight compared to FRC and FL, during training, so they are more suitable at client devices.

3.2.4. *Learner Accuracy Evaluation.* The distributed training method in FSL, SerAVG, is similar to FedAVG (FL) but only averages the model weights of the NN in the (edge) servers (Section 1). In this subsection, we evaluate the convergence speed of SerAVG based on source data distribution, cut indices and data size at each client. Notice that we expect a higher accuracy with further hyper-parameters tuning, given prior results in similar contexts [**93**, **94**]. While our accuracy results show 95% confidence intervals, parameter tuning is out of the scope of this paper.



(A) (B)

FIGURE 3. VGG + CiFar10: Plots show (a) Client and (b) Server memory demands (size of model weights and hidden variables), with batch size 32 and image size (3,32,32). The FSL edge server uses slightly less memory, as less batches of hidden variables are present simultaneously compared to PSL. Also, the FSL client uses less memory compared to FRC/FL as only the Client NN partition is deployed versus the whole model. Notice that the clients of FSL and PSL have the same memory demand as the client models and output sizes are the same given the same cut index and image size.



(A) Accuracy (Cut Index = 3) (B) Accuracy (Cut Index = 5) (C) Accuracy (Cut Index = 7) (D) Accuracy (different number of clients)

FIGURE 4. From plots (a), (b) and to (c), the accuracy of SerAVG, PSL, and FRC decreases from the FedAVG level of accuracy to NonAVG, as fewer convolutional layers are averaged. PSL minimizes loss based on the hidden variables from all clients, so it achieves a little higher accuracy compared to SerAVG and FRC. However, FSL can achieve better privacy as shown in Section 3.3. SerAVG: Average (edge) server NN's weights; NonAVG: Each pair trains on its own; FedAVG: Average the complete NN's weights. In plot (d), we use a cut index of 3 and the FSL and PSL accuracies converge to similar values.

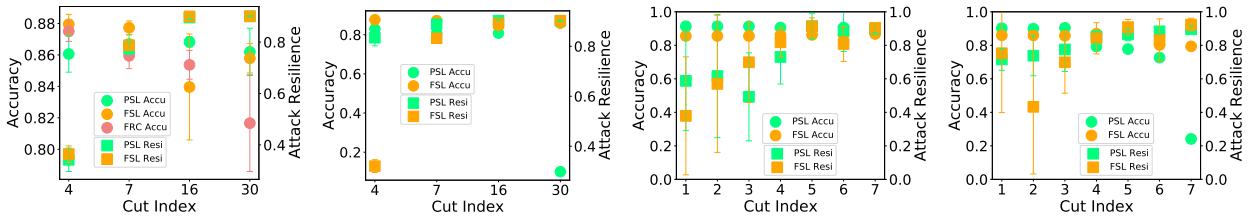
Tradeoff among Non-IID Data, Cut Index and Accuracy: In this experiment set, we evaluate the model accuracy given Non-IID source data and different numbers of feature layers to average. We split the data into two parts, namely part 0 and part 1. Each part includes data corresponding to half of the labels in the MNIST dataset and is divided

into a trainset and a testset. Note that the aforementioned way of splitting the dataset is an extreme Non-IID case. For example, a model trained on part 0 of the training set has no knowledge of the labels in part 1 of the training set. To assess the systems on more realistic data distributions, we further add 10% samples, uniformly and randomly selected from the complete MNIST dataset to both parts of datasets.

Consider Figures 4a, 4b and 4c, the x-axis represents the data partitions that trainset or testset belongs to, i.e., $0 \ \& \ 1$ means trainset of part 0 and testset of part 1 were used. The y-axis shows the validation accuracy. We show the results with cut indices 3, 5, and 7 in a modified LeNet, which correspond to the three convolutional (feature extraction) layers. From left to right, the accuracy decreases from the level of FedAVG (full weights sharing) to NonAVG (no weights are shared) when the server/shared NN for FSL, PSL, and FRC is shallower. When the Cut Index is 3, SerAVG, FedAVG, and PSL perform equally well. When the Cut Index is 5, SerAVG is worse than FedAVG but still higher than NonAVG. FRC has similar accuracy to SerAVG. PSL maintains similar high accuracy as FedAVG. Further, when the Cut Index is 7, all hybrid methods have similar accuracy as NonAVG. We conclude that SerAVG can enhance the accuracy in the Non-IID source data set, while worse than FL and PSL depending on the number of layers to average.

Comparing FedAVG and SerAVG, FedAVG averages all weights, while each FSL client NN is trained with a smaller batch of gradients ignoring other clients, which explains why a shallower (edge) server NN leads to lower accuracy. On the other hand, comparing PSL and FSL, the PSL server optimizes for minimal loss using all clients' batch output. Thus, the gradients applied with SerAVG will be less accurate compared to PSL, as the FSL clients are not updated considering the direction of other clients' gradients.

Note also a similar but smoother drop in accuracy based on Cut Indexes with IID CIFAR10 sources and VGG16 in Figure 5a. The two experiments with MNIST and CIFAR10 datasets suggest that SerAVG can achieve high accuracy by utilizing data of all clients when applied to suitable NN models and Cut Indexes. Specifically, when the



(A) Privacy Oblivious (VGG16+Cifar-10)

(B) DC 1 & Multiplier=2.0 (VGG16+Cifar-10)

(C) Privacy Oblivious (Lenet+MNIST)

(D) DC 1 & Multiplier=0.3 (Lenet+MNIST)

FIGURE 5. Accuracy and Attack Resilience for Privacy Oblivious and Privacy-Aware Architectures based on Cut Index. Note: DC 1 in captions refers to $DC\_Frequency = 1$. These figures show that to reach high accuracy and attack resilience, the Cut Index cannot be too big or too small, and loss multiplier is another way to enhance attack resilience. Furthermore, the loss multiplier can be more practical than DC_Frequency, since it doesn't introduce overhead to the training steps.

Cut Index is small, SerAVG achieves high accuracy while allowing deployments over resource-constrained client devices.

Tradeoff between Resource Demand and Accuracy. In this experiment, we evaluated the accuracy of FSL with small resource demand, i.e., small input data size and limited resources. The dataset is IID across 20, 100, and 500 clients yielding different input data size at each client, and we use cut index of 3 to isolate the accuracy drop caused by SerAVG. Our results are shown in Figure 4d. The x-axis indicates the index of epochs, and the y-axis shows the corresponding test set accuracy after a certain number of epochs. We found that with LENET and MNIST, the validation dataset can reach an accuracy range of 87% to 93%, as long as each client has enough data to train the machine learning model.

3.2.5. *Advantages and Limitations.* Our evaluation of Privacy-Oblivious FSL shows shorter training times and less memory footprint, although its accuracy is consistently lower (but only slightly) to that of other methods. However, when server node resources are not limited, e.g., in the Cloud, FSL may have a marginal gain over PSL. Thus, it is important to consider resource availability when choosing a system design.

3.3. Evaluation Results for Privacy-Aware FSL. In this section, we present the evaluation results of our privacy-aware FSL architecture and show that it can provide

certain privacy guarantees. FSL clients do not share the source data and model weights, so adversaries cannot directly access the source data or reconstruct them with the model weights using model inversion attacks [**78**, **79**]. However, when compared with Federated Reconstruction (FRC) [**80**] which trains a complete model at the client, FSL still sends the intermediate data through a network to complete the forward and backward propagation between clients and (edge) servers. An adversary could use such data to reproduce the source data, e.g., through an Autoencoder [**95**] NN, trained with a specific dataset, in Section 2.1.

To assess how our approach mitigates such vulnerabilities, we first introduce the evaluation setup, the design and usage of the attacker Autoencoder NN, and our experiment methodology. Then, we discuss the results of different privacy approaches, i.e., NoPeek [**82**] and the Client Based Privacy Approach (CPA), and the privacy level of different ways of partitioning the NN. NoPeek solves a multi-objective optimization problem of two loss functions, i.e., one maximizes accuracy, and the other maximizes the differences between source images and intermediate data. For CPA, we evaluate CPA-DC and CPA-DP. CPA-DC (Section 2.3) optimize the two loss function in NoPeek alternatively. CPA-DP applies a DP-SGD [**96**] algorithm in the clients. Finally we evaluate the privacy guarantee of different partitioning of client NN and server NN motivated by Section 2.4. We conclude this section by presenting results that demonstrate the high resilience to privacy attacks of our proposed FSL and the advantages in training efficiency.

3.3.1. *Evaluation Settings.* Our Privacy-Aware FSL and PSL extend our Privacy-Oblivious version by adding the CPA. Partitioning the NN was made easy by considering only sequential NNs (e.g., LeNET and VGG16). We tested both systems with the same image classification workloads (e.g., MNIST and CIFAR10) on the same hardware (i.e., NVIDIA RTX6000 and NVIDIA V100, respectively) as the Privacy-Oblivious setting.

3.3.2. *Setup the Attacker's Auto-Encoder Neural Network.* Following Section 2.1, the attacker trains an autoencoder NN, consisting of an Encoder and a Decoder, with an auxiliary dataset using the MSE loss function to minimize the difference between inputs and outputs, i.e. the input data can be faithfully reconstructed (decoded) from the output. The encoder part uses convolutional layers to extract latent variables from its input

dataset, and the decoder uses the encoder's last activation function outputs and transposed convolutional layers to reproduce the input dataset of the encoder. Consequently, we implement the encoder NN with the client NN structure, and let the decoder strictly mirror the client NN structure, i.e., the i^{th} layer of an encoder (client NN) would be the i^{th} last layer of the decoder, with transposed convolutional layer.

3.3.3. *Privacy Evaluation: Methodology.* In this subset of our evaluation, we want to show both CPA and carefully designed ways of partitioning NN provide high attack resilience while preserving high accuracy, based on different datasets, NNs, for split learning-based systems.

Each experiment includes trials initializing the Autoencoder's weights and data loaders with different random seeds. In each trial, the attacker used a dataset containing similar features to the learner's source dataset. To reconstruct MNIST, we selected EMNIST [**97**], and for CIFAR10 we selected the CIFAR100 [**89**]. The EMNIST dataset contains hand-written characters instead of numbers in MNIST, so features like lines and curves are the same and the attacker can decode those activation function outputs. Similarly, the CIFAR100 dataset contains 100 classes of RGB images instead of the 10 classes in CIFAR10, so the common features, e.g. classifying cat or dog, can be used to reconstruct with the activation function outputs from the CIFAR10 dataset.

For each trial of the experiment, we first let the attacker learn to reproduce her datasets. Based on the MSE loss, *i.e.* a loss function that measures how different the original and reproduced images are, the attacker updates her weights in each epoch. After 20 epochs, we used the decoder to reproduce the learner's dataset from the intermediate data.

When an autoencoder NN is trained, we train a new classifier with the same NN structure and source data as the learner to classify the reproduced images for 20 epochs. The mean and standard deviation of this classifier's *attack resilience* τ in section 2.2.

We show the results comparing Privacy-Oblivious and Privacy-Aware FSL and PSL systems with different CPAs and Cut Indexes. Then, we evaluate the trade-off between accuracy and attack resilience for FSL and PSL.

3.3.4. *Privacy Evaluation using NoPeek.* As we illustrated in Section 3.3.6, NoPeek solves a multi-objective optimization problem that takes in the source data and intermediate data to maximize the difference with a Distance Correlation (DC) loss function, as well as the prediction and labels to maximize the accuracy. To solve such an optimization problem, NoPeek has to share the value of the loss over a network which may cause vulnerability or added complexity in maintaining the gradient graph. As shown in Table 1, this approach has both high attack resilience (i.e., 97% for PSL and 98% for FSL) and high learner's accuracy (i.e., 97% for PSL and 96% for FSL) when trained for the same number of epochs and clients as the Privacy Oblivious experiment with the MNIST dataset and LENET NN.

cases	PSL	FSL
attack_resilience(τ)	0.9733	0.9837
learner_accuracy	0.9702	0.9614

TABLE 1. NoPeek stats with 20 clients training 20 epochs.

3.3.5. *Evaluation Result using Client-Based Privacy Approach via Distance Correlation (CPA-DC).* To mitigate the drawbacks of the loss value sharing, we consider a new approach that prevents transmitting data outside clients, improving upon NoPeek. We optimize for the similar two objectives in NoPeek alternatively in the Client-Based Privacy Approach via DC. As shown in Equation 16, there are *DC Frequency (F)* and *Loss Multiplier (m)* to evaluate. *F* defines how many times the DC loss function is optimized given the Cross Entropy has been optimized once. *m* is applied to the loss function result. These two parameters control how different the intermediate data and source data will be, by changing the frequency of optimizing the DC loss and by changing the learning rate of gradients applied during that optimization, respectively.

We note multiple tradeoffs in CPA-DC. First, CPA-DC is not as training time-efficient as NoPeek. NoPeek can optimize its two objectives with one forward and backward steps while CPA-DC has to solve them sequentially. However, we consider that NoPeek transfers more information than necessary over a network. Second, there are tradeoffs for *DC Frequency (F)* and *Loss Multiplier (m)*. Increasing *F* adds more epochs to optimize for DC loss, so the attack resilience would be higher at the expense of a longer

training time. Instead, by increasing *m*, we can keep a small *F*, which reduces training time while maintaining attack resilience. Intuitively, multiplying the DC loss by a larger *m* is similar to increasing the gradient descent step size. Thus, we need fewer DC epochs to maintain the attack resilience, while the gradients can be sub-optimal. Based on the discussion, we expect that a large *m* combined with *F* of 1 can balance between training time efficiency and DC loss gradients' accuracy. These two parameters should be carefully designed in a production environment.

We first experimented with MNIST classification with different DC Frequencies and a constant loss multiplier of 0.1 (equivalent to reducing the learning rate by 10%) and studied the tradeoff between accuracy and attack resilience, as shown in Fig. 6a. The x-axis represents the DC Frequency, the left y-axis shows the learner accuracy and the right y-axis shows the corresponding attack resilience.

From the top plot of Figure 6[2], as DC Frequency is increasing, for both Privacy-Aware FSL (PAFSL) and Privacy-Aware PSL (PAPSL) systems, the attack resiliency increases and the learner accuracy decreases, as expected. Notice that PAFSL achieves better accuracy and good resilience for most DC Frequency values. For DC Frequency from 10 to 20, given that the attack resilience of PAFSL and PAPSL are close within 10% difference, PAFSL achieves more than 90% accuracy. From 25 to 35, PAPSL does not learn any features while PAFSL still has about 80% accuracy. When DC Frequency is five, the PAPSL has an advantage over PAFSL, with close accuracy, and PAPSL has around 20% more attack resilience.

The result shows that Privacy-Aware FSL with CPA-DC is easier to tune for high accuracy and attack resilience. *Within the wider domain of DC Frequency, PAFSL has higher accuracy and good attack resilience compared to PAPSL.* This is because of the learning rate in SerAVG and PSL. The server weight update rule of PSL is shown in Equation 19.

[2]A DC Frequency of zero corresponds to Privacy-Oblivious FSL and Privacy-Oblivious PSL, i.e. without privacy awareness.

(19) $$W_g^{t+1} = W_g^t - \eta \sum_{i=1}^{N_C} \frac{\partial g(f(x_i))}{\partial W_g}$$

Similarly for FSL, we have Equation 20.

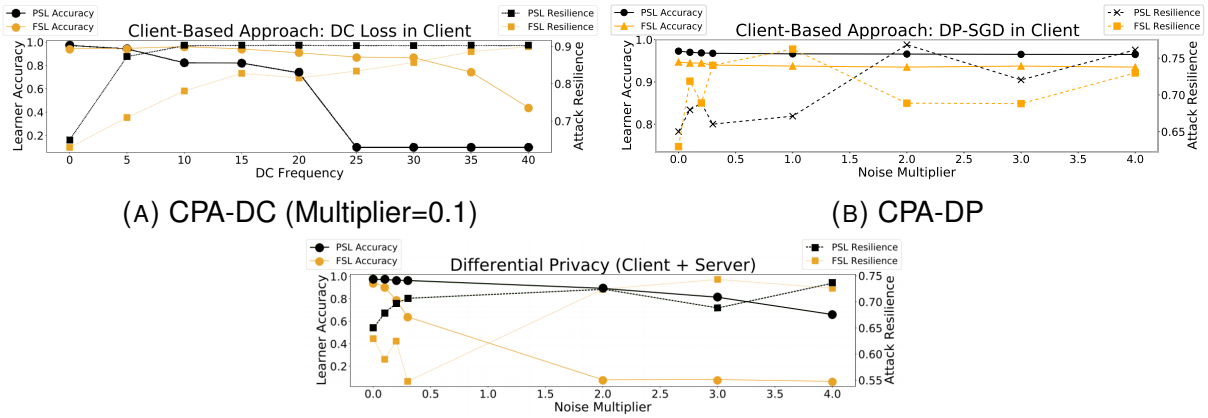
(20) $$\begin{aligned} W_g^{t+1} &= \frac{\sum_{i=1}^{N_C}(W_g^t - \eta \frac{\partial g(f(x_i))}{\partial W_g})}{N_C} \\ &= W_g^t - \frac{\eta}{N_C} \sum_{i=1}^{N_C} \frac{\partial g(f(x_i))}{\partial W_g}, \end{aligned}$$

where W_g^t indicates the weights in the server at iteration t, g is the server NN, f is the client NN, x_i represent the i-th batch of data, N_C is the number of clients, and η is the step size. Intuitively, since PSL has a larger step size, its server NN can be confused quicker than FSL servers by the intermediate data. Moreover, the confused server NN can further confuse the client NN. It justifies our observation that PSL's accuracy and attack resilience become unstable quickly when increasing the *DC Frequency (F)*. Therefore, we conclude that *FSL is easier to tune compared to PSL.*

In Figures 5b and 5d, with a fixed *DC Frequency (F)*, we show the accuracy (left y-axis) and attack resilience (right y-axis) based on different *Loss Multiplier (m)* for different models and datasets. Furthermore, we compared the privacy oblivious cases (Figure 5a and Figure 5c), and the privacy-aware cases at different Cut Indexes (x-axis). As expected, increasing the *Loss Multiplier (m)* enhances attack resilience but reduces accuracy for different datasets prevalently, especially when the client NN is deep.

Overall our evaluation of CPA-DC shows good attack resilience and accuracy with a combination of small *DC Frequency (F)* and big *Loss Multiplier (m)*. And our FSL has better accuracy and similar attack resilience to PSL. We hence conclude that our CPA-DC can defend against our attacker model.

3.3.6. *Privacy Evaluation with Differential Privacy Approach.* The previous section has discussed the CPA-DC, but instead of DC there are other lightweight methods that can enhance the privacy guarantee which adds noise to the client's NN while preventing depletion of the client's battery quickly. In this section, we compare CPA via Differential



(A) CPA-DC (Multiplier=0.1)

(B) CPA-DP

(C) DP-SGD in Complete NN

FIGURE 6. LeNet+MNIST: Accuracy and attack resilience (τ) with 20 clients and a Cut Index of 3 for Client-Based Privacy Approaches (via DC (a) and via DP (b)) and DP-SGD on the global learner model (c). Our FSL with both client-based policies guarantees a high level of privacy and accuracy.

Privacy (CPA-DP uses the popular DP-SGD [**96**] algorithm inside clients) and CPA-DC. Also, we show that naïvely using DP-SGD in an FSL system would lead to low accuracy. The implementation extends the Privacy-Oblivious FSL with a DP-SGD optimizer, provided by the Opacus [**98**] library. This method would add normally distributed random noise to the gradients during backward propagation based on $noise\_multiplier$ ϵ. This parameter controls the magnitude of the noise added. Notice that DC generates the gradients in a specific direction to reduce the correlation between intermediate data and source data in each *Distance Correlation round*. So we expect CPA-DP to have a worse level of privacy, given the same level of learner accuracy, compared to CPA-DC. Thus, the focus of this section is to show that CPA can be applied with other privacy methods like DP, despite DP's worse privacy compared to DC.

We summarize the results of CPA-DP in Fig. 6b. This plot shows the result when Cut Index equals 3. The x-axis is the $noise\_multiplier$. The attack resiliency of FSL and PSL with $noise\_multiplier > 0$ is consistently better by nearly 5% than FSL and PSL with $noise\_multiplier = 0$. At the same time, the accuracy decreases by less than 1% in either FSL or PSL from $noise\_multiplier = 0$ to $noise\_multiplier = 4$.

CPA is a general approach and can be customized with different methods to enhance attack resilience. The evaluation shows that CPA-DP can also improve attack resilience, while learner accuracy does not change much. Compared with CPA-DC, both methods provide similar learner accuracy, but CPA-DC's attack resilience is higher.

We now compare CPA-DP against applying DP-SGD in both client NN and server NN. Figure 6c shows high attack resilience, but the learner accuracy of FSL drops below 60% when $noise\_multiplier \geq 0.3$. Meanwhile, PSL shows a similar behavior as using CPA-DP. So, CPA-DP is considered a better method for FSL to enhance its attack resilience than DP-SGD applied in both clients and servers. The reason for FSL's lower accuracy under DP-SGD and high noise can be attributed to its SerAVG. After applying SerAVG, the distribution of the random noise in the server NN can be arbitrary, as shown in E.q., 21, while the noise in the client NN stays intact. Thus, the resulting complete NN in FSL may not converge.

$$W_g^{t+1} = W_g^t - \frac{\eta}{N_C} \sum_{i=1}^{N_C} (\frac{\partial g(f(x_i))}{\partial W_g} + e_{g_i}^t) \tag{21}$$

The variable $e_{g_i}^t$ indicates the gaussian noise added for the i-th server NN at iteration t according to $e_{g_i}^t \sim \mathcal{N}(0, (noise\_multiplier \times max\_gradient\_norm)^2)$ [**98**].

3.3.7. *Evaluation Using Different ways of Partitioning NN.* In Figure 5a and Figure 5c, the right y-axis shows the attack resilience, and the x-axis indicates the Cut Indexes. Overall, if we have a deeper client NN (i.e., moving from smaller to bigger Cut Indexes), the attack resilience increases and accuracy decreases (consistent with the SerAVG Evaluation in Section 3.2.4). After adding more layers, the intermediate data would have fewer features from the source data, but only keeps those that can improve the classification accuracy. Thus, fewer features are preserved and the attacker's ability to reconstruct the source's data is hindered.

We also want to emphasize that with the CIFAR10 workload, when there are 7 layers in the clients, the attack resilience reaches about 80%. We reach the same result with MNIST workload at the Cut Index of 4. No extra privacy-aware method was used, and

as we discussed earlier, the transmission delay can also be reduced with a deeper NN in the client due to potentially smaller intermediate data.

Cut Index is an important hyper-parameter for training delay, accuracy, and privacy. *Different Cut Indexes bring the following tradeoff: the deeper client NN adds more resource demand at the edge, but reduces the transmission time and enhances attack resilience. On the other hand, a shallower server NN may lead to lower accuracy with SerAVG.*

Furthermore, system architects can combine the approaches mentioned above, e.g., having a moderately deep NN in clients and using the CPA-DC, to find a balance between resource demand and performance. As in Figure 5d, with $cut\_index = 4$, $DC\_Frequency = 1$ and $loss\_multiplier = 0.3$, we still get about 80% attack resilience and more than 90% accuracy.

On the other hand, when comparing FSL and PSL, we note that FSL has more hyper-parameters to tune. But, in all experiments reported in Figure 5, when the Cut Index is large, FSL has better accuracy than PSL. So we conclude that a carefully specified way of partitioning the NN can benefit the most when applied in hybrid federated-split learning systems.

3.3.8. *Advantages and Limitations.* The CPA-DC method introduces a separate *DC round* to minimize DC loss, which adds extra training overhead. We further introduce a *loss multiplier*, to minimize the overhead by changing the step size. Tuning it requires profiling and experience. We thus conclude that FSL and CPA cannot solve all the limitations in Federated Learning (FL) and Split Learning (SL) systems. But, their flexibility allows users to customize a better distributed learning system that meets their objectives in terms of latency, privacy, and accuracy.

CHAPTER 5: Federated Learning with In-Network Aggregation

Deep neural architectures have achieved state-of-the-art performance across multiple domains, but their reliance on decentralized, domain-specific data has made Federated Learning (FL) an essential paradigm [**99**, **100**]. While FL mitigates privacy concerns by keeping raw data local, its deployment over wireless edge networks is fundamentally constrained by network dynamics. The repeated, synchronized exchange of massive model updates over heterogeneous radio links and shared backhaul infrastructure creates severe bottlenecks [**101**, **102**]. Existing model-centric (e.g., compression) and system-centric (e.g., client scheduling) optimizations treat the network as a passive pipeline, failing to resolve the underlying transport limitations.

To address this, we propose *FLAG (Federated Learning with In-Network Aggregation)*, an architecture that transforms the programmable 5G infrastructure into an active participant in the training loop. Instead of transmitting complete updates to a remote parameter server, FLAG performs streaming, on-the-fly aggregation at line rate directly within the 5G gNodeB (gNB) via the SDAP layer. To handle wireless variability, FLAG introduces **Partial Contribution Correction (PCC)** to correct bias from lost updates, and **Deadline-Driven Grouping (DDG)** to bound straggler delay through adaptive, timer-based aggregation.

Our theoretical and empirical evaluations demonstrate FLAG's ability to co-design data transport and model convergence. We prove that FLAG converges to a bounded neighborhood of the optimal model even under packet loss and asynchronous updates, and we derive an analytical expression for the optimal aggregation deadline, τ^*. Furthermore, FLAG achieves an asymptotic communication reduction of $O(N/F)$ (where N is the number of clients and F is the number of gNBs) without losing precision. Validated on a 5G emulation testbed, FLAG delivers up to $5.1\times$ faster time-to-accuracy compared

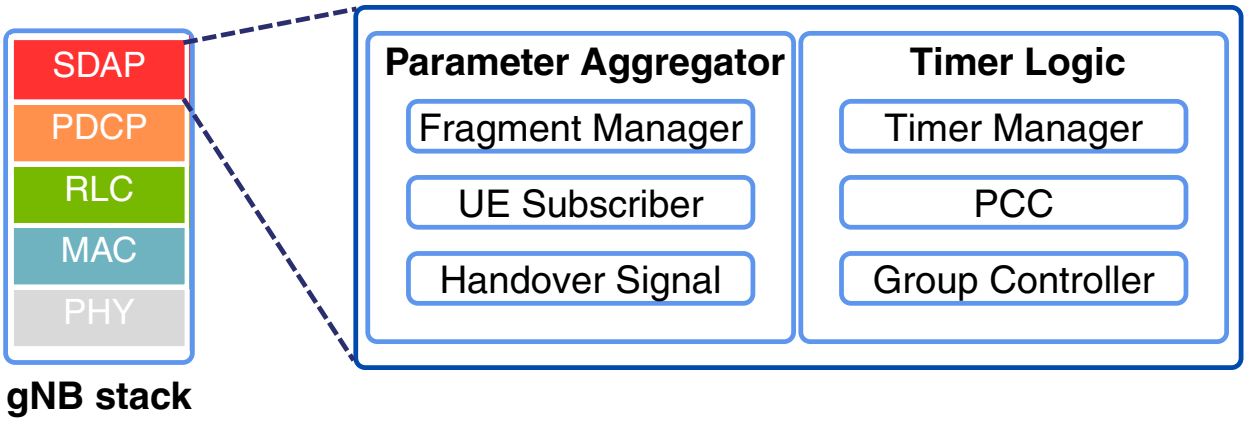


FIGURE 1. 5G gNB extension with FLAG. The Parameter Aggregator (PA) aggregates fragments, while the Timer Logic coordinates aggregation.

to classical FedAvg, while outperforming hierarchical and asynchronous baselines by up to $2.6\times$ under heavy packet loss.

1. FLAG System Design

In conventional FL, the gNB acts as a simple traffic forwarder, sending individual model updates from every device over a shared, and often congested, backhaul link to the PS. This design treats the gNB as a passive conduit, ignoring its strategic position and computational capabilities. This inefficiency creates the central bottleneck that client-side optimizations alone cannot solve.

FLAG is built on a different principle: transforming the gNB from a passive forwarder into an active participant in the learning process. It is a lightweight, 5G-compliant system that brings In-network Aggregation (INA) into the radio access network to alleviate these bandwidth constraints, enabling a broader set of applications to benefit from faster federated communications. It extends the gNB with additional modules to manage FL traffic while requiring no modifications to the client or the PS. This section describes the system architecture, the registration process, and the protocol to support INA.

1.1. System Architecture. The architecture of FLAG is shown in Figure 1. Aggregation is performed inside the gNB, above the SDAP layer, on FL traffic identified by a

dedicated Quality of service Flow Identifier (QFI). Two main building blocks implement this pipeline: the Parameter Aggregator (PA) and the Timer Logic.

Parameter Aggregator (PA). The PA is responsible for processing incoming model updates and managing client mobility. Its core component, the Fragment Manager, partitions large tensors into MTU-sized fragments for transmission and reassembles received fragments into complete updates, while also executing the FLAG INA Algorithm 4, described in Sec. 1.4. The Handover Signaler maintains continuity when a UE moves across cells, transferring the FL state to the new gNB so that the client resumes participation without losing its contribution to the current round. Finally, the UE Subscriber maintains a registry of all UEs involved in training, tracking their identifiers, join status, and activity, which is essential for coordinating aggregation and mobility support at scale.

Timer Logic. The Timer Logic ensures that aggregation proceeds timely across heterogeneous UEs. The *Timer Manager* enforces strict deadlines on fragments, discarding those that arrive too late to avoid delaying the entire round due to stragglers. The *PCC* module (Section 2) complements this by allowing the system to incorporate partial model contributions without compromising accuracy, reducing the impact of missing updates. In parallel, the *Group Controller* selects which UEs participate in each round according to their advertised deadlines and channel conditions. Together, these mechanisms prevent slower devices or poor links from dominating training time while still retaining diversity in model contributions.

Integration into the 5G stack. The INA pipeline is integrated directly into the existing 5G stack, minimizing disruption to standard operations. Aggregated fragments pass through the SDAP, PDCP, and GTP-U layers to the co-located PA, which produces a single averaged update before forwarding it to the Cloud PS. This design ensures that the uplink load from multiple UEs is reduced to a single transmission per gNB, lowering backhaul usage. On the downlink, the updated global model is broadcast by the Parameter Server and relayed transparently through the same stack to all registered UEs. Because FLAG operates entirely within the gNB and respects existing 5G control plane

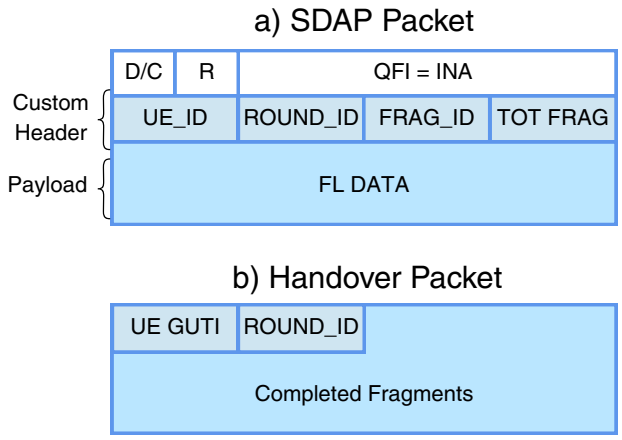


FIGURE 2. a) Custom SDAP header for INA traffic. b) Handover message during mobility events.

procedures, it can be incrementally deployed without modifications to clients or the core network.

UE Registration. The UE Subscriber manages the registration of devices for FL tasks. After completing the standard 5G attach procedure, a UE signals its intent to join FL by transmitting a FLAG-specific identifier (UE-ID). This identifier, aligned with temporary 5G identifiers such as GUTI, ensures unique addressing for FL operations. The UE Subscriber updates its internal registry with the UE's status. The same identifier is also used by the Handover Signaler to migrate a UE's FL state across gNBs during handovers (Section 1.4).

1.2. FLAG Aggregation Protocol. To support INA, FLAG partitions each tensor into fragments that fit within the 5G Maximum Transmission Unit (MTU), typically 1,500 bytes. Fragments are encapsulated at the SDAP layer with a custom FLAG header and the tensor payload, as shown in Figure 2(a).

Fragment format. The header contains metadata including the UE-ID, round number, fragment ID, and the expected number of fragments. This information allows the gNB to synchronize updates, prevent duplicates, and track participation. The payload carries partial model updates.

Algorithm 4 FLAG gNB Aggregation

procedure CLIENTLOCALUPDATE($c_i, w(k-1)$)
  $w_i(k) \leftarrow$ LocalUpdate$(w(k-1))$
  Split $w_i(k)$ into fragments $\{f_i^{(m)}\}_{m=1}^{M}$
  send $\{f_i^{(m)}\}_{m=1}^{M}$ to associated gNB
end procedure

procedure GNBAGG(gNB$_j$, $\{f_i^{(m)}\}_{i\in\mathcal{C}_j,\, m=1,\ldots,M}$)
  Let $M = \frac{\text{modelsize}}{\text{MTU}}$
  $A \leftarrow \mathbf{0}_{M\times\text{MTU}}$ ▷ Row $A^{(m)}$ aggr. fragment $f_i^{(m)}$
  $\mathbf{N} \leftarrow \mathbf{0} \in \mathbb{N}^M$ ▷ N_m counts received frag. for seg. m
  for each client $c_i \in \mathcal{C}_j$ **in parallel do**
    for $m = 1, 2, \ldots, M$ **in parallel do**
      $A^{(m)} \leftarrow A^{(m)} + |D_i| \cdot f_i^{(m)}$
      $N_m \leftarrow N_m + 1$
      if $N_m = |\mathcal{C}_j|$ **then**
        Send $\frac{A^{(m)}}{\sum_{c_i\in\mathcal{C}_j}|D_i|}$ ▷ Send agg. frag. to PS
      end if
    end for
  end for
end procedure

procedure GLOBALAGGREGATION($\{w_j(k), |D_j|\}_{j\in\mathcal{G}}$)
  $w(k) \leftarrow \frac{1}{M}\sum_{j=1}^{M} w_j(k)$
  send $w(k)$
end procedure

procedure BROADCASTGLOBALMODEL($w(k)$)
  for each $c_i \in \mathcal{C}$ **do**
    $w_i(k) \leftarrow w(k)$
  end for
end procedure

[FLAG Aggregation at the SDAP layer algorithm.]Aggregation at the SDAP layer. Embedding INA at the SDAP layer provides flexibility while preserving 5G compliance. Implementing aggregation at PDCP would disrupt encryption contexts, sequence numbers, and state machines [**103**], while performing it at GTP-U would interfere with one-to-one tunnel mappings between UEs and the core network [**104**]. By contrast, SDAP enables the isolation of FL traffic through a dedicated QFI without impacting other layers.

1.3. FLAG Aggregation Formulation. Consider a set of client devices (UEs) $\mathcal{C} = \{c_1, c_2, \ldots, c_N\}$, where each client c_i owns a local dataset D_i and trains a local model w_i.

Let $\mathcal{G} = \{\text{gNB}_1, \text{gNB}_2, \ldots, \text{gNB}_M\}$ denote the set of M gNBs (base stations) participating in the system. Each client c_i is associated with exactly one gNB, and each gNB j serves a disjoint subset of clients $\mathcal{C}_j \subset \mathcal{C}$:

$$\bigcup_{j=1}^{M} \mathcal{C}_j = \mathcal{C}, \quad \mathcal{C}_j \cap \mathcal{C}_k = \emptyset, \quad \forall j \neq k. \tag{22}$$

A central parameter server (PS) aggregates the updates received from the gNBs into the global model w_{global}. In a conventional hierarchical FL design [**46**], a designed edge server acts as a local PS, aggregating the models of its clients (superscript *hier* denotes hierarchical aggregation at the gNB):

$$w_{\text{gNB}_j}^{\text{hier}} = \frac{1}{|\mathcal{C}_j|} \sum_{c_i \in \mathcal{C}_j} w_i, \tag{23}$$

and forwarding the result to the PS. The PS then performs a second-level aggregation:

$$w_{\text{global}}^{\text{hier}} = \frac{1}{M} \sum_{j=1}^{M} w_{\text{gNB}_j}^{\text{hier}}. \tag{24}$$

This approach reduces uplink load but has two main drawbacks. First, it requires geolocated edge servers to aggregate traffic. Client updates must be routed from the device to the edge server and then to the PS, potentially increasing the physical path they must traverse. Second, while the aggregation module at the edge server provides benefits, its design at the application layer introduces computation overheads and software complexity. Instead, the proposed approach leverages packet-processing capabilities directly at the gNB to perform *line rate aggregation* without turning the gNB into a full local PS. Each client $c_i \in \mathcal{C}_j$ transmits its local model w_i to the serving gNB j. The gNB aggregates the updates of its clients in real time:

$$w_{\text{gNB}_j}^{\text{INA}} = \frac{1}{|\mathcal{C}_j|} \sum_{c_i \in \mathcal{C}_j} w_i, \tag{25}$$

and forwards a single aggregated update to the PS. Note that $|\mathcal{C}_j|$ is an unweighted average. The PS then computes the global model as:

$$w_{\text{global}}^{\text{INA}} = \frac{1}{M} \sum_{j=1}^{M} w_{\text{gNB}_j}^{\text{INA}}. \tag{26}$$

Compared to hierarchical FL, our approach reduces the volume of data at the very first step of the network, reducing path length and the saturation on the gNB-to-PS link. Our approach aggregates the parameters at line rate directly on the 5G stack (Section 2), thereby minimizing gNB processing delays and accelerating convergence, which improves time-to-accuracy.

Memory Overhead at the gNB. During each round, the gNB must maintain sufficient memory to store the model being aggregated, as implied by Equation 25. Fragments are stored temporarily until all contributions are combined or until their aggregation deadline expires. To optimize memory usage, the region assigned to a fragment can be recycled once the fragment has been forwarded to the PS and acknowledged for delivery. This reuse is possible because expired fragments are safely discarded by the PCC mechanism (Section 2), ensuring accuracy is not compromised. Once the global model is returned by the PS and distributed to all UEs, buffers are released for the next training round.

1.4. FLAG In-Network Aggregation Algorithm. If model updates are aggregated at programmable network nodes that operate at line rate and adapt to client heterogeneity and link variability, federated training over wireless networks can achieve near-synchronous convergence without modifying client or server software. This subsection describes how FLAG implements this principle through its programmable aggregation algorithm, combining timer-driven grouping, bias-corrected partial updates, and mobility-aware coordination within the existing 5G protocol stack.

The training process in FLAG involves no modifications on the client side and follows the standard FL workflow from the PS's perspective: UEs train locally and send updates,

while the PS aggregates and redistributes the global model. At the gNB, INA transparently aggregates these updates in real time and forwards a single result to the PS. The difference lies at the gNB, which transparently intercepts and aggregates updates in real time. All operations require no modifications to the 3GPP control plane, which makes FLAG deployable on existing 5G networks without altering client or PS software.

Workflow. At each iteration, UEs compute local updates on their datasets and forward the resulting model weights to the serving gNB. Each gNB maintains a temporary aggregation buffer A large enough to hold the model parameters (see Section 1.3). Updates from connected clients are accumulated fragment by fragment: when the gNB has collected all contributions for a given fragment, it forwards the averaged result to the PS. The PS waits for the aggregated updates from all gNBs, computes the global model, and broadcasts it to the network. Each gNB then relays the update to its connected UEs, synchronizing them for the next training round. This process reduces uplink bandwidth consumption between gNBs and the PS while remaining transparent to UEs, which continue to operate as in conventional FL.

Algorithm Structure. Algorithm 4 formalizes this process. Each client performs a local update (Lines 1–5) and transmits the resulting model to its serving gNB in multiple parameter fragments. Let $f_i^{(m)}$ denote the m^{th} fragment of the model update transmitted by client c_i. Upon reception, the gNB aggregates incoming fragments in parallel (Lines 5–19). For each fragment index m, the aggregation buffer $A^{(m)}$ accumulates the weighted client contributions $f_i^{(m)}$, while the counter N_m tracks the number of received fragments. When all expected contributions are collected for a given fragment, or when a timer expires as described in Section 2, the gNB computes the average of the collected fragments and forwards the resulting partial aggregate to the PS. The PS then performs a global aggregation across all gNBs (Lines 20–23), followed by a broadcast of the updated global model to all clients (Lines 24–28). This design guarantees that each update is processed exactly once, even under client mobility, through the handover synchronization protocol discussed in Section 1.4.

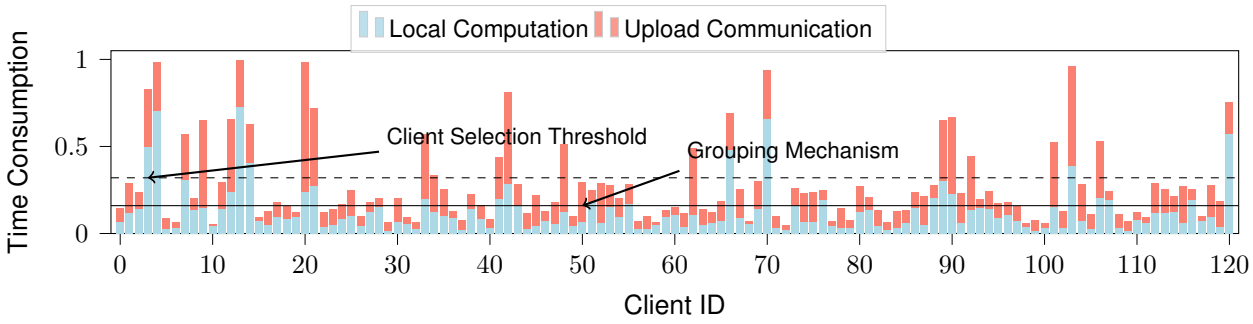


FIGURE 3. FL clients differ in compute and bandwidth as transfer times fluctuate. FLAG limits synchronization overhead and accelerates training through grouping and smarter client selection mechanisms.

Timer mechanism. To prevent straggling UEs from delaying training, we introduce a timer-based mechanism that defines a timestamp and a corresponding deadline for fragment reception at each gNB. Upon receiving the first model segment $f_i^{(m)}$ from any UE c_i in the current round, the gNB records the timestamp and starts a countdown to collect all corresponding fragments $f_i^{(m)}$ for the same segment m from clients $c_i \in \mathcal{C}_j$, where $\mathcal{C}_j$ is the set of UEs associated with gNB j. This approach ensures that the training pipeline is not stalled by the slowest UE. If the deadline expires before all expected fragments are received, the system invokes a recovery mechanism, as discussed in Section 2.

Handover management. User mobility introduces additional challenges when a UE switches gNBs mid-round. The Handover Signaler coordinates this transition using the UE-ID to transfer metadata between gNBs. Fragments already received are aggregated at the source gNB and forwarded to the PS, while remaining fragments are expected at the target gNB. A dedicated handover packet informs the new gNB of the UE's state, preventing duplication or omission of contributions. This mechanism preserves timeliness and avoids fragment forwarding between gNBs.

2. Communication-Aware Programmable FL

Our design goal is guided by the intuition that efficient federated training over wireless networks can be achieved by performing model aggregation at line rate within programmable network nodes, provided that the aggregation logic adapts to client heterogeneity and communication variability. In this section, we demonstrate this principle

through the design of FLAG's programmable aggregation layer, which enables real-time coordination between communication and learning processes via timer-driven grouping and bias-corrected partial updates. In particular, clients participating in FL are highly heterogeneous, not only in their computational speed but also in the quality of their wireless links. Such heterogeneity means that even after applying selection strategies [**24**, **25**], a difference remains in both training and communication times. Fig. 3 illustrates this effect: the blue segments represent training duration, while the red segments show bandwidth-related transfer delays. Unlike training time, which is mainly determined by the client's hardware, transfer time is influenced by wireless channel dynamics and can fluctuate unpredictably. As a consequence, even clients that initially appear well-suited for participation may experience adverse bandwidth conditions, which can hold back the aggregation process at the gNB.

Without mitigation, this leads to straggler effects, where a few slow clients slow down training times and job completion times. To minimize such waiting times, FLAG integrates a timer-based mechanism into the INA process. Upon receiving the first fragment of a round, each gNB starts a timeout during which it expects the remaining fragments from its connected clients. Fragments that arrive before the timeout are aggregated, while those missing or excessively delayed are discarded and handled through the PCC mechanism, which we describe next.

Such as design guarantees that aggregation proceeds at predictable intervals regardless of the slowest participants, striking a balance between fairness and system throughput. This combination of fragment-based transmission and timers enables the gNB and PS to operate at line rate, processing updates as they arrive rather than waiting for full models. By processing fragments independently and in parallel, the system reduces end-to-end latency and enhances throughput, thereby accelerating distributed training convergence and improving robustness in volatile wireless conditions.

Partial-Contribution Correction (PCC). Simply discarding the entire aggregate at the gNB whenever a given timer expires or after a UE disconnection is an inefficient solution. This approach would cancel the benefits of INA, as the contributions of all other

UEs that successfully transmitted their fragments would also be lost. The most straightforward way to precisely "de-aggregate" the contribution of a disconnected UE would be to store all the individual fragments received from each UE at the gNB until the aggregation is complete and transmitted. However, this approach quickly runs into the problem of memory limitations. The number of UEs that can simultaneously connect to a single gNB can be substantial, and each UE might transmit a large number of fragments depending on the application. Storing all these fragments, even temporarily, would require a reserved memory area, dependent on the model, at the gNB, potentially exceeding its available resources and compromising its ability to perform other essential functions, such as managing connections and handling mobility. This memory constraint motivates the need for a memory-efficient approximation to handle partial updates.

We propose PCC, a statistical bias correction mechanism designed to make the aggregated model update at the gNB representative of all clients, even when some clients fail to upload all their fragments before the deadline. In particular, PCC compensates for this missing-data bias without requiring the gNB to store per-client fragments (which would be too memory-intensive). Instead, it rescales the aggregated sum $A^{(m)} = \sum_{i \in S_m} |D_i| w_i^{(m)}(k)$ to approximate the total contribution as if all clients had participated. The scaling uses the ratio between the number of clients sending all fragments ($|S|$) and those sending the specific fragment m ($N_m = |S_m|$) as follows:

$$A^{(m)}_{\text{adjusted}} = A^{(m)} \times \frac{|S|}{N_m}. \tag{27}$$

The final update for fragment m sent by gNB j is hence:

$$w_j^{(m)}(k) = \frac{A^{(m)}_{\text{adjusted}}}{\sum_{i \in S} |D_i|}. \tag{28}$$

This mechanism ensures robustness and supports line rate aggregation.

2.1. PCC Does Not Slow Down Convergence. A key question is whether the Partial-Contribution Correction (PCC) mechanism, while enabling line-rate aggregation, affects

the convergence of the global model $w(k)$ towards a stationary point of the global objective $F(w)$. Since PCC introduces a small deterministic bias due to approximated client contributions, we analyze whether this bias alters the theoretical convergence guarantees of FLAG. To do so, we need the following assumptions.

Assumption 1: $Lipschitz$-Smoothness. The global loss function $F(w)$ is L-smooth, i.e., for some constant $L > 0$:

$$F(w') \leq F(w) + \langle \nabla F(w), w' - w \rangle + \frac{L}{2}\|w' - w\|^2, \quad \forall\, w, w'.$$

This assumption means that the global objective does not change too abruptly; its gradient is Lipschitz continuous with constant L. Intuitively, L quantifies how "curved" is the loss function. In practice, this ensures that the model's update steps are stable: taking a gradient step does not overshoot the minimum, and the descent of $F(w)$ can be bounded in terms of the step size and gradient magnitude.

Assumption 2: Bounded Variance. The variance of the aggregated update $g(k)$ (sum of all $w_j^{(m)}(k)$ from gNBs at the PS) is bounded, i.e., $\mathbb{E}[\|g(k) - \nabla F(w(k))\|^2] \leq \sigma^2$. For brevity, we simplify the second moment bound as $\mathbb{E}[\|g(k)\|^2] \leq \|\nabla F(w(k))\|^2 + \sigma^2$, acknowledging this common simplification.

This assumption captures the inherent randomness of the stochastic and federated updates: clients train on different local datasets, and wireless transmission noise or asynchronous arrivals can introduce additional fluctuations. Bounding this variance guarantees that local and communication noise do not dominate the descent direction over time.

Assumption 3: Bounded Bias. The PCC mechanism introduces a bias. We assume that the expected global update $g(k)$ relates to the true gradient via a bounded bias vector ϵ: $\mathbb{E}[g(k)] = \nabla F(w(k)) + \epsilon$, where $\|\epsilon\|$ is bounded, reflecting the systematic deviation caused by averaging over potentially different sets S_m and S. This bias term quantifies the mismatch between the "ideal" gradient (with all clients participating) and the approximated one obtained under real network dynamics with missing or delayed fragments.

The bounding constant guarantees that this deviation remains small, so that PCC affects only the final accuracy floor rather than the convergence rate.

The global model updates as $w(k+1) = w(k) - \eta g(k)$. Applying L-smoothness (Assumption 1):

$$F(w(k+1)) \leq F(w(k)) - \eta\langle \nabla F(w(k)), g(k)\rangle + \frac{L\eta^2}{2}\|g(k)\|^2. \tag{29}$$

Taking the total expectation and substituting Assumptions 2 and 3 we obtain:

$$\begin{aligned}
\mathbb{E}[F(w(k+1))] &\leq F(w(k)) - \eta\langle \nabla F(w(k)), \mathbb{E}[g(k)]\rangle \\
&\quad + \frac{L\eta^2}{2}\mathbb{E}[\|g(k)\|^2] \\
&\leq F(w(k)) - \eta\langle \nabla F(w(k)), \nabla F(w(k)) + \epsilon\rangle \\
&\quad + \frac{L\eta^2}{2}(\|\nabla F(w(k))\|^2 + \sigma^2) = F(w(k)) \\
&\quad - \eta\|\nabla F(w(k))\|^2 - \eta\langle \nabla F(w(k)), \epsilon\rangle \\
&\quad + \frac{L\eta^2}{2}\|\nabla F(w(k))\|^2 + \frac{L\eta^2\sigma^2}{2}.
\end{aligned} \tag{30}$$

Bounding the bias term using Young's inequality

$$-\eta\langle \nabla F(w(k)), \epsilon\rangle \leq \eta\|\nabla F(w(k))\|\|\epsilon\|$$

and combining the terms we obtain:

$$\begin{aligned}
\mathbb{E}[F(w(k+1))] &\leq F(w(k)) - \frac{\eta(1 - L\eta/2)}{2}\|\nabla F(w(k))\|^2 \\
&\quad + \frac{\eta\|\epsilon\|^2}{2(1 - L\eta/2)} + \frac{L\eta^2\sigma^2}{2}.
\end{aligned} \tag{31}$$

Summing Eq. (31) from $k = 0$ to $T - 1$, dividing by T, rearranging, and taking $T \to \infty$ (assuming $\eta < 1/L$ for $1 - L\eta/2 > 1/2$), we get the average squared gradient norm

bound:

$$\liminf_{T\to\infty} \frac{1}{T}\sum_{k=0}^{T-1} \mathbb{E}[\|\nabla F(w(k))\|^2] \leq \underbrace{\frac{\eta\|\epsilon\|^2}{(1-L\eta/2)^2}}_{\text{Bias term}} + \underbrace{\frac{L\eta\sigma^2}{1-L\eta/2}}_{\text{Variance term}}. \tag{32}$$

This analysis shows that the algorithm converges to a neighborhood of a stationary point, characterized by Eq. (32). The size of this neighborhood depends on the variance σ^2 and the bias $\|\epsilon\|^2$ introduced by the approximate PCC mechanism. Smaller bias and variance, along with an appropriately chosen step-size η, lead to convergence closer to the optimal solution. The 'neighborhood' represents the maximum accuracy penalty the system pays for operating in a volatile wireless environment. By using the PCC mechanism to correct for missing client fragments on the fly, the system avoids stalling for slow devices, ensuring fast, line-rate aggregation. This theoretical bound guarantees that despite dropped packets and statistical approximations, FLAG effectively balances the trade-off between strict model precision and real-world network speed, allowing the global model to successfully stabilize. In our evaluation (Section 4), we observe that the empirical bias term $\|\epsilon\|$ remains below 10^{-3} even under high dropout rates ($> 20\%$), confirming that PCC preserves convergence accuracy while significantly reducing communication overhead.

3. Packet Loss Mitigation Mechanisms

While PCC compensates for missing fragments, it cannot recover updates that were never received due to persistent losses or straggling clients. These incomplete transmissions still reduce the statistical diversity of updates and can bias the aggregated model over time. Therefore, our goal is to reduce avoidable losses in the aggregation path, so that the remaining variability primarily stems from inherent wireless channel dynamics rather than from coordination artifacts. To achieve this, our solution combines two complementary mechanisms: (i) *Deadline-Driven Grouping (DDG)* in the gNB, which adaptively clusters clients based on latency and channel quality, and (ii) *Staleness-Aware*

Algorithm 5 PS Staleness-Aware Aggregation

Require: Current round t; global model w^t; decay $\lambda > 0$; step size $\alpha \in (0, 1]$.
Require: $\mathcal{A} = \{(\overline{w}^{v,h}, N_{v,h})\}$ with $v \leq t$ (round id), h (group label), $N_{v,h} \in \mathbb{N}$ (sample count).

for all $(\overline{w}^{v,h}, N_{v,h}) \in \mathcal{A}$ **do**
  $\delta_{v,h} \leftarrow t - v$ ▷ Staleness in rounds (age)
  $s_{v,h} \leftarrow \max\{0, \cos(\mathrm{vec}(w^t), \mathrm{vec}(\overline{w}^{v,h}))\}$ ▷ Cosine similarity (clipped at 0) measures alignment between incoming and current models;
  $a_{v,h} \leftarrow N_{v,h}\, s_{v,h}\, e^{-\lambda \delta_{v,h}}$ ▷ Weight combines statistical importance (data size), semantic relevance (similarity), and temporal freshness (exponential decay);
end for
$A \leftarrow \sum_{(\overline{w}^{v,h}, N_{v,h}) \in \mathcal{A}} a_{v,h}$ ▷ Total weight for normalization
$\widetilde{w} \leftarrow \dfrac{\sum_{(\overline{w}^{v,h}, N_{v,h}) \in \mathcal{A}} a_{v,h}\, \overline{w}^{v,h}}{A}$ ▷ Weighted blend
$w^{t+1} \leftarrow (1 - \alpha)\, w^t + \alpha\, \widetilde{w}$ ▷ Convex update
return w^{t+1} ▷ Broadcast

Aggregation (SAA) in the PS, which corrects for delayed or postponed updates using age-dependent weighting. Together, these mechanisms limit the need for PCC corrections and improve both fairness and convergence stability.

3.1. Deadline-Driven Grouping. FLAG's timer mechanism prevents indefinite delays by setting a global time budget. However, enforcing a strict deadline for all clients may benefit only faster clients with good connections while causing many updates from slower clients to be dropped. To address this problem, we introduce a dynamic, deadline-driven grouping mechanism.

Grouping Process of gNB. The gNB coordinates the training round, which is uniquely identified by the round id t (also used as the model version tag). The process begins when the gNB invites clients to participate in round t by broadcasting the global model w^t and a deadline. Clients respond with a `JOIN_ROUND` message if they estimate they can meet this deadline (which may be governed by either a target number of ready clients or a time threshold). The gNB forms a fast group as soon as a predefined threshold of K_{fast} clients have joined. It immediately issues a `START` signal to this group and opens a dedicated aggregation buffer for their updates. The remaining, slower clients are designated as the slow group. The gNB waits until this second group is ready and then issues

a separate START signal to them. The gNB maintains a distinct, non-overlapping aggregation for this slow group. Their updates are aggregated independently and forwarded to the PS as a second, potentially more stale update for the same round t. This multi-group mechanism ensures that the fast clients' contributions are never delayed, while the updates from slower clients are still collected and handled without interfering with the primary aggregation process.

PS Staleness Mitigation. However, this flexibility introduces the challenge of staleness: clients that defer a round continue training on an older global model $w^{t-\tau}$. To prevent these stale updates from destabilizing convergence, the PS employs a Staleness-Aware Aggregation (SAA) mechanism.

The intuition behind the DDG mechanism is that fast clients do not wait for slower ones, while the Staleness-Aware Aggregation algorithm (SSA) guarantees that any delayed updates arriving from slow groups are integrated proportionally to their relevance and freshness. As shown in Algorithm 5 (and consistent with the grouping flow in Algorithm 4), the PS operates at current round t with current model w^t, a staleness–decay rate $\lambda > 0$, and an update step size $\alpha \in (0, 1]$. The PS receives a collection $\mathcal{A}$ of per group aggregates $\overline{w}^{v,h}$ where $v \leq t$ is the round tag carried by the aggregate, $h \in \{\mathcal{F}, \mathcal{S}\}$ is the group label (fast/slow as formed at the gNB), and $N_{v,h} \in \mathbb{N}$ is the total sample count that contributed to that per group aggregate at the gNB. Staleness for each received aggregated model from each gNB is measured as $\delta_{v,h} = t - v$ (representing τ), and the PS evaluates a nonnegative similarity score $s_{v,h}$ between $\overline{w}^{v,h}$ and w^t. The PS then forms a blending weight $a_{v,h}$ for each incoming aggregate that increases with data size $N_{v,h}$ and similarity $s_{v,h}$ while decaying with age according to the function $f(\tau)$ (implemented as an exponential with rate λ). After collecting all items, the PS computes the normalizer $A = \sum a_{v,h}$ and constructs a blended target $\widetilde{w}$ by normalizing the weighted contributions; if no usable signal is present (i.e., $A = 0$), the PS safely falls back to $\widetilde{w} = w^t$. Finally, the global model is updated by a conservative convex step toward $\widetilde{w}$ using α, yielding w^{t+1}, which is then broadcast to all gNBs.

The PS stores tagged aggregates $(\overline{w}^{v,h}, N_{v,h})$, interprets $\tau{=}t{-}v$ as round age, uses similarity and decay to weight them ($a_{v,h}$, A), blends into $\widetilde{w}$, and updates w^{t+1} thereby incorporating deferred or slower contributions without sacrificing stability.

Deadline-Driven Grouping (DDG) and Staleness-Aware Aggregation (SAA) form a coordinated pipeline: DDG minimizes waiting time at the gNB by launching fast and slow groups independently, while SAA ensures that delayed aggregates still contribute proportionally to their quality and recency. Together, they reduce communication stalls without discarding useful information, maintaining convergence speed and stability even under fluctuating network conditions.

Section 4.3 quantifies this trade-off, showing that FLAG achieves faster convergence with only a minor accuracy reduction, which can be further tuned via the decay rate λ and update step α.

3.2. Dynamic Local Deadline. Given that the grouping mechanism may not be tolerated in some applications due to the staleness of updates, we propose, as an alternative to our reactive grouping mechanism, a proactive synchronization technique inspired by adaptive training [**24**]. The goal is to reduce the idle time of faster clients by converting it into productive computation, thereby homogenizing the completion time of the round. The process is coordinated by the gNB, which sets a local deadline (T local) for the round. Instead of finishing early and waiting faster clients (i.e., those with the best hardware performance) use their expected spare time to perform additional local training epochs. This strategy ensures that all clients, both fast and slow, are productively engaged for roughly the same duration, causing their model updates to arrive at the gNB in a more tightly synchronized window, thereby speeding up the training process [**24**].

To formally define the dynamic adjustment of local training, we present the foundational model inspired by the adaptive mechanism in PyramidFL [**24**]. This model converts the idle time of fast clients into additional computation. For a client i connected to gNB j, the number of local iterations $I_{i,j}$ is calculated as:

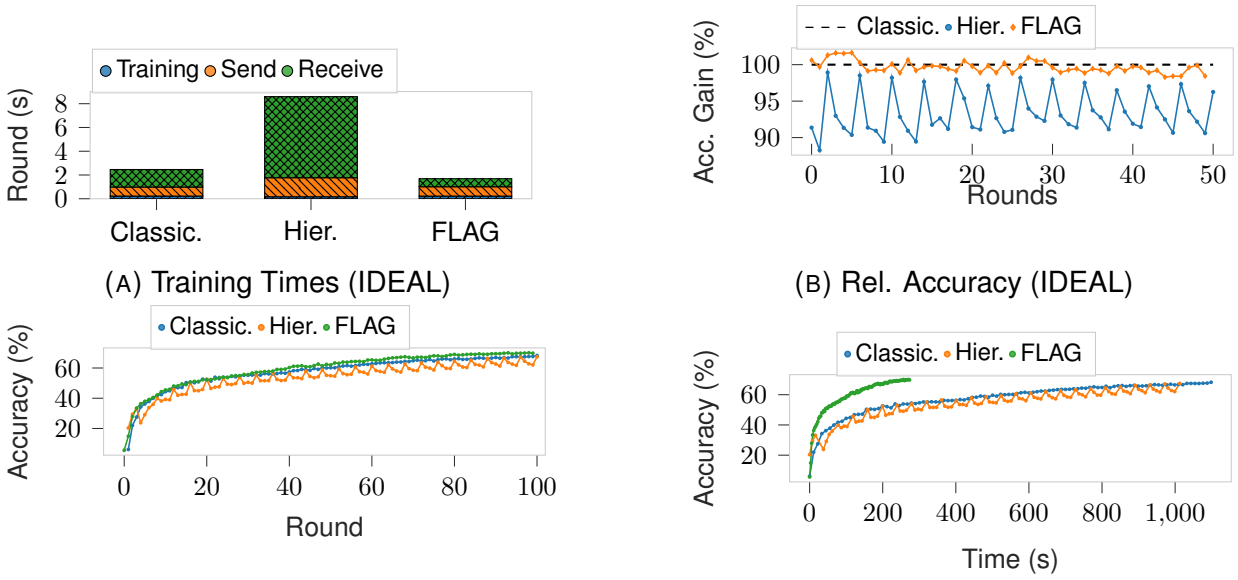
$$I_{i,j} = \left(\beta \times \frac{\max(T_{\text{local},j} - t_{i,j}, 0)}{t_{i,j}^{\text{comp}}} + 1 \right) \times I_{\text{fix}} \tag{33}$$

In this formulation, $I_{i,j}$ is the adaptive number of local iterations, calculated based on the dynamic deadline $T_{\text{local},j}$ set by gNB j, the client's estimated baseline round time $t_{i,j}$ (with computation portion $t_{i,j}^{\text{comp}}$), a fixed baseline number of iterations I_{fix}, and a confidence factor β that moderates the adjustment to account for estimation errors. While this linear model provides a straightforward way to utilize spare time, it does not account for the principle of diminishing returns inherent in local model training. Excessive training epochs on a client's limited local dataset can lead to overfitting, causing the local model to diverge from the global objective and contribute less effectively to the global model.

Hence, we replace the linear term with a logarithmic function, which naturally captures the diminishing returns of extended training. This ensures that clients with a small amount of spare time receive a significant boost, while the fastest clients are prevented from performing an excessive number of epochs.

$$I_{i,j} = \left(\beta \times \log \left(1 + \frac{\max(T_{\text{local},j} - t_{i,j}, 0)}{t_{i,j}^{\text{comp}}} \right) + 1 \right) \times I_{\text{fix}} \tag{34}$$

In both formulations, the $\max(..., 0)$ function calculates the usable spare time, ensuring that only fast clients ($t_{i,j} < T_{\text{local},j}$) perform additional work, while slow clients default to the baseline. The key parameters are the dynamic deadline $T_{\text{local},j}$ set by the gNB, the estimated round time $t_{i,j}$ for the client, its computation portion $t_{i,j}^{\text{comp}}$, and the baseline number of iterations I_{fix}. The confidence factor β moderates the use of spare time to account for potential estimation errors [**24**]. This logarithmic model offers a more robust and realistic approach, striking a balance between utilizing idle time and maintaining the generalizability of the global model. In the evaluation section, we show that FLAG is analogous to the client selection strategy, offering flexibility tailored to the required trade-offs.



(A) Training Times (IDEAL)

(B) Rel. Accuracy (IDEAL)

(C) Round Acc. (Bottleneck)

(D) TTA (Bottleneck)

FIGURE 4. *Ideal vs. bottlenecked FL conditions.* **(HAR vs. MLP) (a)–(b)** show ideal settings: FLAG reduces round time by minimizing send and receive overhead and maintains accuracy close to Classical. **(c)–(d)** reflect backhaul bottlenecks: while the accuracy is similar over rounds, FLAG achieves faster convergence than Classical and Hierarchical.

4. Evaluation

This section evaluates the performance of FLAG by examining how its design addresses key bottlenecks in federated training over wireless networks. Our goal is to assess how design elements such as INA, deadline-based client grouping, and PCC impact the efficiency and robustness of FL in 5G environments. We develop a 5G emulator modeling UE and gNB behavior at the user plane. It implements SDAP, PDCP, and GTP-U layers, with uplink fragments tagged by a custom SDAP header carrying FLAG metadata. Protocol layers enforce MTU and queueing constraints, while a two-state Markov model injects fragment-level losses to capture wireless variability. This provides a realistic yet flexible platform for evaluating FLAG under diverse conditions. We assess FLAG on FedScale [**105**] using two ML tasks: (i) HAR [**106**] with an MLP and (ii) FEMNIST [**107**] with ShuffleNet. To reflect the non-i.i.d. nature of user data in real deployments, FEMNIST is partitioned by writer, yielding skewed class distributions, while HAR

is partitioned by subject, ensuring user-specific activity traces. Unless otherwise specified, experiments use 6 gNBs and 60 UEs evenly distributed across them. Each UE uplink is fixed at 100 Mbps, while the gNB–PS backhaul is set to either 10 Mbps (bottleneck) or 1000 Mbps (baseline). We adopt FedProx to address data heterogeneity, though FLAG remains agnostic to the aggregation algorithm and supports integration with any optimizer or client selection scheme. To capture realistic edge dynamics, each UE periodically updates its compute throughput and uplink bandwidth using traces from AIBench [**108**] (mobile processors with $\geq$2GB RAM) and MobiPerf [**109**] (smartphone uplink statistics). Finally, we compare FLAG against standard FL, which performs best under ideal bandwidth, and HFL [**45**], a hierarchical architecture designed to reduce PS load.

4.1. Homogeneous FL. We begin the evaluation by showing how even in ideal conditions with homogeneous clients, performance degrades under backhaul bottlenecks.

In the HAR task with an MLP (Figure 4), we compare an ideal, unconstrained network against one where the backhaul introduces congestion. Figures 4a and 4b break down per-round time into training, sending, and receiving phases. While training and sending costs are similar across systems, receiving time dominates in Hierarchical FL because multiple local updates occur before global aggregation. By contrast, FLAG achieves the best performance since the backhaul only carries per-gNB flows, reducing bandwidth usage. Figure 4b further shows that while Classical FL maintains stable accuracy, Hierarchical FL exhibits fluctuations, whereas FLAG sustains competitive accuracy with improved efficiency. Under constrained backhaul (Figures 4c and 4d), FLAG reduces training time, achieving up to a $5.1\times$ speedup over Classical FL, which performs worst. We next turn to a more demanding setting: FEMNIST with ShuffleNet (Figure 5), again assuming uniform compute and bandwidth across UEs. Under ideal conditions (Figures 5a and 5b), FLAG and Classical FL show nearly identical round durations (11.8s vs. 12.0s), while Hierarchical FL exceeds 30s due to extra aggregation steps. When bandwidth is limited (Figure 5d), the classical FL round times exceed 50

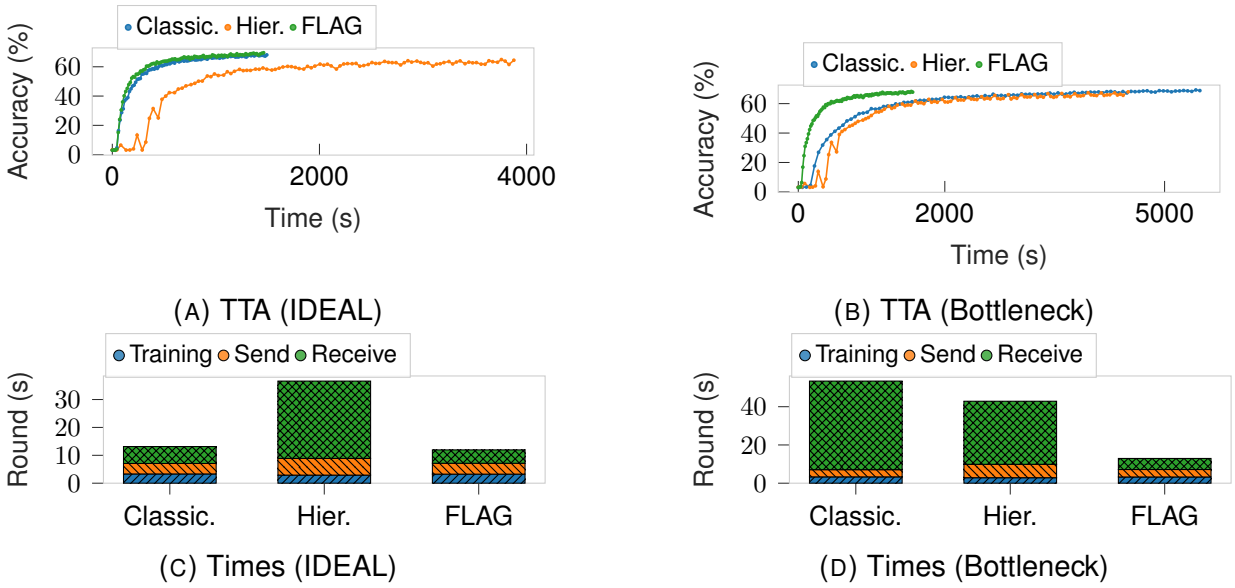
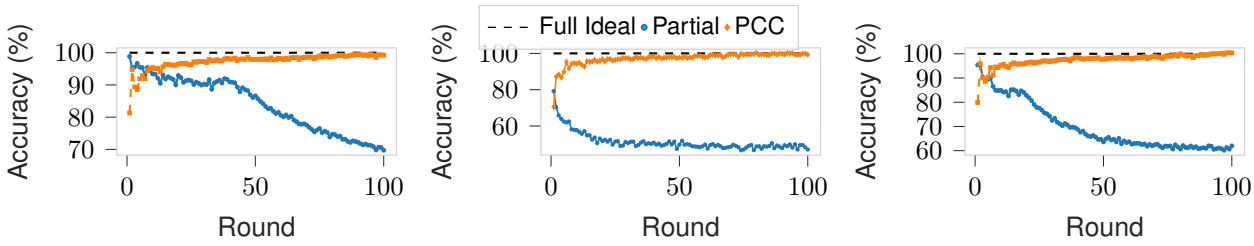


FIGURE 5. *Ideal vs. bottlenecked FL conditions. (Homogeneous)* **(FEMNIST vs. Shufflenet) (a)–(c)** show ideal settings: FLAG reduces the average training rounds by minimizing send and receive overhead and maintains accuracy close to Classical. **(b)–(d)** reflect backhaul bottlenecks: FLAG achieves faster convergence than Classical and Hierarchical with an improvement over time up to $5.1\times$.

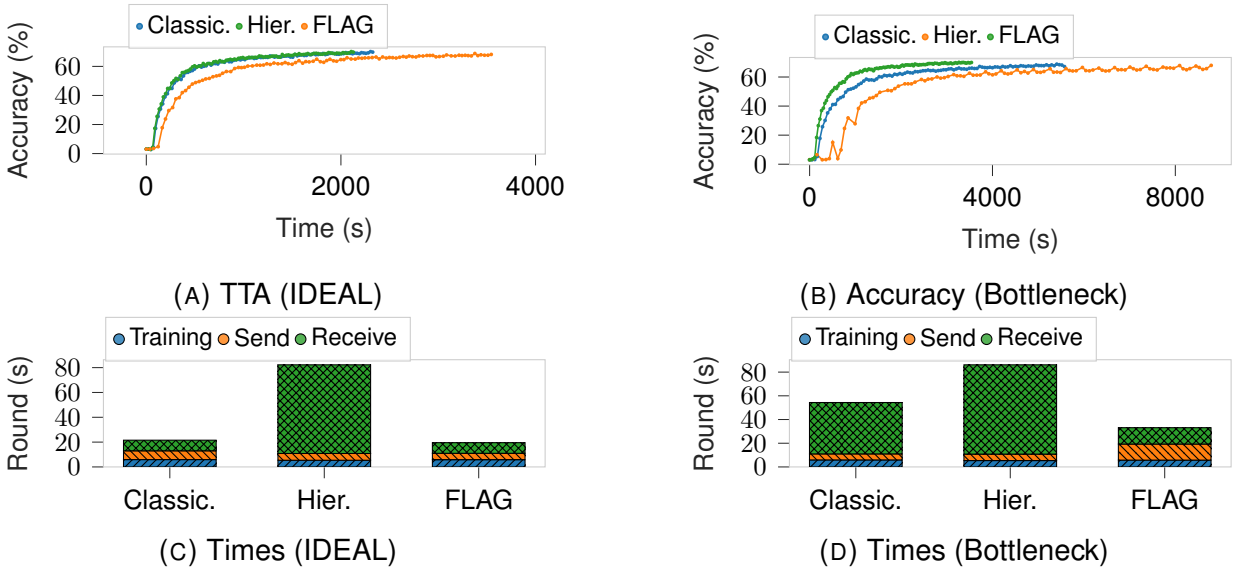
seconds as the send and receive phases oscillate to nearly 20 seconds each. Hierarchical FL mitigates some of this cost but still suffers from long training times. In contrast, FLAG maintains stable send/receive phases of 6s each, for a predictable round duration of 24s. Overall, while Hierarchical FL alleviates part of the communication burden, FLAG outperforms both alternatives in bandwidth-limited settings, delivering up to a $5.1\times$ faster time-to-accuracy through gNB-side aggregation and a streamlined processing pipeline.

4.2. Heterogeneous FL. Real-world FL involves heterogeneous devices with varying data volumes, computational resources, and network stability. To capture this, we simulate FEMNIST training with ShuffleNet while modeling disparities in UE compute and bandwidth. Figure 7 reports time-to-accuracy (TTA) and round-time breakdowns under both ideal and constrained backhaul. In the ideal case (Figure 7c), FLAG and Classical FL perform similarly, with nearly identical send and receive times (e.g., FLAG:



(A) Low Loss (B) High Loss (C) Medium Loss

FIGURE 6. *Flag PCC impact*. PCC is evaluated under three different wireless loss scenarios. While Partial Aggregation struggles to maintain accuracy even in the low-loss scenario (Figure 6a), PCC stabilizes relative accuracy in both the medium and high loss scenarios (Figure 6c, and Figure 6b), maintaining high performances.

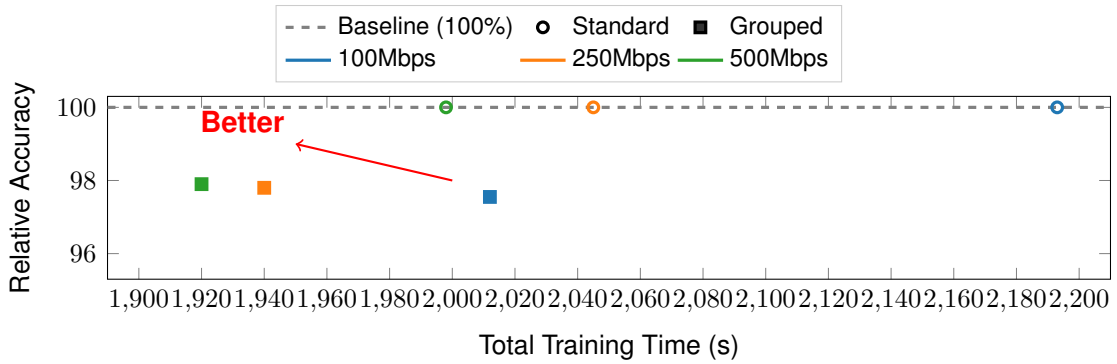


(A) TTA (IDEAL) (B) Accuracy (Bottleneck)

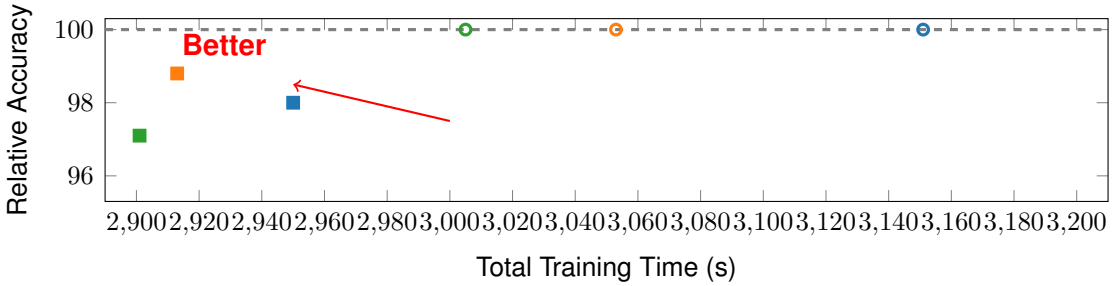
(C) Times (IDEAL) (D) Times (Bottleneck)

FIGURE 7. *Ideal vs. bottlenecked FL conditions (Heterogeneous)* **(FEMNIST + ShuffleNet) (a)–(c)** show ideal settings: FLAG reduces the average training round time by minimizing send and receive overhead, maintaining accuracy comparable to Classical. **(b)–(d)** reflect backhaul bottlenecks: FLAG achieves faster convergence than Classical and Hierarchical FL.

13.3s/13.1s; Classical: 13.5s/13.5s), showing that both systems approach optimal throughput without bottlenecks. Under bandwidth constraints (Figure 7d), however, the gap widens sharply. Classical FL suffers from large communication overheads, with send and receive phases rising to 25.1s and 23.9s. By contrast, FLAG reduces these to 12.2s and 15.4s through in-network aggregation, cutting uplink traffic per round. Consequently,



(A) Accuracy 50 UE.



(B) Accuracy 100 UE.

FIGURE 8. *Grouping time improvements.* Relative accuracy degrades using the grouping mechanism while delivering better training times.

FLAG achieves faster round completion under identical compute heterogeneity and delivers shorter TTA in the bottlenecked setting (Figure 7b).

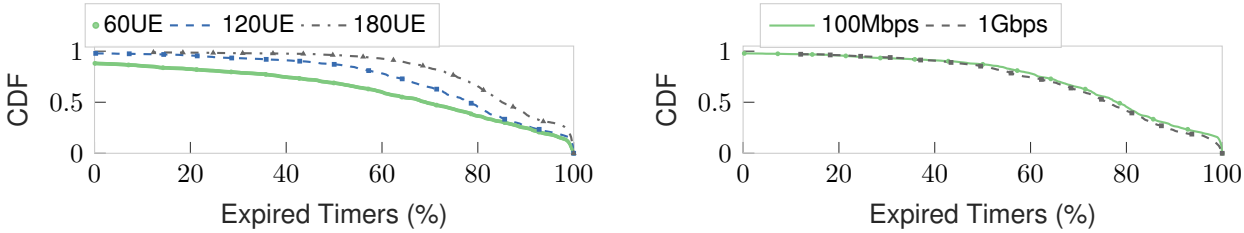
4.3. The Impact of Deadline-Driven Grouping. We next evaluate the performance of our deadline-driven grouping mechanism, focusing on the crucial trade-off between training speed and model accuracy. Asynchronous approaches, such as grouping, are known to accelerate training by mitigating stragglers; however, they also introduce the risk of model divergence due to stale updates from deferred clients. This experiment quantifies this trade-off by comparing FLAG with Grouping against a standard synchronous FL baseline. Figure 8 presents the time-to-accuracy for runs with 50 and 100 UEs. We use the FEMNIST dataset, which is trained on ShuffleNet, and the backhaul connection is set to 100, 250, and 500 Mbps. The results clearly show a reduction in training time for the grouping mechanism. By allowing the PS to process updates from the

fastest available group, FLAG achieves a target accuracy up to 200 seconds faster than the synchronous baseline, representing a speedup of nearly 10%. This acceleration, however, comes at the cost of model accuracxy. As the plots show, the “Grouped” approach maintains a final accuracy that is only 1-3% lower than the synchronous baseline. This accuracy preservation is a direct result of our SAA mechanism. While grouping inherently creates stale updates, our adaptive blending factor, α_{final}, mitigates their negative impact. By dynamically down-weighting stale contributions based on their age and similarity, SAA prevents the global model from diverging and preserves high accuracy. The results show that the FLAG grouping mechanism reduces time-to-accuracy, albeit with an accuracy trade-off. Overall, it enables faster convergence without substantially compromising performance, making it practical for real-world, heterogeneous environments. While some applications may accept this modest loss in accuracy to save time, others may prefer to remain delay-tolerant. We therefore present the results of our next mitigation, the Dynamic Local Deadline, as formulated in 3.2.

4.4. Client Selection Tradeoffs. While the benefits of aggregation were clarified in the previous analysis, we further explore a case study on client selection, highlighting how building custom client-selection strategies on top of FLAG yields significant improvements. Client grouping may not be feasible for some applications in which the speed–accuracy trade-off is acceptable; in such cases, it is preferable to avoid staleness entirely, maximize accuracy, and balance training time. We compare standard random client selection, PyramidFL [**24**] client selection, and our adaptation, PyramidFLAG, which aims to further reduce training time, leveraging the benefits of aggregation with a dynamic local deadline. We adopt the PyramidFL client-selection strategy and adjust the local epoch trainings as described in 34. We use the FEMNIST dataset trained on ShuffleNet and 6 gNB with 10 clients each. The backhaul connection is set to 1Gbps. Figure 10 shows that the benefits of PyramidFL translate into improvements in time-to-accuracy under highly non-IID settings. While in Figure 10a we highlight the benefits of PyramidFL over the classical FL training settings in terms of time to accuracy, we can see that the improvements are consistent also in our PyramidFLAG settings. We also

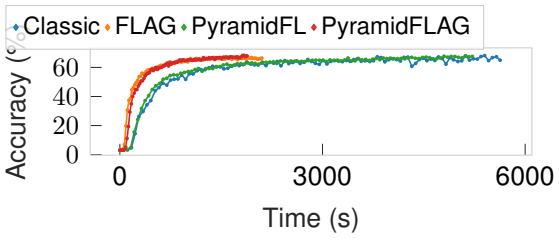
show in Figure 10b that the impact of Pyramid leads not only to faster convergence but also to accuracy improvements compared to the base case. This approach proves especially effective when adapted to our FLAG architecture, yielding additional reductions in time-to-accuracy. Overall, FLAG can be viewed as an agnostic architecture for in-network aggregation in FL, providing the flexibility to adopt client-selection policies that better accommodate task-specific needs.

4.5. Scalability Analysis. To assess how FLAG's timer-based aggregation scales with system load and backhaul capacity, we measure the fraction of fragments missing their deadlines under two dimensions: (a) number of UEs and (b) PS backhaul bandwidth. Figure 9 reports the mean complementary CDF of expired timers. As shown in Figure 9a, timer expirations become more likely as the number of UEs grows (30–90). For example, the probability of 20% of timers expiring increases from 0.8 at 30 UEs to 0.95 at 60 UEs and 0.98 at 90 UEs. This reflects the inherent variability introduced by larger UE populations, where slower devices or weaker channels raise the risk of stragglers. In contrast, Figure 9b highlights the limited impact of backhaul capacity on timer expirations. The gap between ideal and bottlenecked configurations remains bounded even at high expiration rates, suggesting that FLAG scales under constrained links. This robustness stems from the grouping mechanism, which counteracts backhaul limitations and preserves accuracy through the use of PCC.

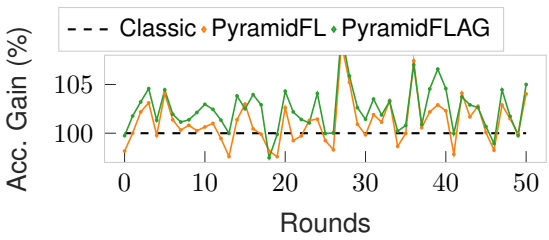


(A) Number of UEs (B) Backhaul bandwidth

FIGURE 9. *Timer expiration.* The average timer expiration rises as the number of UEs increases (Figure 9a), while it remains stable across varying backhaul capacities (Figure 9b). Accuracy is unaffected thanks to PCC.



(A) Time To Accuracy



(B) Rel. Accuracy (IDEAL)

FIGURE 10. FLAG is client selection agnostic; pairing it with ad-hoc mechanisms further improves the convergence. (a) shows the time to accuracy improvement. Under non-IID data and constrained backhaul, the benefits of FLAG are further enhanced, yielding faster time-to-accuracy; (b) highlights how the accuracy can benefit from client selection strategies.

CHAPTER 6: Client Selection via LLM for Efficient Federated Learning

In the previous chapter, we explored how to integrate severely resource-constrained devices at the far edge into the training process through Federated Split Learning (FSL). By leveraging computation offloading, FSL significantly reduces the hardware requirements for devices to participate in the federated network, effectively expanding the pool of eligible training nodes. While FSL successfully resolves the architectural challenge of *how* to train on weak edge devices, this massive expansion of the participant pool introduces a complementary orchestration challenge: deciding exactly *which* subset of these available clients to actively select for each training round to maximize global efficiency.

To understand the necessity of targeted client selection, one must examine the foundational constraints of the FL paradigm as established in the literature. In an ideal, unconstrained environment, a central parameter server would aggregate updates from all available clients in every round to maximize data diversity. However, in real-world cross-device deployments, this full-participation model is physically and computationally impossible due to massive concurrency limits and bandwidth costs [**110**]. Therefore, the system is forced to select a small fraction of clients per round, introducing two critical bottlenecks:

First, from a system perspective, edge networks are plagued by extreme hardware and connectivity variance. Traditional aggregation algorithms, such as FedAvg [**100**], operate on a synchronous barrier. Consequently, the duration of a training round is dictated by the slowest participating node—a phenomenon known as the *straggler effect*. As highlighted by Li et al. [**?**] and Tu et al. [**?**], if client selection is naive or purely random, these stragglers drastically inflate the time-to-accuracy, and dropped wireless connections can waste massive amounts of computational energy.

Second, from a statistical perspective, client data is profoundly non-IID (Independent and Identically Distributed). A naive selection policy that attempts to solve the straggler problem by simply prioritizing the fastest devices inherently introduces severe sampling bias. It risks over-representing specific user demographics while completely ignoring slower devices that might hold rare, highly valuable data features. As demonstrated by Zhao et al. [**?**], training on highly skewed, unrepresentative data distributions causes severe weight divergence, destabilizing the global model and preventing convergence entirely.

Therefore, intelligent client selection is not merely an optional optimization feature—it is a mandatory structural requirement for FL. As emphasized by recent utility-aware frameworks [**24**, **51**], the orchestrator must solve a complex exploration-exploitation problem, actively balancing the trade-off between system velocity (mitigating stragglers) and statistical utility (maximizing data diversity) to ensure the global model converges efficiently.

Resource scheduling and node selection are fundamental optimization problems across all distributed machine learning paradigms. Even in highly controlled environments like centralized data centers, intelligent task placement is required to prevent network bottlenecks and mitigate computational stragglers. However, when distributed training transitions to the edge via Federated Learning (FL), this selection problem becomes exponentially more difficult. Unlike dedicated computing clusters, FL relies on consumer edge devices characterized by extreme system and data heterogeneity. Fluctuating device compute speeds, volatile wireless network bandwidths, and profoundly non-independent and identically distributed (non-IID) local data mean that naive or random client selection will drastically degrade model convergence and significantly increase overall training time. Consequently, there is a critical need for intelligent, dynamic orchestration capable of continuously balancing the statistical utility of a client's local data against their physical system efficiency.

Current intelligent client selection strategies fall into two primary categories, yet both exhibit critical limitations that prevent optimal performance in real-world deployments.

First, static heuristics are structurally rigid. By relying on hard-coded mathematical proxies, they are inherently blind to the dynamic phase shifts of model convergence and cannot adapt to unforeseen network bottlenecks during the training lifecycle [**2**]. Second, while Reinforcement Learning (RL) methods aim to introduce adaptability, their decision-making remains tightly constrained by the specific reward heuristics used to train them, leading them to struggle when deployed in unseen or diverse conditions [**24**, **25**]. Furthermore, RL agents require massive, domain-specific offline datasets to converge and often fail to navigate the high-dimensional, highly non-stationary state spaces characteristic of volatile FL environments. Consequently, a fundamental gap in existing frameworks is the lack of a context-aware orchestrator capable of zero-shot reasoning. There is a critical need for a system capable of intuitively interpreting complex, multi-dimensional state vectors and dynamically adjusting selection strategies on the fly, without relying on static formulas or requiring extensive, environment-specific retraining.

Recently, the application of Large Language Models (LLMs) has expanded significantly beyond traditional natural language processing, demonstrating remarkable efficacy in solving complex, combinatorial decision-making problems within networked systems. Recent studies have successfully employed foundation models for tasks such as cluster job scheduling, adaptive bitrate streaming, and dynamic network routing [**?**,**?**,**56**], frequently matching or outperforming bespoke heuristic algorithms.

The intuitive justification for leveraging LLMs in federated client selection lies in their architectural strength as high-dimensional pattern recognizers. At its core, selecting the optimal subset of training nodes requires balancing a chaotic interplay of non-linear variables—such as fluctuating wireless bandwidths, heterogeneous compute capacities, and skewed data utility. Traditional orchestration frameworks force engineers to compress these complex, real-world realities into rigid mathematical proxies. LLMs, however, possess a deep, pre-trained capacity for contextual reasoning and structural dependency mapping. By representing the network's physical state as semantic context, the LLM can implicitly weigh conflicting trade-offs—for instance, dynamically recognizing when it is mathematically advantageous to tolerate a slower client because it possesses

highly valuable, rare data. This allows the system to navigate edge volatility without relying on handcrafted, fragile threshold equations.

To address these limitations, we propose FLLLM, an online, Large Language Model-driven client selector. By projecting raw system metrics into an LLM's semantic space, FLLLM leverages the model's capacity for sequence modeling and zero-shot generalization to natively handle the non-linear tradeoffs between compute latency and informational utility.

The architecture of the proposed solution is illustrated in Figure 1. At each communication round, the system monitors the available Clients Space to extract a rich set of features. These inputs are partitioned into a Global State (g_t), which captures macro-level environment variables such as the current round and accuracy targets, and individual Client States (s_t), which detail per-device telemetry. Because pre-trained LLMs cannot natively process raw numerical data, an Encoder projects these states into a continuous latent space. The core LLM processes these encoded embeddings and passes their representations to Action Heads, which generate selection scores. Simultaneously, the architecture evaluates an Oort baseline and extracts an Oort Policy heuristic to guide early training stability. A DPO Agent bridges these signals, evaluating the LLM's proposed selection against the baseline to finalize the subset of Round t Clients. Crucially, the agent calculates a DPO Loss based on the chosen utility function. This loss is fed back via a continuous learning loop to a LoRA module, enabling the LLM to dynamically fine-tune its selection strategy online.

Overall, FLLLM makes three primary contributions. First, it introduces preference learning via Direct Preference Optimization (DPO). This overcomes cold-start instability by anchoring the model's learning with a guided heuristic warmup, continuously refining the LLM policy online from historical buffers, and eliminating the need for an explicit, unstable reward model. Second, it uses an Accuracy-Focused Reward Oracle that employs a Pareto-aware utility metric, prioritizing improvements to gradient quality while treating completion time as a secondary efficiency objective. Third, it features a lightweight state encoding that uses an MLP to project heterogeneous, high-variance client

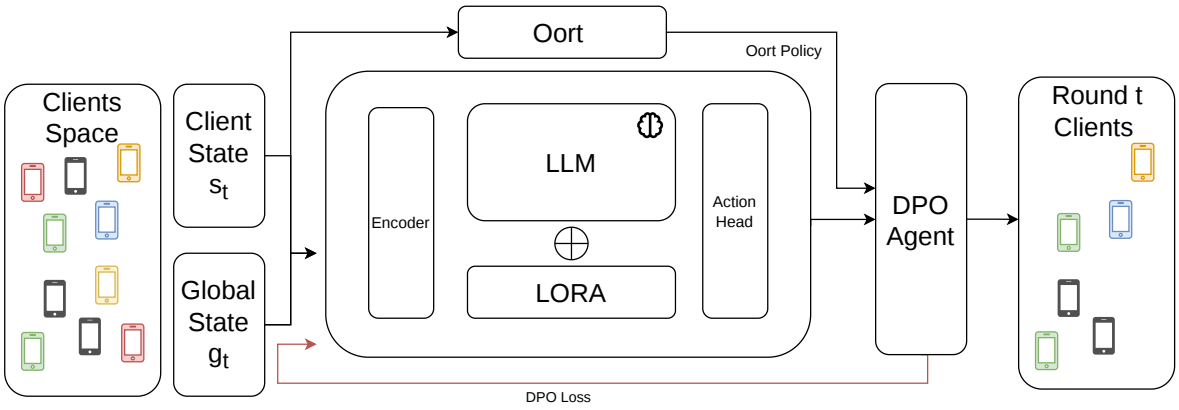


FIGURE 1. FLLLM Architecture. The system takes two primary inputs: the global state, g_t, which captures macroscopic environment variables such as the communication round and target accuracy, and the client state, s_t, which details individual client metrics. An encoder projects these inputs into a latent space that the LLM can process. The architecture employs the Oort [**25**] policy as a baseline for real-time training. A Direct Preference Optimization (DPO) [**111**] Agent evaluates the outputs from both the LLM's action head and the Oort policy, ultimately selecting the set of Round t Clients that maximizes a customizable utility function. The resulting selection generates a DPO Loss, which is fed back to a LoRA module. This feedback loop dynamically refines the LLM's client selection policy over successive rounds to improve upon the baseline.

telemetry into a compact token representation, enabling direct LLM conditioning without costly feature engineering. We proceed by describing the challenges in selecting current FL clients and how the proposed FLLLM framework bridges the gaps.

1. Federated Learning Dynamic Clients Selection Adaptability

The fundamental challenge in deploying Federated Learning across heterogeneous edge environments lies in balancing two often-conflicting objectives: system efficiency and statistical efficiency. This trade-off space can be visualized as a two-dimensional plot, as illustrated in Figure 2, where the axes represent the wall-clock time required per round (system overhead) and the number of rounds required to reach a target accuracy (statistical convergence). The optimal region of this space is the bottom-left quadrant, representing an ideal scenario in which the system achieves the maximum accuracy gain per round while simultaneously minimizing the time required to complete each

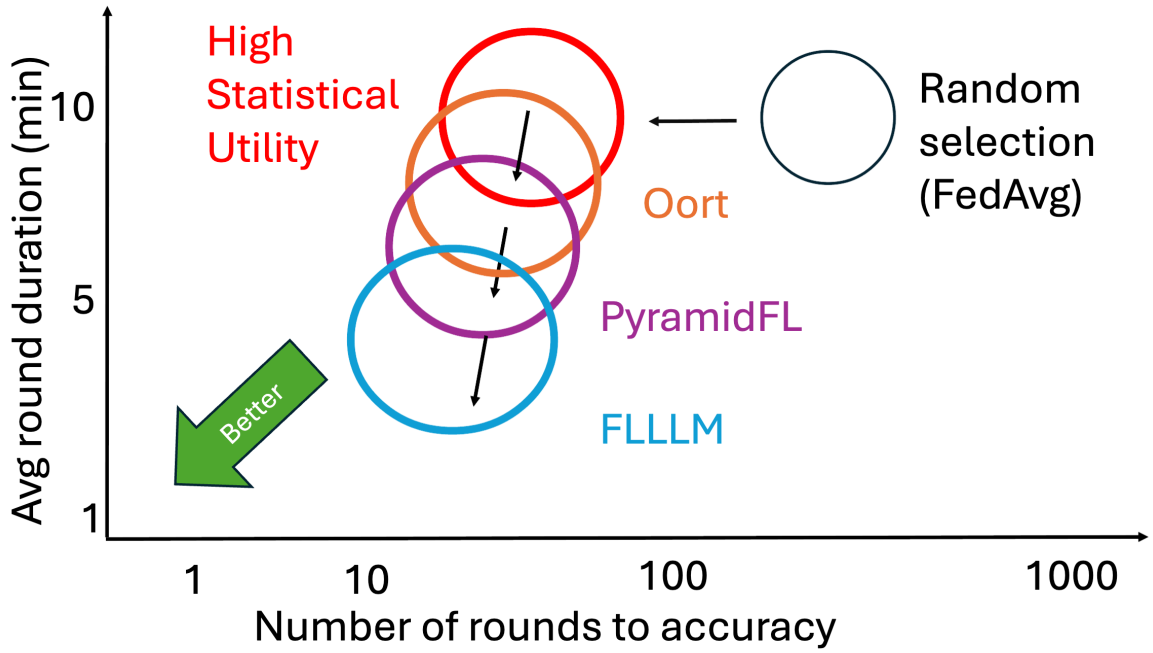


FIGURE 2. Trade-off between system efficiency (time per round) and statistical efficiency (rounds to target accuracy). While heuristic-based frameworks like Oort and PyramidFL offer improvements over random selection baselines (e.g., FedAvg), they are bounded by static utility functions. FLLLM leverages a dynamic LLM-based orchestration—coupled with epoch adaptation and model pruning—to actively push the Pareto frontier toward the optimal bottom-left quadrant, achieving faster convergence times and fewer total communication rounds.

round. Thus, we aim to leverage FLLLM to reduce the number of rounds and the overall training time while increasing the accuracy.

State-of-the-art client selection frameworks, such as Oort [**25**] and PyramidFL [**24**], attempt to navigate this space by prioritizing clients based on defined utility functions. Oort ranks clients by their statistical utility (e.g., training loss) while imposing penalties for exceeding a target system latency. PyramidFL further refines this by profiling the variance of selected clients and actively exploiting the fastest available nodes to accelerate training.

While Oort [**25**] and PyramidFL [**24**] approaches successfully improve upon random selection (e.g., FedAvg [**2**]), they are fundamentally limited by their reliance on static,

hand-crafted heuristics. The formulas dictating client utility in Oort and PyramidFL remain fixed throughout the training lifecycle. Consequently, they fail to dynamically adapt to highly volatile edge environments where the value of a client's data or computational speed changes non-linearly. For instance, the relative importance of exploring diverse data distributions versus exploiting fast, reliable devices shifts dramatically between the early phases of model training and the final fine-tuning epochs. Static heuristics cannot capture these complex, time-varying dynamics, resulting in sub-optimal performance ceilings.

To overcome the limitations of heuristic-based selection, this work introduces FLLLM, which reformulates client selection as an intelligent, dynamic orchestration problem managed by a Large Language Model (LLM). **The primary motivation for leveraging an LLM is its capacity for advanced reasoning and zero-shot generalization over complex state spaces.**

Unlike Oort and PyramidFL, which compress complex client profiles into a single, static scalar metric U_i (for instance, aggregating a client's statistical utility $\mathcal{L}_i$ and system penalty P_i into a rigid linear combination such as $U_i = \alpha\mathcal{L}_i - \beta P_i$), thereby discarding critical contextual nuances, FLLLM continuously evaluates a comprehensive 16-dimensional state vector. This scalar reduction in traditional heuristics creates a fundamental blind spot: a rigid formula can assign the exact same numerical score to a computationally slow device with exceptionally rare data and a lightning-fast device with highly redundant data, masking their vastly different operational values.

To overcome this, the 16-dimensional vector in FLLLM encapsulates real-time system conditions, fluctuating network states device computational capacities, and historical data quality metrics in their raw, uncompressed form. By processing this high-dimensional state representation, the LLM can dynamically infer the non-linear interdependencies between variables that rigid formulas fail to capture. For example, rather than strictly penalizing a computationally slow device based on a static system threshold (like the penalty βP_i, the LLM can contextually recognize its immediate value if that device possesses high-utility data essential for the current training phase. Consequently,

FLLLM moves beyond one-dimensional ranking, enabling a context-aware orchestration that dynamically aligns heterogeneous client capabilities with the immediate statistical requirements of the global model.

The LLM acts as an active agent that learns contextually. If the environment exhibits severe straggler effects, the LLM dynamically shifts its selection strategy to prioritize system efficiency without requiring manual retuning. In contrast, if statistical stagnation is detected, it intuitively prioritizes data diversity.

Furthermore, FLLLM actively shapes the trade-off space rather than merely reacting to it. By jointly outputting intelligent client subsets alongside dynamic Epoch Adaptation and Model Pruning directives, the LLM actively reduces both the computational load and the network payload for constrained clients. As depicted in Figure 2, this multi-dimensional optimization shifts the training trajectory directly toward the optimal bottom-left quadrant, minimizing both the total time-to-accuracy and the communication rounds required, effectively bridging the gap left by rigid, state-of-the-art heuristics.

1.1. Federated Learning Problem Formulation. We consider a synchronous Federated Learning (FL) setting with a set of K distinct clients $\mathcal{K} = \{1, \ldots, K\}$. Each client k holds a private dataset $\mathcal{D}_k = \{(\mathbf{x}_i, y_i)\}_{i=1}^{|\mathcal{D}_k|}$. The goal is to learn global model parameters $\boldsymbol{\theta} \in \mathbb{R}^d$, where $f(\mathbf{x}; \boldsymbol{\theta})$ denotes the model prediction and $\ell(\cdot, \cdot)$ a task-specific loss function, by minimizing the empirical risk:

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{k=1}^{K} p_k F_k(\boldsymbol{\theta}), \tag{35}$$

where

$$F_k(\boldsymbol{\theta}) \triangleq \frac{1}{|\mathcal{D}_k|} \sum_{(\mathbf{x}, y) \in \mathcal{D}_k} \ell(f(\mathbf{x}; \boldsymbol{\theta}), y) \tag{36}$$

denotes the local objective over client dataset $\mathcal{D}_k$, and $p_k \geq 0$ is the aggregation weight, typically defined as $p_k = \frac{|\mathcal{D}_k|}{\sum_{j=1}^{K} |\mathcal{D}_j|}$.

At each communication round t, the server selects a subset of clients $S_t \subseteq \mathcal{K}$ with fixed cardinality $|S_t| = M$. The selection is conditioned on a system state vector $\mathbf{s}_t$ capturing heterogeneous client telemetry (e.g., gradient statistics, data characteristics, and system performance metrics).

We aim to learn a selection policy $\pi_\phi(S_t \mid \mathbf{s}_t)$, parameterized by ϕ, that maximizes the expected cumulative utility over a training horizon T:

$$\max_{\phi} \ \mathbb{E}_{S_{0:T} \sim \pi_\phi} \left[\sum_{t=0}^{T} \gamma^t \, R(S_t, \mathbf{s}_t) \right], \tag{37}$$

where $R(S_t, \mathbf{s}_t)$ measures the joint statistical and system contribution of the selected subset, and $\gamma \in (0, 1]$ is a temporal discount factor.

To explicitly balance the tradeoff between statistical utility and wall-clock time, we carefully define the reward $R(S_t)$. To enforce a strict hierarchy prioritizing high-magnitude updates, we define a linear, velocity-based metric rather than a logarithmic squash. We first calculate the system's mathematical velocity, $V(S_t)$, representing the informational gain per physical unit of time:

$$V(S_t) = \frac{\left(\sum_{k \in S_t} ||\nabla F_k(\boldsymbol{\theta}_t)|| \right) \times \text{ScaleFactor}}{\max_{k \in S_t} \left(T_{comm}^{(k)} + T_{comp}^{(k)} \right)} \tag{38}$$

We then formulate the final reward by combining this velocity with a flat time penalty:

$$R(S_t) = \alpha \cdot V(S_t) - \beta \cdot \max_{k \in S_t} \left(T_{comm}^{(k)} + T_{comp}^{(k)} \right) \tag{39}$$

In this formulation, $V(S_t)$ drives the policy to maximize the absolute gradient improvement relative to the time spent, while β represents a strict linear penalty on system staleness. This ensures the LLM specifically seeks a Pareto-optimal frontier, maximizing accuracy per unit of time rather than merely minimizing raw system latency.

2. The FLLLM Architecture

Having established the limitations of static heuristics—specifically their reliance on rigid, pre-defined hyperparameters that fail to adapt to volatile training environments—we now introduce the proposed FLLLM architecture. Unlike frameworks such as Oort [**25**] and PyramidFL [**24**], which require manual tuning prior to deployment, FLLLM acts as an autonomous orchestrator.

By leveraging a Large Language Model to process a comprehensive 16-dimensional state vector, our architecture eliminates human guesswork. The LLM natively understands real-time environment metrics—such as fluctuating network bandwidths, heterogeneous device compute capabilities, and evolving data quality distributions. By interpreting these complex states, the LLM autonomously adapts the parameters that previously required rigid manual tuning, dynamically inferring the optimal target latency thresholds, statistical penalty weights, and exploration decay rates for each specific training round.

2.1. From Manual Tuning to Autonomous Orchestration. A critical limitation of state-of-the-art client selection frameworks is their heavy reliance on pre-defined mathematical proxies and rigid utility equations. Frameworks such as Oort and PyramidFL require system administrators to manually define several hyperparameters before the commencement of a Federated Learning session.

To illustrate this rigidity, we consider the exact utility equation driving client selection in Oort. Oort attempts to balance data quality and system speed by scoring each client i using a hardcoded proxy function:

$$U_{Oort}(i) = |L_i|\sqrt{|D_i|} \cdot \left(\frac{T_{target}}{\max(T_i, T_{target})} \right)^{\gamma} \tag{40}$$

In this formulation, the statistical utility is represented by the empirical training loss L_i and the dataset size D_i. This value is then aggressively penalized if the client's estimated completion time T_i exceeds a human-defined target latency T_{target}. The severity

of this penalty is controlled by γ, another statically defined coefficient. If a system administrator guesses the wrong T_{target} for the current network weather, the framework will either reject valuable clients unnecessarily or stall the global training process.

Similarly, PyramidFL relies on mathematical heuristics that require pre-defined profiling windows. PyramidFL scores clients by estimating the variance of their local model updates relative to their total execution time:

$$U_{Pyramid}(i) = \frac{Var(\Delta w_i)}{T_{comp}^{(i)} + T_{comm}^{(i)}} \tag{41}$$

Here, $Var(\Delta w_i)$ represents the profiled variance of the client's weight divergence, while the denominator sums the pre-calculated computation and communication times. This approach forces the system to spend initial training rounds simply profiling clients to fill these variables, and the linear division assumes a constant trade-off ratio between variance and time that fails to hold across different phases of model convergence.

The necessity of hardcoding these equations introduces severe brittleness into the training process. In highly volatile edge environments, the optimal value for these parameters is fundamentally non-stationary. A static latency threshold that performs well during periods of high network availability will cause catastrophic client rejection rates if global bandwidth suddenly drops.

FLLLM systematically eliminates the need for these human-defined priors and rigid mathematical proxies. Instead of requiring an engineer to guess the optimal latency threshold or penalty weight a priori, the Large Language Model autonomously infers the optimal operational boundaries in real-time. Because the LLM continuously processes the raw, 16-dimensional state vector, it acts as an autonomous orchestrator. Through its Direct Preference Optimization alignment, the LLM has inherently learned the complex, non-linear trade-offs that those static equations were originally designed to approximate.

If the global network degrades, the LLM intuitively relaxes its internal latency expectations without requiring human intervention to update a hardcoded target variable. If the global model begins to overfit, the LLM naturally increases its exploration of diverse

clients without relying on a static variance threshold. Consequently, FLLLM transitions federated client selection from a manually tuned, mathematically rigid heuristic to a fully autonomous, proactive orchestration system.

2.2. Input State Modeling. We model the FL environment state as a composition of macroscopic context and per-client features, as shown in Figure 1. Specifically, the system receives a Global State ($\mathbf{g}_t$) capturing environment-wide statistical variables, and a Client State ($\mathbf{s}_t$) representing each candidate k as a comprehensive continuous vector $\mathbf{x}_k \in \mathbb{R}^{d_{in}}$, where $d_{in} = 15$.

This 15D vector natively embeds System Metrics, Data Metrics, and historical interactions without relying on arbitrary hardcoded formulas. Because variables like completion times, bandwidth, and empirical gradients naturally form heavy-tailed distributions, we explicitly apply $\log(1 + x)$ transformations to these specific metrics to stabilize them mathematically prior to encoding. The 15 dimensions are categorized as follows:

(1) f_0, f_1, f_2: Data samples, statistical feature distance, and local compute speed.
(2) f_3, f_4, f_5, f_6: Network bandwidth capabilities and measured completion times (total, local, comm), each uniformly log-scaled.
(3) f_7: The expected underlying gradient utility ($\log(1 + g_k)$).
(4) f_8, f_9: Normalized heuristic scores acting as a statistical anchor, alongside projected virtual finish targets.
(5) $f_{10}, \ldots, f_{14}$: Historical interaction metrics including global selection counts, round staleness, reward volatility, trend analysis, and gradient diversity similarity.

To ensure these features safely interface with the internal attention matrices of a pretrained LLM despite the non-stationary nature of shifting network topologies, the 15D vectors undergo Dynamic Rolling Z-Score Normalization to preserve strict Gaussian distribution stability ($\mu = 0, \sigma = 1$).

Because Z-score normalization inherently normalizes arbitrary variance away, the system recovers the physical, absolute scale of the environment via the Global State ($\mathbf{g}_t \in \mathbb{R}^6$). This vector injects the normalized round timestamp, alongside the batch-level

arithmetic mean and standard deviation for both the raw gradient utilities and total local completion times.

We employ a learnable Multi-Layer Perceptron (MLP), acting as an Encoder module $g_\psi : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{model}}$, to mathematically map these continuous variables directly into the deep embedding dimension of the chosen Large Language Model (e.g., $d_{model} = 4096$):

$$\mathbf{e}_k = g_\psi(\mathbf{x}_k), \quad \mathbf{e}_{global} = g_\psi(\mathbf{g}_t) \tag{42}$$

These resulting feature embeddings bypass the NLP subword tokenizer entirely. The variables are fed as a cohesive sequence to the network backbone, allowing the LLM to process physical Federated Learning metrics natively.

2.3. LLM-Driven Policy with Residual Heuristics. The core operational decision-making module relies on a Transformer-based policy π_ϕ. During the forward pass, the LLM processes the sequence of encoded embeddings to produce a context-aware latent representation $\mathbf{h}_k$ for each client. An Action Head subsequently projects this dynamic representation downward to a raw scalar score $z_{LLM}^{(k)}$:

$$\mathbf{H} = \text{Transformer}([\mathbf{e}_{global}, \mathbf{e}_1, \ldots, \mathbf{e}_K]), \quad z_{LLM}^{(k)} = \text{MLP}_{head}(\mathbf{h}_k) \tag{43}$$

To mitigate the “cold-start” algorithmic problem—wherein the untrained LLM initially struggles to prioritize physical network constraints—we introduce a Residual Control mechanism. The framework explicitly incorporates a baseline heuristic utility score $U_{Oort}(k)$, mapped intrinsically from the telemetry vector state. The final executed selection logit $\hat{z}^{(k)}$ is constructed via a convex addition between the standalone Action Head output and the baseline statistical heuristic:

$$\hat{z}^{(k)} = z_{LLM}^{(k)} + \lambda(t) \cdot U_{Oort}(k) \tag{44}$$

Here, $\lambda(t)$ operates as a time-decaying Skip Connection parameter. This architectural bridge guarantees that the network is aggressively guided by a rigidly correct heuristic during the early exploratory phase ($\lambda(t) \approx 0.9$) and smoothly transitions into fully autonomous, LLM-driven exploitation tracking purely optimal policies as $\lambda(t) \to 0.1$.

2.4. Trajectory-Based Optimization via Online DPO. Training a client selector entirely online using traditional scalar surrogate functions is computationally unstable due to the inherently non-differentiable thresholds of discrete client sets. To overcome this, the optimization of the LLM is managed directly through Preference Optimization, applied dynamically via Online Hindsight Experience Replay.

At the onset of round t, the system predicts two competing selection branches: an LLM-generated subset (S_{FLLLM}) and a heuristic-generated baseline (S_{Oort}). The subset projecting the highest theoretical utility is provisionally designated as the Winner (S_w), and the alternative defaults to the Loser (S_l).

Crucially, the provisional Winner subset S_w is then executed on the physical network environment. Upon round completion, the theoretical target metrics are retroactively overwritten by the physically realized gradient magnitudes and measured wall-clock delays. Furthermore, the system imposes strict Hindsight Filtering: if any executed client in S_w severely underperforms its expected velocity—effectively acting as a straggler bottleneck—it is forcefully purged from the Winner set configuration and appended to the Loser set.

This physically verified preference pair is then used to compute the discrete Direct Preference Optimization (DPO) loss. By contrasting the hindsight-corrected optimal subset against the rejected baseline, the objective is mathematically customized for federated orchestration:

$$\mathcal{L}_{DPO}(\phi) = -\mathbb{E}_{(S_w, S_l)} \left[\log \sigma \left(\beta \log \frac{\pi_\phi(S_w)}{\pi_{ref}(S_w)} - \beta \log \frac{\pi_\phi(S_l)}{\pi_{ref}(S_l)} \right) \right] \tag{45}$$

By minimizing this specific loss function, the optimization process actively penalizes the LLM (π_ϕ) if its generated probability distribution mimics the rigid heuristic choices of

the baseline or includes physically verified stragglers. Simultaneously, it maximizes the likelihood of the executed, verified Winner set while preserving bounded similarity to a frozen reference policy (π_{ref}).

Calculating the DPO loss is mathematically elegant, but executing the subsequent backpropagation phase introduces a severe hardware bottleneck. Standard fine-tuning of a Large Language Model requires computing and storing gradients and optimizer states for billions of parameters. This memory footprint is fundamentally incompatible with the hardware constraints of edge orchestration servers. To ensure computational feasibility, FLLLM injects a Low-Rank Adaptation (LoRA) module into the attention architecture of the foundation model.

Instead of updating the massive, pre-trained weight matrices of the base LLM, LoRA hypothesizes that the dynamic adaptations required for specialized tasks possess a low intrinsic dimension. Let $W_0 \in \mathbb{R}^{d\times k}$ represent a frozen weight matrix in the LLM's transformer layer. LoRA constrains the weight update ΔW by representing it as the product of two low-rank decomposition matrices, $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$, where the rank $r \ll \min(d, k)$. The forward pass is thus modified to:

$$h = W_0 x + \Delta W x = W_0 x + BAx \tag{46}$$

Crucially, during the backward pass of the Online DPO algorithm, the foundational weights W_0 remain strictly frozen; no gradients are computed for them. The DPO loss calculated from the physically verified preference pairs ($\mathcal{L}_{DPO}$) backpropagates exclusively to update the parameters within matrices A and B. Consequently, the trainable parameter space ϕ referenced in the DPO objective is radically reduced from billions of parameters to merely a few million.

This targeted gradient routing achieves two critical architectural advantages. First, it completely insulates the foundational weights W_0, thereby immunizing the LLM against catastrophic forgetting. The model retains its general zero-shot reasoning and semantic understanding of network concepts intact within W_0, while the LoRA module (BA) acts

as a specialized, pluggable cognitive layer that maps those concepts to Pareto-optimal client selections.

Second, the combination of DPO and LoRA elegantly eliminates the need for an independent reward model. Traditional Reinforcement Learning from Human Feedback (RLHF) requires hosting a separate, billion-parameter LLM solely to act as a critic, effectively doubling or tripling the memory overhead. By utilizing the physical environment as the deterministic judge and updating only the low-rank matrices via preference optimization, FLLLM achieves dynamic, closed-loop policy alignment within a footprint small enough for edge deployment.

A critical distinction between the standard Direct Preference Optimization (DPO) pipeline and the FLLLM architecture lies in the generation of the preference pairs (S_w, S_l).

In standard natural language alignment, both the chosen and rejected responses are typically sampled directly from the language model's own policy. However, relying purely on self-generated samples for federated client selection limits the exploration space and slows convergence, as the LLM may simply oscillate between equally sub-optimal topological subsets during early training phases.

To accelerate alignment and guarantee state-of-the-art performance, FLLLM alters the preference generation mechanism by introducing bidirectional exogenous sampling. Instead of forcing the LLM to generate both competing subsets, the architecture utilizes the deterministic output of the Oort heuristic as a dynamic anchor.

For a given state context x, the FLLLM policy generates a dynamic selection S_{FLLLM}, while the Oort algorithm deterministically calculates S_{Oort}. This enables a bidirectional evaluation structure: if physical execution verifies that the LLM outperforms the heuristic ($R(S_{FLLLM}) > R(S_{Oort})$), the preference pair is constructed with Oort as the adversarial baseline ($S_w = S_{FLLLM}, S_l = S_{Oort}$). Conversely, if the LLM generates a mathematically inferior topology ($R(S_{Oort}) > R(S_{FLLLM})$), the system actively forces the LLM to learn from the heuristic by injecting Oort as the chosen response ($S_w = S_{Oort}, S_l = S_{FLLLM}$).

Crucially, the foundational DPO loss function requires no structural modification to accommodate this exogenous input. Rather than relying on autoregressive token generation—which is highly inefficient for unordered set selection—the FLLLM policy (π_ϕ) generates independent latent preference scores for the entire client pool simultaneously. Because the log-likelihood of a selected subset is defined mathematically as the sum of its constituent client scores, the active policy can seamlessly evaluate the Oort-derived subset, $\pi_\phi(S_{Oort})$, simply by aggregating the scores of the exogenously provided indices.

To illustrate the exact mechanics of this score aggregation, consider a simplified environment with five available clients. During the selection phase, the active LLM initially selects $S_{LLM} = \{0, 1\}$, while the Oort baseline selects $S_{Oort} = \{3, 4\}$. Upon physical execution, the environment verifies that Oort's subset achieves higher utility. Due to bidirectional sampling, the preference pair is securely mapped as the winner $S_w = \{3, 4\}$ and the loser $S_l = \{0, 1\}$.

During the subsequent forward pass, the environment state vector is fed into both the active model (π_ϕ) and the frozen reference model (π_{ref}). Unaware of the execution outcome, both models independently project the state into a 1D tensor of client preference scores. Assume the active model outputs a score tensor $T_\phi = [+2.5, +3.0, +0.1, -1.0, -0.5]$, reflecting its initial, incorrect bias toward clients 0 and 1. The frozen reference model outputs a similar anchoring tensor, $T_{ref} = [+2.4, +2.9, +0.2, -0.8, -0.4]$.

The framework extracts the four critical DPO variables by mapping the physical subset indices directly to these output tensors via $O(1)$ summation:

$$\pi_\phi(S_w) = T_\phi[3] + T_\phi[4] = -1.0 + (-0.5) = -1.5$$

$$\pi_\phi(S_l) = T_\phi[0] + T_\phi[1] = 2.5 + 3.0 = 5.5$$

$$\pi_{ref}(S_w) = T_{ref}[3] + T_{ref}[4] = -0.8 + (-0.4) = -1.2$$

$$\pi_{ref}(S_l) = T_{ref}[0] + T_{ref}[1] = 2.4 + 2.9 = 5.3$$

These four scalars are injected directly into the DPO loss function. Recognizing the severe discrepancy—where the active model assigned the verified Loser a dominant score of $+5.5$ and the Winner a sub-optimal score of -1.5—the resulting backpropagation mathematically forces the active LoRA weights to update. In subsequent rounds, the active model's latent scores for clients 3 and 4 are driven higher, while its scores for clients 0 and 1 are penalized. The inclusion of the reference scores serves as a mathematical anchor, preventing the active model from drastically over-correcting its neural pathways during a single update and ensuring smooth trajectory alignment.

By continuously injecting a highly optimized heuristic into this preference pipeline, FLLLM forces a highly targeted gradient update. When the LLM underperforms, Oort acts as an expert teacher; when the LLM discovers superior topologies, Oort becomes the rejected baseline. The LLM does not merely learn a vaguely "good" policy; it mathematically contrasts its contextual decisions against the rigid proxies of existing frameworks, driving the neural weights to explicitly route around the straggler bottlenecks that trap traditional algorithms while anchoring its minimum performance to a state-of-the-art standard.

2.5. Real-Time LLM Training and Reward Stabilization. If DPO training is activated at Round 0 blindly without a guided warmup phase, the topology inevitably suffers from the "Expert Demonstrator Problem." Random exploratory schedules yield an infinitesimally narrow informational delta (Δ) between S_w and S_l, forcing the trainer to mistakenly attempt LoRA gradient updates based on insignificant mathematical noise.

The decaying Skip Connection $\lambda(t)$ dynamically resolves this bottleneck analytically. By actively overwhelming the predicted sequence logits with the heuristic utility during the first iterations, the mathematical output is forced to mirror an Expert Demonstrator pattern. This synthetic oversight safely generates High-Contrast Preference Pairs where the physical algorithmic superiority of S_w mapped over S_l is indisputable.

The strategy effectively delivers pristine gradient signals backward to the LoRA framework. Operating via Behavioral Cloning, the LLM actively calibrates internal attention

mechanisms to correctly associate unrefined raw input telemetry (e.g., bandwidth capability f_3, or log-transmission times f_4) continuously with their verified outcome reward probabilities.

Finally, to address the naturally vanishing magnitude of gradient norms encountered as the target global model reliably converges—a state which would otherwise critically stall the mathematical velocity scaling constants—we anchor their variance dynamically through an Exponential Moving Average (EMA):

$$\tilde{g}_t = \frac{||\nabla \boldsymbol{\theta}_t||}{\mu_t}, \quad \mu_{t+1} = \gamma \mu_t + (1-\gamma)||\nabla \boldsymbol{\theta}_t|| \tag{47}$$

By relying upon a smoothly shifting target anchor for μ_t, the physical reward variance remains mathematically distinguishable across all 200 epochs of training operation, permanently ensuring that the LLM continues extracting highly optimal optimization gradients long into system maturity.

3. Experimental Results

To analyze the results of our architecture, we start by defining the baseline experiments, including current state-of-the-art architectures like Oort [**25**] and PyramidFL [**24**], which attempt to improve the time-to-accuracy metric by optimizing the client selection process.

Oort introduces a Multi-Armed Bandit (MAB) [**112**] approach where each client represents an arm. However, its utility function relies heavily on rigid, pre-defined equations that require significant manual tuning of static hyperparameters. These include a strict Target Round Duration to penalize stragglers, an Exploration Probability (ϵ) that dictates the rate of untested client selection, predefined Pacing Variables (window and step size) to scale utility thresholds, and a static Time Penalty Factor (α) that linearly weights system latency against statistical utility.

PyramidFL builds upon the Oort baseline but seeks to exploit intra-client heterogeneity through two main mechanisms. First, it utilizes local adaptation to set a variable

number of local epochs, aiming to create a fair environment in terms of execution time across the client pool. Second, it introduces a parameter dropout mechanism capable of adapting the payload size sent to the Parameter Server when network conditions become unstable or bandwidth is limited. Despite these dynamic mechanics, PyramidFL still requires statically defined bounds beforehand, such as the maximum allowable Parameter Dropout Limits and strict lower/upper bounds for Local Iterations to prevent catastrophic model degradation.

While these approaches monitor environment variables to capture the global state of the infrastructure, their reliance on static, hard-coded constraints inherently limits their ability to adapt to unprecedented shifts in the network environment. FLLLM overcomes these limitations through contextual awareness, learned Pareto optimization, and implicit exploration. Unlike baselines such as Oort, which rely on rigid, predefined thresholds (e.g., a static target round duration) and risk artificially penalizing the system during sudden network shifts, FLLLM evaluates the full 16D state distribution of all clients simultaneously to determine ”fast” or ”slow” clients based on current conditions. Furthermore, instead of requiring developers to manually hard-code the trade-off between system speed and statistical accuracy, FLLLM uses Direct Preference Optimization (DPO) to autonomously explore the Pareto-optimal frontier, maximizing informational velocity by dynamically evaluating whether a client's data justifies the wait time. Finally, FLLLM eliminates the need for arbitrary random exploration variables, such as an exploration rate ϵ. By projecting client telemetry into a continuous semantic space, the model's attention mechanism naturally highlights underexplored clients based on latent similarities to high performers, avoiding the waste of communication rounds inherent in forced, random exploration.

To ensure a rigorous, standardized baseline comparison, our experimental architecture is built on top of PyramidFL, which natively operates within the FedScale [**105**] ecosystem. We evaluate our architecture across two highly contrasting environments: the lightweight Human Activity Recognition (HAR) dataset and the computationally intensive, high-dimensional OpenImages dataset.

Crucially, these experiments are designed to spotlight the distinct architectural advantages of the LLM-driven policy. We demonstrate the model's exceptional capacity for zero-shot generalization and contextual awareness across vastly different system topologies. By continuously optimizing its selection policy online without relying on prior offline training phases, the LLM autonomously learns complex computational and statistical trade-offs in real time. This cognitive flexibility enables the architecture to adapt instantly to shifting network conditions, resulting in a fundamentally more resilient and dynamic client selection process than rigid heuristic baselines.

Experimental Setup and System Configuration All federated training emulations were conducted on a distributed cluster utilizing the Gloo communication backend. The central Parameter Server was allocated to a dedicated CPU environment, while local client computations and the LLM inference engine were distributed across accelerated hardware nodes equipped with V100 GPUs. Everything is built on top of the PyramidFL framework [**24**]. Furthermore, to ensure data quality and stabilize early gradients, strict data filtration thresholds were applied across all experiments, discarding any candidate clients possessing fewer than 30 local data samples.

For the lightweight Human Activity Recognition (HAR) task, we deployed a MobileNetV2 architecture. The global model was optimized using the Yogi gradient policy. Training was conducted over a maximum of 100 communication rounds, selecting 20 distributed workers per round. Each selected client executed 5 local epochs with a local batch size of 16. The global learning rate was initialized at 4×10^{-5} and bounded by a minimum decay threshold of 2×10^{-5}. To configure the baseline algorithms (Oort and PyramidFL) for this environment, the strict time penalty factor (α) was set to 2.0, with an exploration minimum set to 0.4 and a pacing delta of 0.3 to scale utility thresholds gradually.

To stress-test the system under high-dimensional constraints, the OpenImages dataset was trained using a computationally heavy ShuffleNetV2 (2.0x) architecture. The server aggregated updates using proxy averaging over a maximum of 200 communication rounds. For this demanding environment, the participant pool was expanded to 50 active workers per round, executing 5 to 10 local epochs with a batch size of 16. A static global

learning rate of 0.05 was maintained. In the baseline configurations, the pacing delta was aggressively increased to 10, while the time penalty factor remained consistent at 2.0.

LLM Policy Configuration The FLLLM architecture was powered by a Llama-3.2-3B base model serving as the core client selector. To rigorously evaluate the LLM's capacity for exploration versus exploitation, we tested the architecture across a full spectrum of generation temperatures ranging from 0.0 (entirely deterministic) to 1.0 (highly stochastic).

To actively mitigate the cold-start instability inherent in Reinforcement Learning, the LLM was subjected to a carefully guided heuristic warmup phase spanning the initial 5 communication rounds. During this phase, the skip connection weight linking the baseline heuristic to the LLM's raw logits was initialized at a dominant factor of 0.9, enforcing expert demonstrator behavior. This weight then decayed linearly to a stable operational floor of 0.1.

Furthermore, the DPO agent's reward oracle was calibrated to seek the Pareto-optimal frontier by balancing a primary statistical utility coefficient of 0.5 against a time-based latency penalty weight of 0.05. Finally, to ensure the Transformer's attention mechanisms could reliably interpret the heterogeneous telemetry metrics, dynamic per-round metric normalization was applied to the 16D state vectors, standardizing the input distributions before mapping them into the LLM's semantic embedding space.

Performance Analysis The convergence trajectories in Figure 3 validate the efficacy of the FLLLM architecture. In the HAR dataset, which features lower computational overhead, FLLLM matches the rapid statistical convergence of Oort while slightly improving upon the overall execution time. However, the true strength of the dynamic LLM policy is revealed in the computationally demanding OpenImages environment (Figures 3c and 3d). Here, the baselines heavily penalize clients based on static constraints, leading to slow accumulation of accuracy over time. Conversely, the LLM intelligently identifies high-value data distributions that justify slightly longer compute times, resulting in a significantly steeper convergence curve. Notably, mid-range temperatures (e.g., LLM-0.2

and LLM-0.4) strike an optimal balance between exploiting known reliable clients and exploring new data distributions, maximizing overall system efficiency.

3.1. Dynamic Clients Selection Future Work. In this work, we introduced FLLLM, a pioneering architecture that reconceptualizes the client selection process in Federated Learning from a rigid mathematical optimization problem into a dynamic, sequence-modeling task. We demonstrated that integrating a Large Language Model as the central orchestration agent significantly accelerates model convergence and improves overall system efficiency across diverse and computationally demanding environments.

The core achievement of this research lies in demonstrating the superiority of contextual, real-time learning over static heuristics. Current state-of-the-art architectures, such as Oort and PyramidFL, rely on hard-coded mathematical proxies to balance system latency and statistical utility. While effective under predictable conditions, these algorithms are bound by static hyperparameter constraints—such as fixed time penalties or rigid exploration probabilities—which critically limit their ability to adapt to unprecedented network volatility or extreme data heterogeneity.

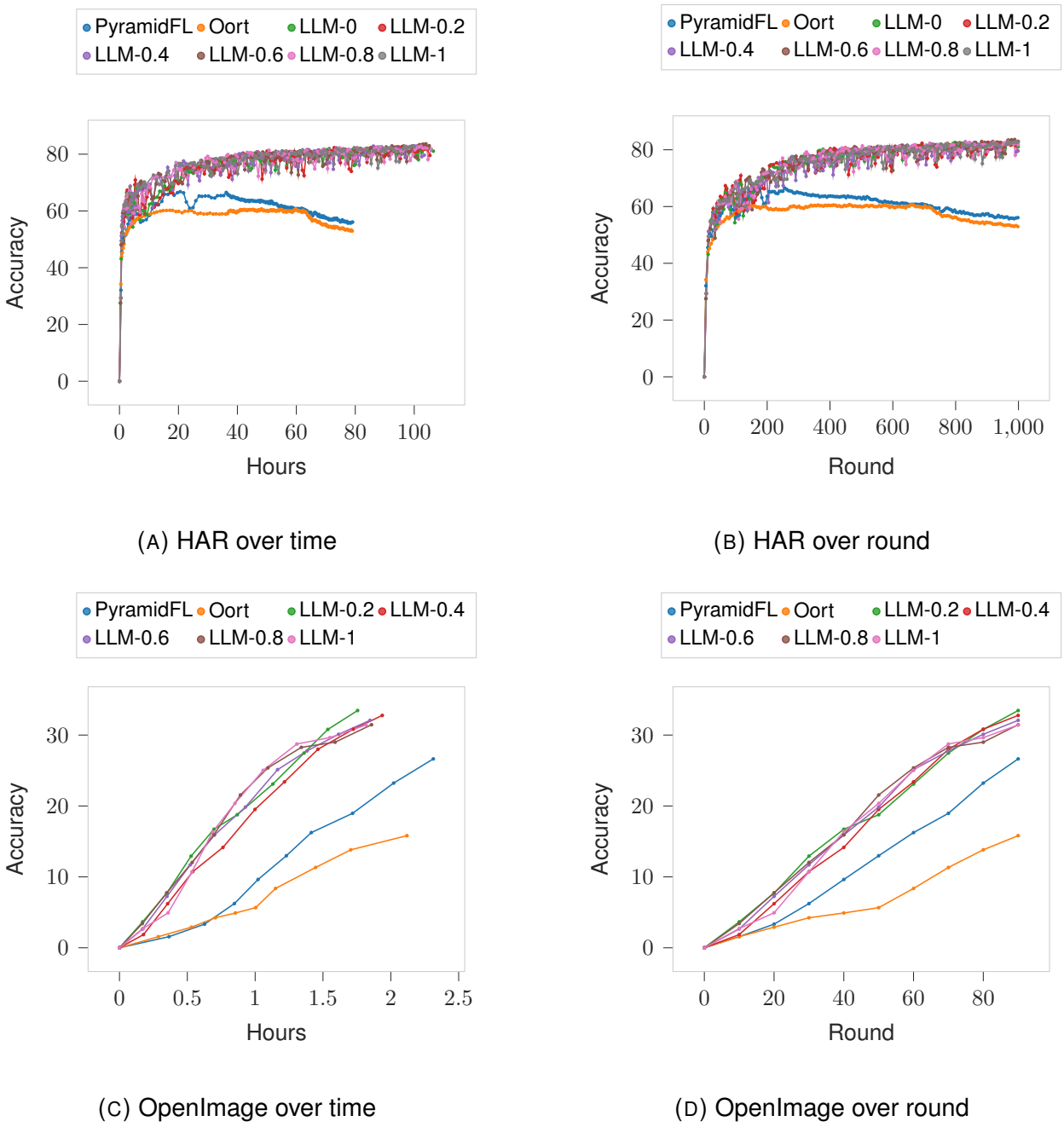
FLLLM fundamentally overcomes this limitation. By projecting a comprehensive 16D state vector of client telemetry into a continuous semantic space, the LLM treats client selection as a contextual reasoning task. Powered by Direct Preference Optimization (DPO) and dynamic LoRA updates, the LLM continuously learns and refines its policy in real-time. Rather than blindly applying a predefined equation, it autonomously interprets the immediate context of the network environment, navigating the Pareto-optimal frontier to prioritize high-value data without human intervention. This zero-shot generalization allows the architecture to implicitly handle exploration and dynamically adjust its threshold for ”fast” or ”slow” clients based on current network weather, resulting in a highly resilient architecture.

Looking forward, several promising avenues remain for extending this research. First, while FLLLM demonstrates exceptional performance in synchronous federated environments, adapting the architecture for Asynchronous Federated Learning (AFL) presents a compelling challenge. Training the LLM to manage extreme staleness and out-of-order

gradient updates could further reduce wall-clock training times in highly unstable edge networks.

Second, the current 16D state vector can be expanded to encompass a broader set of multi-objective constraints. By incorporating metrics such as real-time device energy consumption, thermal throttling, and differential privacy budgets, the LLM could be trained to navigate a complex, multi-dimensional Pareto frontier, optimizing not just for time and accuracy, but for sustainability and security.

Finally, optimizing the LLM selector itself offers a critical path for deployment. Investigating aggressive quantization techniques or knowledge distillation could compress the reasoning capabilities of the multi-billion parameter model into a lightweight control plane. This would allow the FLLLM architecture to be deployed directly on resource-constrained edge servers, fully democratizing intelligent, real-time orchestration in distributed networks.



(A) HAR over time

(B) HAR over round

(C) OpenImage over time

(D) OpenImage over round

FIGURE 3. Comparative performance of FLLLM versus state-of-the-art baselines. Subfigures (a) and (b) display the test accuracy trajectories for the HAR dataset plotted against cumulative wall-clock time and communication rounds, respectively. Subfigures (c) and (d) illustrate the convergence on the high-dimensional OpenImages dataset. The FLLLM policy is evaluated across a spectrum of temperature parameters (denoted as LLM-0 through LLM-1). Across both environments, FLLLM consistently demonstrates superior Pareto-optimal efficiency, reaching target accuracies in significantly less wall-clock time than both Oort and PyramidFL.

CHAPTER 7: Conclusion and Discussion

We present a comprehensive, multi-layer architecture designed to address the critical resource and networking bottlenecks inherent in modern distributed machine learning systems. As models grow in complexity and data generation becomes increasingly decentralized, traditional orchestration paradigms that treat compute and communication as isolated domains are no longer sufficient. By decoupling computational and networking requirements across three distinct operational layers—the Cloud Data Center, the Near Edge, and the Far Edge—this work provides scalable, robust, and network-aware solutions.

At the Cloud Data Center layer, this thesis introduced Plebiscito, a fully decentralized, asynchronous orchestration architecture. Unlike conventional schedulers that rely on centralized control and remain agnostic to network conditions, Plebiscito utilizes a distributed max-consensus auction to track system load and allocate jobs based on a bandwidth-aware utility function. By proactively avoiding network hotspots and co-locating communicating components, Plebiscito significantly reduces job completion times, allocation failure rates, and cluster-wide network contention for Machine-Learning-as-a-Service workloads.

Moving to the Near Edge, we addressed the uplink communication bottlenecks prevalent in wireless Federated Learning with the introduction of FLAG (Federated Learning with In-Network Aggregation). By embedding aggregation logic directly into the user-plane of the programmable 5G stack, specifically at the SDAP layer, FLAG transforms base stations into active participants in the training loop. To ensure robustness against volatile wireless conditions, FLAG incorporates Partial-Contribution Correction to mitigate bias from lost updates and Deadline-Driven Grouping to bound straggler delays.

These mechanisms collectively reduce the volume of data traversing the backhaul and accelerate the overall time-to-accuracy.

At the Far Edge layer, we developed Federated Split Learning (FSL), a hybrid paradigm designed to accommodate severely resource-constrained devices. By partitioning the neural network between the end-user device and an edge server, FSL offloads the bulk of the computational burden while preserving data privacy. This architecture allows multiple client-server pairs to train in parallel, achieving high scalability and overcoming the synchronization and latency limitations inherent to standard split learning and federated learning methods.

Finally, to orchestrate the participation of heterogeneous edge devices, we presented FLLLM, an intelligent client selection mechanism powered by Large Language Models. Moving beyond rigid mathematical heuristics, FLLLM projects real-time client telemetry into a semantic state space, using Direct Preference Optimization to continuously refine its selection policy. This allows the system to autonomously balance statistical utility and system latency, dynamically adapting to network volatility and data heterogeneity to further maximize training efficiency.

While this dissertation establishes a robust foundation for orchestrating distributed learning systems, several promising avenues remain for future exploration.

First, extending the capabilities of the Large Language Model orchestration agent to support Asynchronous Federated Learning presents a compelling challenge. Training the orchestration model to manage extreme staleness, out-of-order gradient updates, and severe client dropouts could further reduce wall-clock training times in highly unstable edge networks. Furthermore, the telemetry state vector can be expanded to encompass a broader set of multi-objective constraints. By incorporating metrics such as real-time device energy consumption, thermal throttling, and differential privacy budgets, the system could be trained to navigate a complex, multi-dimensional Pareto frontier, optimizing for sustainability and security alongside time and accuracy.

Second, the deployment of intelligent control planes on resource-constrained edge servers requires further optimization. Investigating aggressive model quantization techniques and knowledge distillation could compress the reasoning capabilities of multi-billion parameter models into lightweight agents. This would democratize intelligent, real-time orchestration across fully distributed networks without requiring extensive cloud-based computational resources.

Lastly, exploring the holistic integration of these distinct mechanisms deploying Plebiscito, FLAG, and FSL in a unified, end-to-end framework could yield compounding performance benefits. Co-optimizing the data center job placement with the 5G in-network aggregation and edge-device splitting would provide a seamless continuum of machine learning orchestration. Such a unified system would dynamically shift compute and communication workloads across the entire infrastructure hierarchy, paving the way for unprecedented efficiency in next-generation distributed training.

Bibliography

[1] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[2] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," 2023. [Online]. Available: https://arxiv.org/abs/1602.05629

[3] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," 2018. [Online]. Available: https://arxiv.org/abs/1802.09941

[4] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, no. 2, Mar. 2020. [Online]. Available: https://doi.org/10.1145/3377454

[5] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 945–960.

[6] B. Costa, J. Bachiega Jr, L. R. de Carvalho, and A. P. Araujo, "Orchestration in fog computing: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 55, no. 2, pp. 1–34, 2022.

[7] A. Gangidi, R. Miao, S. Zheng, S. J. Bondu, G. Goes, H. Morsy, R. Puri, M. Riftadi, A. J. Shetty, J. Yang, S. Zhang, M. J. Fernandez, S. Gandham, and H. Zeng,

"Rdma over ethernet for distributed training at meta scale," in *Proceedings of the ACM SIGCOMM 2024 Conference*, ser. ACM SIGCOMM '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 57–70. [Online]. Available: https://doi.org/10.1145/3651890.3672233

[8] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," 2023. [Online]. Available: https://arxiv.org/abs/1602.05629

[9] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 167. [Online]. Available: https://doi.org/10.1145/2806777.2809955

[10] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2523616.2523633

[11] P. Moritz et. al, "Ray: A distributed framework for emerging AI applications," in {*OSDI*} *18*, 2018, pp. 561–577.

[12] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 613–627.

[13] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient GPU cluster scheduling," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 289–304.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, “Dominant resource fairness: Fair allocation of multiple resource types,” in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. Boston, MA: USENIX Association, Mar. 2011. [Online]. Available: https://www.usenix.org/conference/nsdi11/dominant-resource-fairness-fair-allocation-multiple-resource-types

[15] S. Rajasekaran, M. Ghobadi, and A. Akella, “Cassini: Network-aware job scheduling in machine learning clusters,” in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1403–1420. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/rajasekaran

[16] Y. Zhao, Y. Liu, Y. Peng, Y. Zhu, X. Liu, and X. Jin, “Multi-resource interleaving for deep learning training,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 428–440.

[17] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, “Scaling distributed machine learning with In-Network aggregation,” in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 785–808. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/sapio

[18] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, “Tiresias: A GPU cluster manager for distributed deep learning,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 485–500.

[19] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, “Federated learning with non-iid data,” 2018. [Online]. Available: https://arxiv.org/abs/1806.00582

[20] O. Gupta and R. Raskar, “Distributed learning of deep neural network over multiple agents,” *Journal of Network and Computer Applications*, 2018.

[21] R. Pathak and M. J. Wainwright, “FedSplit: an algorithmic framework for fast federated optimization,” in *NeurIPS*. Curran Associates, Inc., 2020.

[22] Z. Zhang, A. Pinto, V. Turina, F. Esposito, and I. Matta, “Privacy and efficiency of communications in federated split learning,” *IEEE Transactions on Big Data*, pp. 1–12, May 2023.

[23] J. Jeon and J. Kim, “Privacy-Sensitive Parallel Split Learning,” in *ICOIN*, 2020.

[24] C. Li, X. Zeng, M. Zhang, and Z. Cao, “Pyramidfl: A fine-grained client selection framework for efficient federated learning,” in *Proceedings of the 28th annual international conference on mobile computing and networking*, 2022, pp. 158–171.

[25] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, “Oort: Efficient federated learning via guided participant selection,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 19–35. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/lai

[26] C.-W. Ching, X. Chen, T. Kim, B. Ji, Q. Wang, D. Da Silva, and L. Hu, “Totoro: A scalable federated learning engine for the edge,” in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 182–199.

[27] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, “Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 947–960. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/jeon

[28] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, “Ray: A distributed framework for emerging AI applications,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577.

[29] Q. Weng, L. Yang, Y. Yu, W. Wang, X. Tang, G. Yang, and L. Zhang, “Beware of fragmentation: Scheduling gpu sharing workloads with fragmentation gradient

descent," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 995–1008.

[30] J. Cao, Y. Guan, K. Qian, J. Gao, W. Xiao, J. Dong, B. Fu, D. Cai, and E. Zhai, "Crux: Gpu-efficient communication scheduling for deep learning training," in *Proceedings of the ACM SIGCOMM 2024 Conference*, ser. ACM SIGCOMM '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1–15.

[31] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598.

[32] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters." USENIX Association, Nov. 2020, pp. 463–479. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/jiang

[33] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018.

[34] X. Li et al., "On the convergence of fedavg on non-iid data," *arXiv preprint arXiv:1907.02189*, 2019.

[35] Y. Fraboni, R. Vidal, and M. Lorenzi, "Free-rider attacks on model aggregation in federated learning," in *AISTATS*. PMLR, 2021, pp. 1846–1854.

[36] T. Li et al., "Federated Optimization in Heterogeneous Networks," *MLSys*, pp. 429–450, 2020.

[37] D. H. Mahlool and M. H. Abed, "A Comprehensive Survey on Federated Learning: Concept and Applications," *CoRR*, vol. abs/2201.09384, 2022. [Online]. Available: https://arxiv.org/abs/2201.09384

[38] H. Yang et al., “Privacy-Preserving Federated Learning for UAV-Enabled Networks: Learning-Based Joint Scheduling and Resource Management,” *IEEE Journal on Selected Areas in Communications*, 2021.

[39] C. Thapa et al., “Splitfed: When federated learning meets split learning,” in *AAAI*, 2022, pp. 8485–8493.

[40] Y. Matsubara, S. Baidya, D. Callegaro, M. Levorato, and S. Singh, “Distilled Split Deep Neural Networks for Edge-Assisted Real-Time Systems,” in *HotEdgeVideo*. ACM, 2019.

[41] A. E. Eshratifar, A. Esmaili, and M. Pedram, “Bottlenet: A deep learning architecture for intelligent mobile cloud computing services,” in *ISLPED*. IEEE, 2019.

[42] H. Zhou et al., “BBNet: A Novel Convolutional Neural Network Structure in Edge-Cloud Collaborative Inference,” *Sensors*, 2021.

[43] Y. Matsubara, M. Levorato, and F. Restuccia, “Split Computing and Early Exiting for Deep Learning Applications: Survey and Research Challenges,” *ACM Comput. Surv.*, vol. 55, no. 5, dec 2022.

[44] L. Zhang, M. Jeon, J. Jose, S.-Y. Kim, J. Jeong, and B.-G. Kim, “Switchml: Scaling distributed machine learning with in-network aggregation,” in *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. Boston, MA, USA: USENIX Association, April 2021, pp. 365–381. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/zhang-lei

[45] L. Liu, J. Zhang, S. Song, and K. B. Letaief, “Client-edge-cloud hierarchical federated learning,” in *ICC 2020-2020 IEEE international conference on communications (ICC)*. IEEE, 2020, pp. 1–6.

[46] ——, “Hierarchical federated learning with quantization: Convergence analysis and system design,” *IEEE Transactions on Wireless Communications*, vol. 22, no. 1, pp. 2–18, 2022.

[47] F. Han, W. Chen, Y. Zhu, and C. Yuen, “Federated learning aided dual-side channel estimation in multi-user massive mimo systems,” *IEEE Wireless Communications Letters*, 2025.

[48] L. You, S. Liu, T. Wang, B. Zuo, Y. Chang, and C. Yuen, “Aifed: An adaptive and integrated mechanism for asynchronous federated data mining,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 9, pp. 4411–4427, 2023.

[49] L. Fu, H. Zhang, G. Gao, M. Zhang, and X. Liu, “Client selection in federated learning: Principles, challenges, and opportunities,” *IEEE Internet of Things Journal*, vol. 10, no. 24, pp. 21 811–21 819, 2023.

[50] C. Smestad and J. Li, “A systematic literature review on client selection in federated learning,” in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, 2023, pp. 2–11.

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

[52] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Mallis, A. Kolesnikov *et al.*, “The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale,” *International Journal of Computer Vision*, vol. 128, no. 7, pp. 1956–1981, 2020.

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

[54] S. J. Reddi, Z. Charles, M. Zaheer, Z. Garrett, K. Rush, J. Konečnỳ, S. Kumar, and H. B. McMahan, “Adaptive federated optimization,” in *International Conference on Learning Representations (ICLR)*, 2021.

[55] Y. Deng, F. Lyu, J. Ren, H. Wu, Y. Zhou, Y. Zhang, and X. Shen, “Auction: Automated and quality-aware client selection framework for efficient federated learning,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1996–2009, 2021.

[56] D. Wu, X. Wang, Y. Qiao, Z. Wang, J. Jiang, S. Cui, and F. Wang, “Netllm: Adapting large language models for networking,” in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 661–678.

[57] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, “Direct preference optimization: Your language model is secretly a reward model,” *Advances in neural information processing systems*, vol. 36, pp. 53 728–53 741, 2023.

[58] P. Patarasuk and X. Yuan, “Bandwidth optimal all-reduce algorithms for clusters of workstations,” *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731508001767

[59] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari, “The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale,” *International Journal of Computer Vision*, vol. 128, no. 7, p. 1956–1981, Mar. 2020. [Online]. Available: http://dx.doi.org/10.1007/s11263-020-01316-z

[60] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[61] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge: Cambridge University Press, 2004.

[62] M. L. Fisher, R. Jaikumar, and L. N. Van Wassenhove, “A multiplier adjustment method for the generalized assignment problem,” *Management Science*, vol. 32, no. 9, pp. 1095–1103, 1986.

[63] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, “Multi-resource packing for cluster schedulers,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, p. 455–466, Aug. 2014.

[64] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, “Parameter hub: a rack-scale parameter server for distributed deep neural network training,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’18. ACM, Oct. 2018.

[65] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, “An analysis of approximations for maximizing submodular set functions–i,” *Mathematical Programming*, vol. 14, no. 1, pp. 265–294, 1978.

[66] S. Khuller, A. Moss, and J. S. Naor, “The budgeted maximum coverage problem,” *Information Processing Letters*, vol. 70, no. 1, pp. 39–45, 1999.

[67] M. H. J. Saldanha and P. S. L. de Souza, “High performance algorithms for counting collisions and pairwise interactions,” in *Computational Science – ICCS 2019*, J. M. F. Rodrigues, P. J. S. Cardoso, J. Monteiro, R. Lam, V. V. Krzhizhanovskaya, M. H. Lees, J. J. Dongarra, and P. M. Sloot, Eds. Cham: Springer International Publishing, 2019, pp. 182–196.

[68] Alibaba, “Alibaba public traces,” https://github.com/alibaba/clusterdata, accessed: 2025-05-15.

[69] I. Baldin, A. Nikolich, J. Griffioen, I. I. S. Monga, K.-C. Wang, T. Lehman, and P. Ruth, “FABRIC: A national-scale programmable experimental network infrastructure,” *IEEE Internet Computing*, vol. 23, no. 6, pp. 38–47.

[70] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, “Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 947–960. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/jeon

[71] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.

[72] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[73] A. Krizhevsky, “Learning multiple layers of features from tiny images,” University of Toronto, Tech. Rep., 2009.

[74] FRRouting Project, *FRRouting User Manual*, FRRouting Project, 2025, available at https://docs.frrouting.org/.

[75] V. Turina, Z. Zhang, F. Esposito, and I. Matta, “Federated or Split? A Performance and Privacy Analysis of Hybrid Split and Federated Learning Architectures,” in *IEEE CLOUD*, 2021.

[76] S. Qiu, D. Wang, G. Xu, and S. Kumari, “Practical and Provably Secure Three-Factor Authentication Protocol Based on Extended Chaotic-Maps for Mobile Lightweight Devices,” *IEEE Transactions on Dependable and Secure Computing*, 2020.

[77] Q. Jiang et al., “Unified Biometric Privacy Preserving Three-Factor Authentication and Key Agreement for Cloud-Assisted Autonomous Vehicles,” *IEEE Transactions on Vehicular Technology*, 2020.

[78] M. Fredrikson et al., “Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures,” in *CCS*. ACM, 2015.

[79] S. Hidano et al., “Model Inversion Attacks for Prediction Systems: Without Knowledge of Non-Sensitive Attributes,” in *PST*, 2017.

[80] K. Singhal et al., “Federated Reconstruction: Partially Local Federated learning,” *NeurIPS*, pp. 11 220–11 232, 2021.

[81] J. Li et al., “Ressfl: A resistance transfer framework for defending model inversion attack in split federated learning,” in *CVPR*, 2022.

[82] P. Vepakomma, A. Singh, O. Gupta, and R. Raskar, “NoPeek: Information Leakage Reduction to Share Activations in Distributed Deep Learning,” in *ICDMW*, 2020.

[83] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[84] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, “Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge,” in *ASPLOS*, 2017.

[85] T. Unterthiner et al., “Predicting Neural Network Accuracy from Weights,” *arXiv preprint arXiv:2002.11448*, 2020.

[86] K. K. et. al., “Lessons Learned from the Chameleon Testbed,” in *ATC*, July 2020.

[87] T. Ryffel et al., “A generic framework for privacy preserving deep learning,” *arXiv preprint arXiv:1811.04017*, 2018.

[88] (2020) PyGrid. [Online]. Available: https://github.com/OpenMined/PyGrid

[89] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” *Tech Report, Department of Computer Science, University of Toronto*, 2009.

[90] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-Based Learning Applied to Document Recognition,” *Proceedings of the IEEE*, 1998.

[91] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based Learning Applied to Document Recognition,” *Proceedings of the IEEE*, pp. 2278–2324, 1998.

[92] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang, “End-to-end Encrypted Traffic Classification with One-dimensional Convolution Neural Networks ,” in *ISI*, 2017.

[93] J. Bergstra and Y. Bengio, “Random Search for Hyper-Parameter Optimization,” *JMLR*, 2012.

[94] L. Li et al., “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization,” *JMLR*, 2018.

[95] G. E. Hinton and R. S. Zemel, “Autoencoders, Minimum Description Length and Helmholtz Free Energy,” in *NIPS*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[96] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, “Deep Learning with Differential Privacy,” ser. CCS ’16. ACM, 2016.

[97] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, “EMNIST: Extending MNIST to Handwritten Letters,” in *IJCNN*, 2017.

[98] A. Yousefpour et al., “Opacus: User-Friendly Differential Privacy Library in PyTorch,” *arXiv preprint arXiv:2109.12298*, 2021.

[99] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, "A survey on evaluation of large language models," *ACM transactions on intelligent systems and technology*, vol. 15, no. 3, pp. 1–45, 2024.

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

[101] W. Y. B. Lim, N. C. Luong, D. T. Hoang, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, and C. Miao, "Federated learning in mobile edge networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 2031–2063, 2020.

[102] R. J. Hayek, J. Chung, K. Comer, C. R. Murthy, R. Kettimuthu, and I. Kadota, "Federated learning over 5g, wifi, and ethernet: Measurements and evaluation," in *Proceedings of the 26th Intl. Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing (MobiHoc '25)*, 2025.

[103] T. Henttonen, J. Koskela, B. Sébire, and A. Toskala, "5g radio protocols," *5 G Technology: 3 GPP New Radio*, pp. 149–186, 2020.

[104] J. Collins, "Gprs tunneling protocol (gtp)," in *Encyclopedia of cryptography, security and privacy*. Springer, 2022, pp. 1–3.

[105] F. Lai, Y. Dai, S. Singapuram, J. Liu, X. Zhu, H. Madhyastha, and M. Chowdhury, "Fedscale: Benchmarking model and system performance of federated learning at scale," in *International conference on machine learning*. PMLR, 2022, pp. 11 814–11 827.

[106] Kaggle. (2025) Human activity recognition using smartphones. [Online]. Available: https://www.kaggle.com/datasets/uciml/human-activity-recognition-with-smartphones

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

[108] W. Gao, F. Tang, L. Wang, J. Zhan, C. Lan, C. Luo, Y. Huang, C. Zheng, J. Dai, Z. Cao *et al.*, "Aibench: an industry standard internet service ai benchmark suite," *arXiv preprint arXiv:1908.08998*, 2019.

[109] J. Huang, C. Chen, Y. Pei, Z. Wang, Z. Qian, F. Qian, B. Tiwana, Q. Xu, Z. Mao, M. Zhang *et al.*, "Mobiperf: Mobile network measurement system," *Technical Report. University of Michigan and Microsoft Research*, 2011.

[110] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečnỳ, S. Mazzocchi, B. McMahan *et al.*, "Towards federated learning at scale: System design," *Proceedings of machine learning and systems*, vol. 1, pp. 374–388, 2019.

[111] R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," 2024. [Online]. Available: https://arxiv.org/abs/2305.18290

[112] A. Slivkins, "Introduction to multi-armed bandits," 2024. [Online]. Available: https://arxiv.org/abs/1904.07272