



# Flame: Simplifying Topology Extension in Federated Learning

Harshit Daga\*  
Georgia Institute of Technology

Jaemin Shin<sup>†</sup>  
KAIST

Dhruv Garg  
Georgia Institute of Technology

Ada Gavrilovska  
Georgia Institute of Technology

Myungjin Lee\*<sup>‡</sup>  
Cisco Research

Ramana Rao Kompella  
Cisco Research

## ABSTRACT

Distributed machine learning approaches, including a broad class of federated learning (FL) techniques, present a number of benefits when deploying machine learning applications over widely distributed infrastructures. The benefits are highly dependent on the details of the underlying machine learning topology, which specifies the functionality executed by the participating nodes, their dependencies and interconnections. Current systems lack the flexibility and extensibility necessary to customize the topology of a machine learning deployment. We present **Flame**, a new system that provides flexibility of the topology configuration of distributed FL applications around the specifics of a particular deployment context, and is easily extensible to support new FL architectures. Flame achieves this via a new high-level abstraction Topology Abstraction Graphs (TAGs). TAGs decouple the ML application logic from the underlying deployment details, making it possible to specialize the application deployment with reduced development effort. Flame is released as an open source project, and its flexibility and extensibility support a variety of topologies and mechanisms, and can facilitate the development of new FL methodologies.

## CCS CONCEPTS

• **Computer systems organization** → **Real-time systems; Distributed architectures**; • **Computing methodologies** → **Machine learning**.

\*Equal contribution

<sup>†</sup>Work done at Cisco Research

<sup>‡</sup>Corresponding author



This work is licensed under a Creative Commons Attribution International 4.0 License.

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0387-4/23/11.

<https://doi.org/10.1145/3620678.3624665>

## KEYWORDS

Distributed Machine Learning, Federated Learning

### ACM Reference Format:

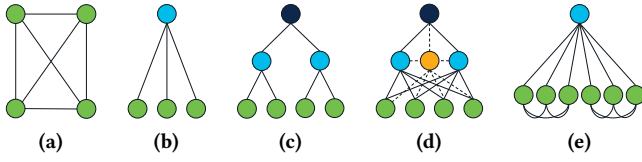
Harshit Daga, Jaemin Shin, Dhruv Garg, Ada Gavrilovska, Myungjin Lee, and Ramana Rao Kompella. 2023. Flame: Simplifying Topology Extension in Federated Learning. In *ACM Symposium on Cloud Computing (SoCC '23)*, October 30–November 1, 2023, Santa Cruz, CA, USA. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3620678.3624665>

## 1 INTRODUCTION

The proliferation of sensors and connected devices such as mobile devices, wearables, and vehicles has resulted in generation of massive amounts of data. In order to quickly and accurately analyze such extraordinarily large and complex data sets to make data-driven decisions, companies have started relying on machine learning techniques. There exist a number of machine learning use cases such as recommendation services [19]; cyber-security breach detection [4]; predictive maintenance and condition monitoring in manufacturing [44]; disease identification in healthcare and life sciences [10] and risk analysis in financial services [24].

Traditional machine learning approaches require collecting all data together in one place, such as a cloud data center. However, with data sources spread geographically, the network becomes the bottleneck. Additionally, user-privacy laws, such as GDPR [35], have resulted in shift towards a federated learning (FL) approach where many clients collectively train a shared model under the orchestration of a central server, also called aggregator.

A classical FL (C-FL) approach adopts a rather simplistic client-server architecture whereby an aggregator (parameter server) builds a global model by combining model updates from training workers (clients). However, not all scenarios fit into this conventional architecture. FL has been a fast-evolving technology and numerous variants [3, 13, 15, 18, 28, 30] have been proposed. Besides accuracy, these are proposed for different performance objectives such as scalability, convergence, training costs, and so on. The designs are also influenced by factors like operation scales and use cases.



**Figure 1: Topologies that can be used in federated learning: (a) distributed, (b) classical FL, (c) hierarchical, (d) hierarchical with replicas and coordinator, and (e) hybrid. ●: training node, ●: aggregation node, ●: global aggregation node, and ●: coordinator. In (d), dotted lines denote connections between coordinator and other components.**

Hence, the system architectures are quite different. Some approaches introduce system components like selectors and coordinators as separate runtime entities [3, 18]; and others introduce edge aggregators [28, 30], enable peer-to-peer collaboration [6, 23], or take a hybrid approach of combining distributed learning and federated learning [13]. As a result, one size (i.e., client-server architecture) doesn't fit all.

The architecture of the distributed machine learning system defines a specific deployment **topology**. Figure 1 illustrates several topologies. The topology is defined by the different nodes participating in the machine learning job, the specific functionality they provide (e.g., train, aggregate, select, etc.), their dependencies and the data or control communication channels that interconnect them (e.g., see Figure 1d). A topology is further defined by the requirements of its deployment, such as concerning specific communication protocol requirements, or the placement of specific functionality relative to the data it should operate on.

Different topologies introduce benefits for different deployment contexts (in terms of scale or geo-distribution, network connectivity, failures or churn), workload properties (e.g., in terms of update frequency or concept drift), and privacy requirements. They also present different tradeoffs in terms of training goals such as time-to-convergence or data transfer costs. It is therefore important to enable flexibility when configuring the distributed training topology. This is further amplified by the continued progress in new training architectures which introduce new types of functionality [22, 33] or cross-participant interactions [23], and opportunities for optimizations that can be realized by updating the topologies of existing systems.

To make it possible to customize the topology of a machine learning deployment, an FL system needs to be flexible and easily extensible; at the same time, it should be able to decouple the management of learning logic, compute and data. The flexibility is provided by many existing solutions, but the latter requirements are not fully met. Current frameworks such as FedScale [21], Flower [2] and PySyft [41] provide low level APIs which make them flexible. However, they cannot be easily extended to support different deployment scenarios

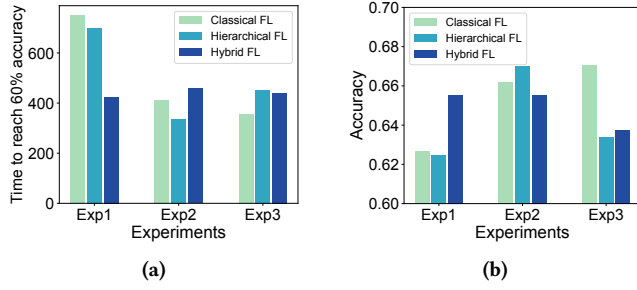
such as hierarchical and hybrid FL, as they lack abstraction suitable for expressing those scenarios.

A recent effort, FedML [17], offers client-server architecture-based abstraction to improve extensibility. This abstraction provides improved expressiveness compared to other frameworks and enables a few templated deployments. However, it quickly becomes difficult to support scenarios where FL components don't fit as either client or server. A canonical example is architectures in [3, 18] where there exist diverse interactions among aggregator, selector and coordinator; classifying them as either client or server gets complicated. Therefore, extending and evolving the deployment scenarios supported by FedML, demands intrusive changes in its codebase, and poses limitations to the flexibility supported by the framework.

In response, we present **Flame**, a new system that provides flexibility of the topology configuration of distributed FL applications around the specifics of a particular deployment context, and is easily extensible to support new FL architectures and algorithms. To achieve this, Flame introduces a new abstraction called *Topology Abstraction Graph (TAG)*. This abstraction enables explicit customization of individual components in the system and supports various designs without requiring modifications to the core system components. The higher level TAG abstraction allows for flexible expression of how these components combine and how they are deployed. Flame also provides interfaces to describe and integrate with different compute infrastructure and dataset providers. This enables support for different resource orchestrators and heterogeneous deployment platforms, as well as to specify and enforce different deployment constraints in terms of data and compute coupling.

The APIs of Flame allow users to express their deployment in a compact TAG representation, provide the machine learning code for the respective roles and select a communication backend. With the APIs, users can extend their use cases easily (§7.1). Flame then expands the TAG to map its physical deployment, using information registered with the system about the properties of nodes and datasets. In §7.2, we demonstrate how users can leverage Flame's flexible backend to adapt to their deployment environment or use cases. The abstract representation supported by Flame allows the users to update their topology by merely updating the TAG graph and providing definitions for any new roles or channel protocols, without changes in the core library. Therefore, users can easily switch from one mechanism/topology to another (§7.3). In §7.5, we compare Flame with other frameworks and showcase that Flame supports a variety of topologies which cannot be easily supported by others. Flame is released as an open source project<sup>1</sup>, and its flexibility and extensibility offer

<sup>1</sup><https://github.com/cisco-open/flame>



**Figure 2: Federated learning experiments where each topology has different time-to-accuracy performance and communication cost: Exp1) straggler node, Exp2) client node failures, and Exp3) aggregator & leader node failures.**

a variety of topologies and mechanisms, and can facilitate the development of new FL methodologies.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Federated Learning

FL [31] has recently emerged as a compelling machine learning practice for meeting data sovereignty and preserving privacy. There exist many variants of federated learning. Many focus on algorithms [26, 27, 31, 32, 39] for improving fairness, accuracy and convergence time. Others propose to select clients intelligently [22, 33] or to carefully sample a client’s dataset [42] for faster convergence. All of these assume the C-FL setting where all clients talk to an aggregation server. In contrast, approaches like hierarchical FL (H-FL) [28, 30], hybrid FL [13] and peer-to-peer FL [23] propose different topologies, and thus entail system architectures and communication patterns different from those of the C-FL.

FL can be used in a cross-device manner, where training is done in-situ on devices, and centrally aggregated. Gboard (a virtual keyboard on Android phone) which recommends keywords that user may type on the keyboard [14, 45] is a canonical example. It can also be used in a cross-silo manner, where training is performed on distributed infrastructure, near data, but possibly on separate compute resources where datasets need to be available. Our work considers such cross-silo settings. Its core concept, however, is generic, and thus it can be applied in other FL settings.

### 2.2 Need for Topology Customization

In FL, the choice of topology can significantly impact the overall efficiency of FL jobs, affecting factors such as time-to-accuracy. Furthermore, the initial conditions assumed for different scenarios may evolve over time, necessitating changes in deployment strategies. These changes can range from simple extensions of a two-level hierarchy to a multi-level hierarchy, or the addition of entirely new components, to modifications in the communication backend.

There are various scenarios that can trigger the need for topology changes in FL. For instance, a one-level C-FL topology may suffice for small-scale experiments. As the experiment scale grows, globally geo-distributed devices favor other topologies, such as hierarchical topologies [28, 30, 43]. In addition, a large-scale hierarchical deployment introduces new components such as selector and coordinator [3, 18], requiring a different topology (e.g., Figure 1d).

We demonstrate trade-offs among three topologies: C-FL, H-FL, and hybrid, with respect to accuracy and convergence time using three scenarios with the CIFAR-10 dataset and 50 training nodes. In C-FL, these nodes are connected to a global aggregator (Figure 1b). For H-FL, the trainers are equally divided into two groups and attached to their corresponding intermediate aggregator (Figure 1c). In the hybrid approach, nodes use distributed learning and a leader node in each group shares updates with the global aggregator (Figure 1e).

As shown in the “Exp1” of Figure 2a, straggling nodes can cause delays in convergence time where hybrid learning is able to reach 60% accuracy up to  $1.77\times$  faster than C-FL and H-FL. In the event of a training node failure (“Exp2” from the same figure), H-FL reached 60% accuracy  $1.36\times$  and  $1.23\times$  faster than hybrid and C-FL, respectively. However, if the intermediate aggregator in H-FL or a group leader in hybrid fails without any failover protection, (shown in “Exp3” of Figure 2b), C-FL reaches 67% accuracy whereas H-FL and hybrid topologies achieve only about 63% accuracy. The experiments illustrate that the “best” topology depends on factors such as operating conditions or target metrics.

Different topologies can require communication backend changes. For instance, existing frameworks employ different protocols (e.g., MQTT and gRPC) as their communication backend. MQTT can be suitable for C-FL as it simplifies operations (e.g., one firewall rule update and proxy setup) since an MQTT broker only needs to be publicly reachable and other entities such as aggregators and trainers can be hidden from the Internet. On the other hand, protocols like gRPC increase management complexity as firewall and proxy settings may need to be updated for each IP or domain name of the entities (e.g., aggregators). For H-FL and hybrid FL, a single MQTT broker causes communication inefficiency as all the traffic routes through the broker. The capability to configure different communication backends can have a direct impact on the operational overheads. In case of the hybrid topology, device-to-device can use gRPC while device-to-server can leverage MQTT as gRPC does not incur more management overhead for the devices in the same LAN or infrastructure while MQTT still preserves the operational simplicity for connectivity across the Internet.

In summary, when selecting a topology and associated training methods, it is crucial to consider that there is *no*

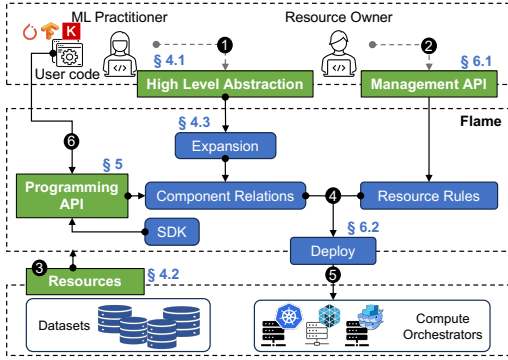


Figure 3: Conceptual overview of the Flame system.

*single topology that applies to all scenarios.* The C-FL architecture is simple and entails lower management overhead but lacks scalability. Hierarchical topologies offer scalability but introduce the risk of failure in intermediate aggregators, which can result in the loss of local updates from multiple training nodes, impacting accuracy and time-to-accuracy. Similarly, hybrid topologies, despite their communication efficiency, carry the risk of failure due to device failures affecting per-group model aggregation when using communication collectives like ring-all-reduce. It is essential to carefully consider the desired goals and trade-offs when customizing the FL topology. To support such flexibility and extensibility there is a need for systems that provide modular support for different components and communication backends.

### 2.3 Current Ecosystem

To create a machine learning system, frameworks such as PyTorch and TensorFlow, provide APIs and low level support for a wide range of models, with focus on speed and optimization. However, creating and deploying such models in geo-distributed settings requires support from resource orchestrators, model registry to store model snapshots and integration with monitoring tools. Alternatively, platforms such as Amazon Sage Maker, Azure ML [7], Vertex AI, offer means for deploying ML applications in the cloud, with a goal to streamline the operation of ML jobs. However, there is no emphasis on support for programmatically extending and adapting the details of the topology of the deployed jobs.

FedML [17] is a recent framework that aims to support easy development of FL-based ML models. Its API and client-server based abstraction allow development of new scenarios, thereby assisting few requirements discussed in this paper. However, the client-server abstraction is not enough to make FedML easily extensible. In FedML, a node is either client or server. Consider the topology shown in Figure 1c designed for hierarchical FL. While training node and global aggregation node fit well with the client-server abstraction, intermediate aggregators don't, because they act as both client as well as server depending on which component they interact

with. In order to handle this dilemma, FedML introduces a concept called *rank* and, based on the rank's value, implements different behaviors in its client codebase. While this enables support for H-FL, it is a stop gap. In hierarchical FL with a separate coordinator (Figure 1d), the rank value can't help because it is unclear which value to assign to rank.

Moreover, while topologies for classical FL (Figure 1b) and hybrid FL (Figure 1e) look similar, behaviors of training and aggregation nodes are dissimilar even though trainers in both topologies are classified as client. Hence, classifying a role as client or server is too coarse to support emerging and diverse FL scenarios. This limitation can require intrusive source code changes in the core codebase. Other frameworks such as FedScale [21], Flower [2] and PySyft [41] follow the same client-server architecture or two-tier topology, and share the same shortcomings of FedML in terms of extensibility.

## 3 FLAME OVERVIEW

**Goal.** The primary objective of Flame is to offer a fine-grained abstraction that eases the composition and extension of machine learning topologies. The system aims to provide modular building blocks that facilitate swift integration and experimentation with new components, new deployment strategies and topologies. Flame also provides precise control over the communication backend to reduce management overheads of FL jobs under different topologies.

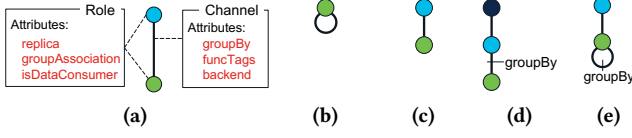
**Need for New System.** The discussion in the previous section points to two key challenges and limitations associated with existing approaches. We summarize them as follows.

**C1:** There is a lack of higher-level, modularized representation of a machine learning application, that explicitly describes all of its components and their dependencies, and could therefore make it possible to flexibly adapt and specialize their behavior and deployment details.

**C2:** In a geo-distributed environment, deploying FL jobs requires coordinating across distinct data and infrastructure providers, possibly owned by different entities. To provide deployment flexibility, there is a need to decouple the infrastructure dependency from the machine learning tasks, yet to provide sufficient meta information that will support mapping and configuration decisions.

**Overview.** The system concept of Flame, as illustrated in Figure 3, revolves around three key components: a high-level abstract description of machine learning job, resource descriptions, and APIs (programming and management) for extending learning techniques and managing jobs. First, a complex machine learning job is described by the developer in the form of *Topology Abstraction Graph* (TAG) (§4.1 and ①). TAG employs a graph-based representation that maps the expanded physical topology to a condensed logical structure, in a manner that captures the functional characteristics of each





**Figure 4: (a) Building blocks of TAG. TAG representation of topologies: (b) distributed, (c) classical FL, (d) hierarchical FL, (e) hybrid. ● is a trainer node with `isDataConsumer` set. The `groupBy` attribute is used to create groups of the same role.**

components and their deployment constraints. Each node in the TAG represents a *role* abstraction, and roles are interconnected using *channels*. Using a graph-based representation, over a client-server architecture, provides expressiveness and extensibility benefits: (i) expressing the responsibility of any component as independent roles; (ii) breaking the components into individual roles provides us with fine-grained abstraction for communication backend between the roles which can be easily switched without impacting the machine learning logic and; (iii) all topologies can be expressed as a graph and TAG can abstract a physical deployment.

Flame is designed to operate on diverse distributed resources (infrastructures and datasets) provided by different entities. These get integrated with Flame through the services provided by the *management plane* (§6.1 and ②). During registration, the resources provide metadata attributes that describe their properties and access constraints. This provides application developers with the flexibility to choose resources based on their specific requirements (§4.2 and ③).

The abstract representation of the TAG is expanded to identify the various components and their relationships (§4.3 and ④). Following the expansion of the TAG into a tangible physical instantiation, Flame leverages the resource metadata and the user-defined infrastructure constraints to facilitate specific deployment decisions (§6.2 and ⑤). Finally, Flame offers support for diverse topologies and provides APIs through a *programming model* to enhance extensibility (§5 and ⑥). These APIs empower developers to develop novel learning techniques or modify existing ones to support different learning techniques associated with distinct roles.

## 4 DESIGN

### 4.1 Topology Abstraction Graph

A central abstraction in Flame is *Topology Abstraction Graph (TAG)*. It represents a simple logical graph, which allows users to express a training workload declaratively for any ML job. It comprises two basic building blocks: *role* and *channel*. A role is a vertex and serves as an abstraction for worker while a channel is an undirected edge between a pair of roles and acts as an abstraction for communication backend. TAG's schema is visually represented as illustrated in Figure 4a and different learning topologies can be represented using a TAG

as shown in Figures 4b-4e. Below we discuss role and channel in detail. To aid understanding of the discussion, a concrete example of a TAG is presented in Figure 5a.

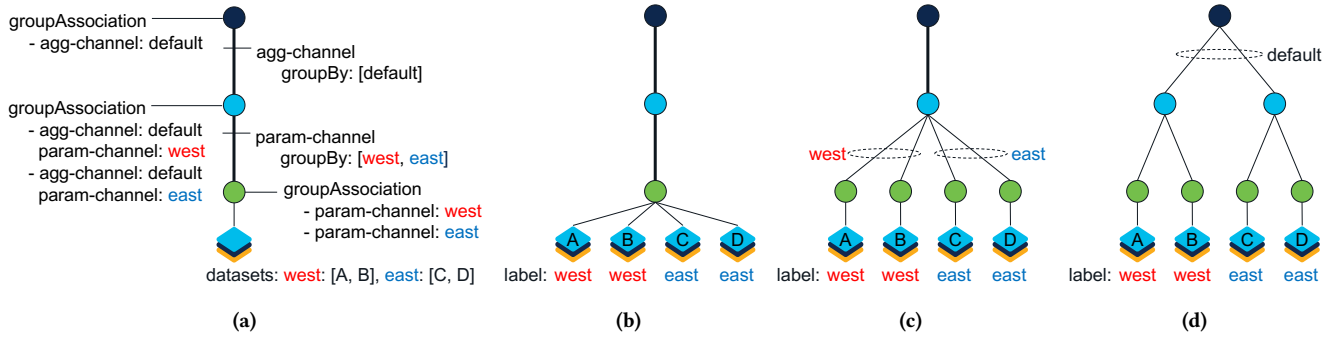
**Role.** An executable worker unit carrying out a specific task in an ML job is defined as a role. Depending on the topology, the task and behavior of a role can vary. For instance, a training worker in FL uses data to build a local model and sends model updates to an aggregation worker, which combines them to create a global model. In hierarchical FL, the aggregator may forward the aggregated model to a global aggregation worker. By exploiting the uniqueness of the tasks associated with these workers, Flame is able to abstract their behavior in the learning process and assigns them roles. This forms the foundation of Flame's flexible and extensible system. These roles are associated with programs that are defined at the job composition stage. The programs are made up of a set of functions such as train, evaluate, load data, and get/distribute model updates, based on the role's responsibility. The program also contains information about the functions execution, as described in Section 5.

The flexible binding between role and program allows Flame to be extensible and support different mechanisms under different topologies. In order to express the specific requirements associated with the deployment of a worker, each role has three attributes: `isDataConsumer`, `groupAssociation` and `replica`. A TAG is expanded into its physical topology by leveraging these attributes, as discussed in §4.3. In order to distinguish training nodes from other components, the boolean attribute `isDataConsumer`, is used to indicate whether a particular role consumes data or not.

In case of topologies like H-FL and hybrid, which involve grouping different nodes to form a cluster, the association of a worker instance (of a role) with channels and their respective groups is determined by the `groupAssociation` attribute (more on channel below). Workers from different roles are connected through channel and the `groupAssociation` governs the connectivity between workers of the same role and those belonging to other roles. This attribute contains a list of the following set:  $\{c_1:g_1, \dots, c_i:g_i\}$  where  $c_i$  is the name of channel  $i$  and  $g_i$  is a group in the channel; an example of this attribute is shown in Figure 5a. The size of the list corresponds to the number of workers for the given role.

Finally, to ensure failure protection or load-balancing, the `replica` attribute is provided to determine the number of workers (i.e., the number of instances) assigned to a particular role. These replicated workers possess identical properties and configurations, and conduct the same type of work. The way they are coordinated however determines the amount of work in each instance. This attribute is particularly valuable for distributing aggregation work among aggregators.

**Channel.** It is an abstraction that links a pair of roles in the TAG and facilitates the exchange of data between them



**Figure 5: Expansion of a TAG into a physical topology.** (a) TAG representation of hierarchical FL (H-FL); each dataset belongs to a group; (b) dataset is expanded; (c) one training worker is allocated per dataset and the worker's group is determined by the dataset's group; and (d) each entry in `groupAssociation` corresponds to one aggregation worker; hence, two aggregators are created; each belongs to a different group for param-channel while both have one "default" group for agg-channel. Since there is one entry for `groupAssociation` in the global aggregation role, expansion finishes by creating one global aggregation worker. Note that the expansion of roles can be done in an arbitrary order since `groupAssociation` has all the necessary information for expansion. A channel's attributes (e.g., `groupBy`) are used to validate the expansion.

through a communication channel. This design choice enables Flame to offer precise control over the communication backend used for each channel, by specifying the desired communication protocol or messaging service. This is in contrast to other systems which allow for configuration of a single backend across the entire ML job, and facilitates the design of efficient ML jobs tailored to the user's specific requirements and resource availability.

Channel has three key attributes: `groupBy`, `funcTags` and `backend`. The `groupBy` attribute is responsible for grouping roles that are connected through the channel. Currently, Flame utilizes a label-based approach, but the attribute can be easily extended to support customized grouping algorithms.

The `funcTags` attribute maps the end-points of the channel to the functions within the connected roles. The attribute maintains key-value pairs where key is a role and value is a list of functions exposed by the application code associated with the role. For instance, in the channel between trainer and aggregator, the `funcTags` has "fetch" and "upload" functions for the training role. Similarly, the attribute has "distribute" and "aggregate" functions for the aggregation role. This attribute helps avoid any ambiguity in identifying which functions to execute on a specific channel when a role is connected to multiple channels.

The `backend` attribute is used to determine the communication protocol for a channel. Users may choose to store their datasets in the cloud of one or multiple providers. Or, they may choose to keep their datasets across different regions of the same provider's cloud. In such cases, co-location of datasets naturally leads to co-location of trainers; and using one type of backend may result in inefficiency (e.g., MQTT traffic over WAN via a broker), increased complexity

(e.g., multi-broker setting for MQTT or complex configuration updates for firewall, ACL and reverse proxy in case of non-broker communication protocol such as gRPC) or both. By allowing per-channel backend, these limitations can be mitigated. We show the utility of this attribute in §7.2.

## 4.2 Resource Annotation for Deployment

In an FL job, resources such as compute and dataset are required but may not be owned by the user deploying their learning tasks. To enable deployment of FL jobs under such situations, Flame allows resource owners to independently register and annotate their resources, which can then be used as needed by the learning job. This approach effectively decouples the infrastructure dependency from any learning tasks and provides greater flexibility in resource usage.

**Compute Access.** Most of the current systems assume that compute nodes are managed through a single provider or utilize a single cluster management tool. However, Flame distinguishes itself by providing an integration service that supports various resource orchestration managers and allows users to register their own cluster (§6.1). During the compute registration phase Flame receives detailed information about computing clusters, including geographical boundaries, resource capacity, and resource types. This information can then be utilized by users when submitting ML jobs or by dataset owners to establish accessibility boundaries for their workers or datasets respectively.

**Metadata for dataset.** For ML training, datasets need to be associated with a configuration for training workers to consume. Flame requires data owners to independently register metadata information with the system, which includes the `realm` and `url` of the dataset. The `realm` attribute plays a

crucial role in defining access restrictions for the dataset. For instance, in order to comply with data privacy regulations like GDPR [35], the dataset owner can utilize information on cluster geographical boundaries to restrict accessibility to specific regions. This design ensures compliance with security and privacy policies associated with the dataset, empowering data owners to maintain control over its accessibility and deployment while allowing others to utilize it. It is important to note that Flame only stores metadata information, not the raw dataset whose location is pointed by the *url*. Upon registering a dataset, Flame assigns a unique *meta-dataID* to it, which is used by the users when submitting ML jobs (§4.3). When users provide a job specification, they can use a *datasetGroups* attribute to combine different datasets into separate groups. This is then used by Flame to map and incorporate the dataset information prior to TAG expansion. To illustrate this, consider the example in Figure 5a, where a user has formed two groups "west" and "east", each consisting of two distinct datasets: (A, B) and (C, D), respectively.

As part of our design, we deliberately separated the infrastructure from the programming logic to facilitate a more organized approach to managing an FL job. By doing so, users can focus solely on composing their ML job, without worrying about the coupling between compute node and dataset. Without this design choice, developers would be unable to complete the composition of an FL job until a data owner makes a dataset available on a compute node. This "human-in-the-loop" model would significantly slow down the composition and deployment of an FL job. Flame, on the other hand, allows for the automatic acquisition of a compute node and access to a dataset, streamlining the process.

### 4.3 TAG Expansion

Algorithm 1 shows the TAG expansion pseudocode. The algorithm expands the abstract representation into a physical deployment topology by creating workers based on the specifications in roles and channels.

The top-level function, *Expand* walks through roles (line 6) and calls *BuildWorkers* for each role. Then, *BuildWorkers* creates the worker configuration. The specification for each role is self-contained. Thus, there is no particular order to iterate roles. If a role is a data consumer, the function iterates on datasets for the role, creates one worker configuration per dataset (lines 15-22) and uses the *datasetGroups* to determine the group. Otherwise, the function takes *groupAssociation* values of role *r* and creates the corresponding worker (lines 24-30). During the expansion, if *replica* is set for a role (not a data consumer), the algorithm creates copies of the role (line 27). Those copies share the same properties (e.g., channel's group). For instance, the topology shown in Figure 1d is implemented by using *replica*. The channel information

---

#### Algorithm 1: TAG expansion

---

```

1 Function Expand(J):
   // J: job specification
2   W ←  $\phi$  // W: a total list of workers
3   if PreCheck(J) is false then
4     return  $\phi$ 
5   R ← GetRoles(J) // R: roles
6   for r ∈ R do
7     X ← BuildWorkers(r, J)
8     W ← W ∪ X
9   if PostCheck(W, J) is false then
10    return  $\phi$ 
11  return W

12 Function BuildWorkers(r, J):
13  W ←  $\phi$ 
14  if r is a data consumer then
15    G ← GetGroupsOfDataSets(r, J)
16    for g ∈ G do
17      D ← GetDataSets(g)
18      for d ∈ D do
19        m ← GetComputeId(d)
20        a ← GetGroupAssocByGroupName(r, g)
21        w ← CreateWorkerConfig(r, m, a)
22        W ← W ∪ {w}
23  else
24    GA ← GetGroupAssociations(r)
25    for a ∈ GA do
26      c ← GetReplicaNum(r)
27      for i = 0; i < c; i++ do
28        m ← DecideComputeId(a)
29        w ← CreateWorkerConfig(r, m, a)
30        W ← W ∪ {w}
31  return W

```

---

is used in pre and post checks to validate the correctness of the TAG and expanded physical deployment.

**Example.** Figure 5 demonstrates the application of Algorithm 1 to expand the high level TAG for H-FL (Figure 5a) to a physical deployment topology. To begin, we associate all datasets in the job specification with the trainer (data consumer) role, resulting in one worker per dataset, as shown in Figure 5b. Then we compare the values of *datasetGroups* and *groupAssociation* to group the training workers into "west" and "east" groups. The next step is to use *replica* and *groupAssociation* associated with the "param-channel" to determine the number of workers required for the aggregator role. By default, *replica* is set to one, unless explicitly stated. In the example, two aggregation workers are created based on the *groupAssociation* values. The same process is applied for the

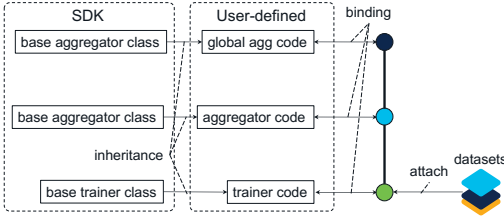


Figure 6: Workload composition for hierarchical topology.

```
from flame.mode.horizontal.trainer import Trainer
class MNistTrainer(Trainer):
    def initialize(self) -> None:
        # Initialize the model
    def load_data(self) -> None:
        # Describe operation to handle data
    def train(self) -> None:
        # Training code
    def evaluate(self) -> None:
        # Testing code
t = MNistTrainer(config)
t.compose()
t.run()
```

Figure 7: Code snippet of user-defined MNistTrainer role to illustrate user programming model. After inheriting a base class (Trainer), user only implements four basic functions: initialize(), load\_data(), train(), and evaluate().

top-level role (global aggregator). Since there is only one value (i.e., *default*) in the *groupAssociation* attribute, a single worker instance is created (Figure 5d).

## 5 PROGRAMMING MODEL

Flame provides two programming models: (1) *user* and (2) *developer*. The *user* programming model is for end users who wish to use the *out-of-the-box* functionalities of Flame to deploy a distributed ML job. The *developer* programming model is intended for developers who want to extend the capabilities of Flame by allowing for different topologies, roles, and training methodologies. The former is useful for those whose needs can be met by the built-in functionalities of Flame while the latter is essential when the built-in features are not sufficient to fulfill the user's needs. Therefore, we refer to users or participants as those who mostly rely on the user programming model whereas we denote developers as those who need more than the built-in features.

**User Programming Model.** The Flame SDK provides a set of base programs (as Python classes). A user builds a job-specific program by implementing a few core functions (e.g., initialize, train, evaluate, etc). The example shown in Figure 6 illustrates the relationship between programs in the Flame SDK and user-defined ones. A user can build the logic for a given role for standard training methodology by inheriting the pre-defined base classes. The base class provides a

```
class Trainer(Role, metaclass=ABCMeta):
    def compose(self) -> None:
        with Composer() as composer:
            self.composer = composer
            tl_load = Tasklet("load", self.load_data)
            tl_init = Tasklet("init", self.initialize)
            tl_train = Tasklet("train", self.train)
            ... ..
            tl_copy = Tasklet("snapshot", self.snapshot)
            loop = Loop(loop_check_fn=lambda: self._work_done)
            tl_load >> tl_init >> loop(tl_get >>
            tl_train >> ... >> tl_copy >> ...)
```

Figure 8: Code snippet that illustrates a composer mechanism via developer programming model. Tasklet accepts *alias* as the first argument to ease the modification of a tasklet chain.

basic workflow for a certain role (such as trainer, intermediate and global aggregator), allowing the user to focus on implementing relevant core functions for their learning task. For instance, for a hierarchical FL (H-FL) topology user can define their custom MNistTrainer as illustrated in Figure 7, by inheriting the *out-of-the-box* base class provided the SDK. Implementing the aggregator role is simpler than the trainer role as Flame's core library includes essential functions like distribute and aggregate. If aggregator roles need to perform validation tests, users can additionally implement the load\_data and evaluate functions.

**Developer Programming Model.** Flame is designed to provide extensibility to support different FL topologies. To achieve this, Flame allows developers to extend or create different roles and accommodate other state-of-the-art learning approaches. Internally, each worker executes the functions in the program associated with its role. In Flame, those functions are specified as an execution unit called tasklet<sup>2</sup>. Tasklets are combined together to finish a worker's task. Inspired by workflow management solutions [12], Flame offers functionality to structure a worker's task as a collection of tasklets and present it as a workflow. Since an ML job typically consists of repeating tasklets, the workflow-like approach helps to formalize the development process of any ML mechanisms, thereby facilitating fast development. To create a workflow, Flame overrides the right shift (>>) operator and provides a composer so that various tasklets can be chained together. An additional Loop primitive, allows repeated execution of chained tasklets until an exit condition is met. This methodology provides easy extensibility for a developer to create standalone tasklets such as taking a snapshot of the model, as shown in Figure 8, or to record various metrics after each step and link them in the workflow.

In addition to its core features, Flame offers a convenient set of API functions through the composer and tasklet

<sup>2</sup>It is to imply that the execution unit is small; it's not one in Linux kernel.



Function	Module	Note
<code>get_tasklet(<i>alias</i>)</code>	composer	Return a tasklet of <i>alias</i>
<code>insert_before(<i>tasklet</i>)</code>	tasklet	Insert <i>tasklet</i> before a tasklet
<code>insert_after(<i>tasklet</i>)</code>	tasklet	Insert <i>tasklet</i> after a tasklet
<code>replace_with(<i>tasklet</i>)</code>	tasklet	Replace <i>tasklet</i> with a tasklet
<code>remove()</code>	tasklet	Remove itself from a chain

Table 1: Composer and Tasklet API.

Function	Note
<code>join()</code>	Join channel and allocate resources for the channel
<code>leave()</code>	Leave channel and deallocate its resources
<code>send(<i>end</i>, <i>msg</i>)</code>	Send <i>msg</i> to <i>end</i>
<code>recv(<i>end</i>)</code>	Receive a message from <i>end</i>
<code>recv_fifo(<i>ends</i>, <i>k</i>)</code>	Receive a message from the first <i>k</i> ends in a list of <i>ends</i> in a FIFO manner
<code>peek(<i>end</i>)</code>	Peek a message from <i>end</i>
<code>broadcast(<i>msg</i>)</code>	Broadcast <i>msg</i> to all the peers at the other end of channel
<code>ends()</code>	Return a list of peers at the other end of channel filtered by a chosen peer selection logic
<code>empty()</code>	Check if peers exist at the other end of channel

Table 2: Channel API.

modules, which are detailed in Table 1. These APIs enable developers to make surgical modifications to the tasklet chain and to quickly develop new functionalities. With class inheritance, the need to re-chain all the tasklets in the child class is avoided, and only a new tasklet is required for the new functionality. This approach reduces redundant lines of code, avoids core library changes, and reduces the risk of introducing bugs, as shown in §7.1.

Finally, in order to ensure compatibility with various communication backends, such as MPI, MQTT, Kafka, and gRPC, Flame’s SDK separates the ML logic from the communication layer. This is achieved by providing a *channel manager* interface with a standardized set of APIs, as exemplified in Table 2, which can be utilized by any two connected roles. This abstracted interface for the communication backend allows roles to send and receive messages uniformly regardless of the underlying protocol. Consequently, this not only enhances the flexibility of Flame, but also provides developers with a consistent means of interacting with the system, irrespective of the chosen communication backend.

## 6 MANAGEMENT PLANE

The management plane of Flame is responsible for managing the lifecycle of FL jobs. Users can create, update, submit and

monitor their jobs via a REST API in the management plane. The complete specification can be found in our repository<sup>3</sup>.

### 6.1 System Components

The management plane consists of the following components: APIserver, controller, notifier, deployer, and agent.

**APIserver.** The APIserver is a front end that exposes the REST API. A CLI tool uses the REST API and allows users to interact with the management plane.

**Controller.** The controller is the core unit in the management plane. It has three primary responsibilities. (i) It processes requests from users and other system components (e.g., agent and deployer) and manages the state via a database. (ii) It performs TAG expansion into a real topology, and interacts with compute cluster managers, such as Kubernetes, for worker deployment and compute resource provisioning/decommissioning. (iii) Finally, it monitors the job’s progress and events such as worker failure.

**Deployer.** The deployer is an integration interface service, which provides abstraction to integrate different compute orchestration solutions such as Kubernetes, Docker Swarm, Apache Mesos, etc. By implementing the APIs defined in the deployer’s interface, Flame can interact with any compute orchestrator. In each compute cluster, the deployer can generate requests for resource allocation and instance (typically in the form of a container) creation, based on instructions received from the controller.

**Agent.** A learning job in Flame consists of multiple roles executing tasks. Each instance of a role in a cluster includes a thin client called agent. The agent is responsible for managing the lifecycle of a task in a given job via the APIserver. The agent fetches the ML code associated with the role, the channel configuration and meta information on the dataset, all of which are needed by a worker to carry out a task. Once obtained, the agent starts the training by executing a worker role as a child process. It monitors the worker’s status and regularly informs the controller. The agent also fulfills the controller’s commands such as task termination.

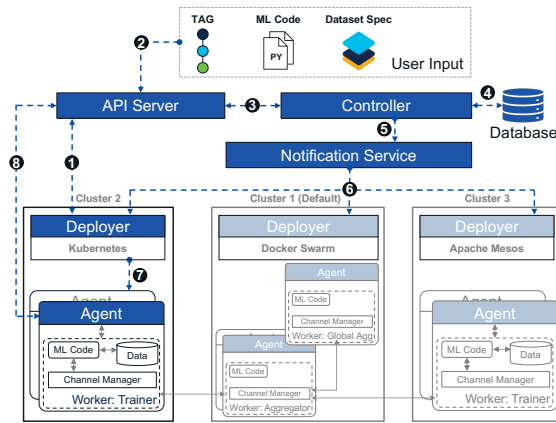
**Notifier.** It is a service that allows the controller to push events to agents and deployers. The notifier enables event-driven management of FL jobs because the agents and deployers maintain an active connection with the notifier.

### 6.2 Workflow

In Figure 9 we describe how Flame is used to register the available compute infrastructure and datasets and to deploy a distributed ML job across those resources.

**Compute Registration.** In order to register a compute cluster, the cluster admin is required to support the Flame’s *Deployer* interface in their cluster. By default, Flame already

<sup>3</sup><https://github.com/cisco-open/flame/tree/main/api>



**Figure 9: Flame architecture and workflow overview.**

integrates the Deployer interface for Kubernetes (K8s). During the bootstrap of the deployer, the admin also assigns a name and provides properties associated with the cluster. Once the deployer is up, it uses a REST API call to register the cluster with Flame ①. This is one time process and the admin has full control of the resources provided by the cluster that can be used by Flame, which implies that the admin can deregister their cluster if needed.

**Job Configuration.** To submit an FL job to Flame, the user needs to provide a job configuration that consists of three main components: (i) a TAG-based high-level abstract description of the machine learning job, (ii) program logic associated with each role, and (iii) data specification configuration containing metadata information about the datasets, which provides deployment constraints. The job is submitted to the system through `APIServer` ②.

**Job Deployment.** Upon receiving the job configuration, it is shared with the controller ③. The controller records this information in the database ④, and expands the TAG to determine where each role should be created, based on the metadata: *datasetGroups*, *groupBy* and *groupAssociation* attributes. The controller then sends a compute creation event to the notifier along with the job information ⑤. The notification service notifies the corresponding deployers where roles need to be created ⑥. Upon receiving such a request, each deployer creates a compute (e.g., a pod in K8s) that contains an agent ⑦. The agent uses the job id to retrieve the code and task configuration files ⑧. It then starts a worker which executes an FL task. Once the task is completed, the agent updates its status via the APIServer. The deployers are subsequently notified through a revoke deploy event to de-allocate the resources from the compute clusters. Our system manages FL jobs in a fully automated manner.

### 6.3 Implementation

We have implemented the management plane of Flame with 12K LOC in Golang while the Flame SDK was developed

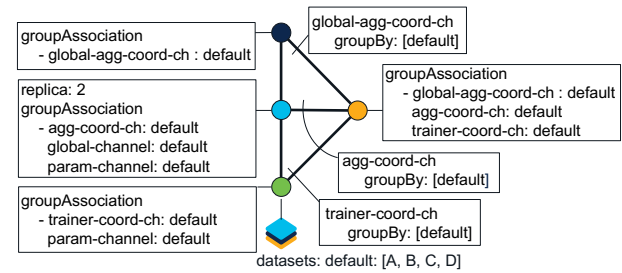






Figure 10: TAG for Coordinated FL (H-FL with coordinator). Only additional changes are shown in the figure on top of the configuration shown in Figure 5a. TAG is expressed in the YAML format. : global aggregator, : aggregator, : trainer, and : coordinator. The expanded form is shown in Figure 1d.

with 7K LOC in Python. The current implementation supports a diverse range of topologies as shown in Table 6. Flame also provides an emulation platform called **Flame-In-A-Box** (fiab). It is a single machine management plane that leverages minikube, a local Kubernetes cluster. All of the system components are deployed as pods on minikube. This single box deployment of our system allows users to easily validate their prototypes of new FL mechanisms and algorithms or to conduct small scale experiments. Moreover, packaging the system components as pods makes the management plane deployment portable, enabling it to be easily deployed in a real world Kubernetes cluster.

## 7 EVALUATION

To demonstrate the flexibility and extensibility of Flame in supporting various FL topologies, our evaluation aims to achieve the following objectives.

- §7.1 demonstrates how developers can extend a sample H-FL topology (as shown in Figure 5a) for complex settings by incorporating a coordinator.
- §7.2 highlights the advantages of selecting communication backends, emphasizing the system’s flexibility.
- §7.3 illustrates the ease of transforming an FL job between different topologies.
- §7.4 presents micro-benchmarking results that quantify the overhead of TAG expansion.
- §7.5 provides quantitative and qualitative comparison of Flame and existing systems.

### 7.1 Extension for New Mechanisms

The *developer programming model* and TAG mechanism of Flame facilitate the extension of topologies and the addition of new mechanisms. An example of topology extension is illustrated in Figure 1d, which shows an H-FL topology with a coordinator. In this paper, we refer to this variant as Coordinated Federated Learning (CO-FL). CO-FL differs from H-FL in two key aspects: (1) the links between aggregator and

```

name: A CO-FL TAG example
roles:
- name: coordinator
  groupAssociation:
  - global-agg-coord-ch: default
    agg-coord-ch: default
    trainer-coord-ch: default
...
- name: aggregator
  replica: 2
  groupAssociation:
  - agg-coord-ch: default
    param-channel: default
    global-channel: default
- name: trainer
  isDataConsumer: true
  groupAssociation:
  - trainer-coord-ch: default
    param-channel: default
channels:
- name: global-agg-coord-ch
  pair: [global-aggregator, coordinator]
  groupBy:
    type: tag
    value: [default]
  funcTags:
    global-aggregator: [coordinate]
    coordinator: [coordinateWithTopAgg]
...
- name: param-channel
  groupBy:
    type: tag
    value: [default]
  pair: [aggregator, trainer]
  funcTags:
    aggregator: [distribute, aggregate]
    trainer: [fetch, upload]

```

**Figure 11: TAG representation for CO-FL topology shown in Figure 10 in the YAML format. Some blocks of the configuration are omitted for brevity.**

trainer form a bipartite graph in CO-FL, and (2) the coordinator is connected to the rest of the roles. Enabling this new variant requires three types of modifications: (i) an update to the TAG, (ii) an update to the implementation of roles in the H-FL TAG to allow communication with the coordinator, and (iii) dataset group update in the dataset specification.

**TAG Changes.** In Figure 10, we illustrate the modifications required to integrate a coordinator into the H-FL topology depicted in Figure 5a. The corresponding TAG representation is illustrated in Figure 11. The transformation process entails modifying only 46 lines of configuration. The majority of the changes (36 lines, 78%) involve configuring new channels for the coordinator, while the rest of the changes are made to existing roles and channels. The addition of a coordinator requires configuring the *replica* attribute to match the number of aggregators (§4.1), and allows for the creation of bipartite-like communication links upon TAG expansion.

```

1 def compose(self) -> None:
2     super().compose()
3
4     with CloneComposer(self.composer) as
       composer:
5         self.composer = composer
6         tl_coord_ends = Tasklet("get_coord_ends",
           self.get_coord_ends)
7
8         tl = self.composer.get_tasklet("distribute")
9         tl.insert_before(tl_coord_ends)

```

**Figure 12: Code snippet for global aggregator for CO-FL.**

<pre> name: Dataset A url: someurl/data_A.npz realm: cluster1 isPublic: true </pre>	<pre> - role: trainer   datasetGroups:     default: [A, B, C, D] </pre>
---	---

(a) An example dataset

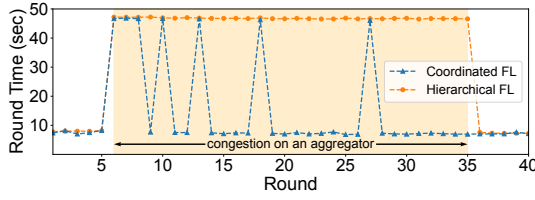
(b) Dataset specification

**Figure 13: Metadata information used in mapping dataset to a correct role and dataset group.**

**Code Changes.** Following the completion of TAG, the next step is to implement each role in the TAG. Flame’s developer programming model allows easy extension without the need for modifying the core library. The developer inherits the base classes of H-FL and implements additional functionality for the coordinator role. In CO-FL, while the global aggregator performs the same steps as it does in H-FL, it must receive information from the coordinator about which intermediate aggregators to send and receive model weights as not all intermediate aggregators may be involved in the training for every round.

Such a functionality, implemented as a *get\_coord\_ends* tasklet, is updated into an inherited tasklet chain of the global aggregator. As shown in Figure 12 (lines 8 and 9), the tasklet is introduced before the *distribute* call using the API in Table 1. We obtain tasklets by using their alias and call appropriate operations (e.g., *insert\_before*, *remove*). Other roles are implemented in a similar fashion, resulting in minor code revisions for the CO-FL implementation, as shown in Table 5.

**Dataset Group Update.** Data owners can register their datasets by providing metadata information. The dataset registration process is independent of composing a job. An example for the metadata is shown in Figure 13a. Upon registration, Flame produces an ID for the metadata. Assuming that datasets are already registered in the system with their IDs as *A*, *B*, *C* and *D*, ML practitioners can use the IDs and group the datasets to different dataset groups. In the H-FL example (Figure 5a), *A* and *B* belonged to the *west* group and *C* and *D* belonged to the *east* group. The modification to support the example of Figure 10 is to group the four datasets into the default group for the trainer role (Figure 13b).



**Figure 14: Performance comparison between Coordinated FL vs Hierarchical FL. Coordinated FL manages the network congestion with its load-balancing scheme.**

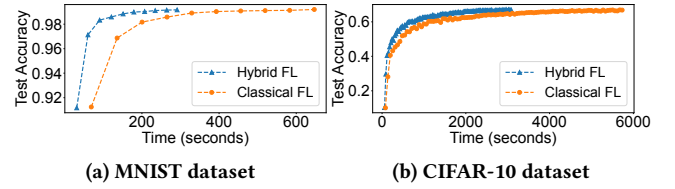
**Setup.** To illustrate the feasibility of CO-FL extension, we implement a toy scenario with 10 trainers, two aggregators, and a coordinator, where a link between the global aggregator and an aggregator becomes a bottleneck over multiple rounds. The coordinator identifies and exclude a slow aggregator as the delay is reported.

**Result.** Figure 14 compares the results of such a scenario with H-FL. In CO-FL, from round #6, the coordinator detects a delayed aggregator based on model upload times. After three consecutive rounds of delays, the slow aggregator is disabled for one round (#9), checked for delay in round #10, and, if delayed, disabled for two rounds (#11-12). As the delay persists, the coordinator disables the straggler in a binary-backoff manner. Without the coordinator, H-FL suffers extended per-round times due to the straggler.

## 7.2 Flexible Backend

We demonstrate the versatility of Flame’s backend configurations and their implications by implementing hybrid FL [13] (Figure 1e). Hybrid FL is good for scenarios where trainers are co-located and network bandwidth among trainers are much higher than bandwidth between trainers and the aggregator. In hybrid FL, instead of individual trainers sending their model updates, co-located trainers form clusters and share their model updates with the aggregator. Within each cluster, trainers utilize the ring-allreduce algorithm [37] to exchange their model updates, resulting in a single copy of the cluster-level model being shared with the aggregator.

**Setup.** In contrast to frameworks limited to a single backend configuration, Flame offers the flexibility to configure the TAG with multiple distinct backends. The hybrid FL example uses a gRPC backend (P2P) for transferring model updates between trainers within the cluster, and an MQTT backend for communication between the aggregator and different clusters. We create a hybrid topology that consists of 50 trainers and emulate different bandwidth on each backend, by utilizing the Linux `tc` tool. We used the MNIST [8] and CIFAR-10 [20] datasets for the experiments, training a 2-layer CNN model and a ResNet-18 model respectively. The trainers are equally divided into five groups and a trainer within one of the cluster is chosen as a straggler where its bandwidth is configured to 1 Mbps for the MNIST and 10 Mbps for the



**Figure 15: Performance comparison between Hybrid FL vs Classical FL. Flame’s flexibility in communication backend selection demonstrates the efficacy of Hybrid FL.**

CIFAR-10 experiment. While all the other gRPC backend is given a maximum bandwidth of 100 Mbps. Additionally, for comparison, we set up a C-FL topology using MQTT as the backend with 50 trainer nodes, where one trainer is designated as a straggler.

**Result.** Figure 15 shows the test accuracy over wall-clock time, where each point represents a round. The results suggest that hybrid FL converges faster than C-FL, by achieving  $2.21\times$  and  $2.01\times$  speedup in reaching 0.985 and 0.650 accuracy in MNIST and CIFAR-10 datasets. This is primarily because hybrid FL allows non-straggling trainers to efficiently upload cluster-aggregated weights to the global aggregator. In contrast, in the case of C-FL, the whole system must wait for the straggler’s weight upload to the global aggregator. Hybrid FL also consumes less bandwidth compared to C-FL to upload model updates (25 MB vs 250 MB/round for MNIST and 223 MB vs 2230 MB/round for CIFAR-10). This experiment shows that Flame allows flexible communication backend configurations and such flexibility can help design and evaluate new FL approaches easily and rapidly.

## 7.3 Topology Transformation: User Perspective

The requirements and constraints for the ML job may change over time, which may require changes in learning topology. To demonstrate how easily these transformations can be made, we walk through the steps of transforming from one topology to another, starting with a basic C-FL topology. Note that these transformations are done offline and we leave the dynamic reconfiguration of topology for future investigation. The transformation steps presented here only involve user application code and TAG changes. In contrast, §7.1 discussed how to extend the logic for new topology.

**Classical→Hierarchical.** C-FL topology consists of trainers and global aggregator. To transform from C-FL to H-FL, a user needs to introduce an (+) aggregator role, a new connecting (+) channel with the new aggregator. Finally, to allow the grouping of trainer nodes the ( $\Delta$ ) `datasetGroups` attribute in metadata information is updated. These modifications each require only a change of up to 16 lines of code (LOC),



	C-FL	C-FL→H-FL	H-FL→H-FL <sup>b</sup>	C-FL→Distributed	C-FL→Hybrid	H-FL→CO-FL
Code	+ trainer + global-agg	+ agg (25 <sup>†</sup> )	N/A	- global-agg Δ inheritance (1)	Δ inheritance (2)	+ coordinator (13 <sup>†</sup> ) Δ inheritance (3)
TAG	+ channel	+ channel (16+N) + role (3+2N) Δ groupAssociation (N)	Δ groupBy (N) Δ groupAssociation (N)	Δ channel (2)	+ channel (12+N) Δ groupAssociation (2N)	+ replica (1) + role (5) + channels (42) Δ groupAssociation (3)
Metadata	+ init info	Δ datasetGroups (N)	Δ datasetGroups (N)	N/A	Δ datasetGroups (N)	Δ datasetGroups (1)

**Table 3: Changes required to transform from one topology to another. Given TAG in the YAML format, required lines of code changes are shown in parentheses.  $N$  denotes the number of groups. The TAG representation of C-FL, H-FL, Distributed and Hybrid is showcased in Figure 4 while TAG for CO-FL (Figure 1d) is shown in Figure 10. H-FL<sup>b</sup> represents H-FL topology with different grouping options. +, - and Δ represent addition, removal and update respectively and N/A indicates no change. <sup>†</sup>: only needs template code. “Δ inheritance” implies the switch of a base class from one to another.**

Topology	Task	Number of Workers					
		1	10	100	1,000	10,000	100,000
Classical FL	Expansion	0.005	0.006	0.036	0.329	3.183	31.990
	DB Write	0.007	0.008	0.037	0.315	2.781	27.971
Coordinated FL	Expansion	0.006	0.012	0.041	0.320	3.190	32.538
	DB Write	0.033	0.035	0.061	0.317	2.901	27.232

**Table 4: TAG expansion latency in seconds.**

with additional updates that varies on the number of newly introduced aggregators.

**Classical→Distributed.** In FL, trainers send their model weights to the aggregator while in distributed learning they are shared among all the nodes directly. Flame SDK provides a separate trainer base class for federated and distributed learning. Thus, from the user’s perspective, C-FL to distributed training change requires, removing the global aggregator, (Δ) updating the inherited base trainer class (1 LOC), and (Δ) altering TAG representation (2 LOC) where the trainer-aggregator channel is updated to trainer-trainer channel as shown in Figure 4b.

**Classical→Hybrid.** Transformation from C-FL to hybrid topology entails two steps: First, it would require (Δ) updating the inherited trainer and global aggregator class. Again, the Flame SDK provides base classes for hybrid topology, thus, a user just needs to change the inherited parent class name in the trainer and global aggregator role’s program (1 LOC each). Then, it needs (Δ) to change the TAG to create appropriate channels with backends and change *groupBy* and *datasetGroups* to group co-located datasets, which only requires up to 12 LOC change with additional updates on varying number of trainer groups.

**Hierarchical→Coordinated.** CO-FL is different from H-FL in that a coordinator oversees a federated learning process. Therefore, in CO-FL, a user needs to introduce the coordinator (13 LOC), (Δ) update the inheritance of classes for the global aggregator, aggregator, and trainer (1 LOC each), add coordinator role (5 LOC), and add new channels between the coordinator and the rest of the roles (14 LOC each). In

addition, grouping between aggregators and trainers can be dynamic based on the coordinator’s logic. For that, the user (Δ) updates *datasetGroups* as a single group (1 LOC) and configures *replica* in the TAG (1 LOC). Note that the reported LOC is for user application code, not the logic for CO-FL.

## 7.4 TAG Expansion Overhead

TAG expansion is the first step in the management plane for deployment preparation. Deployment time varies based on factors like network bandwidth, cluster resources, and job size; our evaluation focuses on TAG expansion overhead rather than deployment issues in geo-distributed scenarios. We conducted experiments to measure the latency of TAG expansion and database write of its results on Flame for two FL topologies (C-FL and CO-FL) by varying the number of trainers. CO-FL was configured with 100 replicas and a coordinator. The results shown in Table 4 demonstrate that the overhead of TAG expansion on Flame is comparable across the two FL topologies. The results also show that Flame is highly scalable, achieving TAG expansion on 100,000 trainers under a minute for both FL topologies. The current implementation can be further optimized since it only uses a single CPU core and data is duplicated during the expansion.

## 7.5 System Comparison

**7.5.1 Comparison with FedML.** FedML uses client-server architecture which limits the system’s ability to be extensible and support flexible topologies. FedML provides native support for C-FL and H-FL ( $n = 1$ ) and wraps the underlying implementation of model, data loading and component logic as part of the core library. Unlike Flame, that allows a user to define their own model and data loader through the *programming model*; users in FedML provide the model and data information through configuration files and introduction of new data loader/model or component requires changes in the core library. Note that, to support H-FL, FedML chooses

Topology	System	Global Agg.	Agg.	Trainer	Coordinator	Core Lib Changes
C-FL*	Flame	231	—	156	—	✗
	FedML	577	—	319	—	✗
H-FL*	Flame	0	200	0	—	✗
	FedML	0	0	68	—	✗
N-level H-FL	Flame	0	0	0	—	✗
	FedML	0	190	0	—	✓
CO-FL	Flame	40	67	73	158	✗
	FedML		Not Achieved (NA)			✓

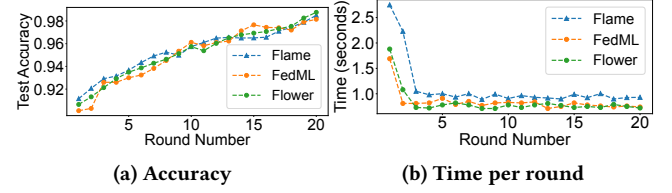
**Table 5: Comparing development effort in number of lines of code between Flame and FedML to enable new topologies by adding/modifying roles. H-FL extends C-FL while  $n$ -level H-FL and CO-FL extends H-FL. \*: natively supported.**

to modify the training client manager class with the appropriate functionality, and a hardcoded method to allow a worker to distinguish whether it needs to act as a trainer or a middle aggregator. This is philosophically different from Flame where we implement a new middle aggregator role and keep the global aggregator and trainer untouched (see LOC changes for H-FL in Table 5).

To further compare the flexibility provided by FedML and Flame, we leverage the native C-FL and H-FL implementations in FedML and extend them to support two new topology (i)  $n$ -level H-FL where  $n$  represents the number of intermediate aggregators, and (ii) CO-FL as described in §7.1. Table 5 illustrates the effort in terms of LOC required to implement different topologies.

**$n$ -level H-FL.** Extending the base H-FL to add new intermediate aggregators in Flame is trivial. It requires changes only in the TAG to introduce new aggregator roles and 0 LOC changes for role’s code. FedML on the contrary, requires significant code changes 190 LOC, across multiple files which are part of the core-library. These changes are required to overcome three limitations. First, the native `client manager` can only synchronize model weights outside its silo (group) with the global aggregator. In  $n$ -level H-FL, the  $n - 1$  level intermediate aggregators synchronize weights with aggregators in different silos, while only the top-most middle aggregator level synchronizes weights with the global aggregator. Secondly, hardcoding in `client manager` for communication with the global aggregator needed to be removed, as the native H-FL ( $n = 1$ ) implementation enforced all middle aggregators to share updates to the global aggregator alone. Thirdly, while the native implementation provided means to organize trainers in data silos, we needed to implement a new configuration mechanism to allow grouping of different aggregators in the  $n$ -level H-FL topology.

**CO-FL.** Implementing CO-FL requires changes in the H-FL TAG configuration and code updates associated with different roles as described earlier and shown in Table 5. However, implementing such a topology using FedML requires more fundamental implementation changes. In case of CO-FL the



**Figure 16: Performance comparison with FedML and Flower.**

coordinator assigns a trainer to intermediate aggregator on the fly. However, FedML uses a torch [36] backend and its process-groups for model update synchronization, and this makes it difficult for FedML to be easily extensible to CO-FL. This is because process-groups are created per silo, and weights are broadcast within the group. Thus, separating out trainers into different groups on the fly is not trivial. It requires significant development effort to support CO-FL in FedML and is marked as not achieved (NA) in Table 5.

**Performance.** Finally, we compare Flame with FedML in terms of accuracy and per-round execution time under the C-FL topology with the MNIST dataset. We also add Flower’s results as another reference point. Figure 16 demonstrates that all of them achieve comparable performance.

**7.5.2 General Comparison.** We compare other FL frameworks by considering their native support for different topologies and support for communication backend between components. For brevity, we focus on highlighting the key differences among FedML, Flower, and FedScale, as presented in Table 5, while omitting other differences with respect to training and aggregation algorithms, client selection algorithms, or supported monitoring capabilities.

Out of the nine listed topologies, Flame natively supports eight of them, whereas the other frameworks only support a few of them. Although Flame does not natively support vertical FL topology, it can be easily implemented due to its extensibility, as demonstrated through the CO-FL use case. However, extending FedML to accommodate additional topologies presents challenges, as discussed earlier. Similar additional effort is required to build other topologies in the other frameworks.

Another crucial distinction between Flame and the other frameworks is that Flame supports different communication protocols on a *per-channel* basis. Both Flame and FedML facilitate integration of different communication protocols, such as MQTT and gRPC, while the other frameworks only support gRPC. The logical graph abstraction employed by Flame breaks down the connections between different roles (workers) into channels, enabling per-channel communication control for any FL topology. In contrast, the other frameworks enforce the use of the same communication backend for all connections between nodes within a job.

Feature	Flame	FedML [17]	Flower [2]	FedScale [21]
C-FL [31]	✓	✓	✓	✓
H-FL [28]	✓	✓	✗	✗
N-level H-FL [43]	✓	✗	✗	✗
Hybrid [13]	✓*	✗	✗	✗
CO-FL [3]	✓*	✗	✗	✗
Vertical [16]	✗	✓	✗	✗
Async H-FL	✓	✗	✗	✗
Async CO-FL	✓	✗	✗	✗
Distributed [17]	✓	✓	✗	✗
gRPC	✓	✓	✓	✓
MQTT	✓	✓	✗	✗
MPI	✗	✓	✗	✗
NCCL	✗	✓	✗	✗

**Table 6: FL framework comparison in terms of topology and protocol; \*: a simplified version of the original design.**

Finally, support for deploying an FL job is classified into two approaches — compute centric v/s compute agnostic. In Flame, the deployer component enables the system to connect with different compute clusters managed by various resource orchestrators. This capability allows for a *compute-agnostic* approach, where the user provides the ML code and deployment instructions/rules, and the system takes care of locating the appropriate compute units, creating the group, deploying the code, and initiating the learning process. In contrast, other frameworks follow a *compute centric* approach, which requires the participant to select appropriate compute resources based on constraints associated with the data, such as GDPR rules, and to manually create groups in the case of H-FL, thereby making them less extensible than Flame.

## 8 RELATED WORK

**Library.** Machine learning libraries provide lower-level interfaces for concisely expressing models. They provide a collection of pre-built algorithms, functions, and tools for developing, training, and deploying machine learning models. TensorFlow [1], PyTorch [36], and scikit-learn [38] are some of the ML libraries providing lower-level interfaces for concisely expressing ML models, with the ability to create custom models and learning algorithms. These libraries are used to create ML models from the ground up while users need to build their system and integrate it with these models. Flame allows developers to use any such ML libraries.

**Frameworks.** Spark ML [46] and Apache MXNet [5] are open source frameworks mainly for distributed learning. Systems such as Flower [2], FedScale [21], and PySyft [41] provide low level APIs which make them flexible. Unlike Flame, they cannot be easily extended to support different deployment scenarios as they lack suitable abstractions. OpenFL [11] is another FL framework based on a client-server architecture with two components: (1) collaborator, which uses a local dataset to train global models, and (2)

aggregator, which receives the model updates and combines them to create the global model. Nvidia Clara [34] is an application framework specifically designed for healthcare use cases. There are other FL frameworks like FedML, which are based on client-server architecture and lack support for diverse FL configurations, required to express and extend the evolving deployment requirements.

**Simulators.** Machine learning simulators enable quick testing of various machine learning algorithms, models, and techniques in a simulated environment. FedJAX [40] is a research-focused federated learning simulator that provides an API for building and training machine learning models using a variety of federated learning algorithms. Flute [9] is another federated learning simulator that focuses on scalability and efficiency. FLSim [25] is also a federated learning simulator that allows users to explore the effects of different federated learning algorithms and hyperparameters on model performance. Flame does not provide a simulator but it supports small scale emulation via the Flame-in-a-box.

## 9 CONCLUSION

We introduce Flame, a system that enables composability and extensibility for federated learning topologies. It relies on a logical TAG representation of the physical topology that exposes new capability to explicitly specialize the behavior and configuration of individual components in any learning system. Its programming model facilitates easy extensions without requiring any modification of its core library. It also provides basic support that makes it possible to deal with heterogeneous deployment environments.

We open sourced the system to help researchers and developers build FL applications in modular fashion and to accelerate the progress of federated learning. In addition, Flame and its TAG abstraction, can be extended to support other learning architectures, including distributed or collaborative learning [6, 29]. Finally, the current implementation of the system assumes that topology changes are made offline, prior to deploying a job. The flexibility provided by Flame is also a step toward enabling dynamic transitions between topologies in real time, such as in response to failures and load fluctuations, and to provide robustness via integration in CI/CD pipelines; we leave this for future work.

## ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, Rodrigo Bruno, for their constructive feedback. We would like to also thank Gaoxiang Luo for his assistance at the early stage of this project. This research was partially supported by NSF projects CCF-2217070 and CNS-1909769, by funding from Cisco Research, and using experimental infrastructure from the NSF Chameleon Cloud testbed.

## REFERENCES

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.
- [2] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchu Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Kwing Hei Li, Titouan Parcollet, Pedro Porto Buarque de Gusmão, and Nicholas D. Lane. 2020. Flower: A Friendly Federated Learning Research Framework. <https://arxiv.org/abs/2007.14390>
- [3] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, et al. 2019. Towards federated learning at scale: System design. *Proceedings of Machine Learning and Systems* 1 (2019), 374–388.
- [4] Anna L Buczak and Erhan Guven. 2015. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications surveys & tutorials* 18, 2 (2015), 1153–1176.
- [5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). [arXiv:1512.01274](https://arxiv.org/abs/1512.01274) <http://arxiv.org/abs/1512.01274>
- [6] Harshit Daga, Yiwen Chen, Aastha Agrawal, and Ada Gavrilovska. 2023. CLUE: Systems Support for Knowledge Transfer in Collaborative Learning with Neural Nets. *IEEE Transactions on Cloud Computing* (2023), 1–14. <https://doi.org/10.1109/TCC.2023.3294490>
- [7] Randy DeFauw and Collin Cudd. December 2021. Applying Federated Learning for ML at the Edge. <https://aws.amazon.com/blogs/architecture/applying-federated-learning-for-ml-at-the-edge/>.
- [8] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.
- [9] Dimitrios Dimitriadis, Mirian Hipolito Garcia, Daniel Madrigal Diaz, Andre Manoel, and Robert Sim. 2022. Flute: A scalable, extensible framework for high-performance federated learning simulations. *arXiv preprint arXiv:2203.13789* (2022).
- [10] Meherwar Fatima, Maruf Pasha, et al. 2017. Survey of machine learning algorithms for disease diagnostic. *Journal of Intelligent Learning Systems and Applications* 9, 01 (2017), 1.
- [11] Patrick Foley, Micah J Sheller, Brandon Edwards, Sarthak Pati, Walter Riviera, Mansi Sharma, Prakash Narayana Moorthy, Shi-han Wang, Jason Martin, Parsa Mirhaji, Prashant Shah, and Spyridon Bakas. 2022. OpenFL: the open federated learning library. *Physics in Medicine & Biology* (2022). <https://doi.org/10.1088/1361-6560/ac97d9>
- [12] Apache Software Foundation. 2015. Apache Airflow. <https://airflow.apache.org>.
- [13] Yuanxiong Guo, Ying Sun, Rui Hu, and Yanmin Gong. 2022. Hybrid Local SGD for Federated Learning with Heterogeneous Communications. In *International Conference on Learning Representations*.
- [14] Andrew Hard, Chloé M Kiddon, Daniel Ramage, Françoise Beaufays, Hubert Eichner, Kanishka Rao, Rajiv Mathews, and Sean Augenstein. 2018. Federated Learning for Mobile Keyboard Prediction. <https://arxiv.org/abs/1811.03604>
- [15] Stephen Hardy, Wilko Henecka, Hamish Ivey-Law, Richard Nock, Giorgio Patrini, Guillaume Smith, and Brian Thorne. 2017. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. <https://arxiv.org/abs/1711.10677>
- [16] Stephen Hardy, Wilko Henecka, Hamish Ivey-Law, Richard Nock, Giorgio Patrini, Guillaume Smith, and Brian Thorne. 2017. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *arXiv:1711.10677 [cs.LG]*
- [17] Chaoyang He, Songze Li, Jinhyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, et al. 2020. Fedml: A research library and benchmark for federated machine learning. *arXiv preprint arXiv:2007.13518* (2020).
- [18] Dzmitry Huba, John Nguyen, Kshitiz Malik, Ruiyu Zhu, Mike Rabbat, Ashkan Yousefpour, Carole-Jean Wu, Hongyuan Zhan, Pavel Ustinov, Harish Srinivas, et al. 2022. Papaya: Practical, private, and scalable federated learning. *Proceedings of Machine Learning and Systems* 4 (2022), 814–832.
- [19] Shristi Shakya Khanal, PWC Prasad, Abeer Alsadoon, and Angelika Maag. 2020. A systematic review: machine learning based recommendation systems for e-learning. *Education and Information Technologies* 25, 4 (2020), 2635–2664.
- [20] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).
- [21] Fan Lai, Yinwei Dai, Sanjay Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha Madhyastha, and Mosharaf Chowdhury. 2022. FedScale: Benchmarking Model and System Performance of Federated Learning at Scale. In *Proceedings of the 39th International Conference on Machine Learning*.
- [22] Fan Lai, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. 2020. Oort: Informed participant selection for scalable federated learning. *arXiv preprint arXiv:2010.06081* (2020).
- [23] Anusha Lalitha, Osman Cihan Kilinc, Tara Javidi, and Farinaz Koushanfar. 2019. Peer-to-peer Federated Learning on Graphs. <https://arxiv.org/abs/1901.11173>
- [24] Martin Leo, Suneel Sharma, and Koilakuntla Maddulety. 2019. Machine learning in banking risk management: A literature review. *Risks* 7, 1 (2019), 29.
- [25] Li Li, Jun Wang, and ChengZhong Xu. 2020. FLSim: An Extensible and Reusable Simulation Framework for Federated Learning. In *International Conference on Simulation Tools and Techniques*. Springer, 350–369.
- [26] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. 2020. Federated optimization in heterogeneous networks. *Proceedings of Machine Learning and Systems* 2 (2020), 429–450.
- [27] Tian Li, Maziar Sanjabi, Ahmad Beirami, and Virginia Smith. 2020. Fair Resource Allocation in Federated Learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26–30, 2020*. OpenReview.net. <https://openreview.net/forum?id=ByexELSYDr>
- [28] Lumin Liu, Jun Zhang, SH Song, and Khaled B Letaief. 2020. Client-edge-cloud hierarchical federated learning. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 1–6.
- [29] Yan Lu, Yuanhao Shu, Xu Tan, Yunxin Liu, Mengyu Zhou, Qi Chen, and Dan Pei. 2019. Collaborative learning between cloud and end devices: an empirical study on location prediction. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. 139–151.
- [30] Siqi Luo, Xu Chen, Qiong Wu, Zhi Zhou, and Shuai Yu. 2020. HFEL: Joint Edge Association and Resource Allocation for Cost-Efficient Hierarchical Federated Edge Learning. *IEEE Transactions on Wireless Communications* 19, 10 (2020), 6535–6548. <https://doi.org/10.1109/TWC.2020.3003744>
- [31] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*.



- [32] John Nguyen, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Mike Rabbat, Mani Malek, and Dzmitry Huba. 2022. Federated Learning with Buffered Asynchronous Aggregation. In *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*.
- [33] Takayuki Nishio and Ryo Yonetani. 2019. Client selection for federated learning with heterogeneous resources in mobile edge. In *ICC 2019-2019 IEEE international conference on communications (ICC)*. IEEE, 1–7.
- [34] Nvidia. 2018. Nvidia Clara. <https://www.nvidia.com/en-us/clara/>
- [35] European Parliament and Council of the European Union. 2018. EU General Data Protection Regulation. <https://eugdpr.org/>.
- [36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
- [37] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distributed Comput.* 69 (2009), 117–124.
- [38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- [39] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and H. Brendan McMahan. 2020. Adaptive Federated Optimization. <https://arxiv.org/abs/2003.00295>
- [40] Jae Hun Ro, Ananda Theertha Suresh, and Ke Wu. 2021. Fedjax: Federated learning simulation with jax. *arXiv preprint arXiv:2108.02117* (2021).
- [41] Theo Ryffel, Andrew Trask, Morten Dahl, Bobby Wagner, Jason Mancuso, Daniel Rueckert, and Jonathan Passerat-Palmbach. 2018. A generic framework for privacy preserving deep learning. <https://arxiv.org/abs/1811.04017>
- [42] Jaemin Shin, Yuanchun Li, Yunxin Liu, and Sung-Ju Lee. 2022. Fed-Balancer: Data and Pace Control for Efficient Federated Learning on Heterogeneous Clients. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (Portland, Oregon) (MobiSys '22)*. Association for Computing Machinery, New York, NY, USA, 436–449. <https://doi.org/10.1145/3498361.3538917>
- [43] Jiayi Wang, Shiqiang Wang, Rong-Rong Chen, and Mingyue Ji. 2020. Demystifying Why Local Aggregation Helps: Convergence Analysis of Hierarchical SGD. In *AAAI Conference on Artificial Intelligence*.
- [44] Thorsten Wuest, Daniel Weimer, Christopher Irgens, and Klaus-Dieter Thoben. 2016. Machine learning in manufacturing: advantages, challenges, and applications. *Production & Manufacturing Research* 4, 1 (2016), 23–45.
- [45] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. 2018. Applied Federated Learning: Improving Google Keyboard Query Suggestions. <https://arxiv.org/abs/1812.02903>
- [46] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.