

Federated Learning Operations Made Simple with Flame

Harshit Daga*
Georgia Institute of Technology

Jaemin Shin†
KAIST

Dhruv Garg
Georgia Institute of Technology

Ada Gavrilovska
Georgia Institute of Technology

Myungjin Lee*‡
Cisco Research

Ramana Rao Kompella
Cisco Research

ABSTRACT

Distributed machine learning approaches, including a broad class of federated learning techniques, present a number of benefits when deploying machine learning applications over widely distributed infrastructures. To realize the expected benefits, however, introduces substantial operational challenges due to required application and configuration-level changes related to deployment-specific details. Such complexities can be greatly reduced by introducing higher-level abstractions – role and channel – using which federated learning applications are described as Topology Abstraction Graphs (TAGs). TAGs decouple the ML application logic from the underlying deployment details, making it possible to specialize the application deployment, thus reducing development effort and paving the way for improved automation and tuning. We present Flame, the first system that supports these abstractions, and demonstrate its benefits for several use cases.

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 [24]; cyber-security breach detection [7]; predictive maintenance and condition monitoring in manufacturing [52]; disease identification in healthcare and life sciences [15] and risk analysis in financial services [28].

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 [40], 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. Instead of sending the raw data to a centralized server, each client uses the data to train their local model, summarizes the changes as a model update and shares it with the aggregator, where updates from all the clients are combined to improve the shared model.

Despite FL’s increasing importance, operationalizing it in production is still challenging. While significant focus has been on FL algorithms and theories, less was given to holistic FL system designs. Towards a holistic FL system, we focus on *federated learning*

operations (FLOps) that are essential to running FL jobs and managing any FL system. FLOps involve support for geo-distributed, heterogeneous environments such as mobile edge computing (MEC) infrastructure, cloud and edge devices. Each environment’s characteristics and constraints therefore entail different tasks in FLOps. In many use cases, a data source is normally different from a compute for training. Similarly, a data owner is unlikely to be a compute cluster admin. These aspects make FLOps more challenging than machine learning operations (MLOps) for centralized learning. Consolidating compute and data management into a centralized facility such as cloud greatly reduces the overheads of MLOps with help of managed services such as AWS Sagemaker, Azure ML and Vertex AI. However, such consolidation is infeasible in FLOps.

At the core of the limitations of existing systems is their lack of support for FL-specific deployment customizations or for the inherent diversity and heterogeneity required for efficient operation of distributed services across federated or disaggregated nodes. In response, we present **Flame**, *a new system that enables fine-grained specialization of distributed FL services around the specifics of a deployment context*.

A number of challenges contribute to the need for a system such as Flame. A classical 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. Indeed, FL has been a fast-evolving technology and numerous variants [5, 17, 19, 22, 34, 35] have been proposed. Besides accuracy, these are proposed for different performance objectives such as scalability, convergence, training costs, and so on. Moreover, the designs are also influenced by factors like operation scales and use cases. Hence, the system architectures are quite different. Some approaches introduce system components like selectors and coordinators as separate runtime entities [5, 22]; and others introduce edge aggregators [34, 35], enable peer-to-peer collaboration [9, 10], or take a hybrid approach of combining distributed learning and federated learning [17]. As a result, one size (i.e., client-server architecture) doesn’t fit all.

Along with the choice among different architectures and their corresponding deployment topologies, the deployment heterogeneity requires FL systems to support different communication backends¹. For instance, a globally visible message queue service is necessary if all workers (aggregator and trainers) cannot easily reach each other, have no fixed IP address or are behind firewall or within private networks. In such a case, a communication backend

*Equal contribution

†Work done at Cisco Research

‡Corresponding author

¹They refer to software components that implement protocols for distributed machine learning. Calling a protocol a communication backend implies the backend implementation with the protocol.

such as MQTT is needed. In a hybrid architecture [17], that combines learning with a centralized aggregator among trainers within a tightly coupled cluster, and distributed learning across clusters, MPI or point-to-point (P2P) backends are more appropriate within a cluster. However, MQTT can still be a viable option for sharing model updates across clusters or with a top-level global aggregator.

Managing distributed data and compute is yet another key FLOps task and the most neglected. In FL, datasets should be available at compute nodes that comply with security and privacy policies. Since a machine learning engineer who wishes to deploy an FL job has neither visibility into participants' compute infrastructures nor control over them, it is typically up to participants to bring compute and datasets together. This coupling can limit when and where FL can be used; instead a system should allow that infrastructure and datasets can be independently registered, and later used by participants to create shared global models.

To support FLOps effectively, 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 [25], Flower [4] and PySyft [48] 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 [21], 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 [5, 22] 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.

To overcome the limitations of current systems and to enable support for FLOps, Flame introduces a new abstraction called the *Topology Abstraction Graph (TAG)*. This abstraction enables explicit customization of individual components in the system and supports heterogeneity and hybrid 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 thin integration interfaces that allow for integration with compute infrastructure and dataset providers, enabling support for different resource orchestrators and heterogeneous deployment platforms. This allows Flame to decouple compute and data management such that a pool of compute resources can be registered independently of data registration, and data can be linked to the pool instead of a single static compute. The actual coupling of compute and data occurs at deployment time.

The modular design of Flame allows participants to express their deployment in a compact TAG representation, provide the machine learning code for the respective roles and select a communication

backend. Flame then expands the TAG to map its physical deployment, using information about the properties of the registered nodes and available datasets in the system. 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 modifications to the core library (§6.1). In §6.2, we demonstrate the benefits of Flame's flexibility. As presented in Table 7 in the appendix, Flame's flexibility and extensibility offer a variety of topologies and mechanisms. Therefore, users can easily switch from one mechanism/topology to another (§6.3). We release Flame as an open source project², which can facilitate the development of new FL methodologies.

2 BACKGROUND AND MOTIVATION

2.1 Federated Learning

Federated learning [36] has recently emerged as a compelling machine learning practice for preserving privacy and meeting data sovereignty. It builds a global model by aggregating model weights at a central location shared from many clients. In FL, clients train a local model while keeping their dataset local. A typical FL job goes over multiple rounds, each of which consists of three broad steps: global model distribution, local training and model aggregation. There exist many variants of federated learning. Some focus on algorithms [30, 31, 36, 37, 45] for improving fairness, accuracy and convergence performance. Others propose to select clients intelligently [26, 38] or to carefully sample a client's dataset [50] for faster convergence. All of these assume the classical federated learning setting where all clients talk to an aggregation server. In contrast, approaches like hierarchical FL [34, 35], hybrid FL [17] and peer-to-peer FL [27] propose different topologies, and thus entail system architectures and communication patterns different from those of the classical FL.

The notion about FL is in-situ training whereby training is done on the device that data is generated. Gboard (a virtual keyboard on Android phone) which recommends keywords that user may type on the keyboard [18, 53] is a go-to example of FL. However, one should note that FL is also widely used by companies with geo-distributed data that cannot be moved to a centralized location because of data privacy laws or network bottleneck. Thus, the requirements for FL job creation and deployment changes in these settings when compared to Gboard use case. In fact, in many cases FL is rather off-site training than in-situ training because datasets can't be readily available for training at a target machine. In such cases, FL relies on separate compute resources where datasets need to be available at those compute resources.

2.2 Federated Learning Operations

We start off our discussion by introducing MLOps briefly. MLOps take place in a central facility such as cloud or on-prem datacenters. Therefore, MLOps are tailored to leverage services or tools available in those infrastructures. At a high level, MLOps can be divided into three phases: (i) *data engineering* that includes sequence of operations on raw data to curate datasets; (ii) *ML model engineering* responsible for writing and executing learning algorithms to obtain

²<https://github.com/cisco-open/flame>

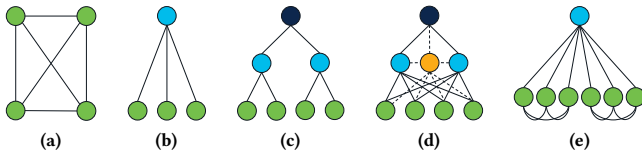


Figure 1: Topologies that can be used in federated learning: (a) distributed, (b) classical FL, (c) hierarchical, (d) hierarchical with replicas and coordinator, and (e) hybrid. ●: training node, ●: aggregation node, ●: global aggregation node, and ●: coordinator. In (d), dotted lines are to tell connections between training nodes and aggregation nodes from those with coordinator.

an ML model; and (iii) *deployment* where the trained model is integrated into application and its performance is monitored. Similar to DevOps principles that combine software development and IT operations, MLOps are a set of principles that aim to build, deploy and maintain machine learning models in production, reliably and efficiently.

FLOps can benefit from some of the MLOps practices. For example, MLOps’s deployment practice can be applicable for FLOps as a model trained through FL can be plugged into a cloud-based deployment pipeline easily. However, there also exist stark differences between FLOps and MLOps. In FL, training takes place in a geo-distributed manner where participants are not co-located physically. Hence, conducting data engineering uniformly is hard as training data may be stored across independent entities. Moreover, model engineering too is conducted across heterogeneous settings (e.g., mobile phones, healthcare use cases, mobile edge computing (MEC) setting, cross-VPC (Virtual Private Cloud) setting in cloud [44], and so on). The heterogeneity makes in-situ training hard too as a machine that generates data may not be one that trains a local model in these different settings.

The heterogeneity indeed led to the development of many variants of federated learning as discussed in §2.1, which makes FLOps more challenging. Our analysis on these approaches suggests that a diverse set of topologies should be supported as part of FLOps. We present some of them in Figure 1. Applying distributed learning mechanisms (e.g., all-to-all, all-reduce, scatter-gather) with a distributed topology shown in Figure 1a can be more effective for cloud-based FL [44] where VPC peering enables high-speed links among participants. On the other hand, classical FL topology shown in Figure 1b may be a reasonable option for mobile phone, healthcare or personalized recommendation use cases. In contrast, Figure 1c depicts a hierarchical topology that may be useful in the MEC environment where training nodes are distributed across different mobile stations which exhibit bandwidth fluctuations. The client selection process can be separated out of a global aggregation node as a coordinator so that the coordinator can be placed closer to training and intermediate aggregation nodes to lower communication overheads. For such a purpose, a variant of the hierarchical topology is a viable option (Figure 1d). Finally, hybrid topology shown in Figure 1e can be desirable in cloud-based FL cases where VPC peering with some participants is disallowed due to security

policy or participants are co-located across several regions in a single cloud or several vendors’ clouds. These are just a few examples, and FLOps may need to support many different topologies and their associated mechanisms as FL environments and technology evolve.

The fact that FL is not in-situ training, requires distributed compute and dataset management, which adds more complexity. This is especially daunting as neither compute nor datasets may be under the control of a model developer, and compute admins may be different from data owners. A common yet inefficient practice is to let data owner who wishes to participate in an FL job bring her own compute along with her data, imposing additional delays and limitations.

2.3 Current Ecosystem

To create a machine learning system, libraries/frameworks such as PyTorch, TensorFlow and Keras, 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 Ray, Amazon Sage Maker, Azure, Vertex AI, integrate such supporting tools around which a system can be developed. Although such platforms are helpful, their goal is to provide a service that focuses on the main phases in the MLOps cycle, and their tight coupling with internal services [11] and APIs does not allow them offer the flexibility and modular support needed for FLOps operations.

FedML [21] is a recent framework aims to support easy development of FL-based ML models. Its API and client-server based abstraction provide the flexibility that allows development of new scenarios, thereby assisting few requirements of FLOps. 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 hierarchical FL, it is unfortunately 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. There are other frameworks such as FedScale [25], Flower [4] and PySyft [48], all of which follow the same client-server architecture or two-tier topology, and share the same shortcomings of FedML in terms of extensibility.

Summary. Similar to DevOps tasks, new use cases and requirements will arise in FLOps as federated learning techniques evolve. Thus, well-defined and finer abstraction boundaries are required

to help express the needs of a federated learning environment. But it is difficult to enforce such abstraction boundaries in the existing frameworks, as their intended goal is to provide support for MLOps, and are thus too rigid to incorporate the constantly changing requirements for a federated learning system. Alternatively, a clean-slate approach and redesign from the ground up to provide flexibility can help. The consequent investment of engineering effort is one that can dramatically reduce ongoing costs and speed further innovation. Thus, there is a need for a system like Flame, that can provide better abstractions for federated settings, while allowing integration with the existing tools.

3 FLAME

Goal. Flame is designed with the goal to provide fine grain abstraction for composability and extensibility for federated learning tasks, thus providing support for FLOP tasks. One way to achieve this is for developers to design a modular system or general purpose solutions using different softwares packages. This often results in in glue code or pipeline jungles, that can inhibit improvements [49]. Additionally, over time not only the requirements for learning job change, but with such a fast developing field new techniques such as models and algorithms are introduced to improve accuracy, convergence rate and reduce communication cost. To adapt to such changes, the system should be able to provide building blocks that can support quick introduction and testing of new models and algorithms, and allow support for various deployment strategies (topologies), such as moving from classical to complex hybrid federated learning approaches. It should also provide fine grain control over communication backend for data movement in different parts of the system.

Overview. The design of Flame enables these decoupling through *Topology Abstraction Graph* (TAG). It employs a graph-based representation that maps the expanded physical topology to a condensed logical TAG. Behavior of each node in the TAG is specified with a *role* abstraction, and roles are interconnected via a *channel* primitive. 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 automatic fine-grain 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 is abstract representation of physical expanded deployment.

Finally, the Flame *management plane* offers support to integrate with different and/or independent orchestrators and resource managers. When expanding a TAG into a concrete physical instantiation, Flame relies on the metadata about the infrastructure resources, in the form of various *attributes* that describe the properties or access constraints of individual components. This allows Flame to coordinate the application deployment with consideration of the requirements of the various stakeholders in the distributed FL systems – application and infrastructure (compute and dataset) providers. Thus, providing an enhanced collaborative environment where Flame participants can focus on FL job and the system takes

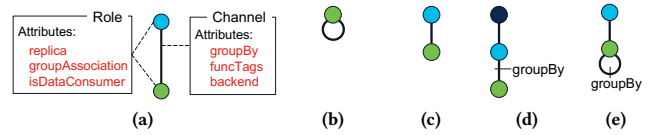


Figure 2: (a) Building blocks of TAG. TAG representation of topologies: (b) distributed, (c) classical FL, (d) hierarchical FL, (e) hybrid. ● is a trainer node with isDataConsumer set. The groupBy attribute is used to create groups of the same role.

care of the operational steps in the FL process, paving the way toward automation for FLOPs.

4 DESIGN

In this section, we introduce the design of Flame. First, we explain a key concept called Topology Abstraction Graph (TAG) in §4.1. Next, we discuss how a TAG is expanded into a real topology for deployment (§4.2). We then describe resource annotation that is essential to drive actual deployment of a real topology (§4.3). Finally, we discuss Flame’s two programming models that can aid fast development of a distributed ML job and easy extension of existing FL methodologies (§4.4).

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 machine learning job. It comprises of 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 an abstraction for communication backends. TAG’s schema is visually represented as illustrated in Figure 2a and different learning topologies can be represented using a TAG as shown in Figures 2b-2e. Below we discuss role and channel in detail. To aid understanding of the discussion, a concrete example of a TAG is presented in Figure 3a.

Role. An executable worker unit carrying out a specific task in a machine learning 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 the uniqueness of the tasks associated with these workers, Flame is able to abstract their behavior in the learning process and assign them roles. This forms the foundation of Flame’s flexible and adaptable 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 4.4.

The flexible binding between role and program allows Flame to be extensible and support different mechanisms under different topologies. Additionally, role has three attributes: *replica*, *isDataConsumer* and *groupAssociation*. A TAG is expanded into its physical

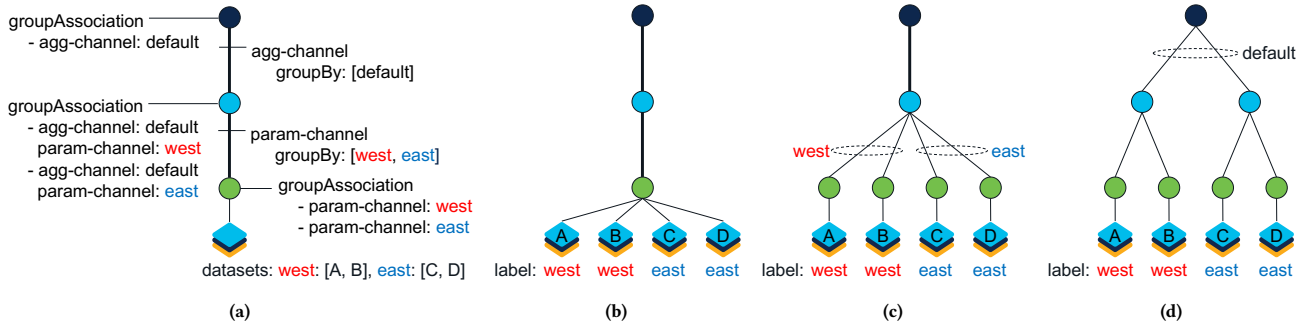


Figure 3: Expansion of a TAG into a physical topology. (a) TAG representation of hierarchical FL (H-FL); each dataset belongs to a group; (b) dataset is expanded; (c) one training worker is allocated per dataset and the worker’s group is determined by the dataset’s group; and (d) each entry in `groupAssociation` 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.

topology by combining these attributes, as discussed in §4.2. *Replica* is to set the number of replicated workers for a role. The replicated workers share the same properties. Thus, this attribute is useful in load-balancing aggregation work among aggregators, for instance (see §6.1). *isDataConsumer* attribute specifies whether or not a role consumes data and *groupAssociation* dictates the way a worker of the role is associated with channels and one of their groups (more on channel below). Each worker of the role has a set of channels and one of their groups. Therefore, *groupAssociation* instructs how workers of a role are connected with those of other roles. This attribute contains a list of the following set: $\{k_1:v_1, \dots, k_i:v_i\}$ where k_i is the name of channel i and v_i is a group in the channel; an example of this attribute is shown in Figure 3a. The size of the list corresponds to the number of workers for the given role.

Channel. It is an abstraction that links a pair of roles in the TAG and facilitates the exchange of data between them through a communication channel. This design choice enables Flame to offer precise control over the communication backend used for each channel, facilitating the design of efficient machine learning jobs tailored to the user’s specific requirements and resource availability. The attributes associated with a channel provide flexibility in creating various topologies, as shown in Figure 2. Additionally, to ensure compatibility with different communication backends such as MPI, MQTT, Kafka, and gRPC, Flame’s SDK separates the ML logic from the communication layer by providing a channel manager interface with a standard set of APIs (e.g., `send`, `recv`, `broadcast`, `join`, `leave`) that any two roles connected by a channel can use. This approach abstracts the implementation details of the communication backend, allowing roles to send and receive messages uniformly, regardless of the underlying communication protocol.

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 grouping approach, but the implementation 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. This attribute helps avoid any ambiguity in identifying which function 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. For instance, users may choose to store their datasets in the cloud of one provider or store them with different cloud providers. It’s also possible that users 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 this attribute’s efficacy in §6.2.

4.2 TAG Expansion

The TAG only specifies the roles and channels, but not the association of datasets with these roles. To allow for flexibility in dataset association, Flame requires users to provide a dataset specification. Additionally, users may need to group datasets to form clusters, as in hierarchical federated learning (H-FL). To enable dataset grouping, Flame provides *datasetGroups* attribute that enables developers to combine different datasets into a single learning group. For instance, in Figure 3a, the user has formed two dataset groups (“west” and “east”), each comprising two distinct datasets (A, B) and (C, D), respectively.

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 using channel information for validation.

The top-level function, `Expand` walks through roles (line 6) and calls `BuildWorkers` for each role. Then, the `BuildWorkers()` function creates the worker configuration. The specification for each

Algorithm 1: TAG expansion

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

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

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 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, a variant of a hierarchical topology shown in Figure 1d is implemented by using *replica*. While pre and post checks are used to validate the correctness of TAG and expanded physical deployment respectively.

Example. Figure 3 demonstrates the application of Algorithm 1 to expand the high level TAG for Hierarchical FL (Figure 3a) 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 3b. 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 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 3d). Thus, completing the TAG expansion.

4.3 Resource Annotation for Deployment

In a machine learning 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 different 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. The majority of 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 interface service that supports various resource orchestration managers and allows developers to register their own cluster (§5.1). It’s necessary to capture specific properties for computing clusters, such as geographical boundaries. This information is provided during registration with the *realm* attribute. For example, certain data can only be accessed in specific regions due to data privacy laws like GDPR [40]. The realm information associated with the compute infrastructure aids the dataset owner in defining the accessibility boundary for their datasets.

Metadata for Dataset. For machine learning training jobs, 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 metadata *realm* attribute helps in defining access restrictions that apply to the dataset. It leverages information from the compute infrastructure to create a logical boundary that defines this accessibility. This design enables compliance with security and privacy policies associated with the dataset, and allows data owners to maintain control over dataset accessibility and deployment while allowing its use by others. It is important to note that Flame only stores metadata information and not the raw dataset. The *url* points to the dataset’s location and can be extended to support various data access methods.

As part of our design, we deliberately separated the infrastructure from the programming logic to facilitate a more organized approach to managing a federated learning (FL) job. By doing so, users can focus solely on composing their machine learning 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.

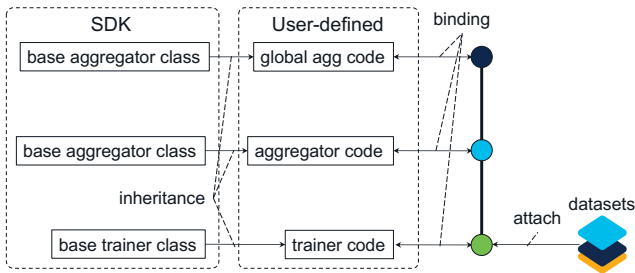


Figure 4: Workload composition for hierarchical topology.

```

from fledge.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 5: 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().

4.4 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. This programming model is useful for those whose needs can be met by Flame’s built-in functionalities. The *developer* programming model is intended for developers who want to extend Flame’s capabilities by allowing for different topologies, roles, and training methodologies. This programming model 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 *role* building block of a TAG performs a task only if it is associated with a program. 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 4 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 implements a basic workflow for a certain role (such as trainer, intermediate and global aggregator), so the user only needs to implement the core functions directly relevant to their learning job. For instance, for a hierarchical FL (H-FL) topology user can define their custom MNistTrainer as illustrated in Figure 5, by extending the out-of-the-box base class provided the Flame SDK. Interestingly, the implementation for aggregator roles

```

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 >>
            ... ..)

```

Figure 6: Code snippet that illustrates a composer mechanism through developer programming model. Loop is a primitive that enables repeated execution, which takes an exit condition as a function. Once tasklets are chained, they are executed in a sequential manner. The first argument of Tasklet is *alias* that can be used to ease the modification of a tasklet chain.

is simpler than that of trainer; user is only required to implement a model architecture since Flame’s core library already provides essential functions such as distribute and aggregate. In case the aggregator roles need to conduct validation test, the user 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 FLOps engineers 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*³. These tasklets are combined together to finish a worker’s task. Inspired by workflow management solutions [1], 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 machine learning mechanisms, thereby allowing fast development. In order to create a workflow, Flame overrides the >> 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 or to record various metrics after each step and link them in the workflow, as illustrated in Figure 6.

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

Finally, in order to expand Flame’s functionalities, developers may require interaction with channels to exchange new types of

³It is to imply that the execution unit is small; it’s not one in Linux kernel.

Function	Module	Note
<code>get_tasklet(<i>aliases</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>)</code>	Receive a message from each end 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.

messages. To facilitate this, Flame provides various functions as part of the channel API, as demonstrated in Table 2. This API not only provides a great deal of flexibility in extending Flame, but also enables developers to interact with the system in a uniform manner, irrespective of the communication backend being utilized.

5 MANAGEMENT PLANE

The Flame SDK facilitates the composition of machine learning jobs and the implementation of new FL mechanisms. The building blocks of Flame, provided via its SDK, offer new abstractions that enable systematic lifecycle management of those FL jobs, performed by its management plane.

5.1 System Components

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

APIserver. The APIserver is a front end that exposes a REST API. A CLI tool uses the REST API and allows users to interact with the management plane. Through the interface, users and FLOPs engineers can operate various functions such as the creation/update of job specifications (topology, machine learning code, etc), submission/termination of a job, etc.

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 database (MongoDB). (ii) It performs TAG expansion into a real topology, and interacts with compute cluster managers, such as Kubernetes, for worker deployment, resource

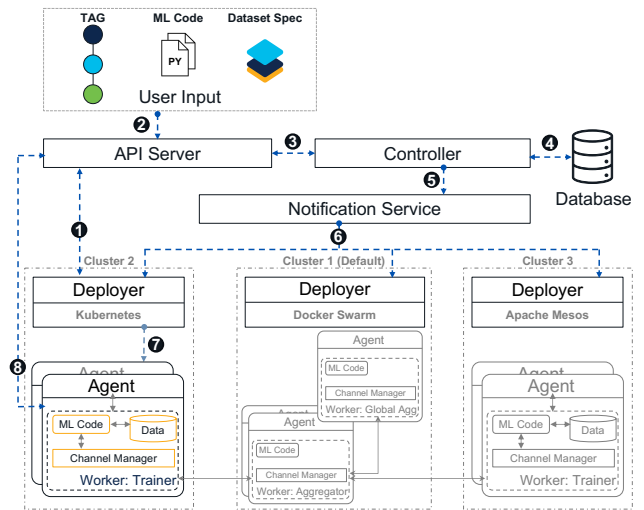


Figure 7: Flame architecture and workflow overview.

provisioning and decommissioning. (iii) Finally, it monitors the job’s progress and is responsible for events such as worker failure.

Deployer. The deployer is an integration interface service, which provides abstraction to integrate different resource 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 resource 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 a lifecycle of a task in a given job by interacting with APIserver. It also prevents any direct interaction of user deployed ML code with the Flame system, thus providing a sandbox environment for the job. During instance initialization, the deployer provides the agent with information related to the job. The agent is then responsible to fetch 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. When the agent receives a terminate event, it stops the worker process and reports the worker’s status to the APIserver.

Notifier. A notification service provides a means for the controller to push event signals to agents and deployers. The notifier enables event-driven management of FLOPs. For instance, upon receiving an event, agents and deployers reach out to the APIserver to obtain job related information.

5.2 Workflow

In Figure 7 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 admins are required to integrate the *Deployer* interface service provided by Flame. For instance, Kubernetes (K8s) uses Kubernetes Deployment that coordinates with K8s to create or modify instances of the pods that hold a containerized application. The integration also results in appropriate permission for creating and deleting pods. The system administrator 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 (step ①). It should be noted that each cluster is owned and managed by its admin who has full control of the resources provided by the cluster that can be used by Flame.

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

Job Deployment. Upon receiving the job configuration, it is shared with the controller (step ③). The controller records this information in the database (step ④), and expands the TAG configuration to determine where each role should be created, based on the metadata *datasetGroups* attribute if the role’s *isDataConsumer* attribute is set (i.e., it is a trainer role) or TAG *groupBy* and *groupAssociation* attributes otherwise. The controller then sends a compute creation event to the notification service (step ⑤) along with the job information. The notification service notifies the corresponding deployers where roles need to be created (step ⑥). Upon receiving such a request, each deployer creates a compute (e.g., a pod in K8s) that contains an agent (step ⑦). The agent uses the job id to retrieve the code and task configuration files (step ⑧). The task configuration file contains role, channel, and dataset access information, which is required by each role to determine its connection with other roles, map the methods to be executed, etc. The agent then starts a worker which executes the FL tasks. Once the task is completed, the agent updates its status by sending a request to the APIServer. The deployers are subsequently notified through a revoke event to de-allocate the resources from the compute clusters. Our system manages FL jobs in a fully automated manner.

5.3 Implementation

We have implemented the management plane of Flame in Golang while the programming model is developed using Python. The current implementation supports a diverse range of topologies and algorithms, as shown in Table 7 in the appendix. 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 packaged as a Kubernetes application and deployed on minikube. This single box deployment of our system allows FLOps engineers to easily validate their prototypes of new FL mechanisms and algorithms or to conduct small scale experiments. Furthermore, this environment provides all the extensibility and flexibility of Flame in an emulation environment, thereby accelerating research. Moreover, packaging

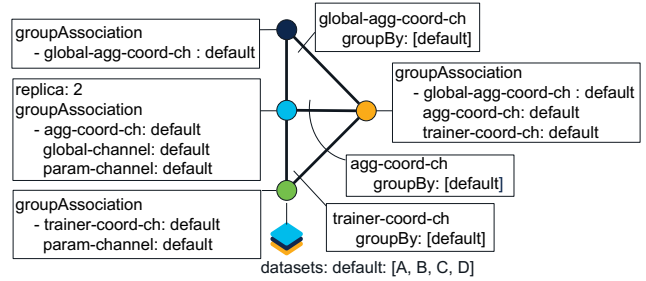


Figure 8: TAG for Coordinated FL (H-FL with coordinator). Only additional changes are shown in the figure on top of the configuration shown in Figure 3a. TAG is represented in YAML format. ●: global aggregator, ●: aggregator, ●: trainer, and ●: coordinator. The expanded form is shown in Figure 1d.

the system components makes the management plane deployment portable, enabling it to be easily deployed in a large K8s cluster.

6 EVALUATION

The goal of the evaluation is to demonstrate how well Flame supports the FLOps requirements for being easily extensible, flexible, and that it decouples the management of compute and data from the ML application. We begin by showing how a developer can extend a sample hierarchical FL topology (as shown in Figure 3a) for a complex setting with a coordinator, in §6.1. We then highlight the benefits of communication backend selection with a Hybrid FL use case, emphasizing the flexibility provided by our system in §6.2. Additionally, we showcase the benefits of TAG in transforming from one topology to another in §6.3. We also compare Flame with the state-of-the-art framework in terms of the complexity associated with creating, modifying and deploying FL applications (§6.4). Finally, we present micro-benchmarking results that demonstrate the overhead of one of the system’s primary tasks—TAG expansion.

6.1 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 a Hierarchical Federated Learning (H-FL) topology with a coordinator. In this paper, we refer to this variant as Coordinated Federated Learning (CO-FL). The CO-FL differs from the H-FL in two key aspects: (1) the links between aggregator and trainer form a bipartite graph in the CO-FL, and (2) the coordinator is connected to the rest of the roles. Enabling this new variant requires an update to the TAG and update to the implementation of roles in the H-FL TAG to allow communication with the coordinator.

TAG Changes. In Figure 8, we illustrate the changes to the TAG required to add a coordinator to the H-FL topology shown in Figure 3a. The transformation from the original TAG to the new one, as shown in the figure, involves modifying around 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. Notably, the addition of a

```

def compose(self) -> None:
    super().compose()

    with CloneComposer(self.composer) as composer:
        self.composer = composer
        tl_coord_ends = Tasklet("get_coord_ends", self.get_coord_ends)

    tl = self.composer.get_tasklet("distribute")
    tl.insert_before(tl_coord_ends)
    tl = self.composer.get_tasklet("end_of_train")
    tl.remove()

```

Figure 9: Code snippet for global aggregator for CO-FL.

	Global Aggregator	Aggregator	Trainer	Coordinator
Hierarchical FL	231	200	156	—
Coordinated FL	40	67	73	158
LOC reduction	82.7%	66.5%	53.2%	—

Table 3: The number of lines of code for each role in Hierarchical and Coordinated FL.

coordinator requires the use of the *replica* attribute in the aggregator role, which is introduced in §4.1 and allows for the creation of bipartite-like communication links upon TAG expansion.

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 a core library. The developer inherits the base classes of H-FL and implements additional functionality for the coordinator role. For example, 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 aggregators to interact within each round. To accomplish this, we implement a *get_coord_ends* function associated with a *funcTag* called *coordinate* (not shown in Figure 8 for brevity). We then use the API in Table 1 and update the inherited *tasklet* chain for the global aggregator, as shown in Figure 9.

In Figure 9, we obtain tasklets by using their alias and call appropriate operations (e.g., *insert_before*, *remove*). Note that we remove *end_of_train* tasklet because a coordinator is now responsible for informing the end of training to all workers. Similarly, the aggregator and trainer are implemented, resulting in minor code revisions for the overall CO-FL implementation, as shown in Table 3.

Result. To demonstrate the feasibility of extending CO-FL, we implement a basic load-balancing scheme in the coordinator. We create a toy scenario with 10 trainers and two aggregators where the link between the global aggregator and an aggregator becomes a bottleneck across multiple rounds. As aggregators report to the coordinator their delays in uploading models, the coordinator can detect delay discrepancies among aggregators. This information allows the coordinator to identify and exclude a slow aggregator. Moreover, the coordinator employs a binary backoff mechanism to give the straggling aggregator a chance to join the training again.

Figure 10 showcases the result of such a scenario in comparison with H-FL. From round #6, the load-balancing scheme detects a significant delay from the straggling aggregator by observing the

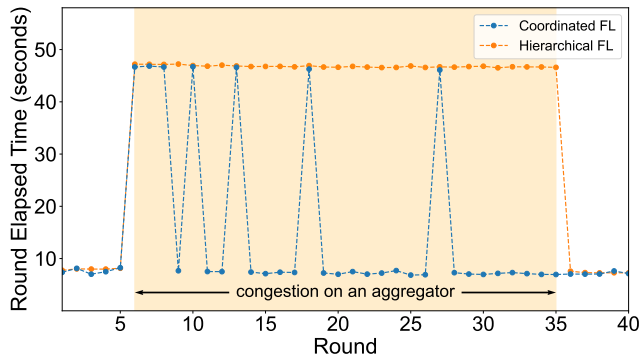


Figure 10: Performance comparison between Coordinated FL vs Hierarchical FL. Coordinated FL manages the network congestion with its load-balancing scheme.

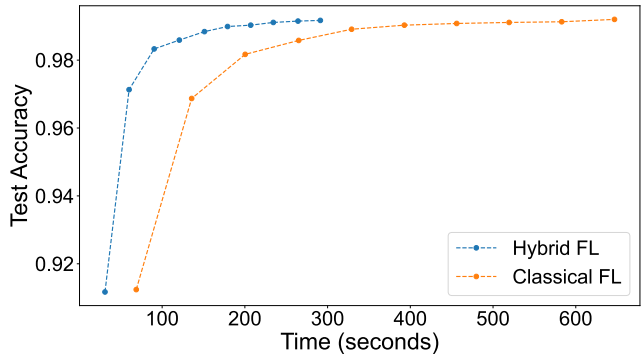


Figure 11: Performance comparison between Hybrid FL vs Classical FL. Flame’s flexibility in communication backend selection demonstrates the efficacy of Hybrid FL.

time to upload model weights to the global aggregator across all aggregators. After observing the delay for three consecutive rounds, the scheme disables the straggling aggregator for one round (round #9). Then, the scheme checks the presence of delay (round #10) by allowing the straggler to join the job again, and, if present, disables the straggling aggregator for two rounds (rounds #11-12). As congestion persists, the coordinator disables the straggler in a binary-backoff fashion (i.e., four rounds at round #14, eight rounds at round #19 and 16 rounds at round #28). H-FL, without such a load-balancing scheme, experiences large per-round time for rounds #6-35 due to the straggling aggregator.

6.2 Flexible Backend

To demonstrate the versatility of Flame on backend configurations and its implications, we use an example of Hybrid FL [17] (Figure 1e) implemented on Flame. Hybrid FL is ideal for scenarios where trainers are co-located and network bandwidth among trainers are much higher than bandwidth between trainers and aggregator. Unlike Classical FL (C-FL), where individual trainers should send their model updates, the co-located trainers in Hybrid FL form a cluster for aggregation. The trainers rapidly aggregate a cluster-level model with algorithms like ring-allreduce [42] and uploading

	C-FL	C-FL→H-FL	H-FL→H-FL ^b	C-FL→Distributed	C-FL→Hybrid	H-FL→CO-FL
Code	+ trainer + global-agg	+ agg	N/A	- global-agg Δ inheritance	Δ inheritance	+ coordinator Δ inheritance
TAG	+ channel	+ channel	Δ groupBy	Δ channel	Δ backend Δ channel Δ groupBy	+ replica + channels Δ groupBy
Metadata	+ init info	Δ datasetGroups	Δ datasetGroups	N/A	Δ datasetGroups	Δ datasetGroups

Table 4: Changes required to transform from one topology to another. The TAG representation of C-FL, H-FL, Distributed and Hybrid is showcased in Figure 2 while TAG for CO-FL (Figure 1d) is shown in Figure 8. H-FL^b represents H-FL topology with different grouping options. +, - and Δ represent addition, removal and update respectively and N/A indicates no change. “Δ inheritance” implies the switch of a base class from one to another, which is a single line change in user’s ML program.

a single copy of cluster-level model enables efficient aggregation at the aggregator.

Flame can easily realize Hybrid FL by configuring the TAG to use two different backends: (1) MQTT backend is used for channel between aggregator and trainer, and (2) P2P backend is for one between trainers. For an experiment, we emulate different bandwidth on each backend, by utilizing Linux `tc` tool. Specifically, we create a hybrid topology that consists of 50 trainers. One trainer is chosen as a straggler with a limited bandwidth of 1 Mbps between an aggregator and itself. P2P backend is given with a maximum bandwidth of 100 Mbps. The trainers are equally divided into five groups. For comparison, we also set up C-FL topology with the exactly same setting except that MQTT backend is only used.

Figure 11 shows the test accuracy over wall-clock time of a job with MNIST dataset [12]. Each point in the figure represents a round. The results suggest that Hybrid FL converges faster than C-FL, by achieving 2.21× speedup in reaching 0.985 in accuracy. This is due to the efficient aggregation of Hybrid FL without straggler. Hybrid FL also consumes less bandwidth (25 MB/round) compared to C-FL (250 MB/round) to upload model updates. This experiment demonstrates that Flame allows flexible communication backend configurations and that such flexibility can help design and experiment new FL approaches.

6.3 Topology Transformation: User Perspective

The topologies introduced in this paper are provided as templates in Flame. The mechanisms for those topologies are available too. Thus, to compose their ML job, users can pick and use one that best suits their needs. However, the requirements and constraints for the ML job may change over time, which may require topology transformations and learning mechanisms. To demonstrate how easily these transformations can be made, we walk through the steps of transforming to other topologies, starting with a basic C-FL 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 and a new connecting (+) channel with the new aggregator. Finally, to allow the grouping of trainer nodes the (Δ) `datasetGroups` attribute in metadata information is updated.

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, and (Δ) altering TAG representation where the trainer-aggregator channel is updated to trainer-trainer channel as shown in Figure 2b.

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. Then, it needs (Δ) to change the TAG to create appropriate channels and backends and change `groupBy` and `datasetGroups` to group co-located datasets.

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, (Δ) update the inheritance of classes for the global aggregator, aggregator, and trainer, and add new channels between the coordinator and the rest of the roles. In addition, grouping between aggregators and trainers can be dynamic based on the coordinator’s logic. For that, the user (Δ) updates `groupBy` and `datasetGroups` as a single group and configures `replica` in the TAG.

6.4 System Comparison

Because a system like FedML [21] is an effort to create improved abstractions for federated learning, we present Table 5 to give an overview of the differences in the functionality provided by both the system. For comprehensive feature comparison among other frameworks, we also present Table 7 in the appendix.

Topology Implementation. The *role* in Flame provides a granular abstraction that allows the developer to implement different components of federated learning as individual roles. This allows ML engineers to focus only on the ML related logic and it further helps in selectively updating different parts of the deployment or topology without touching other components. Contrary to that, FedML adopts a client-server architecture which provides a tight

Feature	Flame	FedML
Standard Topology	FL, H-FL, Distributed, Hybrid, Coordinated	FL, H-FL, Distributed
New/Update Topology	TAG Changes/Role logic	New Implementation in Core library
New Component	Role and its logic	New class, modify the core library server component and redeploy it
Communication	Different protocol for each channel	Same protocol for all channels
Infrastructure	Bring your own or use existing	Bring your own
Deployment	Automatically deploys	Developer select compute and deploys

Table 5: Comparing the changes required in Flame and FedML to implement any topology, introduce new components, update communication backend, and support deployment.

coupling between various tasks executed by a learning job. Creating new components such as coordinator or transforming from one topology to another would require the developer to implement the logic by modifying the core library, while Flame’s programming model allows the developer to introduce new logic without touching any core system library.

Communication Backend. Both systems support easy integration of different communication protocols such as MQTT, MPI, gRPC, etc., however, these systems differ in the extensibility and flexibility granularity they provide. The logical graph abstraction adopted by Flame breaks down the connection between different roles (workers) into *channel* which provides per-channel communication control for any FL topology. The client-server architecture in FedML does not allow per-worker connections to be configurable. It takes the communication configuration at the global level and uses it for all the workers (trainers and aggregators), which means that all nodes use the same communication backend.

Deployment and Grouping Support. The deployment of an FL application can be categorized in two ways. First, *compute centric* approach, where the application developer is responsible to secure the compute and then deploy the ML code to start the learning process. Second, *compute agnostic* approach, where the developer provides the ML code and instruction/rules about the type of compute required by each worker. The system is then responsible to find the compute units to create the group, deploy the code, and start the learning process. FedML follows *compute centric* approach, which requires the participant to select appropriate nodes based on constraints associated with the data, such as GDPR rules, or to create groups such as in the case of H-FL. Flame, by associating the channel’s *groupBy* attribute with metadata *datasetGroups* information, enables groupings among workers in the same role. This permits Flame to dynamically create groups for complex topologies such as hierarchal, distributed, or hybrid. In Flame, *realm* allows the system to automatically determine the compute clusters where specific *role* should be deployed. Then, the deployer component in Flame allows the system to connect to different compute clusters managed by different resource orchestrators, which can spawn workers automatically, thus following a *compute agnostic* approach.

6.5 TAG Expansion Overhead

To prepare for actual deployment, the first step at the management plane of Flame is to expand TAG. Once TAG is expanded, Flame utilizes underlying cluster management solutions, such as Helm

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 6: TAG expansion latency in seconds.

(for Kubernetes), to deploy workers as containers to perform their tasks in an FL job. Deployment time can vary significantly based on several factors, such as the distance between the management plane and remote compute clusters, network bandwidth, cluster resource availability, job size, etc. These types of deployment issues are ubiquitous in geo-distributed job or workload scenarios. As a result, the evaluation of this deployment aspect is beyond the scope of this paper and we instead analyze the overhead of TAG expansion.

We conducted experiments to measure the latency of TAG expansion and database write of its results on Flame for different FL topologies, namely C-FL (Figure 1b) and CO-FL (Figure 1d), as shown in Table 6. We evaluated the latency with varying numbers of trainers, and CO-FL was configured with 100 replicas and a coordinator. The results demonstrate that the overhead of TAG expansion on Flame is comparable across different FL topologies. Additionally, the results 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 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 [2], PyTorch [41], and scikit-learn [43] 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 [54] and Apache MXNet [8] are open source frameworks mainly for distributed learning. Systems such

as Flower [4], FedScale [25], and PySyft [48] 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 [16] 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 [39] 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 [47] 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 [13] is another federated learning simulator that focuses on scalability and efficiency. FLSim [29] 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.

8 CONCLUSION

In this paper we introduce Flame, a system that enables composability and extensibility for federated learning topologies, thus providing support for FLOps tasks. It relies on logical TAG representation of 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 on 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 accelerate the progress of federated learning.

A APPENDIX

Table 7 provides a summary of the key features of existing FL frameworks, based on their public repositories as of May 5th, 2023. While the table showcases Flame’s strengths in comparison to other frameworks, it is not intended to be exhaustive. For instance, FedML offers several other algorithms and FedScale offers other features such as cohort-based learning [33]. Although we took great care in conducting code analysis, we encountered ambiguities in certain aspects during the comparison. We encourage the readers to explore the complete list of features offered by each framework through their project web page or GitHub repositories.

REFERENCES

- [1] [n.d.]. Apache Airflow. <https://airflow.apache.org>.
- [2] Martin 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.
- [3] Durmus Alp Emre Acar, Yue Zhao, Ramon Matas, Matthew Mattina, Paul Whatmough, and Venkatesh Saligrama. 2021. Federated Learning Based on Dynamic Regularization. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=B7v4QMR6Z9w>
- [4] 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://doi.org/10.48550/ARXIV.2007.14390>
- [5] 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.
- [6] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1175–1191. <https://doi.org/10.1145/3133956.3133982>
- [7] 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.
- [8] 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](http://arxiv.org/abs/1512.01274) <http://arxiv.org/abs/1512.01274>
- [9] Harshit Daga, Yiyen Chen, Aastha Agrawal, and Ada Gavrilovska. 2021. Canoe: A System for Collaborative Learning for Neural Nets. *arXiv preprint arXiv:2108.12124* (2021).
- [10] Harshit Daga, Patrick K Nicholson, Ada Gavrilovska, and Diego Lugones. 2019. Cartel: A system for collaborative transfer learning at the edge. In *Proceedings of the ACM Symposium on Cloud Computing*. 25–37.
- [11] 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/>.
- [12] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine 29*, 6 (2012), 141–142.
- [13] 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).
- [14] Cynthia Dwork. 2006. Differential Privacy. In *Automata, Languages and Programming*, Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12.
- [15] Mehervar 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.
- [16] 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>
- [17] 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*.
- [18] 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>
- [19] 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://doi.org/10.48550/ARXIV.1711.10677>
- [20] 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:cs.LG/1711.10677](https://arxiv.org/abs/1711.10677)
- [21] 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).
- [22] 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.
- [23] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. 2020. SCAFFOLD: Stochastic Controlled Averaging for Federated Learning. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Hal Daumé III and Aarti Singh (Eds.), Vol. 119. PMLR, 5132–5143.

	Feature	Flame (ours)	FedML [21]	Flower [4]	FedScale [25]
Topology	Classical FL [36]	✓	✓	✓	✓
	Hierarchical FL [34]	✓	✓	✗	✗
	Distributed FL [21]	✓	✓	✗	✗
	Hybrid FL [17]	✓	✗	✗	✗
	Coordinated FL [5]	✓*	✗	✗	✗
	Vertical FL [20]	✗	✓	✗	✗
	Async Hierarchical FL	✓	✗	✗	✗
	Async Coordinated FL	✓	✗	✗	✗
Protocol	gRPC	✓	✓	✓	✓
	MQTT	✓	✓	✗	✗
	MPI	✗	✓	✗	✗
	NCCL	✗	✓	✗	✗
Aggregation Policy	Synchronous FL [36]	✓	✓	✓	✓
	Asynchronous FL [37]	✓	✓ [†]	✗	✓
Algorithm [‡]	FedAvg [36]	✓	✓	✓	✓
	FedProx [30]	✓	✓	✓	✓
	FedAdam [46]	✓	✗	✓	✗
	FedAdagrad [46]	✓	✗	✓	✗
	FedYogi [46]	✓	✗	✓	✓
	FedDyn [3]	✓	✓	✗	✗
	FedBuff [§] [37]	✓	✓ [†]	✗	✓
	SCAFFOLD [23]	✗	✓	✗	✗
	q-FedAvg [32]	✗	✗	✓	✓
FedNova [51]	✗	✓	✓	✗	
Client Selection	Select All	✓	✓	✓	✓
	Random [36]	✓	✓	✓	✓
	FedBuff [§] [37]	✓	✓ [†]	✗	✓
	Oort [26]	✓	✗	✗	✓
Sample Selection	Select All	✓	✓	✓	✓
	FedBalancer [50]	✓	✗	✗	✗
Security	Differential Privacy [14]	✓	✓	✓	✓
	Secure Aggregation [6]	✗	✓	✗	✗

Table 7: Comparing Flame with other FL frameworks. [†]: simulation; ^{*}: simplified version of an original architecture; [‡]: widely used algorithms listed; others (e.g., FedML) implemented several other algorithms (omitted for brevity); [§]: both aggregation algorithm and client selection are included.

- <https://proceedings.mlr.press/v119/karimireddy20a.html>
- [24] 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.
- [25] 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 (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.), Vol. 162. PMLR, 11814–11827. <https://proceedings.mlr.press/v162/lai22a.html>
- [26] 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).
- [27] Anusha Lalitha, Osman Cihan Kilinc, Tara Javidi, and Farinaz Koushanfar. 2019. Peer-to-peer Federated Learning on Graphs. <https://doi.org/10.48550/ARXIV.1901.11173>
- [28] Martin Leo, Suneel Sharma, and Koilakuntla Maddulety. 2019. Machine learning in banking risk management: A literature review. *Risks* 7, 1 (2019), 29.
- [29] 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.
- [30] 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.
- [31] 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=ByexEISYDr>
- [32] Tian Li, Maziar Sanjabi, Ahmad Beirami, and Virginia Smith. 2020. Fair Resource Allocation in Federated Learning. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=ByexEISYDr>

- [33] Jiachen Liu, Fan Lai, Yinwei Dai, Aditya Akella, Harsha Madhyastha, and Mosharaf Chowdhury. 2022. Auxo: Heterogeneity-Mitigating Federated Learning via Scalable Client Clustering. *arXiv:cs.LG/2210.16656*
- [34] 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.
- [35] 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>
- [36] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*. PMLR, 1273–1282.
- [37] 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 (Proceedings of Machine Learning Research)*, Gustau Camps-Valls, Francisco J. R. Ruiz, and Isabel Valera (Eds.), Vol. 151. PMLR, 3581–3607. <https://proceedings.mlr.press/v151/nguyen22b.html>
- [38] 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.
- [39] Nvidia. [n.d.]. Nvidia Clara. <https://developer.nvidia.com/industries/healthcare>
- [40] European Parliament and Council of the European Union. 2018. EU General Data Protection Regulation. <https://eugdpr.org/>.
- [41] 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).
- [42] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distributed Comput.* 69 (2009), 117–124.
- [43] 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.
- [44] Ujjwal Ratan and Vidya Sagar Ravipati. 2022. Managed federated learning on AWS: A case study for healthcare. https://d1.awsstatic.com/events/aws-re-mars-event-2022/Managed_federated_learning_on_AWS_A_case_study_for_healthcare_MLR312.pdf. Accessed: October 18 2022.
- [45] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and H. Brendan McMahan. 2020. Adaptive Federated Optimization. <https://doi.org/10.48550/ARXIV.2003.00295>
- [46] Sashank J. Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and Hugh Brendan McMahan. 2021. Adaptive Federated Optimization. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=LkFG3B13U5>
- [47] Jae Hun Ro, Ananda Theertha Suresh, and Ke Wu. 2021. Fedjax: Federated learning simulation with jax. *arXiv preprint arXiv:2108.02117* (2021).
- [48] 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://doi.org/10.48550/ARXIV.1811.04017>
- [49] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *Advances in neural information processing systems* 28 (2015).
- [50] Jaemin Shin, Yuan-chun Li, Yunxin Liu, and Sung-Ju Lee. 2022. FedBalancer: Data and Pace Control for Efficient Federated Learning on Heterogeneous Clients. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22)*. Association for Computing Machinery, New York, NY, USA, 436–449. <https://doi.org/10.1145/3498361.3538917>
- [51] Jianyu Wang, Qinghua Liu, Hao Liang, Gauri Joshi, and H. Vincent Poor. 2020. Tackling the Objective Inconsistency Problem in Heterogeneous Federated Optimization. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 7611–7623. https://proceedings.neurips.cc/paper_files/paper/2020/file/564127e03caab942e503ee6f810f54fd-Paper.pdf
- [52] 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.
- [53] 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>
- [54] 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.