# FLIGHT: A FaaS-Based Framework for Complex and Hierarchical Federated Learning

Nathaniel Hudson[a,b], Valerie Hayot-Sasson[a,b], Yadu Babuji[a], Matt Baughman[a], J. Gregory Pauloski[a], Ryan Chard[b], Ian Foster[a,b], Kyle Chard[a,b]

[a]*Department of Computer Science, University of Chicago, 5801 S Ellis Ave, Chicago, 60637, IL, United States*
[b]*Data Science and Learning Division, Argonne National Laboratory, 9700 S Cass Ave, Lemont, 60439, IL, United States*

**Abstract**

Federated Learning (FL) is a decentralized machine learning paradigm where models are trained on distributed devices and are aggregated at a central server. Existing FL frameworks assume simple two-tier network topologies where end devices are directly connected to the aggregation server. While this is a practical mental model, it does not exploit the inherent topology of real-world distributed systems like the Internet-of-Things. We present FLIGHT, a novel FL framework that supports complex hierarchical multi-tier topologies, asynchronous aggregation, and decouples the control plane from the data plane. We compare the performance of FLIGHT against Flower, a state-of-the-art FL framework. Our results show that FLIGHT scales beyond Flower, supporting up to 2048 simultaneous devices, and reduces FL makespan across several models. Finally, we show that FLIGHT's hierarchical FL model can reduce communication overheads by more than 60%.

*Keywords:* Federated Learning, Hierarchical Federated Learning, Function-as-a-Service, Decentralized Systems, Edge Intelligence

## 1. Introduction

Much of today's data is naturally distributed due to the growing ubiquity of systems like the *Internet-of-Things* (IoT) (Shah and Yaqoob, 2016; Baccour et al., 2022) and *Mobile Edge Computing* (Hu et al., 2015). Conventionally, training an Artificial Intelligence (AI) model on distributed data required first transferring the data to a centralized computing system (e.g., high-performance computing cluster). However, in many scenarios this approach is intractable due to large data volumes, data transfer costs, and privacy concerns (Ali et al., 2022). *Federated Learning* (FL) (McMahan et al., 2017; Konečnỳ et al., 2016), provides a potential solution as it implements a distributed training paradigm in which AI models can be trained in a distributed fashion without needing to relocate data.

Unlike conventional deep learning, FL trains individual models directly where data reside (e.g., edge devices, IoT devices, mobile devices, and sensors). A central location (e.g., server) is then tasked with aggregating (or averaging) locally-trained models rather than training a single model itself. Fig. 1 shows the general FL training process. Because no training data are communicated over the network in FL, it provides two key benefits: *(i)* reduced communication cost (Hudson et al., 2022), assuming the size of the model weights are less than the training data; and *(ii)* enhanced data privacy (McMahan et al., 2017).

FL typically assumes a two-tier system made up of a single *aggregator* connected to a flat layer of *workers* that perform local training (see Fig. 1). While reasonable in many cases (Patros et al., 2022), this assumption has notable limitations. First, it prevents an FL process from taking advantage of the naturally hierarchical, often geographically clustered, scale-free topology of many networks (e.g., the Internet) (Barabási et al., 2003).

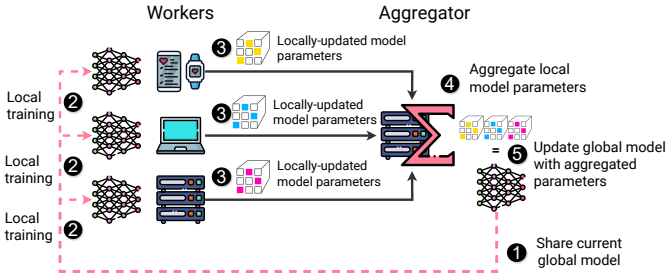


Figure 1: High-level view of a standard two-tier FL system and process.

Second, it ignores the geospatial relationship between the data distributions at end devices. For example, consider a collection of smart homes in which smart meters collect, monitor, and predict energy consumption. The data collected from individual smart homes often follows geospatial patterns related to income levels (Hudson et al., 2021).

*Hierarchical Federated Learning* (HFL) (Yu et al., 2023; Abad et al., 2020; Abdellatif et al., 2022) aims to address these problems. In HFL, the network used to share model parameter updates is multi-tier and hierarchical, and can include more than one aggregator in the network (see Fig. 2). Intermediate aggregators can produce aggregated models that are more *regional* in their context as they ultimately depend on the data distribution of the workers from which local models are obtained. HFL has been found to be particularly well-suited for use in remote environments with limited network connectivity due to its reduction of communication costs (Almurshed et al., 2022; Rana et al., 2022). Despite recent innovation in the development of FL frameworks—e.g., Flower (Beutel et al., 2022), FedML (He

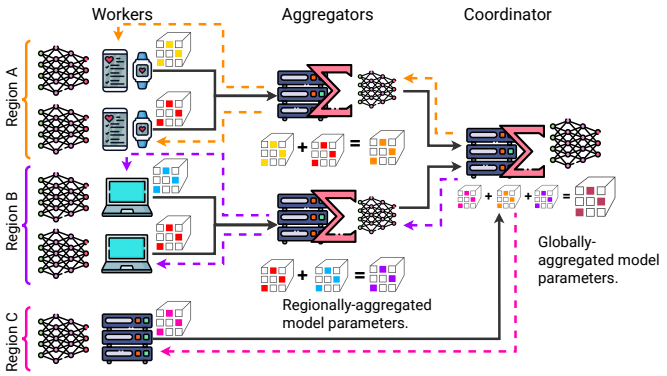


Figure 2: High-level view of a hierarchical FL system and process.

et al., 2020)—to the best of our knowledge there is no robust FL framework that natively supports complex HFL. Further, existing FL frameworks are often device-driven, an assumption that simplifies deployment, but does not scale to large, distributed systems.

We present Flight (**F**ederated **L**earning **I**n **G**eneral **H**ierarchical **T**opologies), an open-source FL framework for implementing arbitrary hierarchies in distributed environments.[1] Flight is the spiritual successor of our earlier FL framework known as FLoX (Kotsehub et al., 2022). Importantly, Flight supports both simulation and deployment on real devices across the computing continuum. To do so, Flight provides modular interfaces for control and data planes. In distributed environments, Flight can combine the Function-as-a-Service (FaaS) paradigm (via Globus Compute) and ProxyStore to decouple control from data and enable flexible, efficient, and performant deployment.

Flight tackles a range of important distributed systems problems in FL. The primary contributions of our work are:

- Flight, an open-source framework capable of defining and deploying hierarchical and asynchronous FL.
- Methods to separate data and control flow in FL using robust compute and data-management frameworks.
- Comprehensive evaluation showing that Flight scales to thousands of concurrent workers, can reduce global data transfer in hierarchical topologies, can reduce training time using asynchronous FL, and can efficiently train in a distributed environment.

The rest of this paper is as follows: Section 2 introduces hierarchical FL and terminology; Section 3 describes the Flight architecture; Section 4 discusses how custom FL strategies can be implemented in Flight; Section 5 evaluates Flight in simulated and real environments; and Section 6 concludes and discusses future directions.

[1] https://github.com/globus-labs/flight

## 2. Background & Related Work

To better contextualize the contributions of Flight, we first formally define *Hierarchical FL* (HFL) and introduce HFL *processes* (i.e., the various steps required to perform HFL, including model training, transferring of parameters, and aggregation). We then survey existing FL frameworks and identify gaps that highlight the need for Flight.

### 2.1. Hierarchical Federated Learning Processes

An HFL process is performed on a network topology made up of various connected devices. The topology of these devices are arranged as a tree (see Fig. 2). The device at the root of this tree—the *global aggregator* in Fig. 2—is responsible for coordinating the HFL process. The first task of the global aggregator is to instantiate an ML model, in this case a deep neural network (DNN), called the *global model*. The global model is initialized with random model parameters (or weights) $\omega_{t=0}$ where $t$ denotes the current *round* (i.e., $t = 0$ is the initialization round). A copy of the global model is sent by the global aggregator to a worker deployed on an end device in the network topology. We refer to the copies of the global model now hosted by workers as *local models*. Workers then locally train their copy of the model on their local data. Local model parameters $\omega_{t+1}^k$ for worker $k$ are updated according to Eq. (1):

$$\omega_{t+1}^k = \omega_t - \eta \nabla \ell(\omega_t, \mathcal{D}_k) \tag{1}$$

where $\eta$ is the learning rate hyperparameter, $\ell(\omega_t, \mathcal{D}_k)$ is the loss from using the global model parameters on the device's local dataset $\mathcal{D}_k$, and $\nabla \ell(\cdot)$ is the gradient from the loss.

In standard two-tier FL, when the workers finish training their local models they send their locally-updated model parameters back to the global aggregator. In HFL processes, workers instead send their locally-updated model parameters to their parent in the topology. This parent will either be an intermediate aggregator or the global aggregator. The intermediate aggregators in the tree are responsible for aggregating the model parameters returned by their topological children—irrespective of whether their children are leaves (workers) or *other* intermediate aggregators. Various strategies can be used to aggregate the local models (see Section 4).The simplest approach is to compute a simple average over the returned model parameters (see Eq. (2)).

$$\omega_{t+1}^k \triangleq \frac{1}{\text{children}(k)} \sum_{k' \in \text{children}(k)} \omega_{t+1}^{k'} \tag{2}$$

Like the intermediate aggregators, the global aggregator will also collect some set of returned model parameters from its immediate children and then perform aggregation. The result of this aggregation is then used to update the global model. Once the global model is updated, the global aggregator might perform other administrative tasks (e.g., testing the global model, checkpointing) before launching a new round of local model training on the end devices. Because we consider the intermediate aggregators as simply returning their own averaged parameters to their parents, we consider a two-tier FL process as a special case of HFL.

| | Hierarchies | | Modularity | | | Deployment | | | Control | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Simulation | | | | |
| Framework | Complex | Simple | Device Selection | Sync Aggr | Async Aggr | Single-Node | Multi-Node | Remote | Coord | Decoupled |
| LEAF (Caldas et al., 2018) | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | N/A | N/A |
| TFF (Bonawitz et al., 2019) | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | N/A | N/A |
| OpenFL (Foley et al., 2022) | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | N/A | N/A |
| FLoX (Kotsehub et al., 2022) | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ |
| APPFL (Li et al., 2023) | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| Flower (Beutel et al., 2022) | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| FedScale (Lai et al., 2022) | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| FedML (He et al., 2020) | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ |
| **Flight (ours)** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 1: Overview of federated learning frameworks. **Hierarchies** captures the topologies supported, from simple two- or three-tier to more complex topologies. **Modularity** captures the extensibility of the framework in terms of selecting participating devices and the use of synchronous or asynchronous aggregation. **Deployment** captures the scenarios supported from single-node to multi-node simulation and real deployment on devices. **Control** captures how the FL framework coordinates the FL process and if data and control planes are decoupled. "N/A" denotes that control coordination and data decoupling are not applicable in frameworks that do not support remote deployment.

The design of aggregation strategies is an active area of research in the FL community. The simplest strategy, `FedSGD`, performs a simple average, similar to Eq. (2), over returned model parameters in a two-tier setting. McMahan et al. proposed the alternative `FedAvg` strategy (McMahan et al., 2017), which generalizes `FedSGD` and is often seen as the standard aggregation strategy in FL. `FedAvg` foregoes a simple average and instead uses a weighted averaging method based on the amount of data at each end device. This can be defined as follows:

$$\omega_{t+1} \triangleq \sum_{k=1}^{K} \frac{n_k}{n} \omega_{t+1}^k. \tag{3}$$

The goal of `FedAvg` is to make the aggregation less sensitive to imbalanced data distributions. Further, Li et al. later proposed `FedProx` (Li et al., 2020) which generalizes `FedAvg` with even greater emphasis on data heterogeneity. Finally, alternative aggregation strategies exist for *Asynchronous Federated Learning* (AFL) (Xie et al., 2019) to incorporate individual local model updates as they arrive. A simple approach is:

$$\omega_{t+1} = \beta \cdot \omega_t + (1 - \beta) \cdot \omega_{t_{k'}}^k \tag{4}$$

where $t_{k'}$ is the time-step of the most recent update from worker $k'$ and $\beta \in (0, 1)$ is the step size.

### 2.2. Federated Learning Frameworks

Here, we briefly survey existing FL frameworks, with prominent frameworks summarized in Table 1.

TensorFlow Federated (TFF) (Bonawitz et al., 2019) and LEAF (Caldas et al., 2018), two of the first FL frameworks, focus on training models using on-premise simulations. TFF is developed and maintained by Google and is meant to simulate FL processes and the underlying statistical qualities of federated datasets. Currently, it provides a lot of foundational abstractions for federated computations and mathematics; it is limited in its use for real-world FL use cases because it is moreso a simulation framework. LEAF is a simple benchmarking framework for different FL scenarios using TensorFlow. It is not designed to be modular framework that enables rapid development of novel FL algorithms (e.g., aggregation algorithms). Additionally, it only provides the standard `FedAvg` algorithm for FL processes.

While FL was initially developed for communication efficiency (McMahan et al., 2017), its ability to operate across sites enables privacy preservation when data cannot be moved from a device (e.g., medical datasets). Some FL frameworks are more pointedly designed around the privacy-preservation benefits of FL. APPFL (Li et al., 2023) is a framework developed and maintained by a team at Argonne National Laboratory that performs cross-silo FL. APPFL is designed to be a platform that requires little technical expertise to use, including a graphical user interface to lessen the burden of entry. Similar to APPFL, SubstraFL (Galtier and Marini, 2019) is FL framework designed to enable medical research on naturally decentralized healthcare data. PySyft (Ziller et al., 2021) is a general framework that provides algorithms for private deep learning. Though not necessarily a framework specifically made for FL, it provides out-of-the-box support for common privacy-preserving algorithms including differential privacy and homomorphic encryption.

Frameworks such as FedLess (Grafberger et al., 2021), Flower (Beutel et al., 2022), FedScale (Lai et al., 2022), and OpenFed (Chen et al., 2023) accommodate FL training over distributed resources and provide interfaces to customize the training process. These frameworks enable larger experiments as well as deployment on devices. Additionally, frameworks like $\lambda$-FL (Jayaram et al., 2022), XFL (Wang et al., 2023), and Parrot (Tang et al., 2023) offer simple user interfaces, but lack support for broad deployments.

The popular Flower (Beutel et al., 2022) FL framework supports a wide range of environments for both simulated and real world experiments. For deployment, Flower uses gRPC for communication. Flower's experiments are client-driven but experimentally configured at the server. In this way, Flower represents a traditional client-server model where each is independently configured and waits on the other.

**Hierarchies:** We know of no other FL framework that, like Flight, supports HFL across hierarchical device networks in which a global aggregator may be connected to worker nodes by multiple intermediate aggregators in a tree topology. One

partial exception is FedML by He et al. (2020), which supports HFL in networks that link a global aggregator with multiple multi-GPU workers, with model training performed across GPUs on each individual worker followed by worker-localized aggregation before worker-aggregated parameters are returned to the global aggregator. While useful for cross-silo FL, this feature is not sufficient for more sophisticated decentralized systems, such as Internet-of-Things, sensor networks, and mobile edge computing.

**Modularity:** The ability to rapidly change FL strategies is crucial for experiments and real-world deployments (e.g., where some devices may be intermittently online). Device selection allows for devices to be sampled in a training round, various strategies can be applied (e.g., based on data distribution, training time, or previous impact on global model). Flower provides extensible interfaces for this purpose.

Most frameworks are designed to support different synchronous aggregation strategies (Bonawitz et al., 2019; Foley et al., 2022; Kotsehub et al., 2022; Beutel et al., 2022; He et al., 2020); however, there are a handful of frameworks, such as APPFL (Li et al., 2023) and FedScale (Lai et al., 2022), that support asynchronous aggregation. This mode of aggregation is necessary in environments with highly heterogeneous or unreliable compute resources, allowing results to be incorporated as they are returned.

**Deployment:** The vast majority of FL framework support single-node simulation and most now also support deploying simulations across several compute nodes and deploying on devices, with some notable exceptions (Bonawitz et al., 2019; Foley et al., 2022; Caldas et al., 2018). Most frameworks adopt a simple multi-processing approach for single node experiments. Support for simulating FL is a necessary feature for a FL framework for simple debugging before remote deployment. In addition to simple debugging, it is necessary for FL frameworks to enable simulation of FL processes for novel research in the field of FL. Not all FL researchers are necessarily interested in deploying on remote devices. For instance, FL researchers interested in developing novel privacy-preserving FL algorithms will only need to simulate FL to analyze the trade-offs of their proposed algorithms. However, an ideal FL framework will have support for both simulations and real-world deployment and interoperable switching between these two modes. Most existing FL frameworks primarily use client-server or RPC models for multi-node and on-device deployment. Notable exceptions include FLoX and APPFL, which can use the FaaS-based Globus Compute platform to deploy training operations, and TFF, which uses Kubernetes.

**Control:** FL frameworks take different approaches to managing the FL process. "Coord" in Table 1 refers to whether the process is driven by the topmost node (i.e., the "Coordinator" in Fig. 4) rather than being driven the bottom-most nodes (i.e., the "Workers" in Fig. 4). The former also suggests that you can change model configuration and training hyperparameters from a centralized location."Decoupled" refers to whether the control plane is decoupled from the data plane, meaning that the communication of logical components (i.e., code for jobs) and data are handled separately. It is important to be able to easily

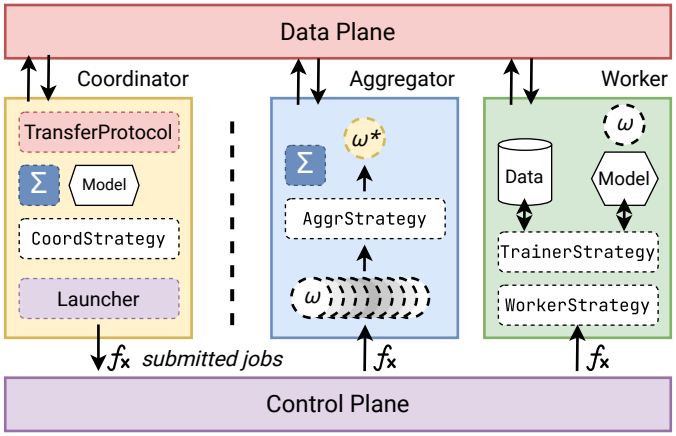


Figure 3: High-level view of Flight architecture. The `Coordinator` launches jobs to be run on `Aggregators` and `Workers` through the control plane, while data (e.g., model parameters, $\omega$) are transferred through a data plane. Each `Worker` trains its local copy of the model and sends back its locally-updated model to its parent (either the `Coordinator` or an `Aggregator`). Each `Aggregator` aggregates the responses of its children (`Workers` and other `Aggregators` alike). The `Coordinator` facilitates the entire process.

reconfigure the FL process, including aggregation topologies, worker workloads, experiment parameters, and the model being used. Most FL frameworks—such as FLoX (Kotsehub et al., 2022), APPFL (Li et al., 2023), FedScale (Lai et al., 2022), and FedML (He et al., 2020)—support such configurations; however device-driven approaches (Beutel et al., 2022) make reconfiguration challenging as device clients must be restarted to change parameters. This contrasts with other frameworks where new models and parameters can be pushed from the coordinator to reinitialize an experiment (Kotsehub et al., 2022).

As we consider more sophisticated FL topologies, decoupling the data and control plane is essential for efficient and scalable deployment. To the best of our knowledge, no existing framework separates control from data, nor do they provide a robust, standardized data communication method that is decoupled from the control plane. Finally, FL frameworks take different approaches to communication with various assumptions regarding connectivity. For example, frameworks relying on gRPC assume inbound network access (Li et al., 2023; Beutel et al., 2022) while others require SSH connections between devices (Lai et al., 2022).

## 3. Flight: Design & Implementation

Flight is an open-source Python library for implementing HFL processes. It is designed to be robust, scalable, and flexible with respect to deep learning models, how aggregation and training tasks are launched, and how parameters are transferred between devices. A high-level overview of Flight's design can be found in Fig. 3.

### *3.1.* Flight *Network Topologies*

Here, we introduce the programmatic abstractions for defining networks of connected devices in Flight. The network topology is defined as a directed graph using Net-

```yaml
MyCoordinator:
  kind: coordinator
  children: [ Aggr, Worker3 ]
  globus_compute_endpoint: null
  proxystore_endpoint: <UUID>
Aggr:
  kind: aggregator
  children: [ Worker1, Worker2 ]
  globus_compute_endpoint: <UUID>
  proxystore_endpoint: <UUID>
Worker1:
  kind: worker
  children: []
  globus_compute_endpoint: <UUID>
  proxystore_endpoint: <UUID>
...
```

Listing 1: FLIGHT network topology definition as a `yaml` file. Each node is defined as a dictionary with its key as its ID and various attributes: `kind`, `children`, `globus_compute_endpoint`, and `proxystore_endpoint`.

workX (NX) (Hagberg et al., 2008). We consider three *entity* types in FLIGHT networks: *(i)* `Coordinator`, *(ii)* `Aggregator`, and *(iii)* `Worker`. The type for each entity is assigned to the nodes in the underlying NX graph as an `enum` attribute.

Each entity type is implemented as a Python class with an extensible API enabling users to customize behavior. Each entity type is responsible for different tasks related to the execution of an HFL process. At a high level, the `Coordinator` has three key responsibilities: *(i)* maintaining the global model; *(ii)* submitting the appropriate training and aggregation jobs (implemented as pure functions) to the `Aggregators` and `Workers`; and *(iii)* acting as the *global aggregator* to aggregate the model parameters returned from each direct child (`Aggregators` or `Workers`). Both the `Aggregator` and `Workers` are entities that wait to receive jobs to run from the `Coordinator`. `Workers` simply train a local copy of the global model on local data. `Workers` then return a `JobResult`—a data class that contains the locally-updated model and other information related to the completion of the job (e.g., loss, time). The `JobResult` is sent to its parent node (either an `Aggregator` or the `Coordinator`). `Aggregator` jobs instantiate a `Future` with a callback that blocks until the `Futures` for the jobs of the `Aggregator`'s children have completed. `Aggregator` then will also return a `JobResult` to its parent. Using a common `JobResult` for `Aggregators` and `Workers` allows for `Aggregators` to aggregate what is returned to them regardless of whether they are returned by a `Worker` or another `Aggregator`. This design choice generalizes to support arbitrary hierarchical scenarios.

We define a legal network topology as follows. 1) The network must be a rooted directed tree. 2) There must be exactly one `Coordinator` node and it must be the root of the tree. 3) There must be at least one `Worker` node and each `Worker` node must be a leaf of the tree. 4) An `Aggregator` node can be neither the root nor a leaf of the tree; `Aggregators` are also not required. These rules enable many different types of hierarchical topologies (see Fig. 4).

FLIGHT topologies are defined using `yaml` files. An example of such a definition is shown in Listing 1. Entities in a FLIGHT topology have three properties: *(i)* an identifier, *(ii)* a flag indicating the entity type (i.e., coordinator, aggregator, or worker), and *(iii)* a list of children's identifiers. Optionally, if deploying on remote resources, the entity may include a Globus Compute endpoint ID and a ProxyStore endpoint ID for managing remote computation and data, respectively. These properties are discussed in Section 3.2.3 and Section 3.3, respectively.

Topology files can be loaded and used in FLIGHT by `flight.Topo.from_yaml(<filename>)`. Using NetworkX (Hagberg et al., 2008) to define system topologies gives FLIGHT the ability to load (or generate) a large variety of topology definitions.

### *3.2. Control Flow: Job Launching*

As mentioned earlier, in FLIGHT, the `Coordinator` is responsible for managing the execution of the HFL process. This includes the task of telling `Workers` to locally train a model and `Aggregators` to aggregate models. Other FL frameworks often apply a `Worker`-driven approach, as discussed in Section 2.2. That is, it is responsible for launching the training and aggregation tasks on the target entities (e.g., a local thread or a process on a remote device). In FLIGHT this is handled by a `Launcher`—a Python object that implements FLIGHT's `Launcher` interface for running jobs on arbitrary computing resources. The `Launcher` is an asynchronous interface to submit a Python function and any input arguments for execution on a target entity (we call the executing training or aggregation function a *task*). The interface returns a *future* to the `Coordinator`, enabling it to monitor execution and receive a callback when the task completes. *Futures* are also passed from the `Coordinator` to `Aggregators` such that they too can wait on a callback from a `Worker` or other `Aggregator`. The result wrapped in the *future* includes returned objects or exceptions. We implement the `Launcher` interface on top of Python's *concurrent.futures* `Executor` interface with additional requirements on the `submit()` method (e.g., submitting a `Job` and returning a `JobResult`). FLIGHT supports several modes for launching jobs necessary for an FL process. For convenience, FLIGHT includes `Launcher` implementations for three use cases: *(i)* local single-node simulation via threads or processes, *(ii)* local multi-node simulation via Parsl (Babuji et al., 2019), and *(iii)* remote execution via Globus Compute (Chard et al., 2020). For convenience, FLIGHT provides a high-level function for launching any type of FL process. A small example of launching an FL process in FLIGHT is shown in Listing 2.

#### *3.2.1. Single-node simulation*

FLIGHT provides a `LocalLauncher` that implements the `Launcher` interface using Python's standard *concurrent.futures* `ProcessPoolExecutor` and `ThreadPoolExecutor`. The `LocalLauncher` enables rapid prototyping for algorithm design, experimentation, simulations, and debugging tasks related to FL/HFL.

#### *3.2.2. Multi-node simulation*

FLIGHT implements a `ParslLauncher` to run FL/HFL processes on high-performance computing systems. Parsl (Babuji

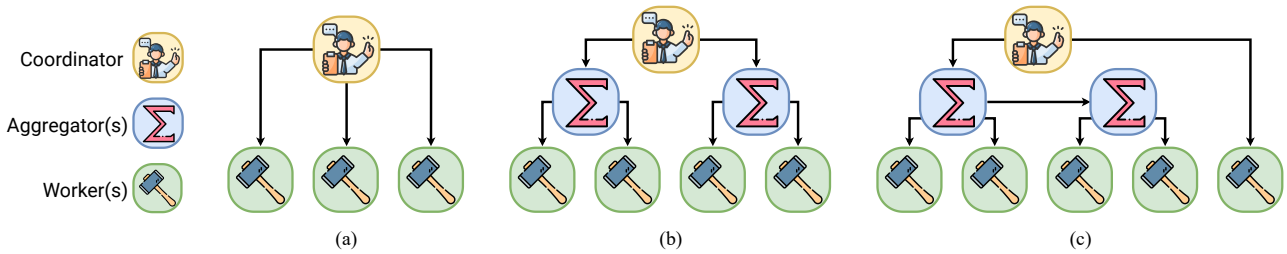


Figure 4: Example legal FLIGHT network topologies: (a) simple two-tier network; (b) simple three-tier hierarchical network; (c) complex hierarchical network.

```python
import torch
import flight as fl

from torchvision.datasets import MNIST

class MyModule(fl.TorchModule):
  # Model definition.

topo = fl.Topo.from_yaml('my-topo.yaml')
mnist = MNIST(...)
fed_data = fl.federated_split(
  topo, mnist, num_classes=10,
  sample_alpha=1.0, label_alpha=1.0
)
trained_model, results = fl.federated_fit(
  topo,
  MyModule(),
  fed_data,
  strategy='fedavg',
  where='local', # 'globus-compute',
  kind='sync',   # 'async'
)
results.to_csv('my-results.csv')
```

Listing 2: Example FLIGHT program that showcases a simple FLIGHT program. Listing 1 provides an example `my-topo.yaml` file.

et al., 2019), a scalable parallel programming library for Python, is designed to run workloads on parallel and distributed computing systems (e.g., institutional clusters, supercomputers, and clouds). Parsl's modular architecture defines an extensible `Executor` interface via which different runtime executors can be used for different scenarios: e.g., HighThroughputExecutor (HTEX) (Babuji et al., 2019), RADICALPilot (Merzky et al., 2022), and WorkQueue/TaskVine (Sly-Delgado et al., 2023). These executors implement the same asynchronous API but differ in how tasks are executed. Parsl supports provisioning of resources from various compute resources, including batch schedulers (e.g., Slurm, PBS), clouds (e.g., AWS), and container orchestration systems (e.g., Kubernetes). FLIGHT's `ParslLauncher` instantiates a Parsl process using a user-defined Parsl configuration (e.g., specifying executor options such as batch queue, account, walltime). FLIGHT tasks are then submitted to Parsl for execution and results are retrieved via the returned future.

#### 3.2.3. *Execution on remote endpoints*

FLIGHT includes a `GlobusComputeLauncher` to enable simple and convenient remote execution of functions for FL/HFL processes. Globus Compute (Chard et al., 2020) implements the Function-as-a-Service (FaaS) paradigm enabling execution of Python functions. It combines a single cloud-hosted service with an ecosystem of user-deployed *endpoints*. Thus, FLIGHT can use the cloud-hosted Globus Compute service to orchestrate execution of tasks (e.g., `Aggregator` and `Worker` tasks) on arbitrary remote devices. As the endpoint software is lightweight—a pip-installable Python agent that communicates via cloud-hosted message queues—it is easily deployed on diverse compute devices. Further, it requires only outbound connectivity to the Globus Compute cloud service, therefore addressing challenges with firewalls and Network Address Translation (NAT) used in many edge environments. The endpoint software builds on Parsl and is therefore equipped to dynamically provision and then execute tasks on diverse systems, including HPC clusters and Kubernetes. Finally, Globus Compute endpoints have been deployed nearly 10,000 times around the world (Bauer et al., 2024); on those systems, FLIGHT can be used without needing to deploy any new infrastructure.

FLIGHT's `GlobusComputeLauncher` uses Globus Compute's Python SDK and executor interface to submit tasks. FLIGHT users must provide OAuth 2 access tokens to instantiate the executor and they must also provide the set of compute endpoints to be used in the network topology definition. FLIGHT submits `Aggregator` and `Worker` tasks to Globus Compute as required and tracks results via the Globus Compute `Future` returned to FLIGHT.

### 3.3. *Data Flow: Federated Data Transfer*

The purpose of HFL is to improve performance by distributing model aggregation to different locations. Thus, it would be both inefficient and costly if all data had to pass through the `Coordinator` rather than be passed directly between the participating entities. Our solution to this problem is to decouple the transfer of data from the execution of the task itself. We effectively use a control layer, via which small function invocations and small function results are sent via the launcher (e.g., via Globus Compute's cloud service or through Parsl's DataFlowKernel). We use ProxyStore (Pauloski et al., 2023, 2024) to move larger data (e.g., models and weights) directly between the tasks running on `Workers` and `Aggregators`. ProxyStore is a framework that uses Python proxy objects to provide *pass-by-reference* semantics in distributed computing environments. Given some data $x$, ProxyStore will generate a proxy object $p(x)$ for the data $x$. This proxy—essentially a small reference to the data—can then be sent with the computing task via

the launcher. When the data is needed by the task being executed on a device, it will then be transferred by ProxyStore using the user's choice of transfer protocol. ProxyStore supports many data transfer protocols, including Redis and *Remote Direct Memory Access* (RDMA). It also implements a peer-to-peer transfer solution, referred to as the `EndpointConnector`, for direct communication between endpoints. This transfer mechanism uses UDP hole punching to establish connections between devices that are behind firewalls and that use NAT for private networks.

### *3.4. Execution Schemes: Synchronous and Asynchronous FL*

Now, we describe FLIGHT's execution schemes to perform FL/HFL either synchronously or asynchronously. Specifically, we describe how jobs are launched and the timing of when their returned results are aggregated. It is worth noting that, for both execution schemes, the role of the `Coordinator` remains the same. Its role is to oversee and manage the execution during the lifetime of the FL/HFL process.

FLIGHT's modular design allows for customization for different FL algorithms and strategies. This modularity is achieved via the `Strategy` abstraction. As discussed more fully in Section 4, a `Strategy` provides callbacks that can be programmed by users to customize the execution of their FL/HFL processes. For the sake of brevity, we forego mentioning *all* available callbacks when describing the execution scheme.

#### *3.4.1. Synchronous HFL (SHFL)*

In the SHFL case, FLIGHT begins a series of "rounds". In each round, the `Coordinator` selects `Workers` to participate in the round (i.e., to train their local model). Depending on the configuration, the selected workers can be either a subset of, or all, workers. The `Coordinator` then identifies all `Aggregators` that are on the path from the `Coordinator` to each selected `Worker`. If all `Workers` are selected to perform local training, then it follows that all `Aggregators` are also selected. Next, the `Coordinator` submits jobs to the selected `Workers` and the relevant `Aggregators` by using its configured `Launcher`. These submissions are generated in a breadth-first search-like fashion where the `Coordinator` (at the root) traverses each of its children and submits the appropriate (training or aggregation) job to each, retaining a corresponding `Future` for each child. The `Coordinator` submits aggregation jobs to each `Aggregator`, including for each a list of the `Aggregator`'s children's `Futures` as an argument. The `Aggregators` begin their aggregation jobs once their children's respective `Futures` have completed. Because we separate the data flow from the control flow via ProxyStore (see Section 3.3), the data from the results of the child `Futures` for each `Aggregator` is never sent back to the `Coordinator`. The data (i.e., model parameters) are transferred to the `Aggregator` that depends on it. When `Workers` are traversed as the leaves of the tree, the `Coordinator` submits a local training job to these entities. Each of which will eventually return a `JobResult` to its parent entity. The `Coordinator`, like the `Aggregators`, waits for the completion of its children's `Futures`. When the `Futures` complete, the `Coordinator` aggregates to update the global model. It then performs additional tasks (e.g., evaluating the global model, processing results) and restarts the process until a set number of rounds concludes.

#### *3.4.2. Asynchronous FL (AFL)*

FLIGHT currently supports only two-tier AFL topologies. Supporting AFL with arbitrary hierarchies would require implementing a new type of launcher with the ability to maintain stateful entities at each level. In SHFL, the `Coordinator` and `Aggregators` remain idle until all their children's `Futures` resolve. Specifically, the `Coordinator` launches local training jobs on each `Worker` and maintains a list of `Futures`. The `Coordinator` takes action as each `Future` is completed. As soon as one `Future` completes, the `Coordinator` retrieves the locally-updated model parameters from the `Future` result and performs a partial aggregation to update the global model—see Eq. (4). The `Coordinator` then launches a new local training job on the worker whose `Future` just completed and appends its new `Future` to its list of `Futures`.

### *3.5. Enabling Machine Learning*

We describe now how FLIGHT works with existing deep learning frameworks to implement, train, and use models and manage datasets.

**Deep learning models.** FLIGHT relies on the PyTorch (Paszke et al., 2019) framework to define, train, and evaluate AI models. We chose PyTorch due to its ubiquity in both deep learning and FL. FLIGHT implements the `FlightModule` class which extends the `torch.nn.Module` class for defining a neural network. This class offers callbacks similar to the `LightningModule` from PyTorch Lightning (Falcon and The PyTorch Lightning team, 2019). Further, the `FlightModule` requires only that two of these callbacks are implemented by the user (i.e., `training_step()` and `configure_optimizers()`). FLIGHT can then train the model by launching remote jobs on `Workers` without further specification from users. We choose to implement the `FlightModule` class rather than use Lightning's default `LightningModule` so as not to require the latter as a dependency given its size and complexity—an important consideration when deploying on edge devices. However, in future work we will add support for PyTorch Lightning. FLIGHT also provides a callable `LocalTrainJob` class that provides a common interface to define how models are trained on `Workers` to allow highly-custom local training loops.

**Datasets** in FLIGHT are based on the PyTorch `Dataset` abstraction. FLIGHT requires that each `Worker` have its own data. For simulated FL processes, this is typically based on partitioning a common dataset (e.g., FashionMNIST (Xiao et al., 2017)) into separate subsets for each worker. FLIGHT simplifies this process by providing a class (`FederatedSubsets`) that allows users to define how a dataset is partitioned and allows users to easily control the distribution of data across `Workers`. This class is effectively a dictionary where the key is the `Worker` ID and the value is a PyTorch `Subset` of the data. Crucial to the needs of HFL deployments, FLIGHT also allows users to load data local on the `Workers`.

```python
@dataclass(frozen=True)
class Strategy:
    coord_strategy  : CoordinatorStrategy
    aggr_strategy   : AggregatorStrategy
    worker_strategy : WorkerStrategy
    trainer_strategy: TrainerStrategy
```

Listing 3: Definition of the `Strategy` class in Flight.

## 4. Strategies

FL is an active research domain and new algorithms are frequently proposed to meet new requirements relating to system heterogeneity, data/statistical heterogeneity, and many other challenges. FL frameworks must thus be flexible and customizable to meet the needs of both FL researchers and practitioners so that novel and custom algorithms can be readily implemented and deployed. Flight implements a modular and extensible abstraction for specifying such algorithms, which we refer to as a `Strategy`. A `Strategy` is essentially a wrapper for an FL algorithm or solution that provides specific implementation details necessary to execute the HFL process. In designing this architecture we considered the needs of various aggregation methods, asynchronous and hierarchical FL, and privacy-preserving FL.

Specifically, a `Strategy` is made up of the following components: *(i)* `CoordinatorStrategy`, *(ii)* `AggregatorStrategy`, *(iii)* `WorkerStrategy`, and *(iv)* `TrainerStrategy` (see Listing 3). We refer to these components as "sub-strategies" because they implement the respective logic that is run on all of the entities in a topology for an HFL process. The `CoordinatorStrategy` is run on the `Coordinator` and provides callbacks for users to implement custom worker selection algorithms based on worker conditions. The `AggregatorStrategy` is run on the `Aggregators` and provides callbacks for custom parameter aggregation algorithms. The `WorkerStrategy` and `TrainerStrategy` sub-strategies are both run on the `Workers`; the former provides callbacks more specific to the execution of the entire local training job performed by the worker (e.g., caching/recording system conditions, pre-processing data on the worker) while the latter provides callbacks specific exclusively to the training loop (e.g., modifying the loss before back-propagation).

A new Flight strategy can be defined simply by composing four existing sub-strategies. Or, a user can provide custom implementations of one or more of the sub-strategy classes. To simplify the development of custom implementations, Flight provides a default implementation for each sub-strategy; a user can inherit from such a default implementation and override only what is needed. As an example, we present our `FedAvg` (McMahan et al., 2017) implementation in Listing 4. For user convenience, Flight provides implementations of strategies for common FL algorithms, including `FedSGD`, `FedAvg` (McMahan et al., 2017), `FedProx` (Li et al., 2020), and `FedAsync` (Xie et al., 2019).

```python
class FedAvgAggr(DefaultAggregatorStrategy):
  def aggregate_params(
    self, state, children_states, children_params
  ) -> Params:
    weights = {}
    for node, child_state in children_states.items():
      weights[node] = child_state['num_data_samples']
    state['num_data_samples'] = sum(weights.values())
    return average_params(
      # Flight-provided utility fn
      children_params, weights=weights
    )

class FedAvgWorker(DefaultWorkerStrategy):
  def before_training(
    self, state: WorkerState, data
  ) -> tuple[WorkerState, t.Any]:
    state['num_data_samples'] = len(data)
    return state, data

class FedAvg(Strategy):
  def __init__(self, **kwargs):
    super().__init__(
      coord_strategy=FedSGDCoordinator(**kwargs),
      aggr_strategy=FedAvgAggr(),
      worker_strategy=FedAvgWorker(),
      trainer_strategy=DefaultTrainerStrategy(),
    )
```

Listing 4: This custom implementation of the `FedAvg` algorithm is constructed by subclassing the default implementations of `AggregatorStrategy` and `WorkerStrategy`. The required callbacks are then overriden and those implementations are wrapped in the `FedAvg` class which contains all the logical components. Note the use of the `FedSGD` algorithm's `CoordinatorStrategy` on line 24.

| Model | Params | Size |
|---|---|---|
| TinyNet | 2 | 8 bytes |
| SmallNet (PyTorch, 2024) | 62K | 242 KB |
| SqueezeNet (Iandola et al., 2016) | 1.2M | 5 MB |
| ResNet-18 (He et al., 2016) | 11M | 45 MB |
| ResNet-50 (He et al., 2016) | 23M | 98 MB |
| ResNet-152 (He et al., 2016) | 60M | 231 MB |

Table 2: Sizes of the models considered in our experiments.

### 4.1. *Defining Custom Strategies*

## 5. Evaluation

We evaluate Flight's scalability in multi-node experiments on an HPC cluster, hierarchical communication costs and asynchronous FL makespan in single node experiments, and performance in on-device experiments conducted on AWS.

### 5.1. Flight *Scaling Tests*

We explore how Flight scales with different model sizes and compare with the state-of-the-art Flower framework. For a fair comparison against Flower, we consider only two-tier topologies and scale the number of workers and model size.

**Testbed:** We use SDSC Expanse, a 5.16 petaflop cluster with 728 CPU compute nodes, each with two 64-core AMD EPYC 7742 processors and 256 GB memory. The nodes are connected with a 100 Gb/s InfiniBand interconnect.

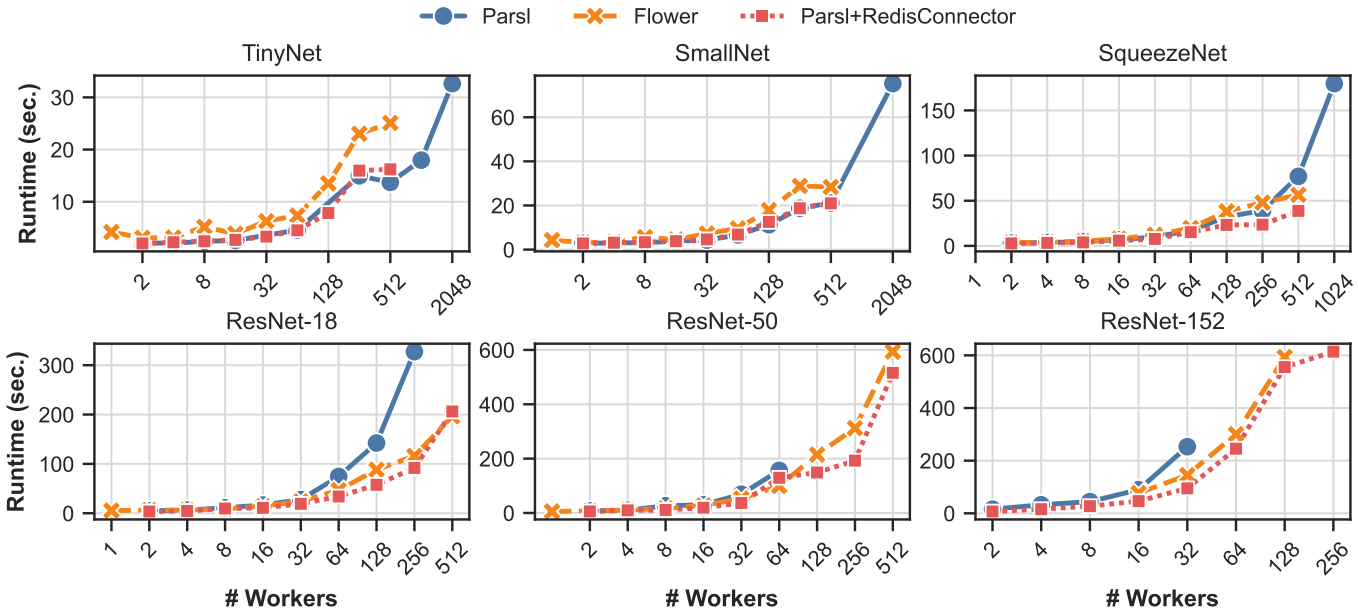


Figure 5: Weak scaling results: Runtime of Flower vs. FLIGHT using Parsl and Parsl+RedisConnector for a series of increasingly complex models (see Table 2). Results confirm that our proposed FLIGHT framework provides better performance and, in some cases, also scales to more workers.

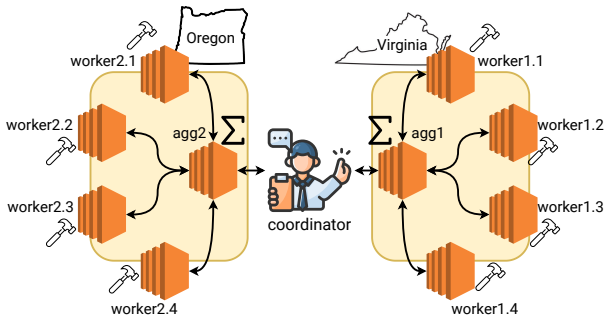


Figure 6: The hierarchical topology of the EC2 testbed employed for remote tests with `GlobusComputeLauncher` and ProxyStore.

**Models:** We use the six image classification model architectures in Table 2 to evaluate FLIGHT performance and overheads. TinyNet, the smallest model, comprises a single linear layer with 1 input channel and 1 output channel. SmallNet is a small neural network from the Pytorch documentation (PyTorch, 2024). The ResNet (He et al., 2016) and SqueezeNet (Iandola et al., 2016) implementations are those included in PyTorch.

**Configuration:** We configure FLIGHT and Flower with a two-tier hierarchy by deploying a `Coordinator` and server, respectively, on a single compute node. We then deploy workers on separate compute nodes and increase the number of workers from 1 to 128 (each using a single core on a single node) and then up to 2048 (on 16 nodes). To evaluate scaling performance and overhead we reduce the fixed processing costs to measure worst-case performance. To do so, we configure both frameworks such that the clients do not train or evaluate the model and instead simply pass a model with randomized weights back to/from the coordinator. We use `FedAvg` to aggregate the models. We record the time from when the clients start until the aggregator has received and aggregated all models.

In FLIGHT, we configure the `Coordinator` with a single model with randomized weights. FLIGHT then distributes that model to each `Worker` who instantiate and then return the model. We configure FLIGHT with the `ParslLauncher` and compare Parsl with and without ProxyStore to measure the benefits of decoupling control from data plane. We configure ProxyStore to use the RedisConnector and deploy Redis on the same node as the `Coordinator`. Flower is a client-driven framework and thus we first initialize random models at each client. The server then picks a client at random and distributes their model weights to the other clients.

**Results:** Fig. 5 shows weak scaling results as we increase the number of workers involved in training. We see that FLIGHT with Parsl outperforms Flower for smaller models, achieving faster results and scaling beyond Flower (2048 compared to 512 workers). As we scaled Flower to 1024 and 2048 workers we observed gRPC errors that prevented aggregation. We did not scale Parsl beyond 2048 nodes due to limited allocation. For the larger ResNet models, Flower slightly outperformed FLIGHT with Parsl, a difference that we attribute to the efficiency of the gRPC protocol and the fact that Parsl is not optimized to move large data volumes over its ZMQ-based protocol. FLIGHT with ProxyStore's RedisConnector overcomes this limitation and we see better runtime and scalability than Flower for all models considered.

### 5.2. *Distributed On-Device Deployment*

We now evaluate the use of FLIGHT in a distributed testbed to replicate a real-world deployment. We use Globus Compute

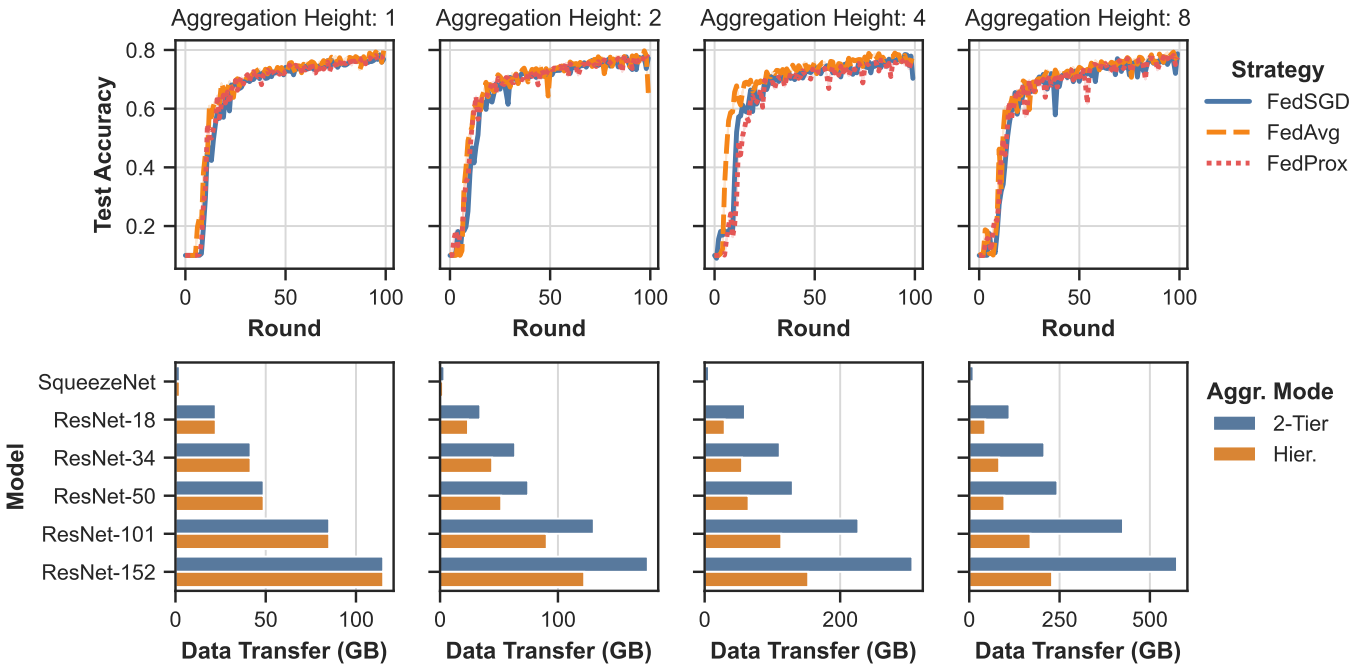


Figure 7: *Top row*: Test accuracy while training over the FashionMNIST dataset using three different strategies. Testing is done on the `Coordinator`. *Bottom row*: Comparison of the total data transferred across the network (implemented as a balanced tree with 256 leaf workers) at each round. For both 2-tier and HFL, the data transfer analysis assumes the same communication topology, though HFL `Aggregators` aggregate the models on return to the `Coordinator`.

and ProxyStore.

**Testbed:** We constructed the three-layer, 11-node hierarchical FLIGHT topology shown in Fig. 6. The coordinator is deployed on an Apple Silicon M1 laptop with 32GB RAM while the eight workers and two aggregators are deployed on 10 `m2.small` EC2 instances (2vCPU, 4GB RAM) on AWS cloud. We deployed one aggregator and four workers in each of AWS's Virginia and Oregon regions. We configured each instance with both a Globus Compute endpoint and a ProxyStore endpoint. We used ProxyStore's `EndpointConnector` for wide-area data transfer.

**Model and Data:** We use `SmallNet` (see Table 2) to perform image classification on the FashionMNIST dataset. For simplicity, we use the standard stochastic gradient descent algorithm for optimizing the model parameters, with a learning rate $\eta = 0.01$. To obtain a non-IID data distribution, as is common in decentralized data (McMahan et al., 2017), we partition data across the eight `Workers` by using a dual Dirichlet distribution. Specifically, the number of data samples at each worker follows the Dirichlet distribution with shape parameter $\alpha = 3.0$, and the distribution of labels across each worker follows a Dirichlet distribution with shape parameter $\alpha = 1.0$.

**Results:** Fig. 8 shows the *training accuracies* obtained over time at each worker (above) and, averaged across workers, at the two coordinators (below) when running this HFL application on our testbed. In the lower plot, we also show the *accuracy* reported by an evaluation step performed on the `Coordinator`. The test accuracy converges with the average training accuracies of the two sets of `Workers`.

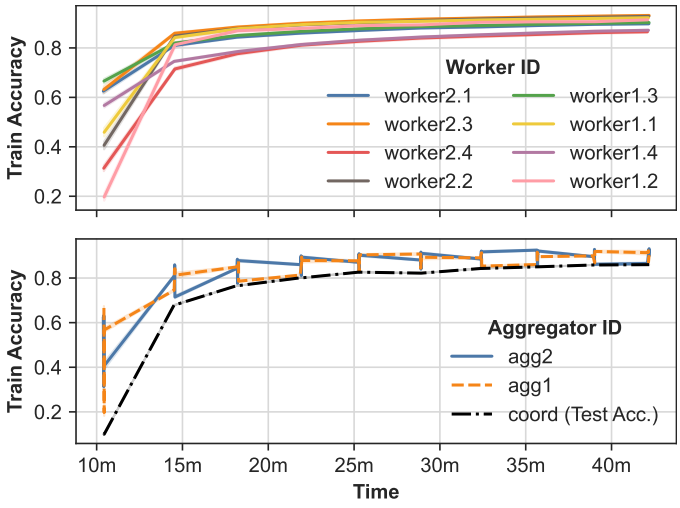


Figure 8: Training accuracy over time at `Workers` and `Aggregators`, and test accuracy at `Coordinator`, for an HFL application (image classification with `SmallNet`) on an 11-node distributed testbed.

### 5.3. Hierarchical FL vs. Two-Tier FL: Communication Cost

HFL takes advantage of the multiple hops necessary in network topologies by using `Aggregators` to perform intermediate aggregation rather than directly traversing all links between `Workers` and the `Coordinator` with all models as in two-tier systems. We use FLIGHT's `LocalLauncher` to simulate the benefits of HFL by comparing the total amount of data transferred over the network. We initialize the simulation with a balanced tree topology with 256 leaves (i.e., 256 `Workers`) and vary its

height from $h = 1, 2, 4, 8$. We generate these tree topologies using the `balanced_tree(·)` function from NetworkX (Hagberg et al., 2008). This function takes a height ($h$) and branching factor ($b$) as an argument, which we compute by $b = 256^{1/h}$. Given the balanced trees, we use the model sizes from Table 2 to numerically compute the data transfer costs for two-tier FL and HFL. The mathematical details behind these calculations are presented in Appendix A. The reduction in transfer volume achieved by HFL is shown in Fig. 7. The benefit of HFL becomes increasingly notable with topology height, saving 60.13% in the case of a height of 8 with the ResNet-152 model. Fig. 7 also shows comparable test accuracy during training across all topology heights, though test accuracy becomes less stable as height increases.

While comprehensively evaluating the impact of hierarchies on FL model performance is beyond the scope of this work, Flight provides a framework to enable research into such questions.

### 5.4. Comparing Synchronous vs. Asynchronous FL

We evaluate Asynchronous FL (AFL) on a two-tier topology with 12 `Workers` using the `LocalLauncher`. We partition the FashionMNIST (Xiao et al., 2017) dataset across these workers with a dual Dirichlet distribution (via the Flight-provided `federated_split()` utility function) such that the workers have non-IID distributions of labels and modestly imbalanced numbers of samples. We use the `LocalLauncher` to start 12 workers, and train SmallNet (see Table 2) with both `FedAvg` (synchronous) and `FedAsync` (asynchronous). In both cases, we configure the FL process to run for 20 rounds (i.e., each worker locally trains the model 20 times).

The visualization in Fig. 9 shows via solid colors when the workers are engaged in training in each case. We see that synchronous FL (SFL) leaves some workers idle for long periods, whereas AFL keeps the workers more active. Overall, AFL has a nearly 20% shorter makespan than SFL. Workers in the AFL run were idle, on average, just 0.061% of the time (standard deviation 0.013%) until they completed their final round, while workers in SFL run were idle 14.858% of the time (standard deviation 11.313%). While a comprehensive analysis of the trade-offs between SFL and AFL is beyond the scope of this paper—such analyses are the focus of related work (Chen et al., 2020; Xie et al., 2019)—we find that AFL achieved a training accuracy of 87% whereas SFL achieved 90% accuracy.

## 6. Conclusion & Future Directions

We have presented Flight, the first FL framework to enable on-device and simulated FL in complex hierarchies. Flight is designed for distributed deployment and adopts novel approaches for decoupling FL control from data movement as well as use of asynchronous communication. Support for various launchers allows for simple deployment in different scenarios and scaling from local processes, to parallel execution on a cluster, to wide-area deployment. Flight's extensible Python implementation and integration with ML frameworks allows it to be easily adopted and customized for various scenarios. Our experiments demonstrate that Flight can scale in multi-node simulations beyond state-of-the-art frameworks. We also show the ability to reduce data communication requirements by more than 60% percent through the use of HFL. Our simulation of AFL demonstrates a nearly 20% reduction in overall makespan as well as substantially decreased resource idle time. Finally, deployment of HFL in a distributed environment of 11 nodes shows that Flight enables effective learning across `Workers` and `Aggregators`.

In future work we will investigate the generalizability of FL algorithms to large scale experiments, the efficient exchange of model weights across different systems, and automated topology configuration to best suit a hierarchical FL workflow.

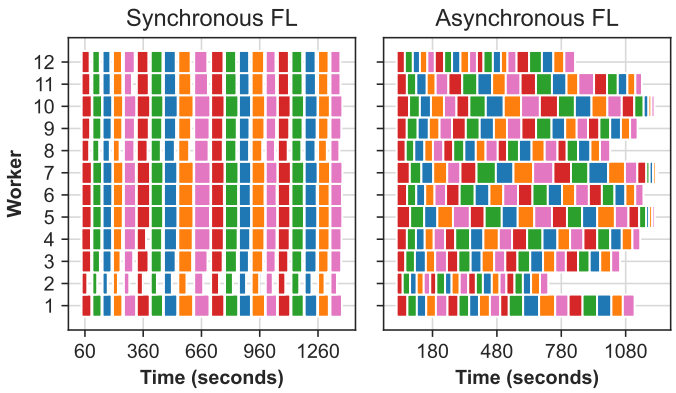


Figure 9: Synchronous vs. asynchronous FL in Flight. Each horizontal bar indicates a worker performing local training, with color indicating round number. Vertically aligned white regions in the Synchronous case indicate when workers sync during aggregation before beginning the next round. In contrast, the Asynchronous case has clear overlaps where workers are working.


## Acknowledgements

This work was supported in part by NSF 2004894 and Laboratory Directed Research and Development funding from Argonne National Laboratory under U.S. Department of Energy under Contract DE-AC02-06CH11357 and used resources of the Argonne Leadership Computing Facility and SDSC's Expanse Supercomputer.


## Appendix A. Method for Calculating Data Transfer Costs in Hierarchical Topologies

Following the description of the the balanced tree topologies in Section 5.3. The data volume can be computed by $2 \cdot E \cdot M$ where $E$ is the number of edges/links/hops in the topology and $M$ is the size of the model. By comparison, the total transfer volume in the two-tier case can be estimated by $(E \cdot M) + (l \cdot h \cdot M)$ where $l$ is the number of leaves (i.e., `Workers`) and $h$ is the height of the topology. The first term is the initial broadcast and the second term is the response. From this formulation, it is clear why the total data transfer in two-tier aggregation is larger than hierarchical aggregation. Further, even without

considering transfers across multiple heights, the hierarchical model reduces the data volume handled by the `Coordinator`.

## References


Abad, M.S.H., Ozfatura, E., Gunduz, D., Ercetin, O., 2020. Hierarchical federated learning across heterogeneous cellular networks, in: IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 8866–8870.

Abdellatif, A.A., Mhaisen, N., Mohamed, A., Erbad, A., Guizani, M., Dawy, Z., Nasreddine, W., 2022. Communication-efficient hierarchical federated learning for IoT heterogeneous systems with imbalanced data. Future Generation Computer Systems 128, 406–419.

Ali, M., Naeem, F., Tariq, M., Kaddoum, G., 2022. Federated learning for privacy preservation in smart healthcare systems: A comprehensive survey. IEEE Journal of Biomedical and Health Informatics 27, 778–789.

Almurshed, O., Patros, P., Huang, V., Mayo, M., Ooi, M., Chard, R., Chard, K., Rana, O., Nagra, H., Baughman, M., et al., 2022. Adaptive edge-cloud environments for rural AI, in: IEEE International Conference on Services Computing, pp. 74–83.

Babuji, Y., Woodard, A., Li, Z., Katz, D.S., Clifford, B., Kumar, R., Lacinski, L., Chard, R., Wozniak, J.M., Foster, I., Wilde, M., Chard, K., 2019. Parsl: Pervasive parallel programming in Python, in: 28th International Symposium on High-Performance Parallel and Distributed Computing, pp. 25–36.

Baccour, E., Mhaisen, N., Abdellatif, A.A., Erbad, A., Mohamed, A., Hamdi, M., Guizani, M., 2022. Pervasive AI for IoT applications: A survey on resource-efficient distributed artificial intelligence. IEEE Communications Surveys & Tutorials 24, 2366–2418.

Barabási, A.L., Dezső, Z., Ravasz, E., Yook, S.H., Oltvai, Z., 2003. Scale-free and hierarchical structures in complex networks, in: AIP Conference Proceedings, AIP Publishing. pp. 1–16.

Bauer, A., Pan, H., Chard, R., Babuji, Y., Bryan, J., Tiwari, D., Foster, I., Chard, K., 2024. The Globus Compute dataset: An open function-as-a-service dataset from the edge to the cloud. Future Generation Computer Systems 153, 558–574.

Beutel, D.J., Topal, T., Mathur, A., Qiu, X., Fernandez-Marques, J., Gao, Y., Sani, L., Li, K.H., Parcollet, T., de Gusmão, P.P.B., NicholasD.Lane, 2022. Flower: A friendly federated learning framework. arXiv preprint arXiv:2007.14390 .

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

Caldas, S., Duddu, S.M.K., Wu, P., Li, T., Konečnỳ, J., McMahan, H.B., Smith, V., Talwalkar, A., 2018. LEAF: A benchmark for federated settings, in: Workshop on Federated Learning for Data Privacy and Confidentiality. ArXiv preprint arXiv:1812.01097.

Chard, R., Babuji, Y., Li, Z., Skluzacek, T., Woodard, A., Blaiszik, B., Foster, I., Chard, K., 2020. FuncX: A federated function serving fabric for science, in: 29th International Symposium on High-performance Parallel and Distributed Computing, pp. 65–76.

Chen, D., Tan, V.J., Lu, Z., Wu, E., Hu, J., 2023. OpenFed: A comprehensive and versatile open-source federated learning framework, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 5017–5025.

Chen, Y., Ning, Y., Slawski, M., Rangwala, H., 2020. Asynchronous online federated learning for edge devices with non-iid data, in: 2020 IEEE International Conference on Big Data (Big Data), IEEE. pp. 15–24.

Falcon, W., The PyTorch Lightning team, 2019. PyTorch Lightning. URL: `https://github.com/Lightning-AI/lightning`, doi:`10.5281/zenodo.3828935`.

Foley, P., Sheller, M.J., Edwards, B., Pati, S., Riviera, W., Sharma, M., Moorthy, P.N., Wang, S.h., Martin, J., Mirhaji, P., Shah, P., Bakas, S., 2022. OpenFL: The open federated learning library. Physics in Medicine & Biology URL: `http://iopscience.iop.org/article/10.1088/1361-6560/ac97d9`, doi:`10.1088/1361-6560/ac97d9`.

Galtier, M.N., Marini, C., 2019. Substra: A framework for privacy-preserving, traceable and collaborative machine learning. arXiv preprint arXiv:1910.11567 .

Grafberger, A., Chadha, M., Jindal, A., Gu, J., Gerndt, M., 2021. FedLess: Secure and scalable federated learning using serverless computing, in: IEEE International Conference on Big Data, IEEE. pp. 164–173.

Hagberg, A., Swart, P., S Chult, D., 2008. Exploring network structure, dynamics, and function using NetworkX. Technical Report. Los Alamos National Lab, Los Alamos, NM (United States).

He, C., Li, S., So, J., Zeng, X., Zhang, M., Wang, H., Wang, X., Vepakomma, P., Singh, A., Qiu, H., Zhu, X., Wang, J., Shen, L., Zhao, P., Kang, Y., Liu, Y., Raskar, R., Yang, Q., Annavaram, M., Avestimehr, S., 2020. FedML: A research library and benchmark for federated machine learning. arXiv preprint arXiv:2007.13518 .

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778. doi:`10.1109/CVPR.2016.90`.

Hu, Y.C., Patel, M., Sabella, D., Sprecher, N., Young, V., 2015. Mobile edge computing—A key technology towards 5G. White paper 11. ETSI. `https://www.etsi.org/images/files/etsiwhitepapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf`.

Hudson, N., Hossain, M.J., Hosseinzadeh, M., Khamfroush, H., Rahnamay-Naeini, M., Ghani, N., 2021. A framework for edge intelligent smart distribution grids via federated learning, in: International Conference on Computer Communications and Networks, IEEE. pp. 1–9.

Hudson, N., Oza, P., Khamfroush, H., Chantem, T., 2022. Smart edge-enabled traffic light control: Improving reward-communication trade-offs with federated reinforcement learning, in: IEEE International Conference on Smart Computing, IEEE. pp. 40–47.

Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J., Keutzer, K., 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size. `arXiv:1602.07360`.

Jayaram, K., Muthusamy, V., Thomas, G., Verma, A., Purcell, M., 2022. Lambda FL: Serverless aggregation for federated learning, in: International Workshop on Trustable, Verifiable and Auditable Federated Learning, p. 9.

Konečnỳ, J., McMahan, H.B., Ramage, D., Richtárik, P., 2016. Federated optimization: Distributed machine learning for on-device intelligence. arXiv preprint arXiv:1610.02527 .

Kotsehub, N., Baughman, M., Chard, R., Hudson, N., Patros, P., Rana, O., Foster, I., Chard, K., 2022. FLoX: Federated learning with FaaS at the edge, in: 2022 IEEE 18th International Conference on e-Science (e-Science), IEEE. pp. 11–20.

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

Li, T., Sahu, A.K., Zaheer, M., Sanjabi, M., Talwalkar, A., Smith, V., 2020. Federated optimization in heterogeneous networks. Proceedings of Machine learning and systems 2, 429–450.

Li, Z., He, S., Chaturvedi, P., Hoang, T.H., Ryu, M., Huerta, E., Kindratenko, V., Fuhrman, J., Giger, M., Chard, R., Kim, K., Madduri, R., 2023. APPFLx: Providing privacy-preserving cross-silo federated learning as a service, in: IEEE 19th International Conference on e-Science, IEEE. pp. 1–4.

McMahan, B., Moore, E., Ramage, D., Hampson, S., y Arcas, B.A., 2017. Communication-efficient learning of deep networks from decentralized data, in: Artificial Intelligence and Statistics, PMLR. pp. 1273–1282.

Merzky, A., Turilli, M., Titov, M., Al-Saadi, A., Jha, S., 2022. Design and performance characterization of RADICAL-Pilot on leadership-class platforms. IEEE Transactions on Parallel & Distributed Systems 33, 818–829. doi:`10.1109/TPDS.2021.3105994`.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S., 2019. PyTorch: An imperative style, high-performance deep learning library. Advances in Neural Information Processing Systems 32.

Patros, P., Ooi, M., Huang, V., Mayo, M., Anderson, C., Burroughs, S., Baughman, M., Almurshed, O., Rana, O., Chard, R., et al., 2022. Rural AI: Serverless-powered federated learning for remote applications. IEEE Internet Computing 27, 28–34.

Pauloski, J.G., Hayot-Sasson, V., Ward, L., Brace, A., Bauer, A., Chard, K., Foster, I., 2024. Object Proxy Patterns for Accelerating Distributed Applications. URL: `https://arxiv.org/abs/2407.01764`, `arXiv:2407.01764`.

Pauloski, J.G., Hayot-Sasson, V., Ward, L., Hudson, N., Sabino, C., Baughman, M., Chard, K., Foster, I., 2023. Accelerating communications in federated

applications with transparent object proxies, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15.

PyTorch, 2024. Deep learning with PyTorch: A 60 minute blitz. URL: https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html.

Rana, O., Spyridopoulos, T., Hudson, N., Baughman, M., Chard, K., Foster, I., Khan, A., 2022. Hierarchical and decentralised federated learning, in: 2022 Cloud Continuum, IEEE. pp. 1–9.

Shah, S.H., Yaqoob, I., 2016. A survey: Internet of Things (IoT) technologies, applications and challenges. 2016 IEEE Smart Energy Grid Engineering (SEGE) , 381–385.

Sly-Delgado, B., Phung, T.S., Thomas, C., Simonetti, D., Hennessee, A., Tovar, B., Thain, D., 2023. TaskVine: Managing in-cluster storage for high-throughput data intensive workflows, in: Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, Association for Computing Machinery, New York, NY, USA. p. 1978–1988. URL: https://doi.org/10.1145/3624062.3624277, doi:10.1145/3624062.3624277.

Tang, Z., Chu, X., Ran, R.Y., Lee, S., Shi, S., Zhang, Y., Wang, Y., Liang, A.Q., Avestimehr, S., He, C., 2023. FedML Parrot: A scalable federated learning system via heterogeneity-aware scheduling on sequential and hierarchical training. arXiv preprint arXiv:2303.01778 .

Wang, H., Zhou, Y., Zhang, C., Peng, C., Huang, M., Liu, Y., Zhang, L., 2023. XFL: A high performace, lightweighted federated learning framework. arXiv preprint arXiv:2302.05076 .

Xiao, H., Rasul, K., Vollgraf, R., 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747 .

Xie, C., Koyejo, S., Gupta, I., 2019. Asynchronous federated optimization. arXiv preprint arXiv:1903.03934 .

Yu, X., Cherkasova, L., Vardhan, H., Zhao, Q., Ekaireb, E., Zhang, X., Mazumdar, A., Rosing, T., 2023. Async-HFL: Efficient and robust asynchronous federated learning in hierarchical IoT networks, in: 8th ACM/IEEE Conference on Internet of Things Design and Implementation, pp. 236–248.

Ziller, A., Trask, A., Lopardo, A., Szymkow, B., Wagner, B., Bluemke, E., Nounahon, J.M., Passerat-Palmbach, J., Prakash, K., Rose, N., Ryffel, T., Reza, Z.N., Kaissis, G., 2021. PySyft: A library for easy federated learning. Federated Learning Systems: Towards Next-Generation AI , 111–139.