FederatedScope: A Flexible Federated Learning Platform for Heterogeneity

Yuexiang Xie*, Zhen Wang*, Dawei Gao, Daoyuan Chen†, Liuyi Yao†, Weirui Kuang‡,
Yaliang Li‡, Bolin Ding‡, Jingren Zhou
Alibaba Group

ABSTRACT
Although remarkable progress has been made by existing federated learning (FL) platforms to provide infrastructures for development, these platforms may not well tackle the challenges brought by various types of heterogeneity, including the heterogeneity in participants’ local data, resources, behaviors and learning goals. To fill this gap, in this paper, we propose a novel FL platform, named FederatedScope, which employs an event-driven architecture to provide users with great flexibility to independently describe the behaviors of different participants. Such a design makes it easy for users to describe participants with various local training processes, learning goals and backends, and coordinate them into an FL course with synchronous or asynchronous training strategies. Towards an easy-to-use and flexible platform, FederatedScope enables rich types of plug-in operations and components for efficient further development, and we have implemented several important components to better help users with privacy protection, attack simulation and auto-tuning. We have released FederatedScope at https://github.com/alibaba/FederatedScope to promote academic research and industrial deployment of federated learning in a wide range of scenarios.

1 INTRODUCTION
As one of the feasible solutions to address the privacy leakage issue when utilizing isolated data from multiple sources, Federated Learning (FL) [44, 61, 97] has rapidly gained enormous popularity in both academia and industry [34, 49, 94, 96]. Such widespread adoption of FL is inextricably tied to the support of FL platforms, such as TFF [12], FATE [97], PySyft [105] and FedML [37], which provide users with functionalities to get started quickly and focus on developing new FL algorithms and applications.

Although the existing FL platforms have made remarkable progress, there are more burgeoning demands from FL research and deployment, which are mainly brought by the heterogeneity of FL. Specifically, we summarize the heterogeneity of FL as the following four aspects.

1. Heterogeneity in Local Data. The isolated data in FL, which are often owned by different parties or generated by different edge devices, vary a lot among the FL participants in terms of quality, quantity, underlying distributions, etc. Such heterogeneity in data can lead to the sub-optimal performance when applying the vanilla FedAvg [61], i.e., producing one global model for all the participants by the same local training process. Recent studies on

- Co-first authors.
- Equal contribution, listed in alphabetical order.
- Corresponding authors, email addresses: yaliang.li, bolin.ding@alibaba-inc.com

Personalized FL [28, 76] customize the local training processes of the participants according to their local data, including applying client-specific parameters, training schedules, submodules, and fusion approaches. Supporting these customizations brings challenges to the extensibility and flexibility of FL platforms.

2. Heterogeneity in Participants’ Resources. Apart from the heterogeneity of data, the participants’ resources can also be very different, including computation resources, storage resources, communication bandwidths, reliability, and so on. However, most of the existing FL platforms [12, 37, 97] adopt a synchronous training strategy for FL, which might lead to additional overhead caused by the heterogeneity in participants’ resources. For example, with the synchronous training strategy, the whole FL system could usually suffer from the slow clients that might be caused by network congestion, sluggish local training, or even device crash. Thus, it would be better if FL platforms allow users to implement/execute FL with asynchronous training strategies [19, 90] to ensure both efficiency and effectiveness in real-world FL applications.

3. Heterogeneity in Participants’ Behaviors. In the vanilla FedAvg [61], the participants only exchange homogeneous information (e.g., model parameters or gradients) and have the same behaviors (e.g., updating models based on the local data via SGD). However, the practical and recent FL applications often require to exchange various types of information among participants and execute diverse training processes, which leads to rich behaviors. For example, in the emerging federated graph learning applications [86, 89, 93, 100], participants exchange and handle multiple types of information, including model parameters, gradients, node embeddings, etc; Real-world FL applications might involve participants with different backends, which indicates that participants need backend-dependent computation graphs and accordingly perform diverse training processes. Besides, due to the heterogeneity in local data discussed above, participants could locally train with client-specific configurations that are suitable for their local data to achieve a better model utility. The heterogeneity in participants’ behaviors, caused by handling various exchanged information, executing diverse training processes and applying client-specific configurations in local training, prompts the FL platforms to support flexible expression for rich behaviors of participants.

4. Heterogeneity in Learning Goals. Towards a more general scope of utilizing isolated data, some recent FL studies [60, 74, 93] propose to allow participants to collaboratively learn common knowledge while optimizing for different learning goals. For example, several research institutes can federate a graph neural network to capture the generalizable structural patterns of molecules by using their own molecule data that correspond to different learning goals, such as predicting solubility, enzyme type, penetration,
Different Levels of Programming Interfaces for Users

Event Handler Message Communicator

Server Client

Deep Customization Simple Configuration

Plug-ins
Auto-Tuning
Privacy Protection
Attack Simulation

Workers

Algorithms
Personalization
Asynchronous Training
Hetero-Task Learning
… …

Server
Message... Levels of Programming etc. Another example, which can benefit from such heterogeneous federated learning, is the pre-trained language models in natural language processing [79], since participants can collaboratively train with different objectives based on their private corpora that cannot be directly shared in some scenarios such as medicine and finance. To handle the heterogeneity in learning goals, FL platforms should allow participants to locally train with different learning objectives and only share parts of the local models for collaboratively learning.

The aforementioned aspects of heterogeneity are commonly observed in real-world FL applications. Although we discuss them in the above four aspects, they can appear jointly in a single application. Facing such mixed heterogeneity, users are eager for an FL platform that is flexible: Participants should be allowed to express their diverse behaviors and different learning goals according to their own local data and system resources, and these participants can be effortlessly coordinated with synchronous or asynchronous training strategies for completing the federal training procedure based on a pre-defined consensus. Towards such a purpose, in this paper, we propose FederatedScope, a novel FL platform to handle the heterogeneity of FL.

To provide such flexibility, we propose FederatedScope, a novel FL platform that employs the event-driven architecture [45, 65] to frame FL courses into <event, handler> pairs. Note that it is not trivial to build up a comprehensive FL platform with such a formalization. In particular, considering the heterogeneity of federated learning, such formalization is expected to express diverse behaviors of servers and clients for handling the heterogeneity, and be well-modalized so that users can conveniently develop new FL algorithms and applications. To fulfill this goal, the events provided in FederatedScope can be categorized into two types, i.e., events related to message passing and events related to condition checking, which are used to describe what happens in the FL courses from the perspective of an individual participant. For example, in a vanilla FedAvg, a typical event of clients is “receiving_models”, which indicates that clients receive the global model from the server, and the corresponding handler can be “train the received global model based on the local data, and then return the model updates”. The handlers, triggered by events, describe what actions should be taken when a specific event happens. These events happen in the intended logical order and naturally trigger the corresponding handlers, which can precisely express various FL algorithms and procedures. All the participants can be coordinated with the pre-defined events related to message passing and condition checking to construct suitable FL course for specific scenarios and applications.

Besides the flexibility, as an FL platform, FederatedScope also provide great usability, that is, FederatedScope provides different levels of programming interfaces to meet different requirements from users, as demonstrated in Figure 1. For the users who want to design new FL algorithms, as discussed above, FederatedScope allows them to add new <event, handler> pairs to implement their ideas. For the users who want to directly apply existing FL techniques to certain application scenarios, FederatedScope provides rich sets of events and corresponding handlers, core functionalities and several important plug-in components, all of which can be directly called, and thus users only need to focus on a necessary set of interfaces to be integrated or implemented. For example, we have implemented several personalization federated algorithms, including applying client-wise configuration, maintaining client-wise sub-modules, global-local fusing, etc, for users’ convenient usage. Besides personalization, FederatedScope provides users with functionalities such as asynchronous training, privacy protection and cross-backend FL, and several important plug-ins such as attack simulation for protection verification, and auto-tuning for helping users to automatically seek suitable hyperparameters.

Last but not least, FederatedScope also has great extensibility, which is brought by the fact that the set of <event, handler> pairs can be easily extended by adding new ones. Take personalization again as an example: to add a new personalization, users only need to add new behaviors (e.g., adopting client-specific training course) in the corresponding handlers. Such extension convenience also holds for all the other functions such as federated aggregators, asynchronous training, privacy-protection, etc. Through this way, FederatedScope can be easily extended to include new functions or plug-ins to satisfy new requirements brought by new developments and support a variety of new scenarios.

Contributions. Our contributions can be summarized as: (1) Motivated by the heterogeneity challenges from a wide range of FL applications, we propose and release FederatedScope, a novel FL platform to handle heterogeneity in FL. The proposed FederatedScope promotes the development of FL techniques and the deployment of FL applications. (2) With the event-driven architecture, FederatedScope provides users with rich yet extendable sets of events and corresponding handlers, core functionalities such as asynchronous training, personalization and cross-backend FL, and several important plug-in components. These implementations make it easy for users to apply FL algorithms in both academia and industry applications. (3) FederatedScope brings great flexibility, usability and extensibility to users, broadens the application scope and enables more tasks that would otherwise be infeasible due to challenges brought by various types of heterogeneity in FL.

2 PRELIMINARY

2.1 Problem Definition

Federated Learning (FL) [44, 61, 97], a learning paradigm for collaboratively training models from dispersed data without directly sharing private information, involves multiple participants who are willing to contribute their local data and computation resources.
We use server to denote the participant(s) who are responsible for coordinating and aggregating, while other participants are clients. During a typical training round of an FL course, clients update the global model received from the server by training it with local data, and send the model updates back to the server for collaborative aggregation. In repeated training rounds, the (possibly sensitive) training data is always kept locally in each client; the server and clients only exchange aggregated and meta information, such as model parameters, gradients, public keys, hyperparameters, etc., which, to some degree, alleviates concerns about data privacy. To further satisfy different types of formal privacy protection requirements, various privacy protection techniques can be integrated into FL, such as Differential Privacy (DP) [80, 88], Homomorphic Encryption (HE) [29, 35], and Secure Multi-Party Computation (MPC) [13, 63]. In short, the goal of FL is to jointly train a global model in a privacy-preserving manner and achieve a better performance compared to that without collaboration.

Formally, there are $M$ clients, and the $m$-th client owns a private training dataset $D_m = \{(x_i^{(m)}, y_i^{(m)}) \in X \times Y, i = 1, 2, \ldots, |D_m|\}$, where $X$ and $Y$ are the input feature space and the label space, respectively. $D_m$ is stored in the $m$-th client's private space, and $n = \sum_{m=1}^{M} |D_m|$ is the total number of training instances. Without sharing $D_m$ directly with each other and the server, the $M$ clients together aim to train a model $h_\theta : X \rightarrow Y$ parameterized by $\theta$, with the loss $F : Y \times Y \rightarrow \mathbb{R}^+ \cup \{0\}$. The FL loss function is:

$$L = \frac{1}{n} \sum_{m=1}^{M} \sum_{(x_i^{(m)}, y_i^{(m)}) \in D_m} F(h_\theta(x_i^{(m)}), y_i^{(m)}).$$

**Extensions.** For the simplicity of presentation, we focus on a vanilla FL to minimize the loss function in Equation (1) in most parts of this paper. Our FederatedScope easily supports different federated settings in real-world FL applications, with more complicated loss functions, in order to handle the heterogeneity as discussed in Section 1. For example, for the purpose of personalization, the input feature spaces, the label spaces, and the underlying learning goals can be different for different clients. In FederatedScope, clients can adopt different models and loss functions in local training, and only federally train the shared parts of the models. We will discuss more details in Section 3.4.

### 2.2 Related Works

**Comparisons with existing FL platforms.** In the recent years, growing along with the development of federated learning, federated learning platforms, including TFF [12], FATE [97], LEAF [14], PySyft [105], FedML [37], FedScale [47], etc., are proposed to support various kinds of applications. These federated learning platforms provide data, models, and algorithms, which saves users’ effort on implementing from scratch and makes it easy for developers. Most of these existing platforms adopt a procedural programming paradigm, which requires users to explicitly declare a sequential training process and computational graph from the global perspective. However, such a design makes the existing FL platforms unable to provide the required flexibility and extendability for the burgeoning demands from FL research and deployment, which limits the promotion and broadened the application scenarios of federated learning. Meanwhile, users also expect FL platforms to become more convenient and simple to handle the aforementioned heterogeneity in real-world FL applications. To tackle these challenges, we propose FederatedScope to provide great flexibility and extendability for users to handle the heterogeneity in FL. FederatedScope provides rich implementations of FL algorithms for convenient usage, and provides different levels of programming interfaces for users to develop new algorithms.

**Comparisons with distributed machine learning.** In distributed machine learning, the server has the rights to control the behaviors of clients; While in federated learning, all the participants could have their own behaviors following the achieved consensus to collaboratively train the model. For example, given the consensus that participants only need to share parts of the model parameters, clients can apply client-wise training configurations (e.g., training steps, learning rate, regularizer, optimizer, and so on) to locally train the model, and keep the non-shared model parameters and the learning goals (might be different among clients) private. To satisfy such requirements, FederatedScope gives the rights to the participants to describe behaviors from their own respective perspectives. Besides, in federated learning, the quality, quantity, and distributions of clients’ local data can be very diverse. Such heterogeneity in data makes it challenging to collaboratively learn, which motivates FederatedScope to provide novel functionalities, such as asynchronous training strategies (in Section 3.3) and personalized federated learning algorithms (in Section 3.4), to make better use of their isolated data. Furthermore, there exist more privacy/security protection requirements in federated learning compared to distributed machine learning. To tackle this, FederatedScope provides some Byzantine fault tolerance algorithms to defend the malicious participants (in Section 3.6), and privacy protection component (in Section 4.1) and attack simulation component (in Section 4.2) to enhance and verify the privacy protection strength of FL applications.

### 3 DESIGN OF FEDERATEDSCOPES

In this section, we introduce the design of FederatedScope, showing how an FL training course can be framed and implemented using an event-driven architecture and why FederatedScope makes it easy to handle heterogeneity in federated learning. The overall structure is illustrated in Appendix D.

#### 3.1 Overview

An FL course consists of multiple rounds of training, and a typical round implemented with FederatedScope is illustrated in Figure 2, which includes four major steps: (1) **Broadcast Models:** the server broadcasts the up-to-date global model to the involved clients; (2) **Local Training:** once received the global model, clients perform local training using their trainer based on their private data; (3) **Return Updates:** after local training, clients return the model updates to the server; (4) **Federated Aggregation:** with the help of an aggregator, the server performs federated aggregation on the received model updates, and optimizes the global model. To facilitate efficient development and deployment of such an FL course with multiple computation/communication rounds and different roles, there are two important design principles of FederatedScope.
receiving (both synchronous and asynchronous) FL algorithms, as well as new ones we implement in FederatedScope and handlers are sufficiently expressive to describe all the existing aggregation’s in the server is triggered. It turns out that the pairs of events that describes the behavior of a participant is triggered. For example, when “local training” in a client is triggered; when “goal_achieved” occurs, “federated aggregation” in the server is triggered. It turns out that the pairs of events and handlers are sufficiently expressive to describe all the existing (both synchronous and asynchronous) FL algorithms, as well as new ones we implement in FederatedScope.

Based on these principles, we first give an overview of our design. **Basic infrastructure.** FederatedScope employs an event-driven architecture within which the behaviors of different clients and the server in an FL course can be programmed (relatively) independently. The information exchange among participants and conditions to be checked by participants during the FL course are described as events (trapezoids within pink areas in Figure 2); when an event occurs, the corresponding handlers (hexagons within pink areas) that describes the behavior of a participant is triggered. For example, when “receiving_models” occurs, “local training” in a client is triggered; when “goal_achieved” occurs, “federated aggregation” in the server is triggered. It turns out that the pairs of events and handlers are sufficiently expressive to describe all the existing (both synchronous and asynchronous) FL algorithms, as well as new ones we implement in FederatedScope.

With such an infrastructure, FederatedScope can easily support different machine learning backends (e.g., PyTorch [70] and TensorFlow [5]). All users need to do is to transform exchanged information (called messages), which might be related to participants’ local backends, into backend-independent ones before sharing, and parse the received messages according to the receiver’s backend for further usage. We call this procedure message translation.

**Programming interfaces.** Within the above infrastructure FederatedScope provides, for each client or server, we only need to implement a Trainer (green dashed rectangles in Figure 2) or Aggregator (blue dashed rectangles), respectively, which encapsulates the details of local training or federated aggregation with well-defined interfaces, e.g., the loss function, optimizer, training step, aggregation algorithms, etc. A Trainer can be implemented as if a machine learning model is trained on the local data owned by a client.

Besides Trainer and Aggregator, the design of FederatedScope allows flexible plug-in operators and components. For example, in order to ensure differential privacy, noise injection operators can be plugged to perturb the messages to be sent, where the amount of noise can be customized for different training tasks. More details of the programming interfaces can be found in Section 3.6.

**Why event-driven?** Benefited from such event-driven design, FederatedScope provides users with expressiveness and flexibility to handle various types of heterogeneity in FL. Users would not be required to implement the FL courses from a centralized perspective as in a procedural programming paradigm, which might be rather complicated for real-world FL tasks due to the heterogeneity. Instead, each individual participant (a server or a client) is instantiated with its own events and handlers to independently describe its behaviors, such as what actions to take when receiving a certain type of message from others, how to perform local training and what information should be returned (for a client), and when

---

**Figure 2: Overview of an FL round implemented with FederatedScope.**

- **Minimal dependency between different roles.** In FederatedScope, each client or the server takes care of only the minimal portion of job it needs to collaboratively accomplish, including the model to be collaboratively learned and the exchanged messages. While allowing both synchronous and asynchronous training, we want to avoid introducing too much duty of coordinating and scheduling to the server. This is important especially when we consider the heterogeneity of resources and learning goals for the clients.

- **Flexible and expressive programming interfaces for algorithm development and plug-in.** FederatedScope aims to enable efficient development of FL algorithms via proper abstraction of FL courses and providing a necessary set of interfaces that developers need to implement. Moreover, for the purpose of privacy protection and other functionalities, operators (e.g., for noise injection and encryption) and components (e.g., for auto-tuning) need to be plugged into the FL course in a flexible way.

---

(Continued...
to perform federated aggregation and start/terminate the training process (for a server).

In this way, the server performs federated aggregation under flexibly triggered conditions, which can prevent the training process from being blocked by unreliable or slow clients (more details in Section 3.3). Different clients may customize their training configurations according to their own data distributions, tasks, and resources, such as training with different trainers for personalization (Section 3.4.1), learning toward different goals (Section 3.4.2), and running on different backends (Section 3.5). FederatedScope also provides some native plug-in modules (Section 4) for various important functionalities, including privacy protection, attack simulation, and auto-tuning. Before diving into these parts, we first provide more details about the event-driven design of FederatedScope in Section 3.2.

### 3.2 Event-driven Architecture

Event-driven architectures are widely adopted in distributed systems [45, 65]. With such an architecture, an FL training course in FederatedScope can be framed into <event, handler> pairs: the participants wait for certain events (e.g., receiving model parameters broadcast from the server) to trigger corresponding handlers (e.g., training models based on the local data). Hence, developers can express the behaviors of a participant (a server or a client) independently from its own perspective, rather than sequentially from a global perspective (considering all the participants together), and the implementations can be better modularized.

The events in FederatedScope are categorized into two classes. One is related to message passing (e.g., exchanging information with others), which is also considered in previous FL platforms, e.g., receiving user-defined messages in FedML [37] and invoking requests in FedKeeper [15]. The other class of events checks the satisfaction of customizable conditions (e.g., whether a pre-defined percentage of feedback from clients has been received). Some examples of events provided in FederatedScope are presented in Appendix B. Next we will introduce these types of events in more detail.

- **Events Related to Message Passing.** The exchanged information among participants are abstracted as messages, and an FL training course consists of several rounds of message passing. Multiple types of messages are involved in an FL course, including but not limited to building up (e.g., join_in and id_assignment), training (e.g., model_param and gradients), and evaluating (e.g., metrics). For the participants, receiving a message can be regarded as an event, and their follow-up behaviors can be described in handling functions (i.e., the handlers) to handle the received messages. A handling function can be invoked by the event of receiving one or more types of messages, while receiving a certain type of message should only trigger one handling function directly.

  Take the vanilla FedAvg as an example, the clients’ handling function for the event “receiving_models” can be described as “train the received global model based on the local data, and then return the model updates”, and the servers’ handling function for the event “receiving_updates” can be described as “save the model updates, and check whether all the feedback has been received”.

  Generally, by defining the events related to message passing, FederatedScope provides users with expressiveness to flexibly describe heterogeneous message exchange, such as exchanging model parameters, gradients, public keys, embeddings, generators, and so on. Meanwhile, through customizing the operations in the corresponding handlers, users can conveniently describe rich behaviors of participants, including training models based on the local data with personalized configurations, performing federated aggregation, predicting, clustering, generating, etc. We will discuss more about this in Section 3.4.

- **Events Related to Condition Checking.** Apart from the events related to message passing, the events related to condition checking are also indispensable for FL implementations. These events and the corresponding handlers describe the participants’ behaviors when certain conditions are satisfied. For example, in an FL course, for the purpose of synchronization in training, the server checks whether the updated gradients or model parameters have been received from all the clients; if yes, it invokes an event “all_received”, and this event triggers the federated aggregation and pushes forward the training process.

  One important usage of the events related to condition checking is to express the customizable conditions for triggering the federated aggregation. Besides “all_received”, in order to support asynchronous training, FederatedScope also provides events “goal_achieved” and “time_up” for such purpose. Specifically, “goal_achieved” indicates that a certain percentage of feedback (so-called aggregation goal) has been received, and “time_up” denotes that the user-allocated time budget for each training round has run out. Different from the event “all_received” that forces the server to wait for feedback from all the clients, “goal_achieved” allows the training process to move forward once the server has received enough feedback, while “time_up” encourages the server to collect as much feedback as possible within the time budget, both of which enable different asynchronous training strategies in FL.

  Furthermore, the events related to condition checking also can be used to describe the behaviors of participants. For example, the server can be equipped with the events “all_joined_in” (i.e., all the clients have joined in the FL course) and “early_stop” (i.e., predefined early stop conditions are satisfied) to describe when to start and terminate the training process, respectively, while the clients can use the events “performance_drop” to trigger personalization when the received global model causes the performance drop, and use “low_bandwidth” to reduce the communication frequency when the available bandwidth is not enough.

  FederatedScope provides warnings if there exist conflicts, and adopts a default resolution following the “overwriting” principle. Specifically, in an FL course implemented with FederatedScope, each event is only permitted to be linked with one handler directly during the execution process. If an event is linked with more than one handler, which might cause conflicts in an FL course, a warning would be raised for users by FederatedScope, and the latest linked handler would overwrite the older ones (e.g., the default handler is overwritten by the user-customized handlers). Finally, the handlers that take effect in an FL course would be printed out and recorded in the experimental logs. Users can remove some handlers or adjust...
the asynchronous training strategies have been successfully applied in order to handle the heterogeneity of FL. The most important advantage is that the event-driven design of FederatedScope provides users with expressiveness and flexibility to implement and customize diverse FL algorithms. Next, with FederatedScope, we will demonstrate how to execute asynchronous federated training (Section 3.3), how to describe rich behaviors of the participants (Section 3.4) and how to conduct cross-backend FL (Section 3.5) in order to handle the heterogeneity of FL.

3.3 Supporting Asynchronous Training

The asynchronous training strategies have been successfully applied in distributed machine learning to improve training efficiency [19, 57, 101]. Considering the aforementioned heterogeneity of FL in Section 1, the asynchronous training strategy is important to balance the model performance and training efficiency, especially in cross-device scenarios that involve a large number of unreliable and diverse clients. With the provided events and handlers, which specify what actions to take (i.e., handlers) when certain customizable conditions are satisfied (i.e., events), FederatedScope supports users to conveniently design and implement suitable asynchronous training strategies for their FL applications.

3.3.1 Behaviors for asynchronous FL. Compared with the synchronous training strategy, several unique behaviors of participants might happen in asynchronous FL, which are modularized and provided in FederatedScope as follows:

(i) Tolerating staleness in federated aggregation. The term “staleness” denotes the version difference between the up-to-date global model maintained at the server and the model that a client starts from for local training, which should be tolerable to some extent in asynchronous FL. Specifically, in the federated aggregation, the staled updates from slow clients might be discounted in the aggregator but they still contribute to the aggregation. Of course, when the staleness is larger than a pre-defined threshold, the updates become outdated and thus can be directly dropped out.

(ii) Sampling clients with responsiveness-related strategies. The uniform strategy for sampling clients [61] might bring model bias in asynchronous FL, since the clients with low response speeds would contribute staled updates with higher probabilities compared with those who respond fast, which implies that the contributions of slow clients would be discounted or even dropped out in federated aggregation. Similar phenomena are happened in synchronous FL using over-selection mechanism [12], as pointed out by previous studies [41, 53, 67]. To tackle such an issue, with the prior knowledge of response speeds (it can be estimated from device information or historical responses), FederatedScope provides a responsiveness-related sampling strategy (i.e., the sampled probabilities are related to the response speeds) and a group sampling strategy (i.e., clients with similar response speeds are grouped).

(iii) Broadcasting models after receiving update. With the synchronous training strategy, the server broadcasts the up-to-date model to the sampled clients after performing federated aggregation. Such a broadcasting manner, denoted as “aggregating here”, can also be adopted in asynchronous FL [90]. We also provide another broadcasting manner to achieve asynchronous FL, named “aggregating after receiving” [67], in which the server sends out the current (up-to-date) model to a sampled idle client once the feedback is received. Compared with “aggregating after receiving”, the “aggregating after receiving” manner can keep the consistent concurrency and promotes an efficient FL systems [41].

To the best of our knowledge, with the provided events and the corresponding handlers that describe the above behaviors, most of the existing studies on asynchronous FL can be conveniently implemented with FederatedScope. For example, FedBuff [67] proposes to register the event “goal_achieved” and apply the “after receiving broadcasting” manner, while SAFA [90] suggests to equip “aggregating after receiving” broadcasting manner with event “goal_achieved” and manages clients based on their stalenesses. Particularly, a synchronous FL course with the over-selection mechanism can be easily implemented in FederatedScope by using event “goal_achieved” and setting the toleration to 0 (i.e., dropout all staled update).

In a nutshell, FederatedScope is well-modularized toward flexibility and extensibility for handling the heterogeneity of FL via applying asynchronous training strategies.

3.3.2 An example of asynchronous FL with FederatedScope.

Example 3.1. An example of asynchronous training strategy for FL is demonstrated in Figure 3. At the beginning of training process, the server samples a subset of the clients and broadcasts the model parameters (denoted as $\theta_i^{(r)}$ for the $r$-th round in the figure) to the sampled clients (e.g., Clients A/B/C in round 0 and A/C/D in round 1). These sampled clients perform local training based on their local data, and return the updated model parameters (e.g., $\theta_A^{(1)}$, $\theta_B^{(0)}$ and $\theta_C^{(0)}$) to the server once they finish training.

If the server could receive all the feedback from the sampled clients in time, as demonstrated in round 0, event “all_received” happens, and the corresponding handler is triggered and federated aggregation is performed to generate the new global model (e.g., $\theta_D^{(1)}$) for the next training round.

However, in some cases, such as round 1, some of the clients (e.g., Client D) fail to return the updated model in time due to some exceptions such as sluggish local training or device crash. With the asynchronous training strategy implemented in FederatedScope, when the allocated time budget has been run out, the “time_up” event would occur, and the server starts performing federated aggregation if the feedback has been received from a sufficient number of clients, otherwise the server executes “remedial measures”, such as restarting the training round, sampling reliable clients additionally, or adaptively adjusting the time budget. The staled feedback, such as $\theta_D^{(1)}$ in round 2, can be saved and contributed to the aggregation if such staleness is still tolerable according to the user-defined threshold in the corresponding events.

As shown in this example, applying asynchronous training can improve the learning efficiency and ensure the effectiveness of the learned model when facing the aforementioned heterogeneity of FL.
FL. More fancy strategies for asynchronous machine learning and FL [19, 57, 90, 101] have been integrated into FederatedScope, such as discounting the staled update, grouping clients according to their responsiveness, and so on.

3.3.3 Convergence analysis. We provide a theoretical analysis of convergence when applying the asynchronous training strategies in FL, under some widely-adopted assumptions [16, 67, 83] including smoothness and convexity of the loss function, the bounded gradient variances, and the bounded staleness.

**Proposition 1.** Consider the optimization problem defined in Equation (1). For each sampled client, it takes $Q$ local SGD steps with learning rate $\eta$ and returns model update to the server for federated aggregation. Assume that the loss function $F$ is $L$-smooth and $\mu$-strongly convex, when setting $0 < \mu Q \eta < 1$, the convergence of global model $\theta^{(t)}$ to the optimum $\theta^*$ after $T$ rounds satisfies:

$$
\mathbb{E}[F(\theta^{(T)}) - F(\theta^*)] \leq (1 - \mu Q \eta)^T \mathbb{E}[F(\theta^{(0)}) - F(\theta^*)]
+ \frac{3LQ\eta}{\mu} \left( \frac{\sigma_1^2 + \sigma_2^2}{2} + C \right) \left[ \eta Q L (\tau_{\max}^2 + 1) + \frac{1}{2} \right],
$$

(2)

where $\tau_{\max}$ denotes the maximum value of staleness caused by the asynchronous training strategies.

The above result shows the value of implementing asynchronous training strategies with FederatedScope. The detailed proof can be found in Appendix A. Compared to previous studies [16, 83], we extend the convergence analysis to more challenging asynchronous FL, where we quantify the effect of staled model updates for the federated aggregation and give a general analysis on the convergence rate rather than an ergodic version [67].

3.4 Supporting Personalization & Multi-Goal

In many real-world applications, handling the heterogeneity of FL brings the requirements of the flexibility of participants’ training behaviors. That is, clients need client-specific training processes and/or different formats of loss functions to meet their resource limitations, data properties and learning goals, all of which can be diverse as discussed in Section 1. Formally speaking, for the $m$-th client, the local training dataset $D_m$ might correspond to client-specific feature space $X_m$ and label space $Y_m$, which can lead to sub-optimal performance of the global model $h_0$ or even makes it unusable. To tackle this, the client could (1) maintain a local model $h_m$ with personalized parameters $\theta_m$ (i.e., personalization) and/or (2) minimize the local loss function $F_m$ (i.e., multiple learning goals), while only sharing parts of the models with others for federal training. Therefore, the loss function in Equation (1) can be extended as:

$$
\mathcal{L}' = \frac{1}{n} \sum_{m=1}^{M} \sum_{(x_i^{(m)}, y_i^{(m)}) \in D_m} F_m(h_{\theta_m}(x_i^{(m)}), y_i^{(m)}),
$$

(3)

Note that there exists some shared parameters among clients, i.e., $\bigcap_{m=1}^{M} \theta_m \neq \emptyset$, and all the clients collaboratively learn $\theta_1, \theta_2, \ldots, \theta_M$ to jointly minimize $\mathcal{L}'$.

Benefited from the event-driven architecture, FederatedScope provides users with flexible expressiveness to describe the behavior of an individual participant (a server or a client) from its own perspective, which is crucial for handling the heterogeneity of FL via allowing the differences among participants. In this section, we present how FederatedScope supports such differences among participants for handling the heterogeneity of FL through the following two ways.

3.4.1 Personalized training behaviors. As discussed by previous work [17, 28, 76], the heterogeneity of FL (e.g., the heterogeneity in local data and participants’ resources) might hurt the model performance for some clients and lead to the sub-optimal performance when sharing the same global model among all participants, such as vanilla FedAvg [61], which motivates the study of personalized federated algorithms [52, 55, 60, 75]. Specifically, personalized federated algorithms are proposed to apply client-specific local training courses based on their private data, including client-wise training configuration, sub-modules, local-local fusion weights, etc. The convergence of the federal learning process is not be determined solely by the learning process of the global model. Each participant can independently choose the most suitable snapshot of
the global model. Therefore, users are expected to describe diverse behaviors of clients to develop personalized federated algorithms, which might be rather complicated and inconvenient when using a procedural programming paradigm since lots of effort is put into sequentially coordinating and describing the participants.

With the event-driven architecture, FederatedScope allows users to describe the behaviors of participants independently, which provides great flexibility to develop new personalization algorithms. Users are able but not limited to (1) specify the training configurations, such as local training steps and learning rate, for an individual client; (2) define new events related to new types of exchanged messages and/or events related to customized conditions to apply personalization algorithms (e.g., performance drop); (3) add personalized behaviors into handlers that are triggered for local training, such as fusing the received global model with local models before performing local training. In most cases, such customization can be inherited from the general training behaviors and only need to focus on the differences. Considering that clients might have different privacy protection requirements, some privacy protection techniques can be adopted. For example, clients might choose to inject noise into the model parameters before sharing them. More details of the privacy protection of messages can be found in Section 4.1.

We provide several representative personalized federated algorithms [17] in FederatedScope for handling the heterogeneity in FL, including pFedMe [75], FedBN [55], FedEM [60], and Ditto [52]. These built-in algorithms serve as examples for showing how to easily and flexibly develop new personalized federated algorithms, and can also be conveniently adopted via configuring by users in real-world applications.

### 3.4.2 Multiple Learning Goals

Note that the scope of FL also covers the scenarios where participants learn common knowledge while optimizing different learning goals [60, 74, 93, 98]. The participants of an FL course reach a consensus on what needs to be shared while keeping other learning parts private, especially in cross-silo scenarios. For example, several medical research institutes would like to collaboratively learn a graph neural network for capturing the common structure knowledge of molecules, but they will not disclose what is the usage of the learned structure knowledge. They might exchange the update of the graph convolution layers while maintaining the encode layers, readout layers, and headers (such as classifier) private. In this and more similar scenarios from model pre-training, it can be difficult or even intractable for users to develop with a procedural programming paradigm via defining the static computation graph of the FL course.

Fortunately, the event-driven design of FederatedScope makes it easy to express and implement the FL courses with multiple learning goals. Each participant owns its local model and private data, defines its computation graph, locally trains with private learning objective, and only exchanges messages of the shared layers with others through FL.

Currently, FederatedScope provides three representative scenarios of FL with multiple learning goals, including graph classification, molecular property inference, and natural language understanding (NLU). In the graph classification scenario, clients own different graph classification tasks and aim to collaboratively improve their own performance due to the limitation of available training data. In the molecular property inference scenario, different clients have different property inference goals, such as the solubility (regression task), the enzyme type (classification task), and the penetration (classification task), which leads to heterogeneity in terms of task type. In the NLU scenario, clients are also heterogeneous in terms of task type, and they own different NLU tasks, including sentiment classification, reading compression, and sentence pair similarity prediction. Since the development of FL with multiple learning goals is still in the early stage, FederatedScope provides these scenarios to broaden the scope of FL applications and promote the development of innovative methods. More details of these scenarios of FL with multiple learning goals can be found in our open source repository [4].

In summary, FederatedScope allows users to describe participants’ behaviors from their respective perspectives and thus provides flexibility in applying different training processes and learning goals to the participants to handle the heterogeneity of FL.

### 3.5 Supporting Cross-backend FL

Motivated by the strong need from real-world applications, FederatedScope supports constructing cross-backend FL courses. For example, in an FL task, some of the involved clients are equipped with TensorFlow while others might run with PyTorch. Thanks to the event-driven architecture, FederatedScope can conveniently provide such functionality via a mechanism called message translation. Note that such support of cross-backend FL is different from those provided by the universal languages such as ONNX [9] and the existing FL platforms such as TFF [12].

Conceptually, ONNX and TFF adopt a global perspective of constructing an FL course, which implies that the complete computation graph is globally defined and shared among all participants. In order to make it compatible with different (versions of) machine learning backends on different clients, the global computation graph is serialized into platform-independent and language-independent representations, sent to the clients, and interpreted or compiled accordingly for different backends.

**Message translation.** FederatedScope, in contrast, gives each participant the right of describing the computation graph on its own (for the portion it takes charge of). Hence, participants can define the computation graph based on their running backends. Following a pre-defined consensus on the format of messages, the participants transform the messages, e.g., gradients and model parameters, generated from the local backends into the pre-defined backend-independent format, e.g., an array of pairs of parameters and values, before sharing them with others. This procedure is called **encoding**. For the other direction, once an encoded message is received, the participant parses the message, e.g., the above array, into backend-dependent tensors in its own computation graph and backend, which is called the **decoding** procedure.

The encoding and decoding procedures are abstracted as two special programming interfaces in FederatedScope with default implementations; they can also be customized for each participant based on its backend and the FL algorithm to be deployed. FederatedScope provides several examples of constructing cross-backend federated learning [3].
as illustrated in Figure 5, the trainer encapsulates updates received global models based on local data, and then returns the model. Users also can add some new events related to condition checking. For the synchronous training strategy, such event can be “all_received” and the corresponding handler will be “perform federated aggregation, and broadcast the updated global models”. For the asynchronous training strategies, the event “all_received” can be replaced with “goal_achieved” or “time_up”, which adds flexible behaviors during sampling clients or performing aggregation (More details can be found in Section 3.3). The federated aggregation is executed by an aggregator, which is also decoupled with the server for flexibly supporting various state-of-the-art (SOTA) aggregation algorithms, such as FedOpt [7], FedNova [84], FedProx [54], etc.

Note that when events such as “all_received” or “goal_achieved” happens, the clients would receive the up-to-date global models after the server performs federated aggregation, which naturally causes the following event “receiving_models” and triggers the corresponding handler for performing a new round of local training. In this way, although we have not explicitly declared a sequential training process, the events happen in the intended logical order to trigger the corresponding handlers, which can precisely express the FL procedure and promote modularization. Further, events such as “maximum_iterations_reached” or “early_stopped” can be adopted to specify when the FL courses should be terminated.

With such event-driven architecture, FederatedScope allows users to use existing or add new <event, handler> pairs for flexible customization, rather than inserting the new behaviors into the sequential FL course carefully as those in the procedural programming paradigm. For example, by simply changing the event “all_received” to other events related to condition checking such as “goal_achieved”, users can conveniently apply asynchronous training strategies. Users also can add some new events related to message passing to enable the heterogeneous information exchange, such as node embeddings in graph federated learning [93] and encrypted results in cross-silo federated learning [35].

```python
class CustomizedServer(Server):
    def custom_handler(args):
        # Describe the operations for handling the event
        ... ...
        # Register the custom handlers for custom events
        registered_handlers = dict()  # Expected type {event: handler}
        register(customized_event, custom_handler)

        registered_handlers[customized_event](args)

    if customized_event == True:
        # Call the corresponding custom_handler
        registered_handlers[customized_event](args)

class Client(object):
    def handler_for_receiving_models(args):
        # Perform local training when receiving the global models
        model_update = trainer.train(args.model, args.data)
        send(message=model_update, receiver=server)

    def train(received_models, data):
        # Describe training behaviors (same as centralized training)
        ... ...
        local_model = update_from_global_models(received_models)
        preds = local_model.forward(data.x)
        args = [optimizer, loss_function, regularizer, ...]
        model_updates = local_model.backward(data.y, preds, args)
        return model_updates

class CustomizedTrainer(Trainer):
    # Register the custom handlers for custom events
    ... ...
    def handler_for_receiving_models(args):
        # Perform local training when receiving the global models
        model_update = trainer.train(args.model, args.data)
        send(message=model_update, receiver=server)

    def train(received_models, data):
        # Describe training behaviors (same as centralized training)
        ... ...
        local_model = update_from_global_models(received_models)
        preds = local_model.forward(data.x)
        args = [optimizer, loss_function, regularizer, ...]
        model_updates = local_model.backward(data.y, preds, args)
        return model_updates
```

Figure 4: Behaviors description with events and handlers.

Figure 5: The training behaviors and clients are decoupled for supporting flexible customization.

3.6 Usage of FederatedScope

In this section, we give a full example of how to set up an FL course, so that users can gain a clear and vivid understanding of FederatedScope. At a high level, users should define a series of events and their corresponding handlers, which characterize the behaviors of participants. As shown in Figure 4, the handlers are expressed as callable functions and bound to the corresponding events with a register mechanism. When an event happens, the corresponding handler will be called to handle it. The example is as follows:

Example 3.2. Consider that a server and several clients would like to construct an FL course and they agree to exchange certain model parameters during the training process. For clients, the event related to message passing is "receiving_models", and the corresponding handler can be "train the received global models based on local data, and then return the model updates". The local training process is executed by a Trainer object held by the client. As illustrated in Figure 5, the trainer encapsulates the training details, entirely decoupled from the client’s behaviors. Hence, the training process can be described as those of the centralized learning case, and the trainer can be flexibly extended with fancy optimizers, regularizers, personalized algorithms, etc. Such a design makes it easy for user customizations.

For the server, the event related to message passing is "receiving_updates" and the corresponding handler can be "save the model updates, and check the aggregation condition", which requires another event related to condition checking. For the synchronous training strategy, such event can be "all_received" and the corresponding handler will be "perform federated aggregation, and broadcast the updated global models". For the asynchronous training strategies, the event "all_received" can be replaced with "goal_achieved" or "time_up", which adds flexible behaviors during sampling clients or performing aggregation (More details can be found in Section 3.3).
representative examples from end-user perspective can be found on our website, including constructing FL courses via simple configuring [2] and developing new functionalities [1].

**Trainer & Aggregator.** The details of the adopted algorithms in trainer and aggregator are decoupled with the behaviors of participants. Therefore, when users develop their own trainer/aggregator with FederatedScope, they only need to care about the details of training/aggregating algorithms. For example, users are expected to implement several basic interfaces of trainers, including train, evaluation, update model, etc., which is the same as those in centralized training and serves as "must-do" items. For the aggregator, which takes the received messages as inputs and returns the aggregated results, users only need to implement how to aggregate.

**Programming Interfaces and Completeness Checking.** FederatedScope provides base classes to aware users of the necessary interfaces for an FL course, such as BaseTrainer and in BaseWorker. These base classes can be used to check the completeness of the defined FL courses, since an "Not Implementation Error" would be raised to abort the execution if users fail to implement the necessary interfaces. With the base classes, FederatedScope provides rich implementation of existing FL algorithms. Therefore users can inherit the provided implementation and focus on the development of new functions and algorithms, which also ensures the completeness of FL courses. Besides, FederatedScope provides a completeness checking mechanism to generate a directed graph to verify the flow of message transmission in the constructed FL course (an example is illustrated in Appendix E).

**Robustness Against Malicious Participants.** To defend malicious participants and make the system more robust, some Byzantine fault tolerance algorithms are provided in FederatedScope. For example, we can apply the Krum [11] aggregation rule in federated aggregation. Note that these Byzantine fault tolerance algorithms can be regarded as the aggregation behaviors of server and implemented in the aggregator, which is decoupled with other behaviors to make it flexible and extendable for users to develop their own fault tolerance algorithms.

4 IMPORTANT PLUG-IN COMPONENTS

In this section, we present several important plug-in components in FederatedScope for convenient usage. These components provide functionalities including privacy protection, attack simulation, and auto-tuning, all of which are tightly coupled with the design of FederatedScope and serve as plug-ins.

4.1 Behavior Plug-In: Privacy Protection

Real-world FL applications might prefer different privacy protection algorithms due to their diversity in types of private information, protection strengths, computation and communication resources, etc., which motivates us to provide various privacy protection algorithms in FederatedScope.

With the design of FederatedScope, privacy protection algorithms can be implemented as behavior plug-ins, which indicates that the privacy protection algorithms bring new behaviors of participants. For example, before the participants share messages, the encryption algorithms might be applied on the messages, or the messages would be partitioned into several frames, or certain noise can be injected into the messages. These behaviors have been predefined in FederatedScope (so-called the behavior plug-in), and can be easily called to protect privacy via simple configuration.

Specifically, we implement a widely-used homomorphic encryption algorithm Paillier [69] and apply it in a cross-silo FL task [35]; and we develop a secret sharing mechanism for FedAvg. These provided examples demonstrate how to apply privacy protection algorithms with FederatedScope. Furthermore, to satisfy the heterogeneity in privacy protection strengths, we provide tunable modules for applying Differential Privacy (DP) in FL, which has been a popular technique for privacy protection and has achieved great success in database and FL applications [24, 25, 80, 88]. An example is illustrated in Figure 6, from which we can see that users can utilize the configuration to modify the client’s behavior: injecting certain noise into the messages before sharing. Users can combine different behaviors together to implement fancy DP algorithms such as NbAFL [88]. Note that to achieve a theoretical guarantee of privacy protection, users still need to specify some necessary settings according to their own data and tasks, including the noise distribution [26, 71] and privacy budget allocation [56, 59, 85].

4.2 Participant Plug-In: Attack Simulation

Attacks, growing along with the development of FL, are important for users to verify the availability and the privacy protection strength of their FL systems and algorithms. Typical attacks include privacy attack and performance attack: the former aims to steal the information related to clients’ private data, while the latter aims to intentionally guide the learned model to misclassify a specific subset of data for malicious purposes such as back-door. However, most of the existing FL platforms ignore such an important functional component.

Note that it is non-trivial to provide attack simulation in an FL platform, since the diversity of privacy and performance attacks brings challenges to the platform’s flexibility and extensibility. Benefited from the design of FederatedScope, the behaviors of malicious participants can be expressed independently, thus the attack simulation can be implemented as the participant plug-in in FederatedScope. To be more specific, as shown in Figure 7, users can conveniently choose some of the participants to become malicious clients via configuring, and attack algorithms can be added to their own trainers. These malicious clients are able to collect or inject certain messages among victims, and further recover or infer the target information accordingly. The simulated attacks provided in FederatedScope can be used to verify the privacy protection strength of their FL systems and algorithms. For example, when users develop

```python
class Client(object):
    def handler_for_receiving_models(args):
        if config.inject_noise_before_sharing == True:
            # Inject certain noise before sharing the message
            args = [noise_distribution, budget, ...]
            protected_messages = add_noise(messages, args)
            send(message=protected_messages, receiver=server)
        else:
            send(message=messages, receiver=server)
```

Figure 6: Behavior plug-in: injecting noise.
which incorporates various HPO methods. Conceptually, Bayesian
optimization (HPO) methods, both traditional methods (e.g., Bayesian
optimization [72] and multi-fidelity methods [6, 8, 27, 51]) and
Federated-HPO methods [22, 42] (denoting very recent ones that
take into account) can help users manage their FL applications well. Hyperparameter op-
teratively affects their performance. Without suitable configurations,
FL algorithms generally expose hyperparameters that can signifi-
cantly affect their performance. Without suitable configurations,
users cannot manage their FL applications well. Hyperparameter op-
timization (HPO) methods, both traditional methods (e.g., Bayesian
optimization [72] and multi-fidelity methods [6, 8, 27, 51]) and
Federated-HPO methods [22, 42] (denoting very recent ones that
take into account) can help users manage their FL applications well.


class Fed_Runner(object):
    ...
    def setup_client(config):
        if config_is_malicious == True:
            # Instantiate a malicious client with attack behavior
            client = MaliciousClient(attack_algorithms, args)
        else:
            # Instantiate a normal client
            client = Client(args)
            client.join_in_FL_course()

Figure 7: Participant plug-in: malicious client.

a new FL algorithm, they want to know the protection level of the
proposed algorithm from some perspectives, such as whether the
dataset properties or private training samples would be inferred
by attacks. They can use several state-of-the-art attack algorithms,
which have been provided in FederatedScope for convenient usage,
to check the privacy protection strength of their FL algorithms, and
enhance the privacy protection strength if necessary according to
the results of simulated attacks.

FederatedScope provides rich types of attack. For privacy attack,
FederatedScope provides the implementation of the following algo-
rithms: (i) Gradient inversion attack [66] for membership inference;
(ii) PLA [64] for property inference attack; (iii) DMU-GAN [38]
for class representative attack; (iv) DLG [104], iDLG [103], GRAD-
INV [30] for training data/label inference attack. In terms of perfor-
mance attack, FederatedScope currently focuses on the backdoor
attack, a representative type of performance attack, whose objective
is to mislead the model to classify some selected samples to the
attacker-specified class. The implementations of SOTA backdoor
attacks include: (i) Edge-case backdoor attacks [82], BadNets [32],
Blended [18], WaNet [68], NARCISUS [99], which perform back-
door attack by poisoning the dataset; (ii) Neurotoxin [102] and
DBA [91], which perform back-door attack by poisoning the model.

4.3 Manager Plug-In: Auto-tuning

FL algorithms generally expose hyperparameters that can signifi-
cantly affect their performance. Without suitable configurations,
users cannot manage their FL applications well. Hyperparameter op-
timization (HPO) methods, both traditional methods (e.g., Bayesian
optimization [72] and multi-fidelity methods [6, 8, 27, 51]) and
Federated-HPO methods [22, 42] (denoting very recent ones that
deliberately take the FL setting into account) can help users manage
FL applications by automatically seeking suitable hyperparameter
configurations.

Therefore, in FederatedScope, we provide an auto-tuning plug-in,
which incorporates various HPO methods. Conceptually, Bayesian
optimization, multi-fidelity, and Federated-HPO methods treat a
complete FL course, a few FL rounds, and client-wise local update
procedures as black-box functions to be evaluated, respectively. Fed-
eratedScope provides a unified interface to manage the underlying
FL procedure in various granularities so that different HPO methods
can interplay with their corresponding black-box functions. This
unification is nontrivial for the last case, where we leverage our
event-driven architecture to achieve the client-wise exploration of
Federated-HPO methods. When they are plugged in, the exchanged
messages are extended with HPO-related samples/models/feedback,
and the participants would handle them with extended behaviors
accordingly.

For Bayesian optimization methods, we showcase applying vari-
os open-sourced HPO packages to interact with FederatedScope. Each
time they propose a specific configuration, FederatedScope
executes an FL course accordingly and returns a specified metric
(e.g., validation loss) as the function’s output.

As for multi-fidelity methods, we have implemented Hyper-
fid, PBT [50] in FederatedScope. Specifically, Federated-
Scope can export the snapshot of a training course to a correspond-
ing checkpoint, from which another training course can restore.
With such a checkpoint mechanism, these multi-fidelity methods
can evaluate the configurations that have survived previous low-
fidelity comparisons by restoring from the last checkpoints rather
than learning from scratch.

Furthermore, FederatedScope provides FedEx [42] as an exem-
plary implementation of Federated-HPO methods. Specifically, once
FedEx is plugged in, we sample configurations for each client inde-
dependently in each FL round. Then each client re-specifies its
native configuration and conducts local updates accordingly, as
shown in Figure 8. Finally, the client-wise feedback is aggregated
on the policies responsible for determining the optimal config-
uration(s).

In summary, the auto-tuning plug-in can manage FL applica-
tions in various granularities. Traditional HPO methods interplay
with FederatedScope by configuring and running one or more com-
plete FL rounds, while Federated-HPO methods explore client-wise
configurations concurrently in a single FL round. With flexibility
provided by the event-driven architecture, we have implemented
these HPO methods in a unified way [87], and novel HPO methods
can be easily developed and contributed to FederatedScope.

5 EXPERIMENTS

5.1 DataZoo and ModelZoo

For convenient usage, we collect and preprocess ten widely-used
datasets from various FL application scenarios, including computer
vision datasets (FEMNIST [21], CelebA [58] and CIFAR-10 [46]), nat-
ural language processing datasets (Shakespeare [61], Twitter [31]
and Reddit [62]) from LEAF [14], and graph learning datasets

... ...
we embed the texts with a bag-of-words model [36] and collabora-

tors. With FL, a CNN with two convolutional layers is trained

Table 1: The comparison between applying synchronous and asynchronous training strategies in federated learning, in terms of the virtual time cost (hours) to achieve the targeted test accuracy.

| Dataset (Target Acc.) | Sync. | Async. |
|-----------------------|-------|--------|
|                       | Vanilla | OS | OS (FedScale) | Goal-Aggr-Unif | Goal-Rece-Unif | Time-Aggr-Unif | Goal-Aggr-Group |
| FEMNIST (85%)         | 61.46 | 27.34 | 2.25× | 28.78 | 2.14× | 11.29 | 5.44× | 11.36 | 5.41× | 11.70 | 5.25× | 10.42 | 5.90× |
| CIFAR-10 (70%)        | 66.99 | 26.42 | 2.54× | 28.98 | 2.31× | 7.73  | 8.67× | 7.98  | 8.39× | 8.87  | 7.55× | 7.54  | 8.88× |
| Twitter (69%)         | 9.41  | 3.84 | 2.65× | 4.14 | 2.27× | 0.78  | 12.06× | 0.64 | 14.70× | 0.50  | 18.82× | 0.65  | 14.48× |

(DBLP [78], Ciao [77] and MultiTask [93]) from FederatedScope-

gnn (FS-G) [86]. The statistics of these datasets can be found in

FederatedScope [47]. Meanwhile, we plot the learning curves in Figure 9. Due to the space limitation, we only show some

equalization mechanism [12]). As Sync-OS is originally proposed and implemented in FedScale [47], we also adapt it for our experiments and report its performance (denoted as Sync-OS (FedScale)) for correctness verification.

Asynchronous FL, which can be regarded as the implementation of the vanilla synchronous training strategy on all the benchmark datasets. Meanwhile, we plot the learning curves in Figure 9. Due to the space limitation, we only show some asynchronous training strategies on CIFAR-10 dataset and omit other similar results. From Figure 9, we can observe the existence of noticeable gaps between synchronous and asynchronous training strategies for a long time during the training process. These experimental results are consistent with previous studies [41, 92] and confirm that the asynchronous training strategies provided in FederatedScope can significantly improve the training efficiency while achieving competitive model performance.

Both our implementation Sync-OS and the original implementation in FedScale show that applying over-selection mechanism in synchronous FL can improve the efficiency to some degree. However, it might cause unfairness among participants and then lead to model bias, as demonstrated in Figure 10. From the figure we can observe that when applying over-selection mechanism Sync-

Baselines. We implement FedAvg with two synchronous training strategies including Sync-vanilla (i.e., the vanilla synchronous strategy) and Sync-OS (i.e., the synchronous strategy with over-selection mechanism [12]). As Sync-OS is originally proposed and implemented in FedScale [47], we also adapt it for our experiments and report its performance (denoted as Sync-OS (FedScale)) for correctness verification.

For asynchronous FL, we instantiate different asynchronous behaviors discussed in Section 3.3.1, and different strategies are named in the format of Async-AdoptedEvent-BroadcastManner-SampleStrategy. For example, Async-Goal-Rece-Unif denotes that this strategy adopts the event “goal_achieved”, the after receiving broadcasting manner and the uniform sampling strategies for asynchronous FL, which can be regarded as the implementation of FedBuff [67]; and Async-Time-Aggr-Group denotes we adopt the event “time_up”, the after aggregating broadcasting manner and a group sampling strategy (the client would be grouply sampled according to their responsiveness [16]).

Analysis. We adopt the virtual time cost (hours) to achieve the targeted test accuracy as the performance metric for comparing synchronous and asynchronous FL. The experimental results are shown in Table 1 (more experimental results can be found in Appendix G and I), from which we can observe that asynchronous training strategies achieve significant efficiency improvements (5.25×-18.82×) compared to the vanilla synchronous training strategy on all the benchmark datasets. Meanwhile, we plot the learning curves in Figure 9. Due to the space limitation, we only show some asynchronous training strategies on CIFAR-10 dataset and omit other similar results. From Figure 9, we can observe the existence of noticeable gaps between synchronous and asynchronous training strategies for a long time during the training process. These experimental results are consistent with previous studies [41, 92] and confirm that the asynchronous training strategies provided in FederatedScope can significantly improve the training efficiency while achieving competitive model performance.

Both our implementation Sync-OS and the original implementation in FedScale show that applying over-selection mechanism in synchronous FL can improve the efficiency to some degree. However, it might cause unfairness among participants and then lead to model bias, as demonstrated in Figure 10. From the figure we can observe that when applying over-selection mechanism Sync-

Virtual Timestamp. Following the best practice in prior FL works [47], we conduct the experiments by simulation while tracking the execution time with virtual timestamps. Specifically, the server begins to broadcast messages containing initial model parameters at timestamp 0. Then each client sends updates back with a timestamp as the received one plus the execution time of local computation and communication estimated by FedScale [47]. The server handles the received messages in the order of their timestamps and lets the next broadcast inherit the timestamp from the message that triggers it, assuming the time cost of the server is negligible. Along with an FL course, we record the performance of the global model with respect to such virtual timestamps.

5.2 Experiment Settings

Here we conduct a series of experiments with FederatedScope on three representative datasets as follows:

FEMNIST. FEMNIST consists of 805,263 handwritten digits in 62 classes, which are partitioned into 3,597 clients according to the writers. With FL, a CNN with two convolutional layers is trained for image classification task on this dataset.

CIFAR-10. As suggested by previous studies [40], we partition the dataset into 1,000 clients with a Dirichlet distribution, and federally train a CNN with two convolutional layers for image classification.

Twitter. We sample a subset from Twitter, which consists of 6,602 twitter users’ 16,077 texts. Each twitter user can be regarded as a client for constructing an FL course. Following previous study [14], we embed the texts with a bag-of-words model [36] and collaboratively train a logistic regression model on these sampled clients for sentiment analysis.

More implementation details can be found in Appendix F and H.

5.3 Results and Analysis

5.3.1 Asynchronous Federated Learning. We first conduct experiments to compare the performance of applying synchronous and asynchronous training strategies in FL.

Virtual Timestamp. Following the best practice in prior FL works [47], we conduct the experiments by simulation while tracking the execution time with virtual timestamps. Specifically, the server begins to broadcast messages containing initial model parameters at timestamp 0. Then each client sends updates back with a timestamp as the received one plus the execution time of local computation and communication estimated by FedScale [47]. The server handles the received messages in the order of their timestamps and lets the next broadcast inherit the timestamp from the message that triggers it, assuming the time cost of the server is negligible. Along with an FL course, we record the performance of the global model with respect to such virtual timestamps.

Baselines. We implement FedAvg with two synchronous training strategies including Sync-vanilla (i.e., the vanilla synchronous strategy) and Sync-OS (i.e., the synchronous strategy with over-selection mechanism [12]). As Sync-OS is originally proposed and implemented in FedScale [47], we also adapt it for our experiments and report its performance (denoted as Sync-OS (FedScale)) for correctness verification.

For asynchronous FL, we instantiate different asynchronous behaviors discussed in Section 3.3.1, and different strategies are named in the format of Async-AdoptedEvent-BroadcastManner-SampleStrategy. For example, Async-Goal-Rece-Unif denotes that this strategy adopts the event “goal_achieved”, the after receiving broadcasting manner and the uniform sampling strategies for asynchronous FL, which can be regarded as the implementation of FedBuff [67]; and Async-Time-Aggr-Group denotes we adopt the event “time_up”, the after aggregating broadcasting manner and a group sampling strategy (the client would be grouply sampled according to their responsiveness [16]).

Analysis. We adopt the virtual time cost (hours) to achieve the targeted test accuracy as the performance metric for comparing synchronous and asynchronous FL. The experimental results are shown in Table 1 (more experimental results can be found in Appendix G and I), from which we can observe that asynchronous training strategies achieve significant efficiency improvements (5.25×-18.82×) compared to the vanilla synchronous training strategy on all the benchmark datasets. Meanwhile, we plot the learning curves in Figure 9. Due to the space limitation, we only show some asynchronous training strategies on CIFAR-10 dataset and omit other similar results. From Figure 9, we can observe the existence of noticeable gaps between synchronous and asynchronous training strategies for a long time during the training process. These experimental results are consistent with previous studies [41, 92] and confirm that the asynchronous training strategies provided in FederatedScope can significantly improve the training efficiency while achieving competitive model performance.

Both our implementation Sync-OS and the original implementation in FedScale show that applying over-selection mechanism in synchronous FL can improve the efficiency to some degree. However, it might cause unfairness among participants and then lead to model bias, as demonstrated in Figure 10. From the figure we can observe that when applying over-selection mechanism Sync-

unfairness among participants, and then causes the learned models to bias towards those clients with fast response speeds. In contrast, the asynchronous learning strategies provided in FederatedScope can improve the efficiency without introducing such unfairness and model bias, due to the fact that staled feedback would be tolerated in the federated aggregation. Hence the distribution of effective aggregation count of asynchronous learning strategies plotted in Figure 10 is more concentrated and similar to that of the vanilla synchronous training strategy.

Further, in Figure 11, we illustrate the characteristics of different asynchronous training strategies in terms of staleness (i.e., the version difference between the up-to-date global model and the model used for local training) of the updates when performing federated aggregation. By comparing Async-Goal-Aggr-Unif and Async-Goal-Rce-Unif, we can see that after aggregating broadcasting manner causes less staleness than after receiving. It implies that after aggregating is more suitable for those FL tasks with a low staleness toleration threshold, but such a broadcasting manner requires more available bandwidths at the server since multiple messages are sent out at the same time.

5.3.2 Personalization. To demonstrate how personalization can handle the heterogeneity among participants, we compare FedAvg [61] with several built-in SOTA personalized FL algorithms, including FedBN [55], FedEM [60], pFedMe [75] and Ditto [52].

The experimental results are illustrated in Figure 12, which shows the client-wise test accuracies on FEMNIST dataset. We can observe that the average accuracy (denoted as the red dots) and the 90% quantile accuracy (denoted as the red horizontal lines) of vanilla FedAvg are both significantly lower than those of personalized FL algorithms. This indicates that applying personalized FL algorithms can improve the client-wise performance, also covering the bottom clients, and then lead to a better overall performance. Besides, in terms of the standard deviation among the client-wise accuracy (shown as σ at the top of the figure), personalized FL algorithms can reduce the performance differences to some degree, which confirms the advantages of enabling personalization behaviors for handling the heterogeneity among participants in real-world FL applications.

Personalized federated learning algorithms might need different computation and computation resources compared to vanilla FedAvg [61]. The computation and communication costs in each training round are determined by the adopted algorithms. Take the comparisons between vanilla FedAvg and two representative personalized federated learning algorithms FedBN [55] and Ditto [52] as examples, in each training round [17], (i) FedBN needs the same computation but fewer communication costs, since it proposes to not share the parameters of BatchNorm layer; and (ii) Ditto needs the same communication but more local computation costs, since it suggests to train the local models additionally. Further, from the perspective of an FL course, i.e., iteratively performing the FL training rounds until termination, the communication and computation costs depend on the convergence of learned models.

5.3.3 Privacy Protection and Attack. We conduct an experiment to show the effect of applying privacy protection algorithms provided in FederatedScope. We take DP as an example, and study its effect on the utility of learned model and the effectiveness in defending against privacy attack. Specifically, we train a ConvNet2 model on FEMNIST, and randomly choose some of the clients to inject Gaussian noise into the returned model updates to strengthen their privacy. We construct multiple FL courses with respect to varying the percentage of clients that injects noise, changing from 0% to 100%, and plot the performance of the learned models in Figure 13. From this figure we can observe that as more and more clients choose to inject the noise into the returned model updates, the test accuracy achieved by the learned global model decreases gradually, from 84% to 65%, which shows the trade-off between the privacy protection strength and model utility.

Moreover, we apply the DLG algorithm [104] implemented in FederatedScope to conduct privacy attack, aiming to reconstruct private training data of other users. As shown on the left-hand side of Figure 13, the reconstructed images from the clients who have not injected noises are clear and the privacy attacker successfully recovers clients’ training data to the extent that the groundtruth digits are exposed. On the right-hand side of the figure, we plot the reconstructed images from those clients injecting noises, which confirms the effectiveness of the privacy protection provided by DP since the attacker fails to recover meaningful information.

5.3.4 Auto-Tuning. As mentioned in Section 4.3, we have implemented several HPO methods in FederatedScope, which enables users to auto-tune hyperparameters of FL courses. Here we experimentally compare some representative HPO methods, including random search (RS) [10], successive halving algorithm (SHA) [51] and recently proposed Federated-HPO method FedEx [42], by applying them to optimize hyperparameters of FedAvg on FEMNIST dataset. We follow the protocol used in FedHPO-B [87], where RS
and SHA try configurations one by one, and FedEx wrapped by RS/SHA manages the search procedure in a fine-grained granularity to explore hyperparameter space concurrently.

We present the results in Figure 14, where the best-seen validation loss is depicted, and the test accuracy of the searched optimal configuration is reported in the legend. The best-seen validation losses of wrapped FedEx decrease slower than their corresponding wrappers, where such a poorer regret seems to indicate poorer search efficiency. However, their searched configurations’ test accuracies are remarkably better than their wrappers, implying the superiority of managing the search procedure in a fine-grained granularity.

6 CONCLUSIONS

In this paper, we introduce FederatedScope, a novel federated learning platform, to provide users with great supports for various FL development and deployment. Towards both convenient usage and flexible customization, FederatedScope exploits an event-driven architecture to frame an FL course into <events, handlers> pairs so that users can describe participants’ behaviors from their respective perspectives. Such an event-driven design makes FederatedScope suitable for handling various types of heterogeneity in FL, due to the advantages that (i) FederatedScope enables participants to exchange rich types of messages, express diverse training behaviors, and optimize different learning goals, and (ii) FederatedScope offers rich condition checking events to support various coordinations and corporations among participants, such as different asynchronuous training strategies. Further, the design of FederatedScope allows us to conveniently implement and provide several important plug-in components, such as privacy protection, attack simulation, and auto-tuning, which are indispensable for practical usage. We have released FederatedScope to help researchers and developers quickly get started, develop new FL algorithms, and build new FL applications, with the goal of promoting and accelerating the progress of FL.

REFERENCES

[1] The examples of constructing FL courses via developing new functionalities in FederatedScope. https://federatedscope.io/docs/own-case/

[2] The examples of constructing FL courses via simple configuring in FederatedScope. https://federatedscope.io/docs/examples/

[3] The examples of cross-backend federated learning in FederatedScope. https://github.com/alibaba/FederatedScope/tree/master/federatedscope/cross_backends

[4] The examples of three scenarios of FL with multiple learning goals in FederatedScope. https://github.com/alibaba/FederatedScope/tree/master/benchmark/BFTL

[5] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. arXiv preprint arXiv:1603.04467 (2015).

[6] Takuya Akiba, Shotaro Sano, Toshikiyo Yanase, Takeuru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’19). 2623–2631.

[7] Muhammad Asad, Ahmed Moustafa, and Takayuki Ito. 2020. FedOpt: Towards Communication Efficiency and Privacy Preservation in Federated Learning. Applied Sciences 10, 8 (2020).

[8] Noor Awad, Neeratyoy Mallik, and Frank Hutter. 2021. DEHB: Evolutionary Hyperband for Scalable, Robust and Efficient Hyperparameter Optimization. In Proc. of the International Joint Conference on Artificial Intelligence (IJCAI’21). 2147–2153.

[9] JunFjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[10] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. Journal of machine learning research 13, 2 (2012), 281–305.

[11] Peva Blancard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. 2017. Machine learning with adversaries: Byzantine tolerant gradient descent. Advances in Neural Information Processing Systems 30 (2017).

[12] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dmitriy Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konecný, Stefano Mazzocchi, Brendan McMahlan, et al. 2019. Towards federated learning at scale: System design. Proceedings of Machine Learning and Systems 1, 0 (2019), 374–588.

[13] 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 Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS’17). 1175–1191.

[14] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konecný, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. 2018. Leaf: A benchmark for federated settings. arXiv preprint arXiv:1812.01097 (2018).

[15] Mohak Chadha, Anshul Jindal, and Michael Gerndt. 2020. Towards federated learning using faas fabric. In Proc. of the the 2020 Sixth International Workshop on Serverless Computing. 49–54.

[16] Zheng Chai, Yujing Chen, Ali Arwaz, Liang Zhao, Yue Cheng, and Huzefa Rangwala. 2021. FedAT: a high-performance and communication-efficient federated learning system with asynchronous tiers. In Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (ResilientFL’21). 1–16.

[17] Daoyuan Chen, Dawei Gao, Weirui Kuang, Yaliang Li, and Bolin Ding. 2022. pFL-Bench: A Comprehensive Benchmark for Personalized Federated Learning. In Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track.

[18] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. 2017. Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning. arXiv preprint arXiv:1712.05526 (2017).

[19] Yujing Chen, Yue Ning, Martin Slawski, and Huzefa Rangwala. 2020. Asynchronous online federated learning for edge devices with non-iid data. In Proc. of the IEEE International Conference on Big Data (BigData’20). 15–24.

Figure 12: Client-wise test accuracy on FEMNIST dataset.

Figure 13: Accuracy w.r.t. varying protection strength and recovered images.

Figure 14: Best-seen validation loss over time on FEMNIST dataset.
A CONVERGENCE ANALYSIS

Without loss of generality, we assume the numbers of training instances are same among clients and simplify Eq. (1) as the following optimization problem:

$$
\min_{\theta \in \mathbb{R}^d} F(\theta) := \frac{1}{M} \sum_{i=1}^{M} F_i(\theta),
$$

where $M$ is the number of participated clients, $F_i$ denotes the loss function of client $i$. We use $S^{(t)} \subseteq [M]$ to denote the index set of the clients that participate in the $t$-th training round. For each client $i \in S^{(t)}$, it takes $Q$ local SGD steps with learning rate $\eta$. The model update at the $t$-th round is:

$$
\theta^{(t+1)} - \theta^{(t)} = \Delta^{(t)}
= \frac{1}{|S^{(t)}|} \sum_{i \in S^{(t)}} \Delta_i^{(t)}
= \frac{1}{|S^{(t)}|} \sum_{i \in S^{(t)}} \sum_{q=1}^{Q} \eta g_i(\theta_i^{(t-\tau_i)}),
$$

where $g_i(\cdot)$ denotes stochastic gradient of client $i$. We use $\tau_i(t)$ to denote the version staleness of initialization model that is used for local training in client $i$ at $t$-th training round. And we simplify it as $\tau_i$ when it would not cause any ambiguity.

Firstly, we give some widely-adopted assumptions \cite{16,67,83}:

**Assumption A.1 (Smoothness).** The loss function $F$ has Lipschitz continuous gradients with a constant $L > 0$, i.e., $\forall \theta_1, \theta_2$:

$$
\left\| \nabla F(\theta_1) - \nabla F(\theta_2) \right\|^2 \leq L^2 \left\| \theta_1 - \theta_2 \right\|^2,
$$

and further:

$$
F(\theta_1) - F(\theta_2) \leq \langle \nabla F(\theta_2), \theta_1 - \theta_2 \rangle + \frac{L}{2} \left\| \theta_1 - \theta_2 \right\|^2.
$$

**Assumption A.2 (Convexity).** The loss function $F$ is $\mu$-strongly convex with a constant $\mu > 0$, i.e., $\forall \theta_1, \theta_2$:

$$
F(\theta_1) - F(\theta_2) \geq \langle \nabla F(\theta_2), \theta_1 - \theta_2 \rangle + \frac{\mu}{2} \left\| \theta_1 - \theta_2 \right\|^2.
$$

**Assumption A.3 (Unbiasedness).** The estimation of stochastic gradient is unbiased: $\mathbb{E}[g_i(\theta)] = \nabla F_i(\theta)$.

**Assumption A.4 (Bounded variances).** The local and global variances are bounded for all clients, i.e., $\forall i$, there exist constants $\sigma_i$ and $\sigma_g$, s.t.,

$$
\mathbb{E} \left[ \left\| g_i(\theta) - \nabla F_i(\theta) \right\|^2 \right] \leq \sigma_i^2,
$$

and further:

$$
\frac{1}{M} \sum_{i=1}^{M} \left\| \nabla F_i(\theta) - \nabla F(\theta) \right\|^2 \leq \sigma_g^2.
$$

**Assumption A.5 (Bounded gradients).** The gradients of clients are bounded, i.e., $\left\| \nabla F_i \right\|^2 \leq C, \forall i \in [M]$. Specifically, $\left\| \nabla F \right\|^2 = \frac{1}{M} \sum_{i=1}^{M} \left\| \nabla F_i \right\|^2 \leq \frac{1}{M} \sum_{i=1}^{M} \left\| \nabla F_i \right\|^2 \leq C$.

**Assumption A.6 (Bounded staleness).** $\forall i \in [M]$, the staleness $\tau_i$ is not larger than $\tau_{\text{max}}$.

Based on these assumptions and inspired by previous studies \cite{16,67}, we have two lemmas:

**Lemma 1.** The expectation of the $L_2$ norm of the estimated gradients for all clients are bounded:

$$
\mathbb{E} \left[ \left\| g_i(\theta^{(t)}) \right\|^2 \right] \leq 3 \left( \sigma_i^2 + \sigma_g^2 + C \right).
$$

**Proof.**

$$
\mathbb{E} \left[ \left\| g_i(\theta^{(t)}) \right\|^2 \right]
= \mathbb{E} \left[ \left\| g_i(\theta^{(t)}) - \nabla F_i(\theta^{(t)}) + \nabla F_i(\theta^{(t)}) - \nabla F(\theta^{(t)}) + \nabla F(\theta^{(t)}) \right\|^2 \right]
\leq 3 \mathbb{E} \left[ \left\| g_i(\theta^{(t)}) - \nabla F_i(\theta^{(t)}) \right\|^2 + \left\| \nabla F_i(\theta^{(t)}) - \nabla F(\theta^{(t)}) \right\|^2 + \left\| \nabla F(\theta^{(t)}) \right\|^2 \right]
= 3 \left( \sigma_i^2 + \sigma_g^2 + C \right),
$$

where the last equality follows Assumption A.4 and A.5.

**Lemma 2.** The expectation of $L_2$ norm of $\Delta^{(t)}, \forall t \in [T]$, is bounded:

$$
\mathbb{E} \left[ \left\| \Delta^{(t)} \right\|^2 \right] \leq 3 Q^2 \eta^2 \left( \sigma_i^2 + \sigma_g^2 + C \right),
$$

where $Q$ denotes the local SGD steps and $\eta$ denotes the learning rate.

**Proof.** According to Eq. (5), we have:

$$
\mathbb{E} \left[ \left\| \Delta^{(t)} \right\|^2 \right] \leq \frac{1}{|S^{(t)}|^2} \sum_{i \in S^{(t)}} \sum_{q=1}^{Q} \mathbb{E} \left[ \left\| g_i(\theta_i^{(t-\tau_i)}) \right\|^2 \right]
\leq \frac{Q \eta}{|S^{(t)}|^2} \sum_{i \in S^{(t)}} \sum_{q=1}^{Q} \left\| g_i(\theta_i^{(t-\tau_i)}) \right\|^2.
$$

Take the expectation over the randomness w.r.t. the client participation and the stochastic gradients:

$$
\mathbb{E} \left[ \left\| \Delta^{(t)} \right\|^2 \right] \leq \frac{Q \eta^2}{|S^{(t)}|^2} \sum_{i \in S^{(t)}} \sum_{q=1}^{Q} \mathbb{E} \left[ \left\| g_i(\theta_i^{(t-\tau_i)}) \right\|^2 \right]
\leq 3 Q^2 \eta^2 \left( \sigma_i^2 + \sigma_g^2 + C \right),
$$

where the last inequality follows from Lemma 1.

Next we provide the convergence analysis for the proposed asynchronous training protocol in federated learning, inspired by previous studies \cite{16,67,83}. With Eq. (5) and the assumption of smoothness (Assumption A.1), we have:

$$
\mathbb{E} \left[ F(\theta^{(t+1)}) - F(\theta^{(t)}) \right]
\leq \langle \nabla F(\theta^{(t)}), \theta^{(t+1)} - \theta^{(t)} \rangle + \frac{L}{2} \left\| \theta^{(t+1)} - \theta^{(t)} \right\|^2
= \langle \nabla F(\theta^{(t)}), \Delta^{(t)} \rangle + \frac{L}{2} \left\| \Delta^{(t)} \right\|^2.
$$

Take the total expectation:

$$
\mathbb{E} \left[ F(\theta^{(t+1)}) - F(\theta^{(0)}) \right]
\leq \mathbb{E} \left[ \langle \nabla F(\theta^{(t)}), \Delta^{(t)} \rangle \right] + \frac{L}{2} \mathbb{E} \left[ \left\| \Delta^{(t)} \right\|^2 \right]
\leq \mathbb{E} \left[ \langle \nabla F(\theta^{(t)}), \Delta^{(t)} \rangle \right] + \frac{3LQ^2\eta^2}{2} \left( \sigma_i^2 + \sigma_g^2 + C \right) \quad \text{(With Lemma 2)}.
$$

\[17\]
Consider the bound of $H_1$:

$$H_1 = \langle \nabla F(\theta(t)), -\frac{1}{|S(t)|} \sum_{i \in S(t)} \sum_{q=1}^{Q} \eta g(\theta_i^{t \to t_q}) \rangle$$

$$= -\frac{1}{|S(t)|} \sum_{i \in S(t)} \sum_{q=1}^{Q} \eta \langle \nabla F(\theta(t)), g(\theta_i^{t \to t_q}) \rangle.$$  \hspace{1cm} (18)

We take the total expectation $\mathbb{E}[\cdot] := \mathbb{E}_F \mathbb{E}_{i \in [M]} \mathbb{E}_{g_i, \theta_i, T} [\cdot]$ ($F$ denotes the historical information):

$$\mathbb{E}[H_1] = -\mathbb{E}_F \left[ \frac{1}{M} \sum_{i=1}^{M} \sum_{q=1}^{Q} \eta \nabla F(\theta_i^{t \to t_q}), \frac{1}{M} \sum_{i=1}^{M} \nabla F_i(\theta_i^{t \to t_q}) \right].$$  \hspace{1cm} (19)

Given $(a, b) = \frac{1}{2}(|a|^2 + |b|^2 - |a - b|^2)$, we have:

$$\mathbb{E}[H_1] = \frac{1}{2} \mathbb{E}_F \left[ -||\nabla F(\theta(t))||^2 - ||\frac{1}{M} \sum_{i=1}^{M} \nabla F_i(\theta_i^{t \to t_q})||^2 ight. 
+ \left. ||\nabla F(\theta(t)) - \frac{1}{M} \sum_{i=1}^{M} \nabla F_i(\theta_i^{t \to t_q})||^2 \right]$$

$$\leq -\mu \eta \mathbb{E}_F \left[ F(\theta(t)) - F(\theta^{(s)}) \right]$$

$$+ \frac{1}{M} \sum_{i=1}^{M} \mathbb{E}_F \left[ ||\nabla F_i(\theta_i^{t \to t_q})||^2 \right]  \hspace{1cm} (20)$$

where the last inequality follows from Lemma 1 and Assumption A.6. By inserting Eq. (22) and Eq. (20) into Eq. (17), we have:

$$\mathbb{E}[F(\theta(t+1)) - F(\theta(t))]$$

$$\leq -\mu \eta \mathbb{E}_F \left[ F(\theta(t)) - F(\theta^{(s)}) \right]$$

$$+ 3LQ^2 \eta^2 \left( \sigma^2 + \sigma_g^2 + C \right) \mathbb{E}[Q_L(t_{\max}^2) + 1]  \hspace{1cm} (23)$$

By rearranging, we have:

$$\mathbb{E}[F(\theta(T+1)) - F(\theta(T)))]$$

$$\geq \frac{3LQ^2 \eta^2}{\mu} \left( \sigma^2 + \sigma_g^2 + C \right) \mathbb{E}[Q_L(t_{\max}^2) + 1]  \hspace{1cm} (24)$$

which implies a geometric series with ratio $1 - \mu \eta$. Thus we can obtain:

$$\mathbb{E}[F(\theta(T)) - F(\theta(0))]$$

$$\leq (1 - \mu \eta)^T \mathbb{E}[F(\theta(0)) - F(\theta^{(s)})]$$

$$+ \frac{3LQ \eta}{\mu} \left( \sigma^2 + \sigma_g^2 + C \right) \mathbb{E}[Q_L(t_{\max}^2) + 1]  \hspace{1cm} (25)$$

When $\mu \eta < 1$, we have the following model convergence conclusion:

$$\mathbb{E}[F(\theta(T)) - F(\theta^{(s)})]$$

$$\leq (1 - \mu \eta)^T \mathbb{E}[F(\theta(0)) - F(\theta^{(s)})]$$

$$+ \frac{3LQ \eta}{\mu} \left( \sigma^2 + \sigma_g^2 + C \right) \mathbb{E}[Q_L(t_{\max}^2) + 1].  \hspace{1cm} (26)$$
Table 2: Examples of events in FederatedScope.

| Category                  | Event                | Event Description                                                                 |
|---------------------------|----------------------|-----------------------------------------------------------------------------------|
| Related to Message Passing| receiving_join_in    | The server receives join-in requirements from clients.                            |
|                           | receiving_model      | Clients receive the global model from the server.                                 |
|                           | receiving_updates    | The server receives model updates from clients.                                    |
|                           | receiving_eval_request| Clients receive the request of evaluation from the server.                         |
|                           | all_received         | All the model updates have been received.                                          |
| Related to Condition Checking| time_up             | The allocated time budget for the training round has run out.                    |
|                           | early_stop           | The pre-defined early stop conditions are satisfied.                               |
|                           | performance_drop     | The received global model causes a performance drop.                               |

Table 3: The statistics of the datasets provided in DataZoo.

| Dataset                | Task                  | Number of Instance | Number of Clients |
|------------------------|-----------------------|--------------------|-------------------|
| FEMNIST                | Image Classification  | 817,851            | 3,597             |
| CelebA                 | Image Classification  | 200,288            | 9,323             |
| CIFAR-10               | Image Classification  | 60,000             | 1,000             |
| Shakespeare            | Next Character Prediction | 4,226,158                | 1,129             |
| Twitter                | Sentiment Analysis    | 1,600,498          | 660,120           |
| Reddit                 | Language Modeling     | 56,587,343         | 1,660,820         |
| DBLP (partitioned by venue) | Node Classification | 52,202             | 20                |
| DBLP (partitioned by publisher) | Node Classification | 52,202             | 8                 |
| Ciao                   | Link Classification   | 565,300            | 28                |
| MultiTask              | Graph Classification  | 18,661             | 7                 |

B  EXAMPLES OF EVENTS
Some examples of events provided in FederatedScope are presented in Table 2. These events and the corresponding handlers are used to describe participants’ behaviors in FederatedScope, which is introduced in Section 3.2.

C  DATAZOO
The statistics of the datasets provided in DataZoo are summarized in Table 3. Our DataZoo contains ten widely-used datasets collected from various FL applications and standardizes the data preprocessing.

D  OVERALL STRUCTURE
The overall structure is illustrated in Figure 15.

E  COMPLETENESS CHECKING
As shown in Figure 16, FederatedScope provides a completeness checking to verify the flow of message transmission in the constructed FL courses. A complete FL course should contain at least one path from the "start" node to the "termination" node.

F  IMPLEMENTATION DETAILS
We conduct the experiments show in Section 5 on 8 GeForce RTX 2080Ti GPUs.

For the experiments on FEMNIST and CIFAR-10, we train a CNN with two convolutional layers. We set the hidden size to 2048 and the dropout ratio to 0.5 to prevent overfitting. At each round of training, the clients execute 4 SGD steps with a batch size of 20, and the learning rate is set to 0.5. The concurrency number (i.e., the number of clients that perform local training at each training round) is 100. For the experiments on Twitter, we set the embedding size to 300 and the maximum length to 140 to embed the texts with a bag-of-words model, and train a logistic regression model. At each
round of training, the clients execute 4 SGD steps with a batch size of 2, and the learning rate is set to 0.05. The concurrency number is set to 200.

For all the implemented synchronous and asynchronous strategies, we tune the hyperparameter on the validation set. For SyncOS, at each training round, we over-select 30% more clients upon the corresponding concurrency number, i.e., 130 on FEMNIST and CIFAR-10, and 260 on Twitter. For asynchronous settings, we set the aggregation goal to 40, 20, and 40 for the experiments conducted on FEMNIST, CIFAR-10, and Twitter, respectively. And the threshold of staleness toleration is set to 20 for all the datasets. For the asynchronous strategies equipped with "time_up", the time budget of each training round is set to the same value as the averaged time cost for achieving the defined aggregation goal when using "goal_achieved".

G DATASETS WITH IID DISTRIBUTION VERSUS NON-IID DISTRIBUTION

We split CIFAR-10 into 100 clients, following the uniform distribution and the Dirichlet distribution (the Dirichlet factors $\alpha$ are set to 1.0, 0.5, and 0.2, a smaller factor value implies a higher heterogeneous degree) to synthesize distributed datasets with IID distribution and non-IID distribution, respectively. Then we federally train a CNN with two convolutional layers, adopting vanilla FedAvg [61] and two representative personalized federated learning algorithms, i.e., FedBN [55] and Ditto [52]. The experimental results are demonstrated in Table 4.

From the experimental results we can observe that, although vanilla FedAvg can achieve competitive performance on CIFAR-10 with IID distribution, it cannot perform well on non-IID distribution datasets and has larger performance drops when the heterogeneity degree increases. The heterogeneity in clients’ data, which is widespread in FL applications, could lead to sub-optimal performance when participants learn a simple global model as what they do in distributed machine learning. Therefore, to handle such heterogeneity, clients are encouraged to apply client-specific training based on their private data, and share parts of the global model while locally maintaining others. As a result, FedBN and Ditto obtain noticeable improvement on datasets with non-IID distributions compared to FedAvg, which indicates their effectiveness in handling such heterogeneity of FL.

H EXAMPLE OF AMOUNT OF PROGRAMMING

For the users who construct FL courses based on the built-in functionalities, they can adopt provided data, models, and plug-in operations via simply configuring, or using their own data or models without changing the implementation of the federated behaviors in the provided FL courses. For example, excluding the codes for dataset preprocessing, users only need to modify two lines of code for adopting a customized dataset: one line for registering the dataset class and one line for changing the dataset name in the configuration.

For the users who aim to do further customization, they can inherit from clients or servers (e.g., from the vanilla FedAvg) and add new <event, handler> pairs to express the new behaviors of servers and clients accordingly. For example, to implement the
personalized federated algorithms FedBN and Ditto based on the vanilla FedAvg, users only need to add/modify 19 and around 100 lines of code respectively.

I EXPERIMENTS AND ANALYSIS ON ASYNCHRONOUS TRAINING STRATEGIES

We add more experimental results of the asynchronous training strategies provided in FederatedScope, as shown in Figure 17. From the figures we can observe that, asynchronous training strategies can achieve better performance compared to synchronous training strategies.

Note that different asynchronous training strategies might have different characteristics. For example, after aggregating broadcasting manner causes less staleness than after receiving (as shown in Figure 11), but after aggregating requires more available bandwidths at the server since multiple messages are sent out at the same time. These different characteristics can guide users to choose the more suitable asynchronous training strategies for their own application, considering model performance, system resource, staleness toleration, etc.

Generally speaking, there is no conclusion on which provided sampling strategy is better than another one, and the effectiveness of the sampling strategies is case dependent. Instead of providing the “best” asynchronous training strategies (in fact it is impossible because of “No Free Lunch”), FederatedScope aim to provide a good abstraction and modularization of asynchronous federated learning, which can both cover most of the existing studies and provide flexibility for customization. Users can conveniently choose the suitable asynchronous training strategies accordingly.

As shown in Figure 17, from the experiments conducted on CIFAR-10, we find that the performance of applying response-related and group sampling strategies are similar to that of applying uniform strategy for this specific case. It is not a surprising result, since both response-related and group sampling strategies are proposed to alleviate the model bias caused by the heterogeneity in participants’ system resources, e.g., clients with weak responsiveness might contribute little to federated aggregation as their (staled) updates might be discounted or even dropped out. However, the distributions of client data are independent of their distributions of system resources, which causes that when clustering the clients according to their system resources, the data distributions among different clusters are similar, as shown in Figure 18. Such similarities among clusters limit the effectiveness of response-related and group sampling strategies compared to the uniform sampling strategies, since there might not exist model bias caused by the heterogeneity in participants’ system resources.

Furthermore, we re-split CIFAR-10 dataset and make the distributions of clients’ data related to their system resources: We randomly select some labels as “rare” labels, and instances with these “rare” labels are only owned by clients with weak responsiveness. The distributions among clients on this dataset (called bias-CIFAR-10) are shown in Figure 19, and the experimental results are shown in Figure 20. We can observe that applying response-related and group sampling strategies can achieve noticeable improvement compared to uniform sampling strategies, which empirically confirms our analysis above.