NVIDIA FLARE Introduction & Roadmap

Chester Chen

Senior Product & Engineering Manager

NVIDIA Federated Learning

NVIDIA FLARE DAY

September 18, 2024

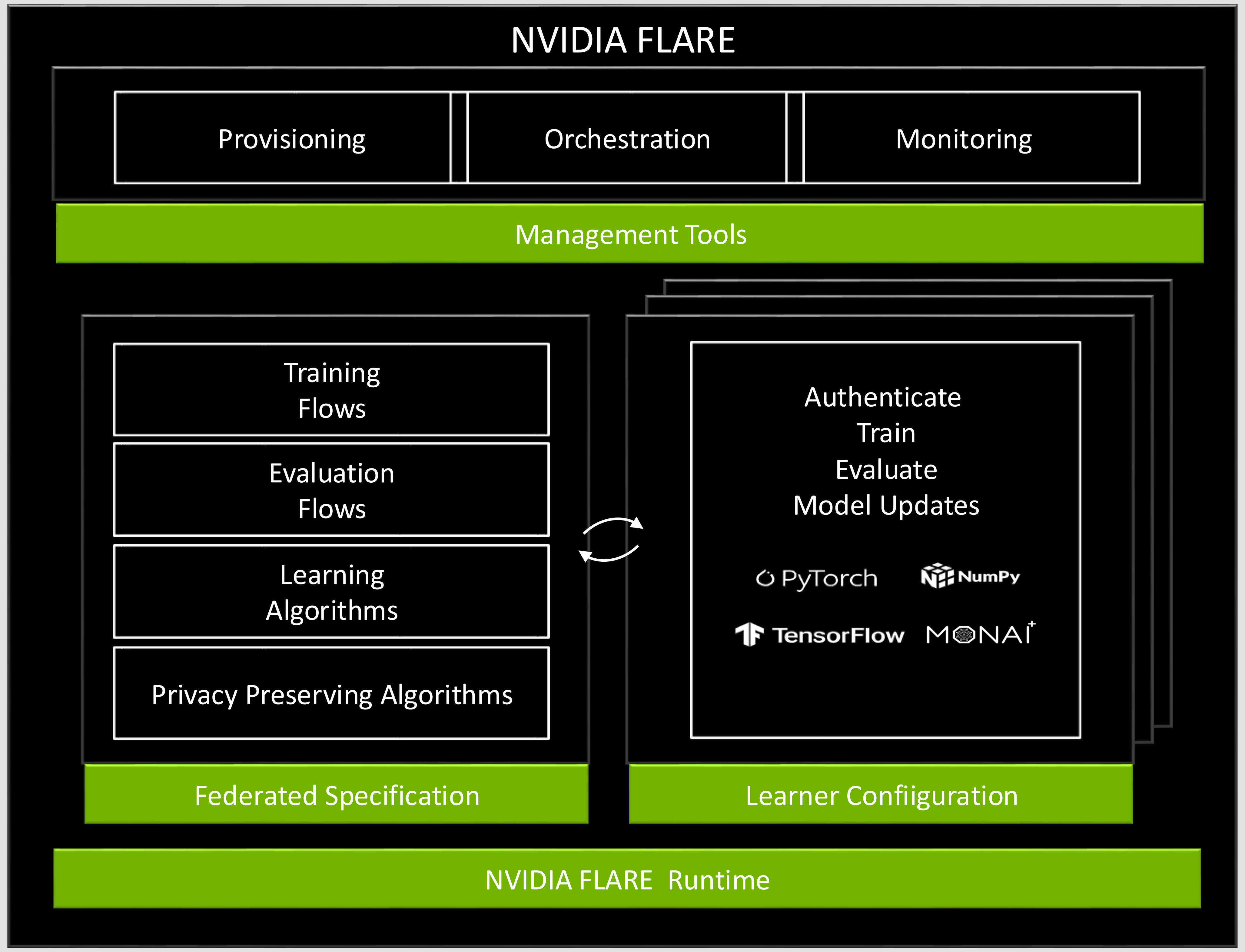
NVIDIA FLARE

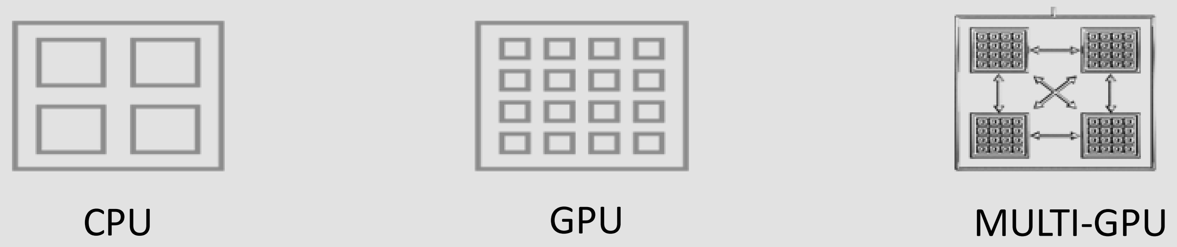
Open-Source, Enterprise Federated Learning & Compute Framework

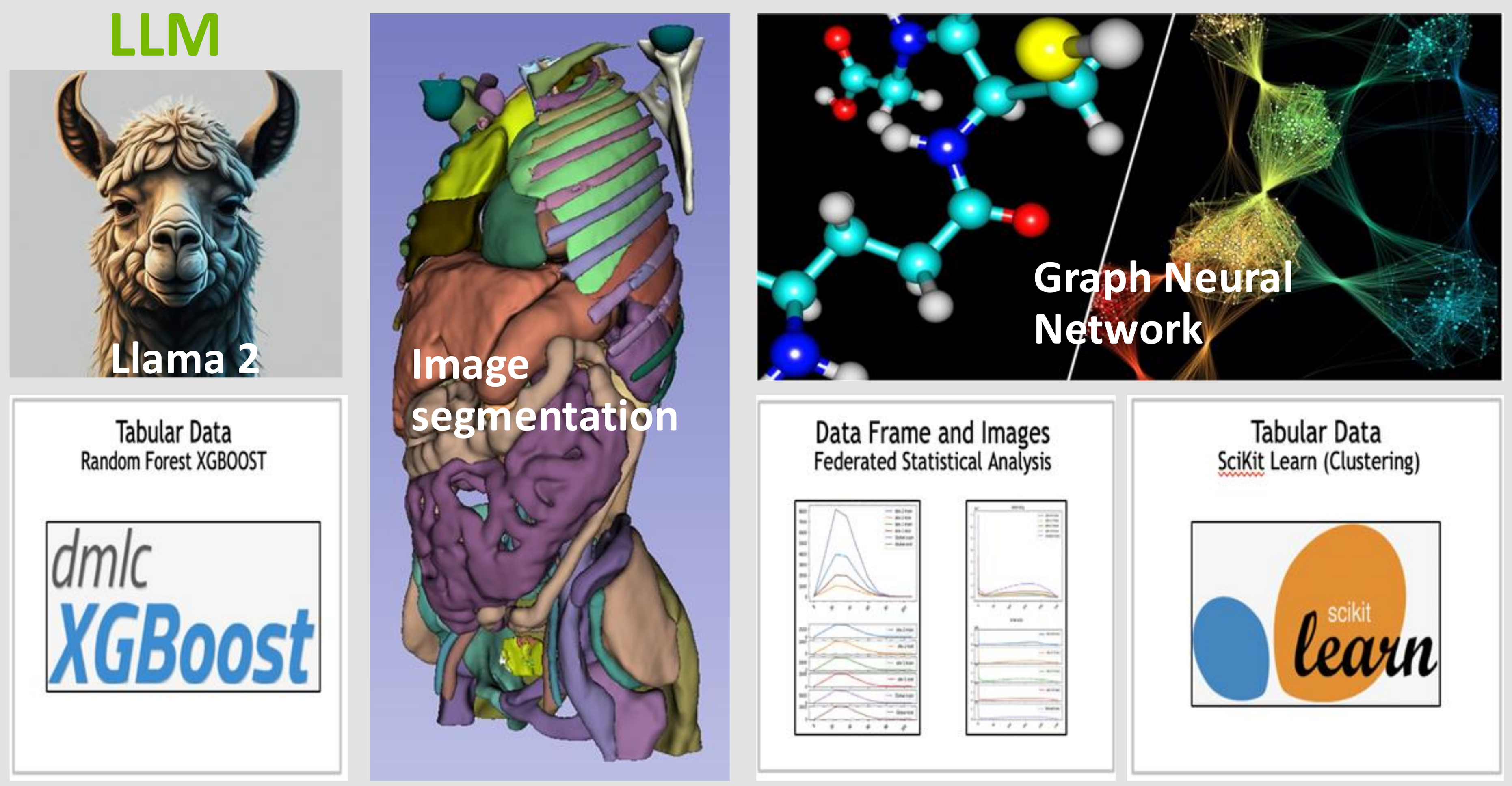
- **Apache License 2.0** to catalyze FL research & development
- **Designed for production**, not just for research
- **Enables cross-country**, distributed, multi-party collaborative Learning
- **Production scalability** with HA and concurrent **multi-task** execution
- **Easy to convert** existing ML/DL workflows to a Federated paradigm with few lines of code changes
- **LLM streaming, LLM fine tuning**
- **Framework, model, domain and task agnostic**
- **Flower Integration**
- **Confidential FL**: end-to-end Federated Learning with Confidential Computing
- **Layered, pluggable, customizable** federated compute architecture
- Secure Provisioning, Orchestration & Monitoring

GitHub: https://github.com/nvidia/nvFlare

Web: https://nvidia.github.io/NVFlare/





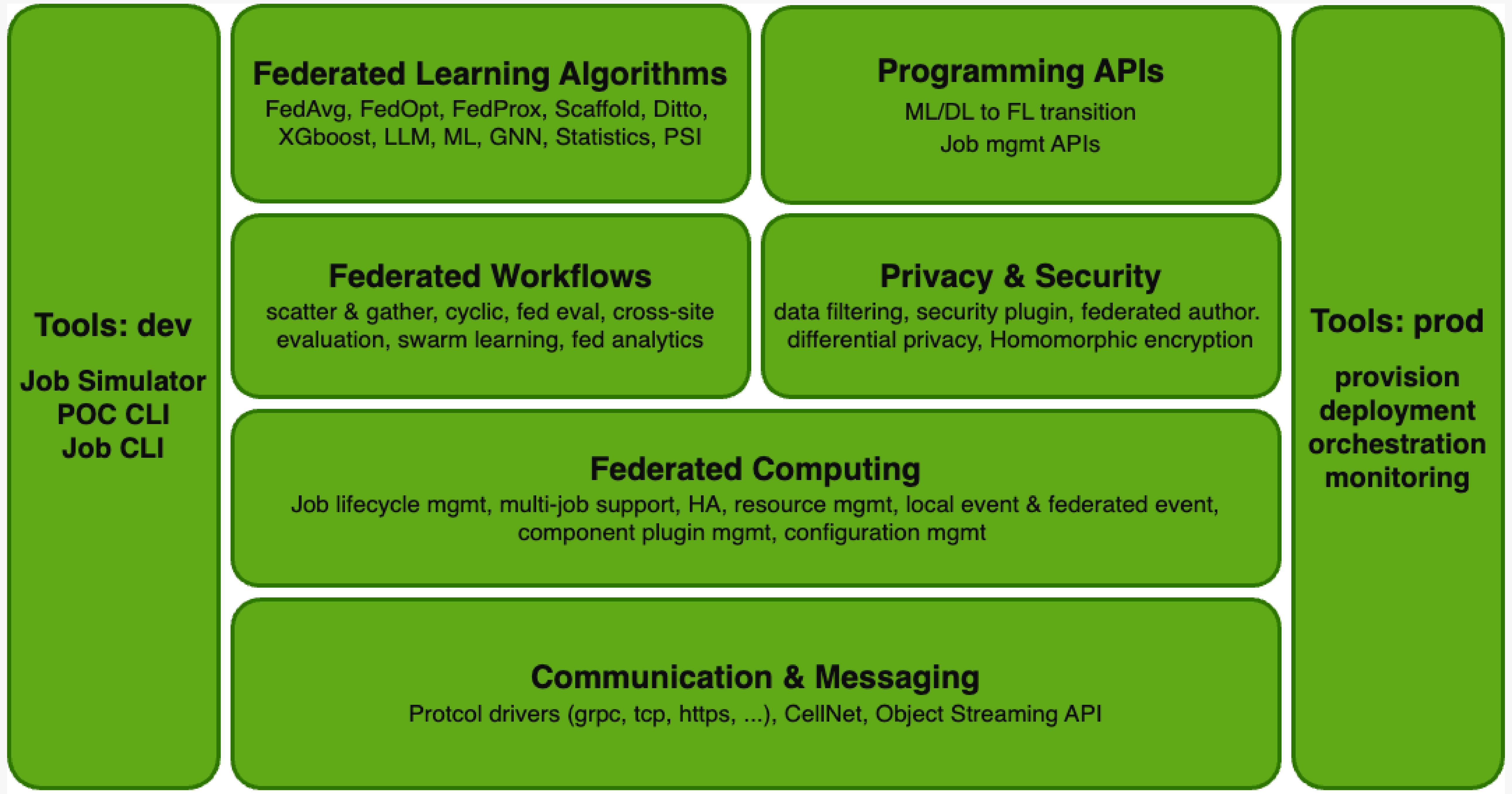


Framework agnostic | Model agnostic | Domain agnostic | Task agnostic

NVIDIA FLARE Architecture

Federated Computing Engine

- **Layered, Pluggable Open Architecture**
  - Each layer's component are customizable and pluggable
- **Network: Communication & Messaging layer**
  - Drivers ➔ gRPC, http + websocket, TCP, any plugin driver
  - CellNet: logical end point-to-point (cell to cell) network
  - Message: reliable streaming message
- **Federated Computing Layer**
  - Resource-based job scheduling, job monitoring, concurrent job lifecycle management, High-availability management
  - Plugin component management
  - Configuration management
  - Local event and federated event handling
- **Federated Workflow**
  - SAG, Cyclic, Cross-site Evaluation, Swarm Learning, Federated Analytics
- **Federated Learning Algorithms**
  - FedAvg, FedOpt, FedProx, Scallfold, Ditto, XGBoost, GNN, PSI, LLM (p-tuning, SFT, PEFT), KM, Scikit-Learn
- **Pythonic Programming APIs**
  - Client API, Controller API, Job Construction API, Job Monitoring API
- **Productivity & Deployment Tools:**
  - Simulator, provision, POC, Cloud deployment, preflight check, more



NVFLARE 2.5.0 Released

- **End-to-End Pythonic APIs**
- **Flower Integration**
- **Secure XGBoost**
  - open sourced libcuda-paillier
- **Developer Tutorial Page**
  - https://nvidia.github.io/NVFlare/

- **New Examples**
  - Secure Federated Kaplan-Meier Analysis
  - BioNemo example for Drug Discovery
  - Federated Logistic Regression with NR optimization
  - Hierarchical Federated Statistics.
  - FedAvg Early Stopping Example
  - Tensorflow Algorithms & Examples
  - FedOpt, FedProx, Scaffold implementation for Tensorflow.
  - FedBN: Federated Learning on Non-IID Features via Local Batch Normalization
  - End-to-end Federated XGBoost example including federated ETL for feature engineering
  - Hello-Flower: example of running flower in NVFLARE

FL Made Easy with NVIDIA FLARE

Converting DL code to FL in minutes

- Client: Client API ➔
  - Lightning Example: 4 lines code changes from DL to FL
- Job API ➔
  - No more editing configuration file
  - End-to-end Python Job construction
- Server: Controller API ➔
  - Simplify FL Algorithm customization

Client API

```
import torch
import torchvision
import torchvision.transforms as transforms
from lit_net import LitNet
from pytorch_lightning import LightningDataModule, Trainer, seed_everything
from torch.utils.data import DataLoader, random_split

# (1) import nvflare lightning client API
import nvflare.client.lightning as flare

seed_everything(7)


DATASET_PATH = "/tmp/nvflare/data"
BATCH_SIZE = 4

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])


def main():
    model = LitNet()
    cifar10_dm = CIFAR10DataModule()
    if torch.cuda.is_available():
        trainer = Trainer(max_epochs=1, accelerator="gpu", devices=1 if torch.cuda.is_available() else None)
    else:
        trainer = Trainer(max_epochs=1, devices=None)

    # (2) patch the lightning trainer
    flare.patch(trainer)

    while flare.is_running():
        # (3) receives FLModel from NVFlare
        # Note that we don't need to pass this input_model to trainer
        # because after flare.patch the trainer.fit/validate will get the
        # global model internally
        input_model = flare.receive()
        print(f"\n[Current Round={input_model.current_round}, Site = {flare.get_site_name()}]\n")

        # (4) evaluate the current global model to allow server-side model selection
        print("--- validate global model ---")
        trainer.validate(model, datamodule=cifar10_dm)

        # perform local training starting with the received global model
        print("--- train new model ---")
        trainer.fit(model, datamodule=cifar10_dm)

        # test local model
        print("--- test new model ---")
        trainer.test(ckpt_path="best", datamodule=cifar10_dm)

        # get predictions
        print("--- prediction with new best model ---")
        trainer.predict(ckpt_path="best", datamodule=cifar10_dm)
```

FL Made Easy with NVIDIA FLARE

Construct FL Job via python code

- Client: Client API ➔
  - Lightning Example: 4 lines code changes from DL to FL
- Job API ➔
  - No more editing configuration file
  - End-to-end python Job construction
- Server: Controller API ➔
  - Simplify FL Algorithm customization

Job API

```
from src.net import Net

from nvflare.app_common.widgets.intime_model_selector import IntimeModelSelector
from nvflare.app_common.workflows.fedavg import FedAvg
from nvflare.app_opt.pt.job_config.model import PTModel

from nvflare.job_config.api import FedJob
from nvflare.job_config.script_runner import ScriptRunner

if __name__ == "__main__":


  n_clients = 2
  num_rounds = 2
  train_script = "src/cifar10_fl.py"

  job = FedJob(name="cifar10_fedavg")


  controller = FedAvg( num_clients=n_clients,  num_rounds=num_rounds )


  job.to(controller, "server")

  # Define the initial global model and add to server
  job.to(PTModel(Net()), "server")

  job.to(IntimeModelSelector(key_metric="accuracy"), "server")


  # Add clients
  for i in range(n_clients):
    executor = ScriptRunner(
      script=train_script, script_args=""  # f"--batch_size 32 --data_path /tmp/data/site-{i}"
    )
    job.to(executor, target=f"site-{i}")

  job.export_job("/tmp/nvflare/jobs/job_config")
  job.simulator_run("/tmp/nvflare/jobs/workdir", gpu="0")
```

FL Made Easy with NVIDIA FLARE

Customizing server-side FL logics is just a for loop logics

- Client: Client API ➔
  - Lightning Example,
  - 4 lines code changes from DL to FL
- Job API ➔
  - No more editing configuration file
  - End-to-end Python Job construction
- Server: Controller API ➔
  - Simplify FL Algorithm customization for researchers who like experiment with new FL Algorithms

Controller API

```
class ModelController(BaseModelController, ABC):

    @abstractmethod
    def run(self)

    def send_model_and_wait( …) -> List[FLModel]

    def send_model( ...,  callback: Callable[[FLModel], None] = None) -> None

    def load_model(…) -> FLModel

    def save_model(…, model: FLModel) -> None:

    def sample_clients(…, num_clients: int = None) -> List[str]:

class FLModel:
    def __init__(
        self,
        params_type: Union[None, str, ParamsType] = None,
        params: Any = None,
        optimizer_params: Any = None,
        metrics: Optional[Dict] = None,
        start_round: Optional[int] = 0,
        current_round: Optional[int] = None,
        total_rounds: Optional[int] = None,
        meta: Optional[Dict] = None,
    ):
```

NVIDIA FLARE: Summary

A domain-agnostic, open-source, extensible FL framework

- **Federated Computing** -- a federated computing framework at core
- **Built for productivity** -- designed for maximum productivity, providing a range of tools to enhance user experience
- **Built for security & privacy** -- prioritizes robust security and privacy preservation
- **Built for concurrency & scalability** -- designed for concurrency, supporting resource-based multi-job execution
- **Built for customization** -- structured in layers, with each layer composed of customizable components
- **Built for integration** -- multiple integration options with third-party system
- **Built for production** -- robust, production-scale deployment in real-world federated learning and computing scenarios
- **Rich examples repository** -- wealth of built-in implementations, tutorials and examples
- **Growing application categories --** medical imaging, medical devices, edge device application, financial services, HPC and autonomous driving vehicles

GitHub : https://github.com/NVIDIA/NVFlare

Web: https://nvidia.github.io/NVFlare/

NVIDIA FLARE Product 2024-2025 Road Map

Release plan

2024 – Sept.

Release 2.5.0 (Released 9/9)

Major User Experience upgrade

Secure XGBoost

2024 – Oct.

FLARE 2.5.1

Python 3.11+ support

2025 –Q1

Release 2.6.0

Confidential FL Release

Additional LLM support

Thank You !

Chester Chen, chesterc@nvidia.com

NVIDIA FLARE Getting Started

Holger Roth

Principal Federated Learning Scientist

NVIDIA Federated Learning

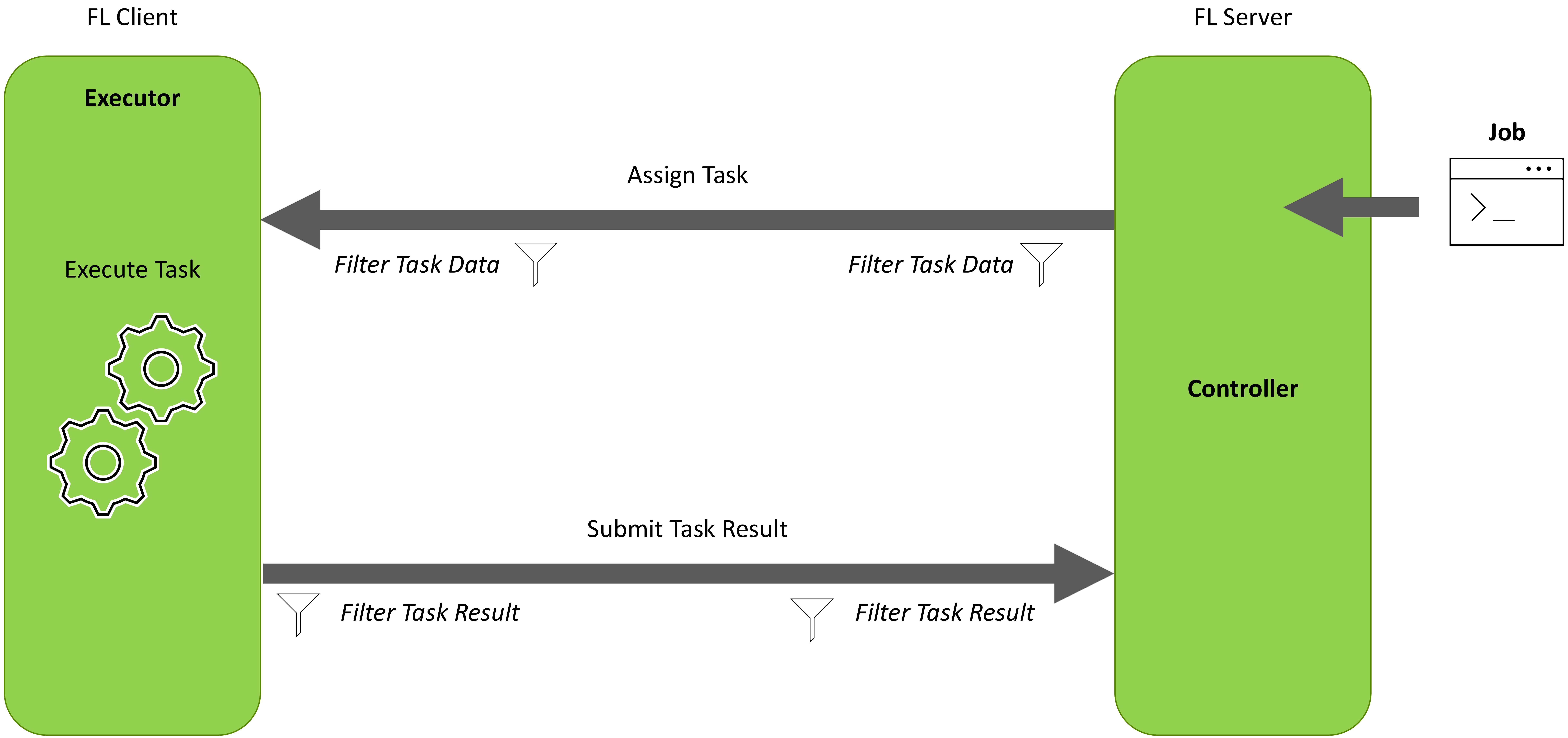
NVIDIA FLARE DAY

September 18, 2024

NVIDIA Federated Learning

Applications across industries

Basic Concepts

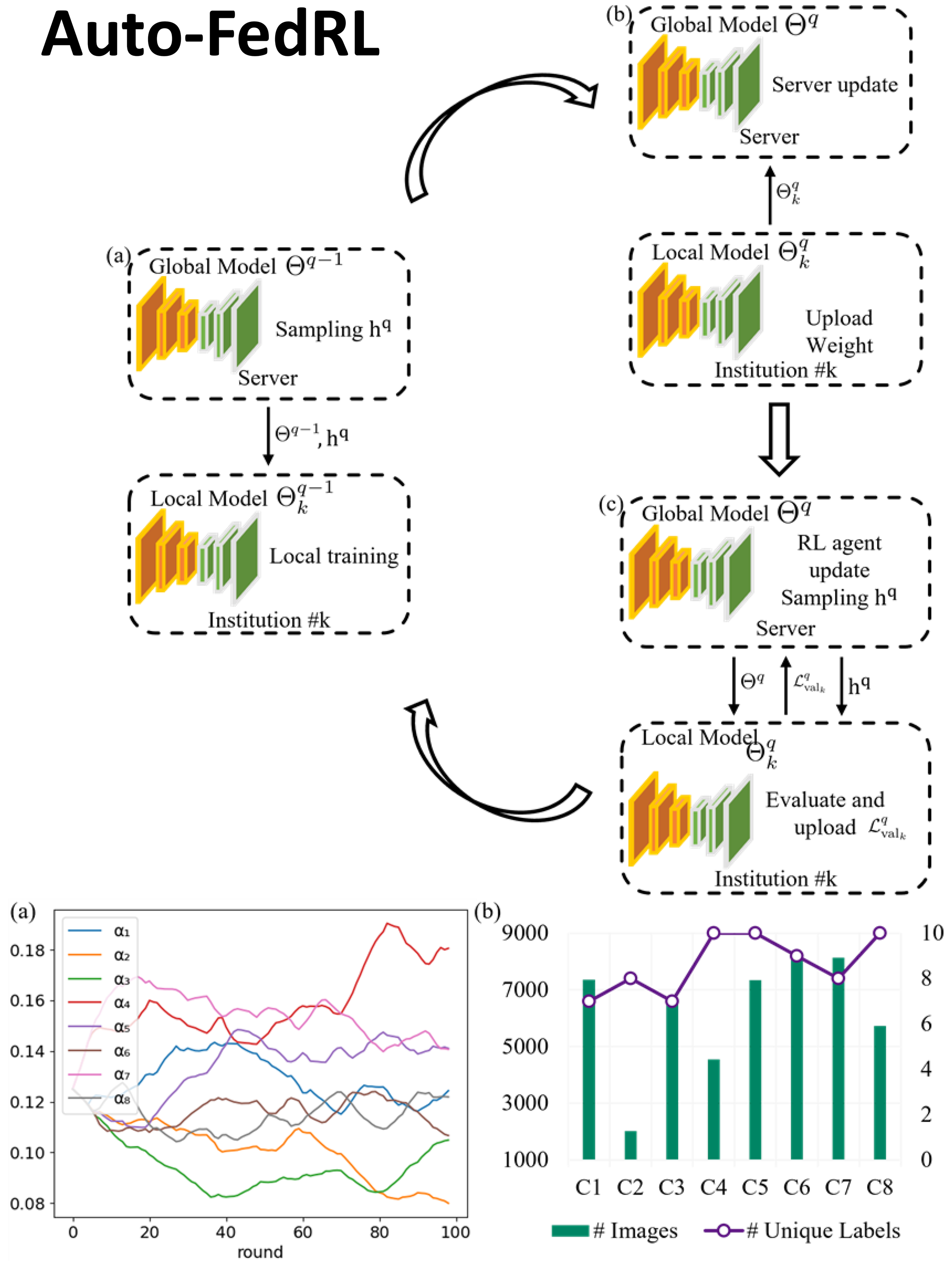


Note: Filters can be enforced by the data owners!

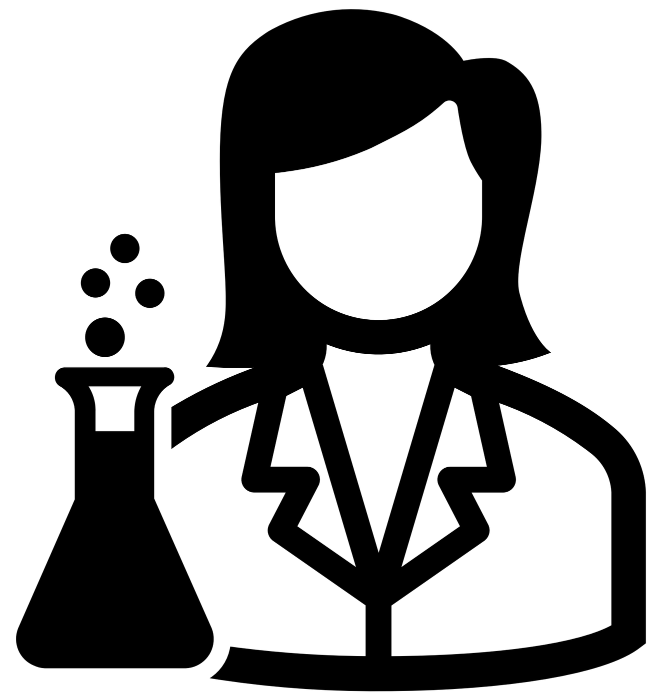
Research With NVFlare

https://github.com/NVIDIA/NVFlare/tree/dev/research

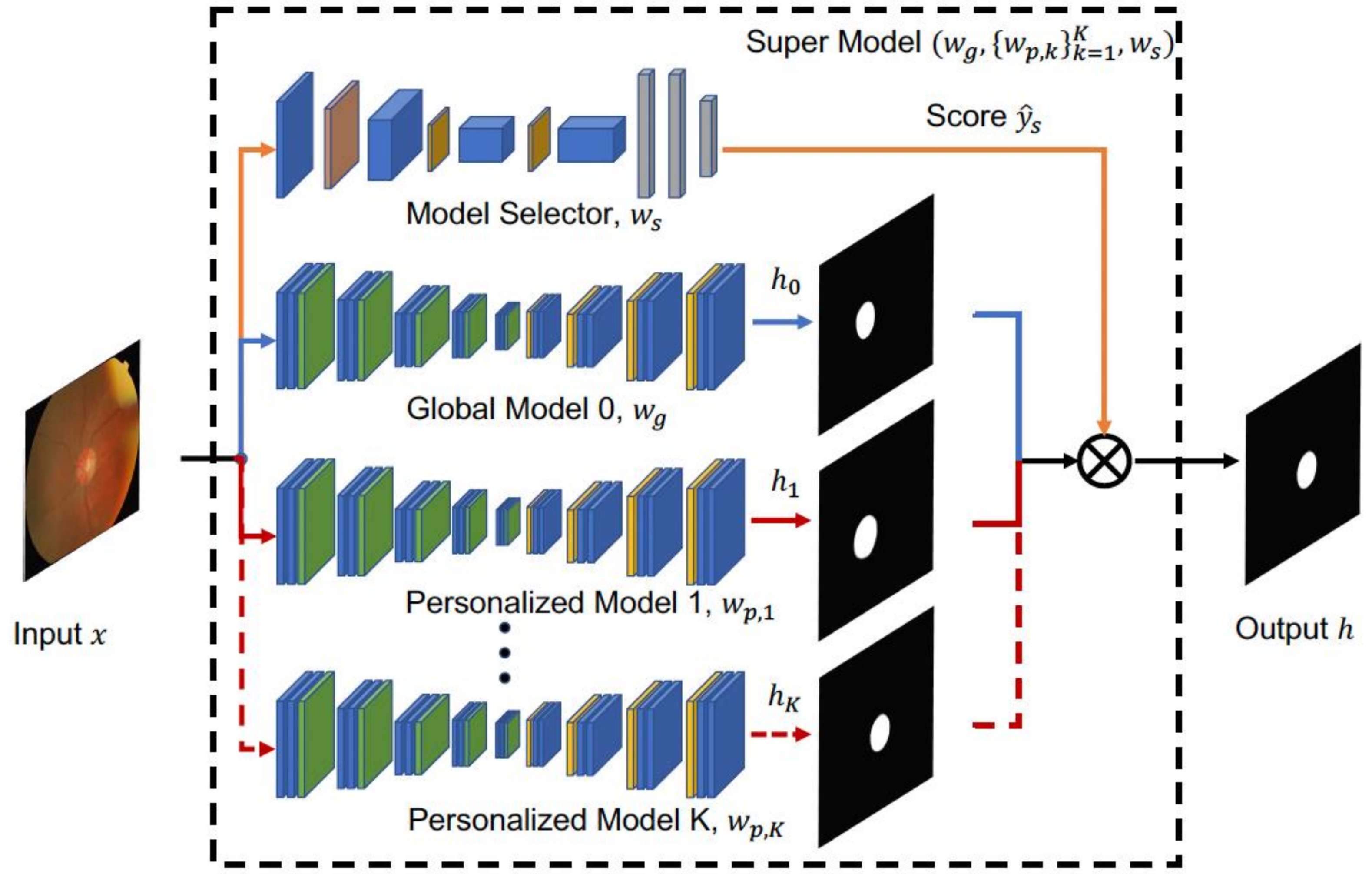
Auto-FedRL



P. Guo et al. ECCV 2022



FedSM: Personalized FL

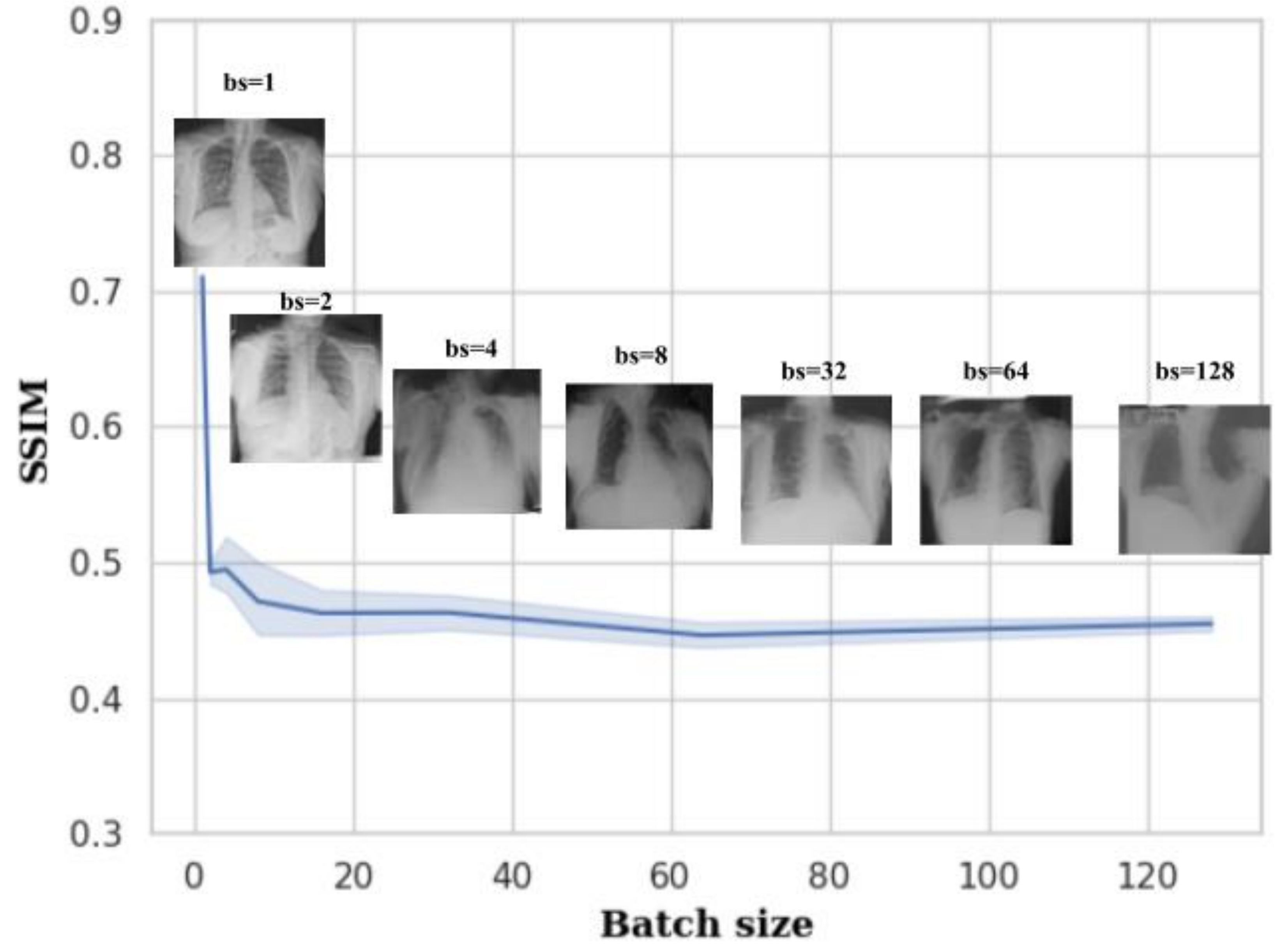


A. Xu et al. CVPR 2022

Baseline Implementations

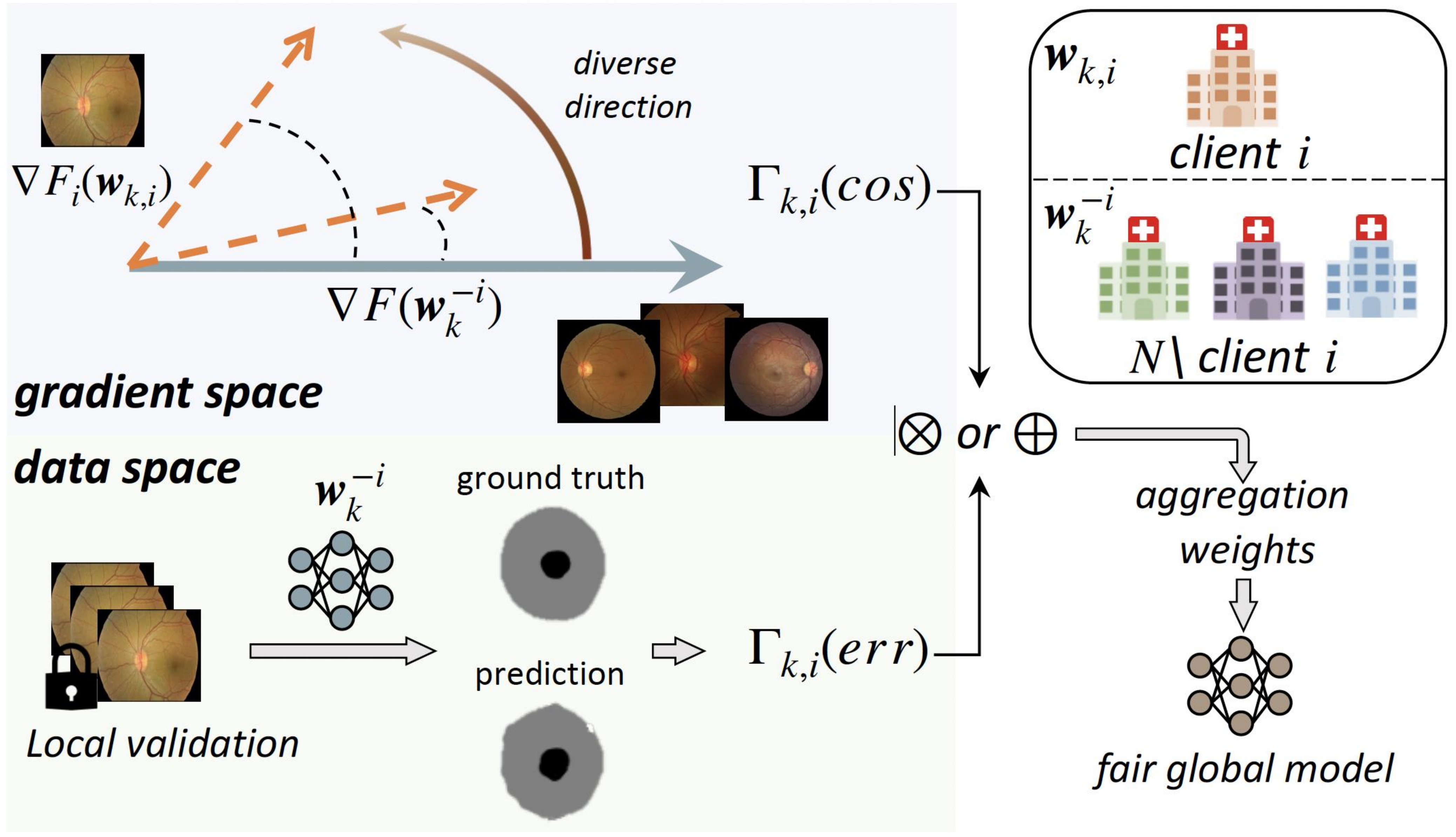
- FedAvg
- FedProx
- FedOpt
- SCAFFOLD
- Ditto
- Cyclic Weight Transfer
- Swarm Learning

Quantifying data leakage



A. Hatamizadeh et al. TMI 2022

FedCE: Contribution Estimation



M. Jiang et al. CVPR 2023

Server Code: Controller

```python
class FedAvg(BaseFedAvg):

    def run(self) -> None:
        self.info("Start FedAvg.")

        model = self.load_model()
        model.start_round = self.start_round
        model.total_rounds = self.num_rounds

        for self.current_round in range(self.start_round, self.start_round + self.num_rounds):
            self.info(f"Round {self.current_round} started.")
            model.current_round = self.current_round

            clients = self.sample_clients(self.min_clients)

            results = self.send_model_and_wait(targets=clients, data=model)

            aggregate_results = self.aggregate(
                results, aggregate_fn=None
            )  # if no `aggregate_fn` provided, default `WeightedAggregationHelper` is used

            model = self.update_model(model, aggregate_results)

            self.save_model(model)

        self.info("Finished FedAvg.")
```

Client Code:
Convert PyTorch to NVFlare

PyTorch CIFAR-10 Tutorial

```python
from net import Net


def main():
    transform = transforms.Compose([...])

    trainset = torchvision.datasets.CIFAR10(...)
    trainloader = torch.utils.data.DataLoader(...)

    testset = torchvision.datasets.CIFAR10(...)
    testloader = torch.utils.data.DataLoader(...)

    net = Net()

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(...)

    # Train loop
    for epoch in range(epochs):
        ...

    print("Finished Training")
```

```python
from net import Net

import nvflare.client as flare


def main():
    transform = transforms.Compose([...])

    trainset = torchvision.datasets.CIFAR10(...)
    trainloader = torch.utils.data.DataLoader(...)

    testset = torchvision.datasets.CIFAR10(...)
    testloader = torch.utils.data.DataLoader(...)

    net = Net()

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(...)

    flare.init()

    while flare.is_running():
        input_model = flare.receive()
        print(f"current_round={input_model.current_round}")

        net.load_state_dict(input_model.params)

        for epoch in range(epochs):  # loop over the dataset multiple times
            ...

        print("Finished Training")

        output_model = flare.FLModel(
            params=net.cpu().state_dict(),
            metrics={"accuracy": accuracy},
            meta={"NUM_STEPS_CURRENT_ROUND": epochs * len(trainloader)},
        )
        flare.send(output_model)
```

1. import client API
2. Initialize
3. Receive global model
4. Load global model
5. Send back the updated model

Create a FedJob and Run Simulation

```python
if __name__ == "__main__":
    n_clients = 2
    num_rounds = 2
    train_script = "src/cifar10_fl.py"

    # Create basic fed Job with initial model
    job = BaseFedJob(
        name="cifar10_pt_fedavg",
        initial_model=Net(),
    )

    # Define the controller and send to server
    controller = FedAvg(
        num_clients=n_clients,
        num_rounds=num_rounds,
    )
    job.to_server(controller)

    # Add clients
    for i in range(n_clients):
        runner = ScriptRunner(script=train_script)  # script_args=f"--batch_size 32 --data_path /data/site-{i}"
        job.to(runner, target: f"site-{i}")

    # job.export_job("/tmp/nvflare/jobs/job_config")    # Exported jobs can be used in real deployment!
    job.simulator_run( workspace: "/tmp/nvflare/jobs/workdir", gpu="0")
```

Client Code:
Lightning client API

Transform your script to FL with a few lines of code changes:

1. Import NVFlare lightning API
2. Patch your lightning trainer
3. (Optionally) validate the current global model
4. Train as usually

```
from nemo.core.config import hydra_runner
from nemo.utils import AppState, logging
from nemo.utils.exp_manager import exp_manager
from nemo.utils.model_utils import inject_model_parallel_rank

# (0): import nvflare lightning api
import nvflare.client.lightning as flare

mp.set_start_method("spawn", force=True)
```

...

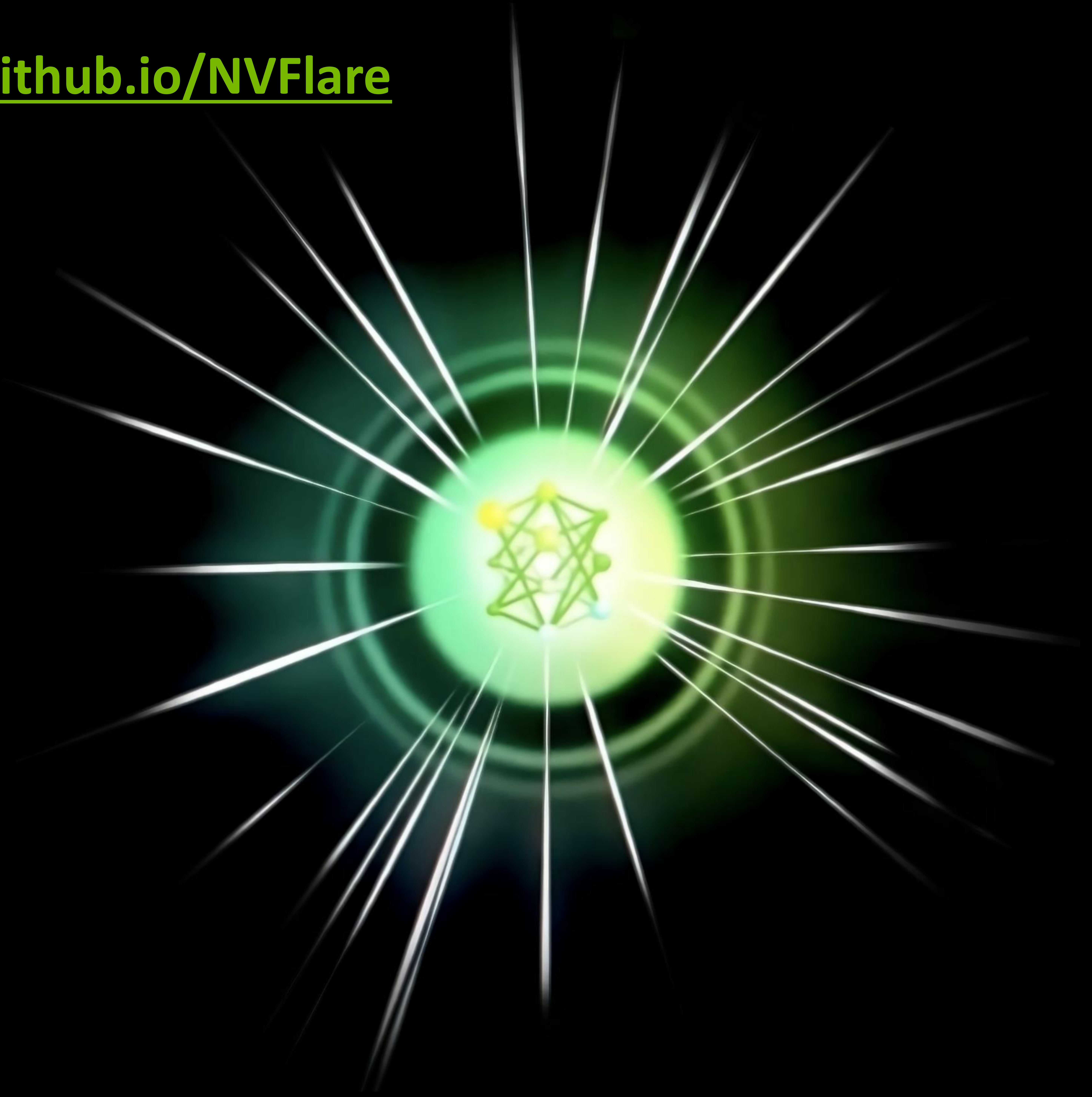
```
# (1): flare patch
flare.patch(trainer)

while flare.is_running():

    # (2) evaluate the current global model to allow server-side model selection
    print("--- validate global model ---")
    trainer.validate(model)

    # (3) Perform local training starting with the received global model
    print("--- train new model ---")
    trainer.fit(model)
```

What's new Just shipped v2.5.0 >

https://nvidia.github.io/NVFlare

NVIDIA FLARE

NVIDIA FLARE™ (NVIDIA Federated Learning Application Runtime Environment) is a domain-agnostic, open-source, and extensible SDK for Federated Learning. It allows researchers and data scientists to adapt existing ML/DL workflow to a federated paradigm and enables platform developers to build a secure, privacy-preserving offering for a distributed multi-party collaboration.



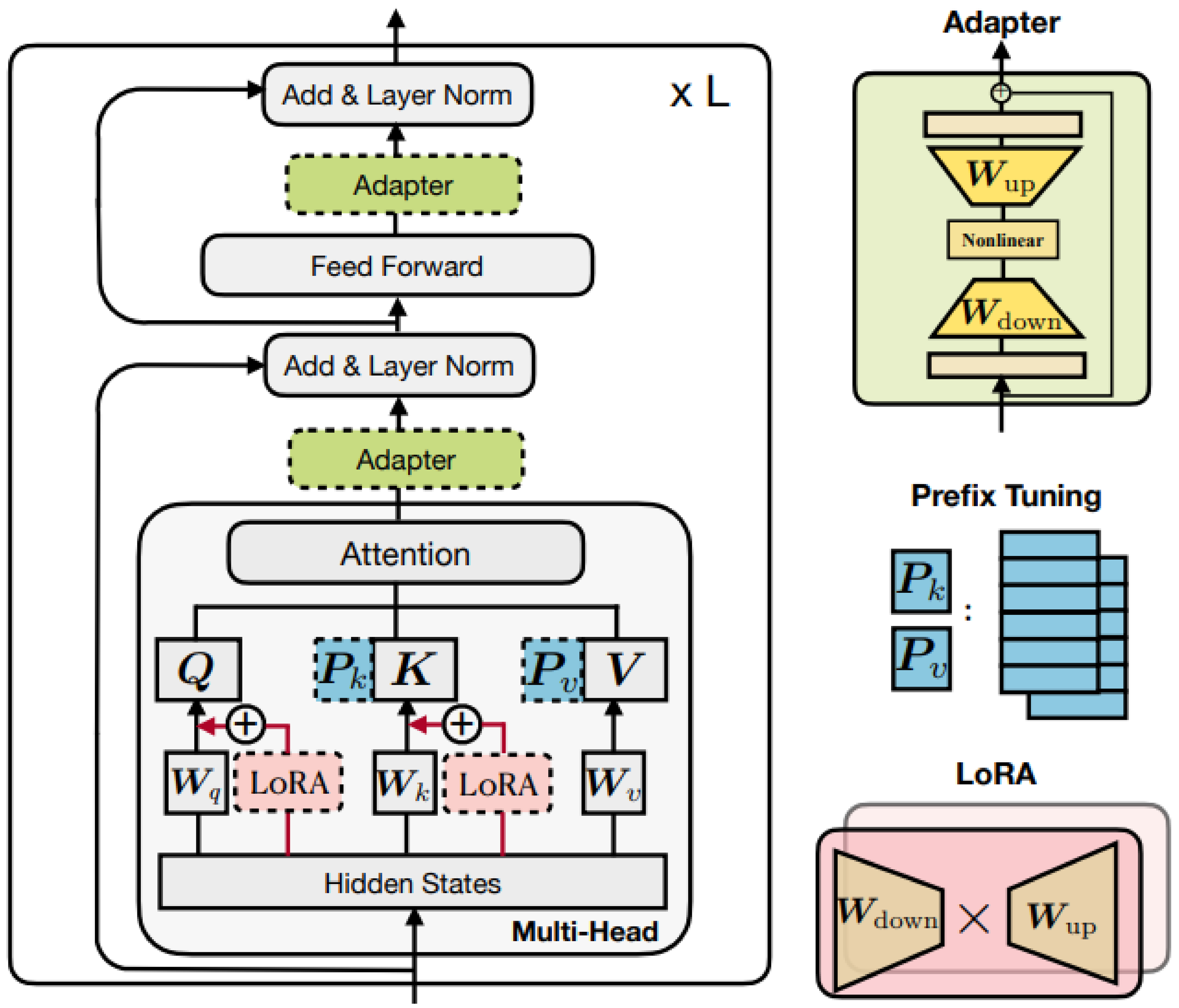
Documentation | Tutorial Catalog | GitHub→

LLM Support – PEFT, SFT, RAG

Compare PEFT Methods With NeMo

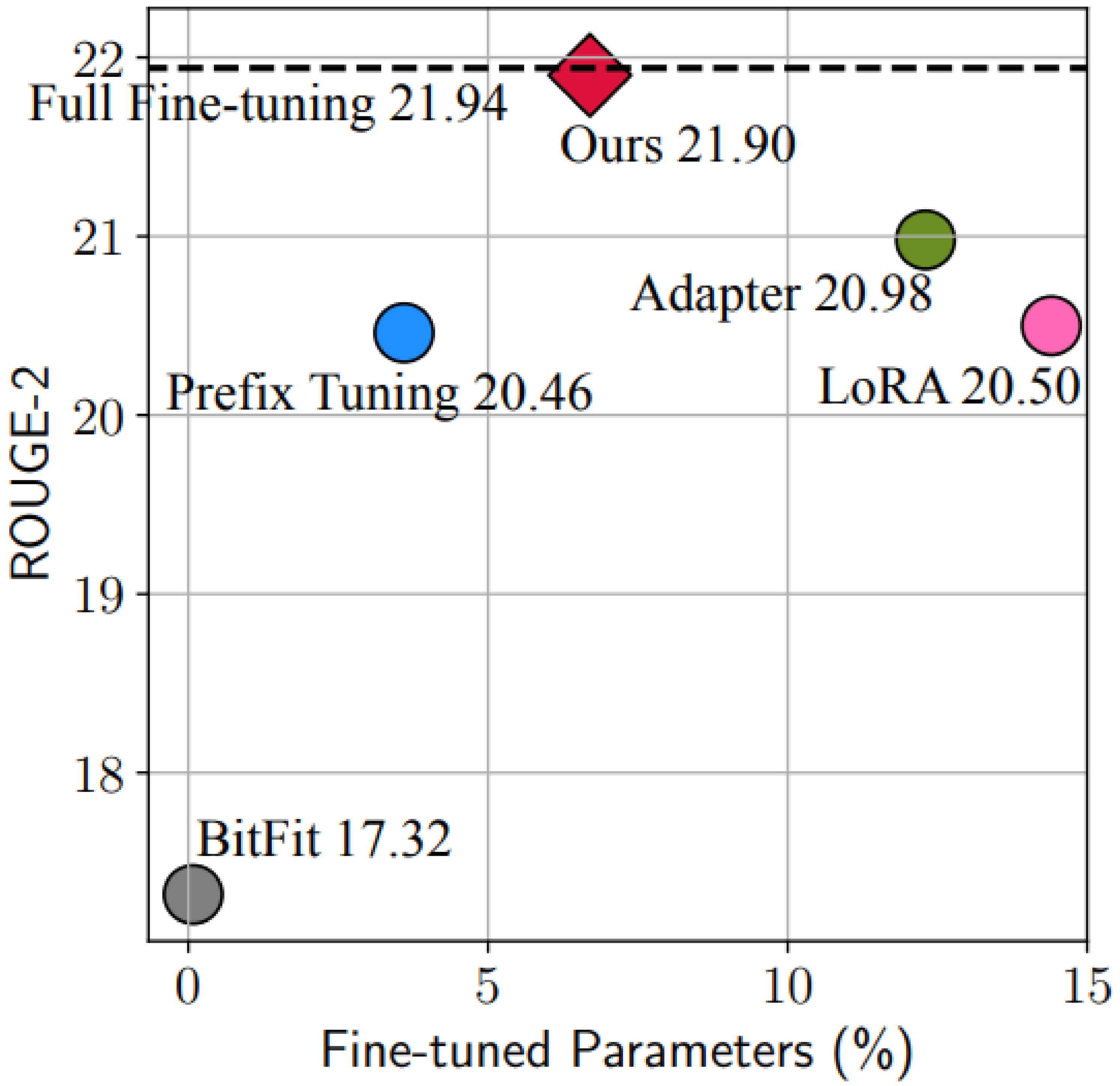
Only 1 line configuration change

Transformer and PEFT methods:



https://arxiv.org/abs/2110.04366

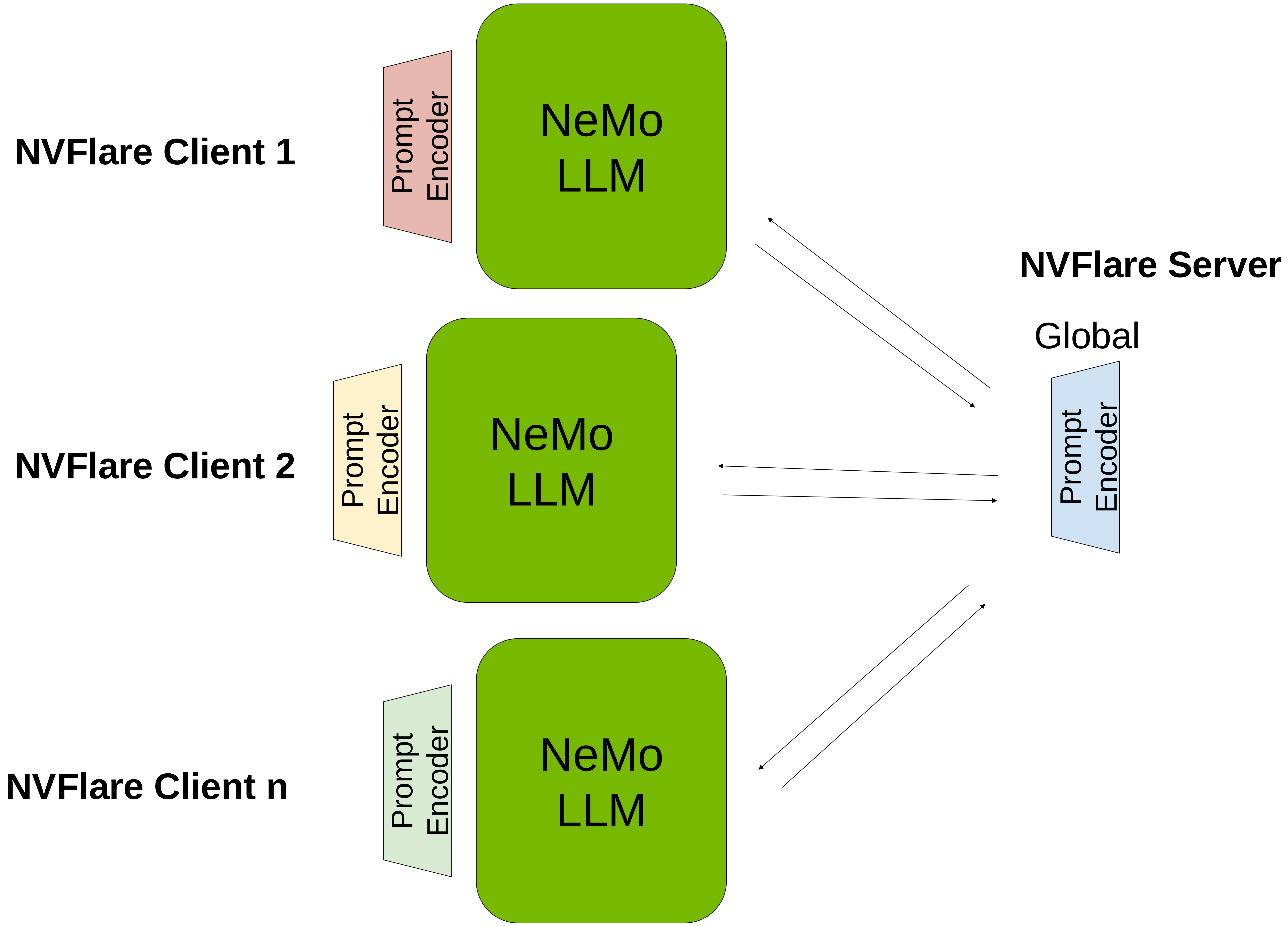
Different PEFT methods on the XSum summarization task:




```yaml
peft:
  peft_scheme: "adapter"  # can be either adapter,ia3, or ptuning
  restore_from_path: null

  # Used for adapter peft training
  adapter_tuning:
    type: 'parallel_adapter' # this should be either 'parallel_adap
    adapter_dim: 32
    adapter_dropout: 0.0
    norm_position: 'pre' # This can be set to 'pre', 'post' or null
    column_init_method: 'xavier' # IGNORED if linear_adapter is use
    row_init_method: 'zero' # IGNORED if linear_adapter is used, op
    norm_type: 'mixedfusedlayernorm' # IGNORED if layer_adapter is
    layer_selection: null  # selects in which layers to add adapter
    weight_tying: False
    position_embedding_strategy: null # used only when weight_tying

  lora_tuning:
    adapter_dim: 32
    adapter_dropout: 0.0
    column_init_method: 'xavier' # IGNORED if linear_adapter is use
    row_init_method: 'zero' # IGNORED if linear_adapter is used, op
    layer_selection:  null  # selects in which layers to add lora a
    weight_tying: False
    position_embedding_strategy: null # used only when weight_tying

  # Used for p-tuning peft training
  p_tuning:
    virtual_tokens: 10  # The number of virtual tokens the prompt e
    bottleneck_dim: 1024  # the size of the prompt encoder mlp bott
    embedding_dim: 1024  # the size of the prompt encoder embedding
    init_std: 0.023

  ia3_tuning:
    layer_selection:  null  # selects in which layers to add ia3 ad
```

NeMo YAML configuration

NVFlare for P-Tuning With NeMo



LLM parameters stay fixed; Prompt encoder parameters are trained/updated

Example: Sentiment Analysis

Downstream task example:

- Financial PhraseBank dataset (Malo et al.) for sentiment analysis.
- The Financial PhraseBank dataset contains the sentiments for financial news headlines from a retail investor's perspective.

Example prompts and predictions:

The products have a low salt and fat content . ***sentiment: neutral***

The agreement is valid for four years . ***sentiment: neutral***

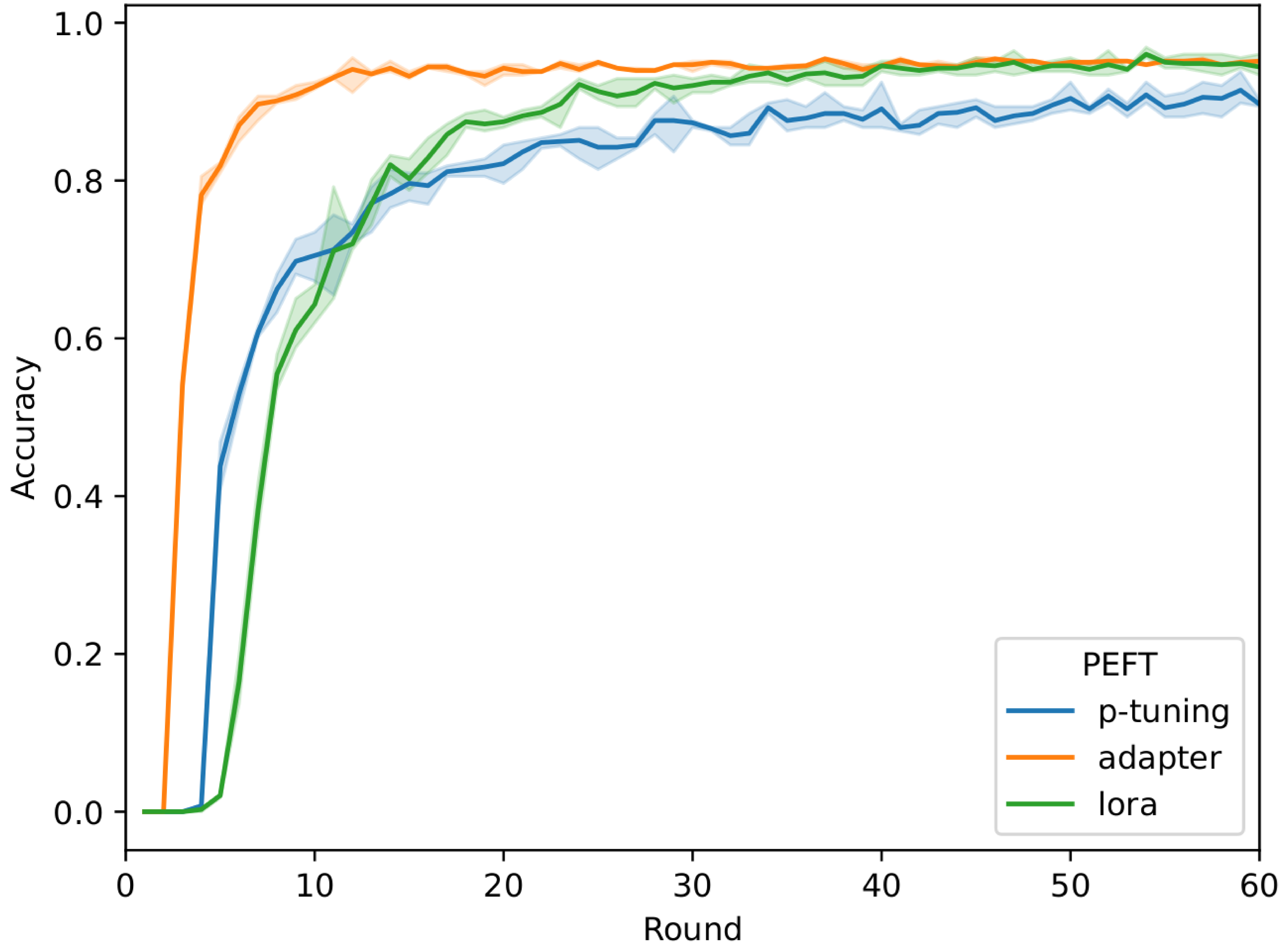
Diluted EPS rose to EUR3 .68 from EUR0 .50 . ***sentiment: positive***

The company is well positioned in Brazil and Uruguay . ***sentiment: positive***

Profit before taxes decreased by 9 % to EUR 187.8 mn in the first nine months of 2008 , compared to EUR 207.1 mn a year earlier . ***sentiment: negative***

Compare PEFT Methods With NeMo

P-tuning vs. Adapter vs. LoRa



Tensor parallel with **2 GPUs** per client

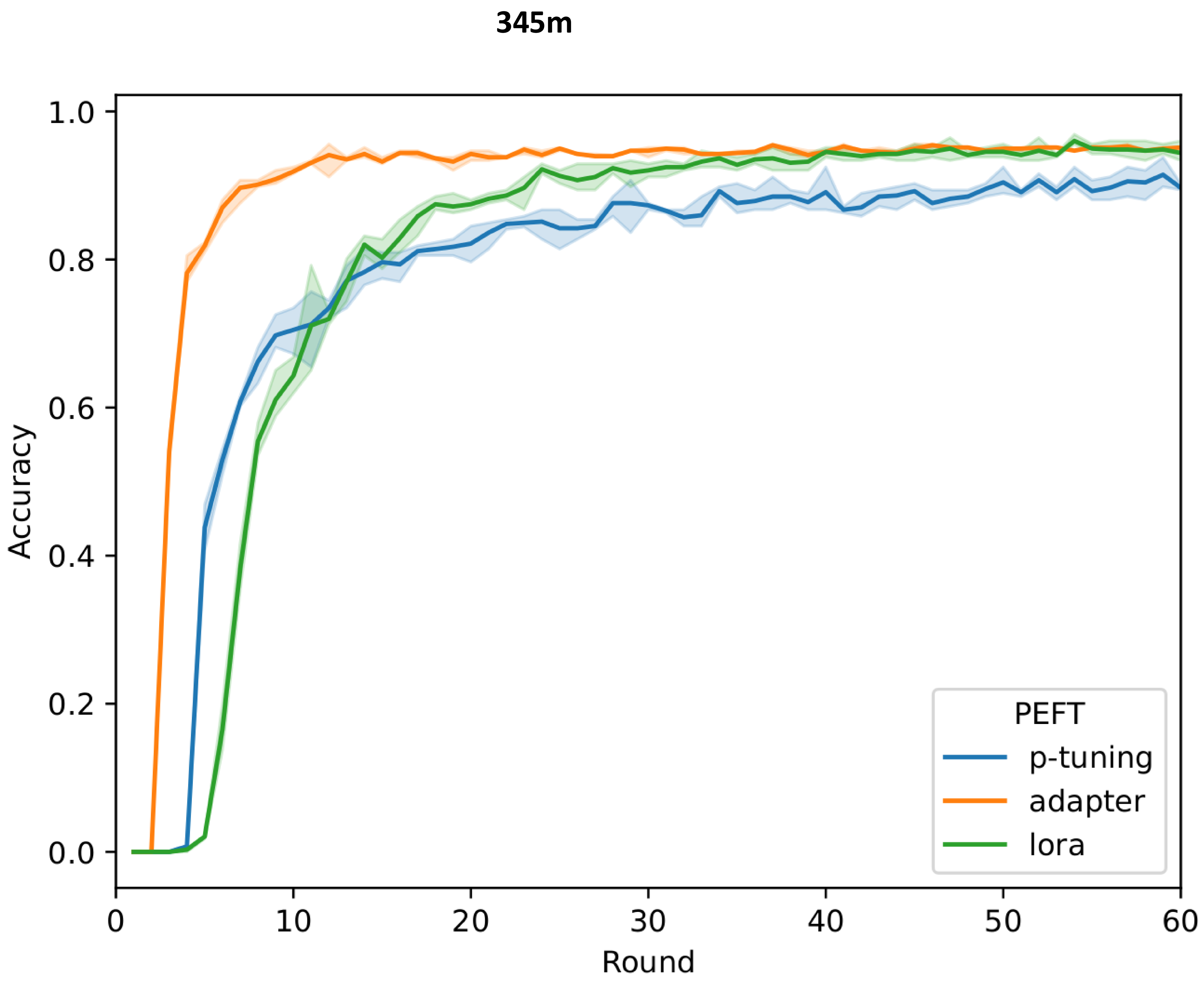
345M Param NeMo GPT Megatron model

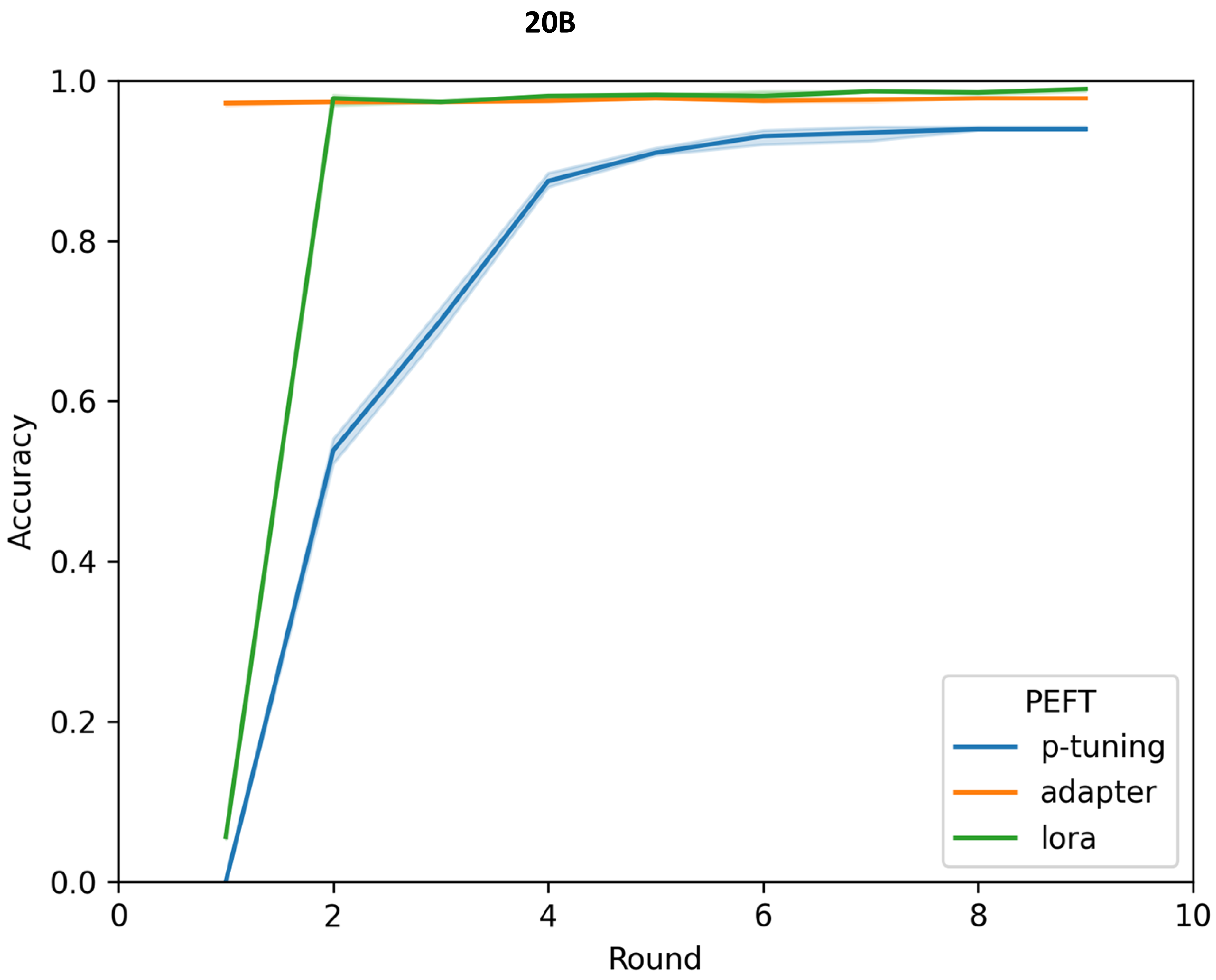
PEFT Method	**Execution time**
P-tuning	4h 59m
Adapter	11h 25m
LoRA	7h 27m

notebook

Compare PEFT Methods With NeMo

P-tuning vs. Adapter vs. LoRa



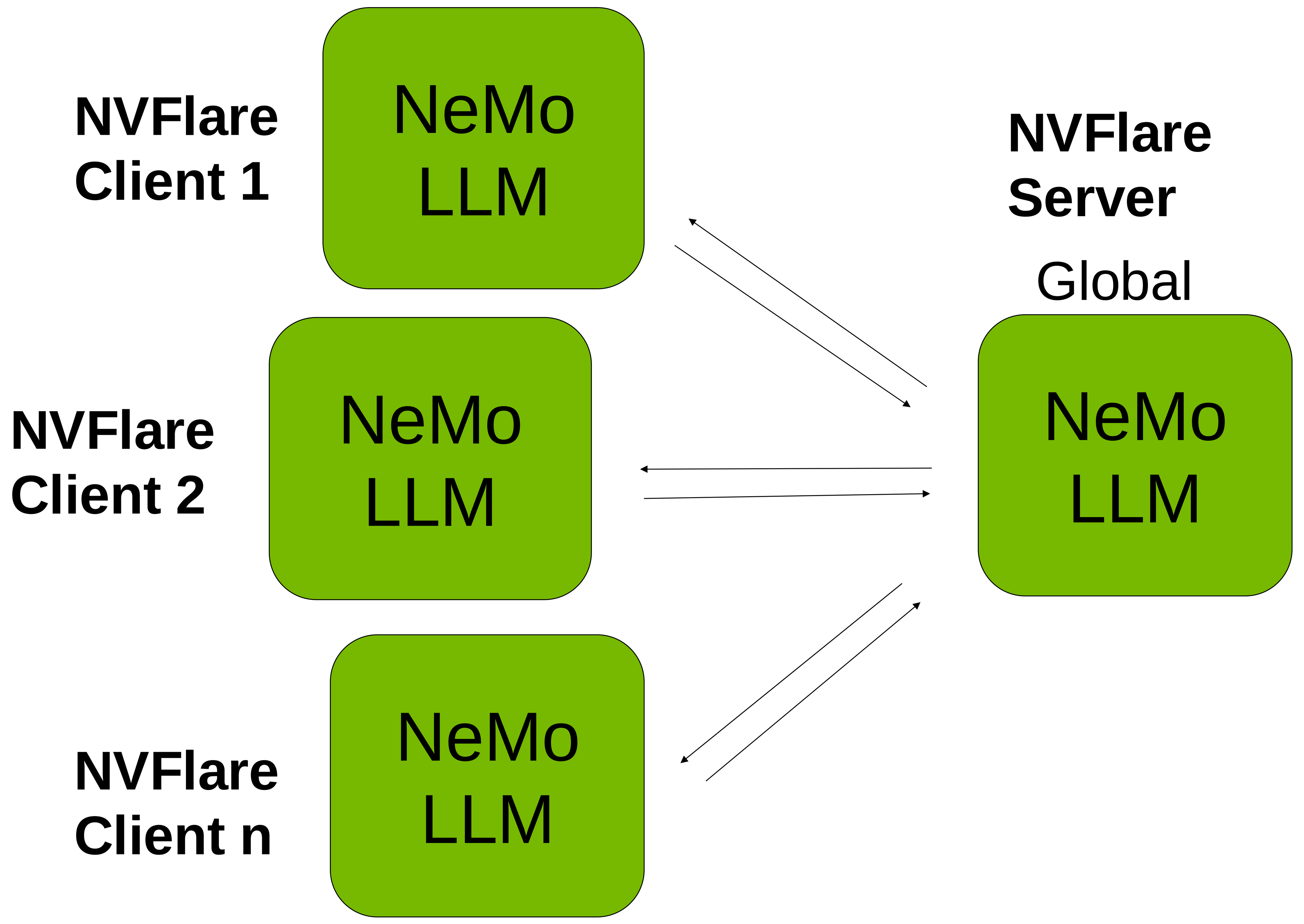


https://github.com/NVIDIA/NVFlare/tree/main/integration/nemo/examples

Supervised Fine-tuning (SFT)

Learning a “instruction-following” LLM

Unlike PEFT, SFT finetunes the entire network



The first step of “Chat-GPT training scheme”.

NVFlare Streaming

Support Large Model Transmission

- Model size of mainstream LLM can be huge: 7B -> 26 GB (beyond the 2 GB GRPC limit)
- In order to transmit LLMs in SFT, NVFlare supports **large object** streaming

https://nvflare.readthedocs.io/en/2.5.0/real_world_fl/notes_on_large_models.html#notes-on-large-models

SFT for Instruction Tuning

3 open datasets

We use three datasets:

- Alpaca
- databricks-dolly-15k
- OpenAssistant

with **instruction tuning data**:

- Full conversations
- Instructions (w/ and w/o context) & responses

SFT Model Evaluation

LLM Benchmark Performance

Evaluation under zero-shot setting. BaseModel - before SFT.

	H_acc	**H_acc _norm**	**P_acc**	**P_acc _norm**	**W_acc**	**Mean**
BaseModel	0.357	0.439	0.683	0.689	0.537	0.541
Alpaca	0.372	0.451	0.675	0.687	0.550	0.547
Dolly	0.376	**0.474**	0.671	0.667	0.529	0.543
Oasst1	0.370	0.452	0.657	0.655	0.506	0.528
Combined	0.370	0.453	0.685	**0.690**	0.548	0.549
FedAvg	**0.377**	0.469	**0.688**	0.687	**0.560**	**0.556**

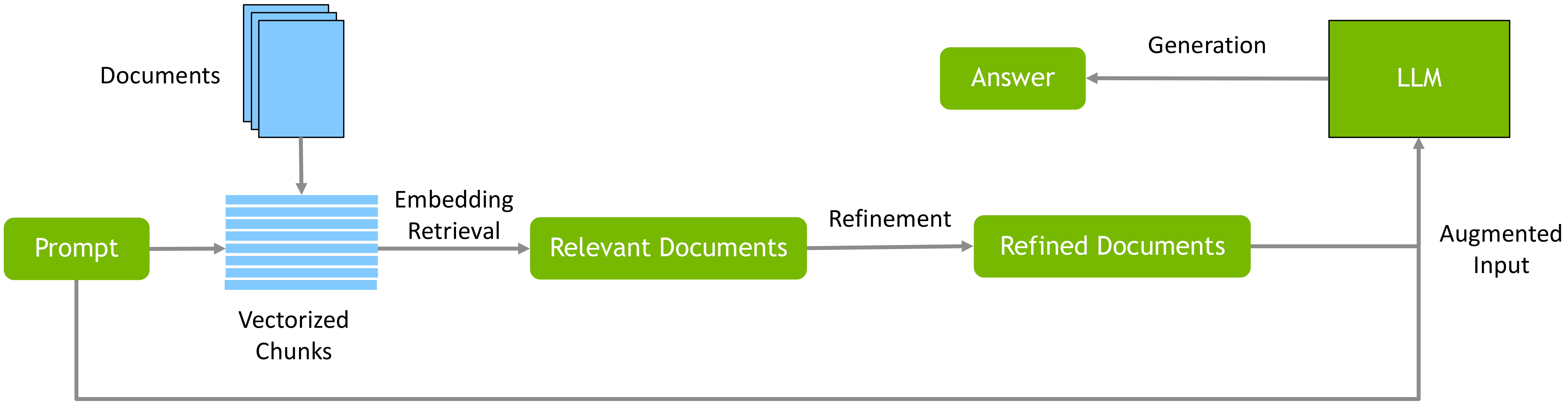
Table 1. Model performance on three benchmark tasks: HellaSwag (H), PIQA (P), and WinoGrande (W)

Blog: https://developer.nvidia.com/blog/scalable-federated-learning-with-nvidia-flare-for-enhanced-llm-performance

Retrieval Augmented Generation (RAG)

Basics

- Three models:
  - **Embedding model:** “vectorizes” a database into information “chunks” that can be searched
  - **Ranking model:** refine chunks relevant to input prompt
  - **Generation model:** Gives answer using the retrieved ”information context” and user prompt
- Three stages:
  - Training / finetuning of the three models
  - Vectorization of database
  - Retrieval, Augmentation, Generation



Federated Embedding Model Training

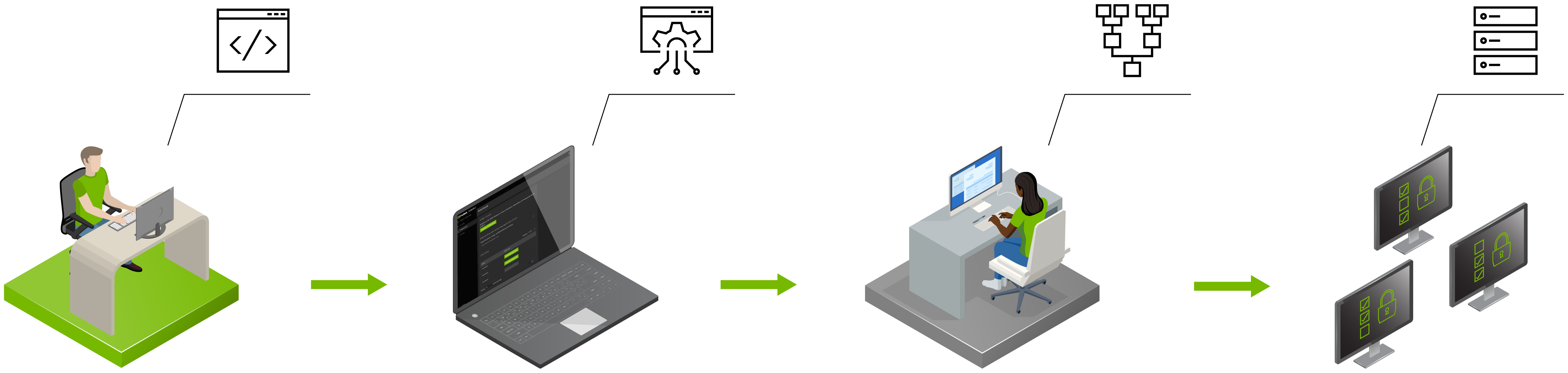
Embedding models can benefit from more diverse data

- Federated training of embedding model using ***Sentence Transformers***
  - Local training (NLI, Squad, Quora)
  - Centralized (All)
  - Federated
- Federated learning can generate results close to centralized training

TrainData	STSB_pearson_cos	STSB_spearman_euc	NLI_cos_acc	NLI_euc_acc
NLI	0.7586	0.7895	0.8033	0.8045
Squad	0.8206	0.8154	0.8051	0.8042
Quora	0.8161	0.8121	0.7891	0.7854
All	0.8497	0.8523	0.8426	0.8384
Federated	0.8444	0.8368	0.8269	0.8246

NVIDIA FLARE Workflow

From rapid research prototyping to streamlined real world deployment



Rapid Development

Streamlined Deployment

Simplified Operations

Real-World Site Security

Thank You !

Holger Roth, hroth@nvidia.com

Secure Federated XGBoost with Homomorphic Encryption

Ziyue Xu

Senior Scientist

NVIDIA Federated Learning

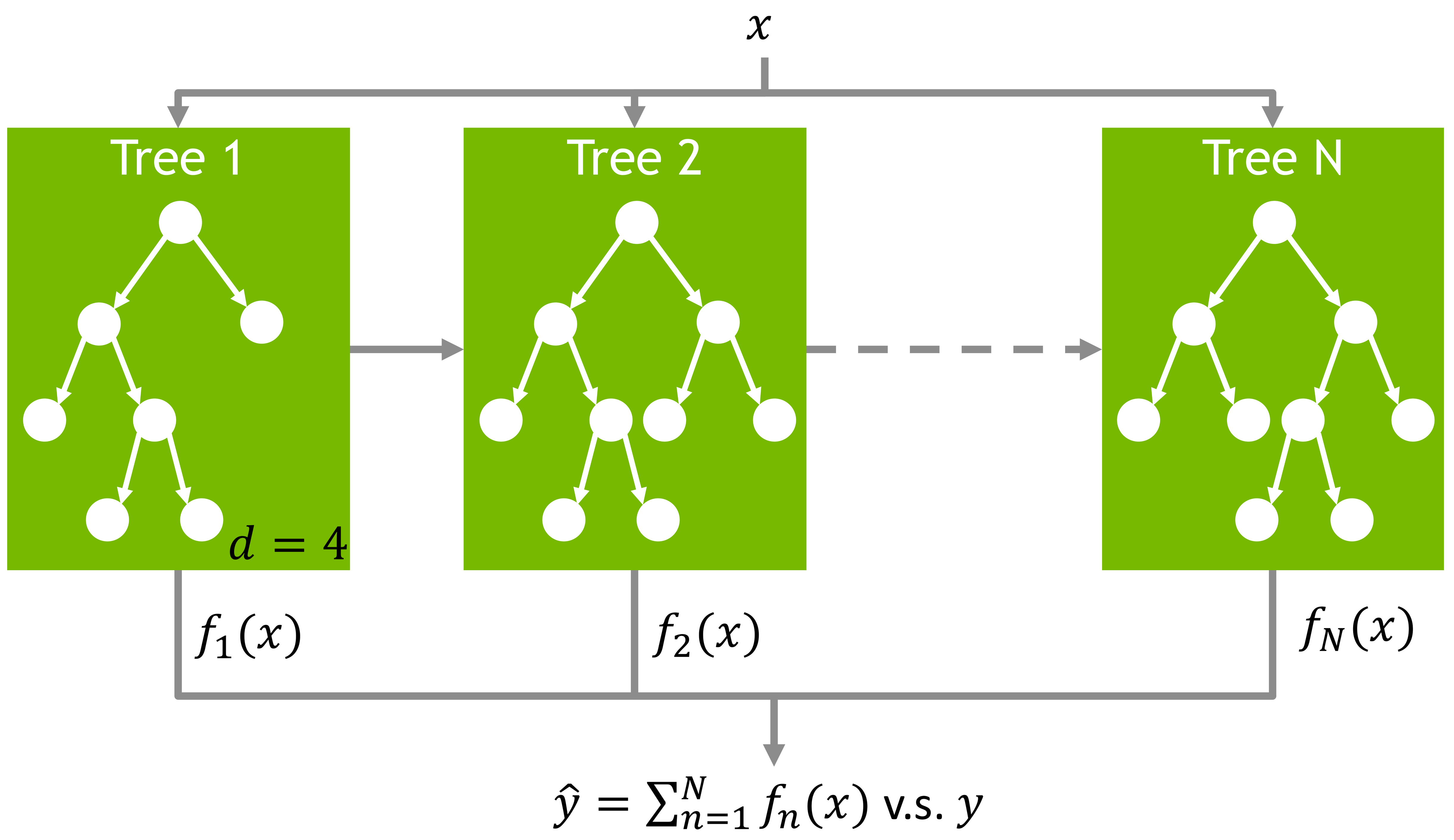
NVIDIA FLARE DAY

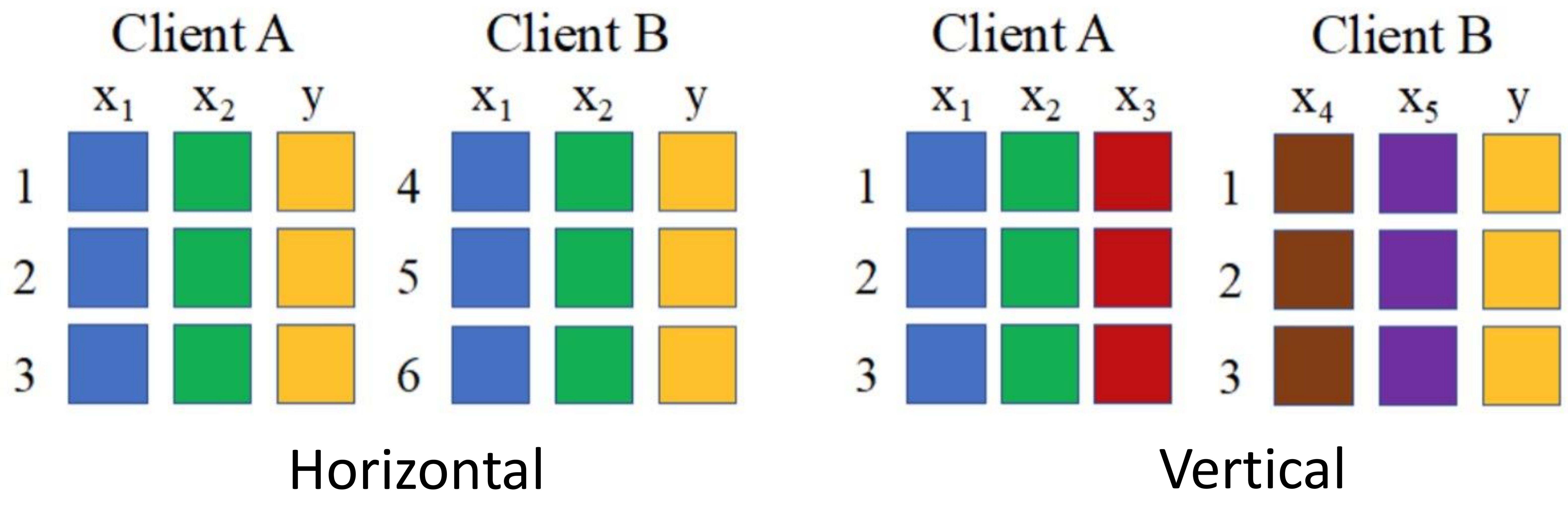
September 18, 2024

XGBoost

Basics and Federated

- Basics
  - Tree-based method, mapping a vector of feature values to its label prediction
  - Even in the age of LLM, still widely used and even SOTA for many tabular data use cases
  - Important in application domains like financial industry
  - Fully explainable, efficient, GPU accelerated with advance features from official DMLC implementation
  - Distributed schemes available, sharing and syncing intermediate results, expect almost identical accuracy
- Federated under two data split settings – following the distributed schemes
  - Horizontal – clients have access to the same features of different data samples / population
  - Vertical – clients have access to different features of the same data samples / population





XGBoost

Federated – Security Concerns and Existing Solutions

- Horizontal – same set of features, different population
  - Each client will compute **partial gradient statistics for full features** over its own data
  - Server performs aggregation to compute global statistics
  - **Security concern**: gradient statistics contains local data distribution information, exposed to server and others
- Vertical - same population, different features, one holds label information (“active party”), other do not (“passive parties”)
  - Passive party A, active party B (label owner – “y”), only active party is able to compute base gradients g&h
  - Each client will be able to compute **full gradient statistics for partial features** upon receiving g&h from active party
  - **Security concern**: the label y can be inferred from g&h, exposed to others
- Existing solutions
  - Third party:
    - Secure pipeline, addressing the potential information leakages
    - Limitation: without DMLC XGBoost support
  - Official XGBoost + NVFlare (previous version for both):
    - Full functionalities from XGBoost (GPU acceleration, etc.)
    - No support for secure features, and therefore the above concerns are not addressed
- Key contribution in this release:
  - Secure federated XGBoost by enabling homomorphic encryption (HE) in both XGBoost and NVFlare implementations
  - Data **privacy secured** with access to all advanced features from **DMLC** XGBoost

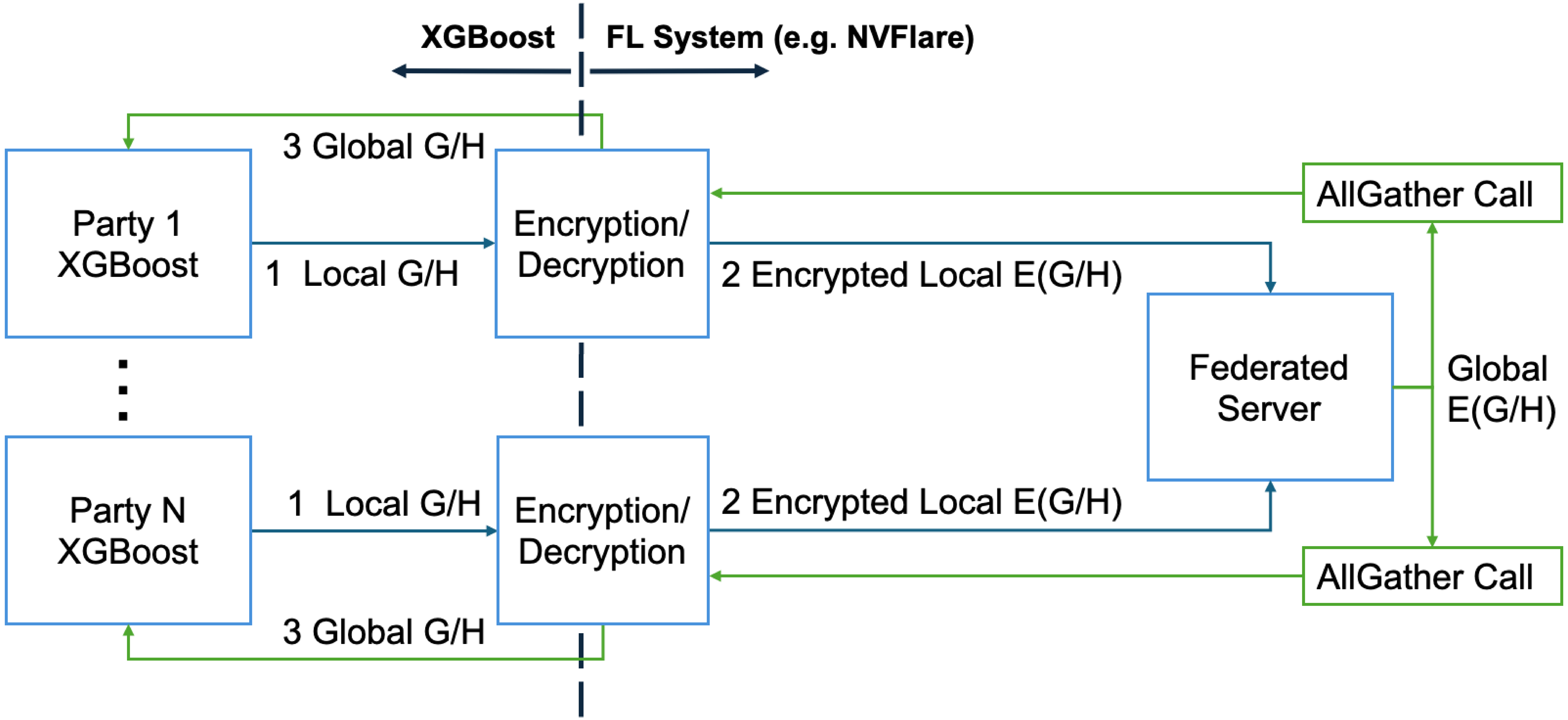
Secure Federated XGBoost

Secure Pattern and Risk Mitigation

- Horizontal

  To prevent client's histogram information leaking to server and others:

  - clients encrypt local G&H histograms (**partial stats for full feature**) with HE, and send to server
  - server adds the partial histograms to a global histogram within HE and send back to clients
  - clients decrypt and perform best split finding



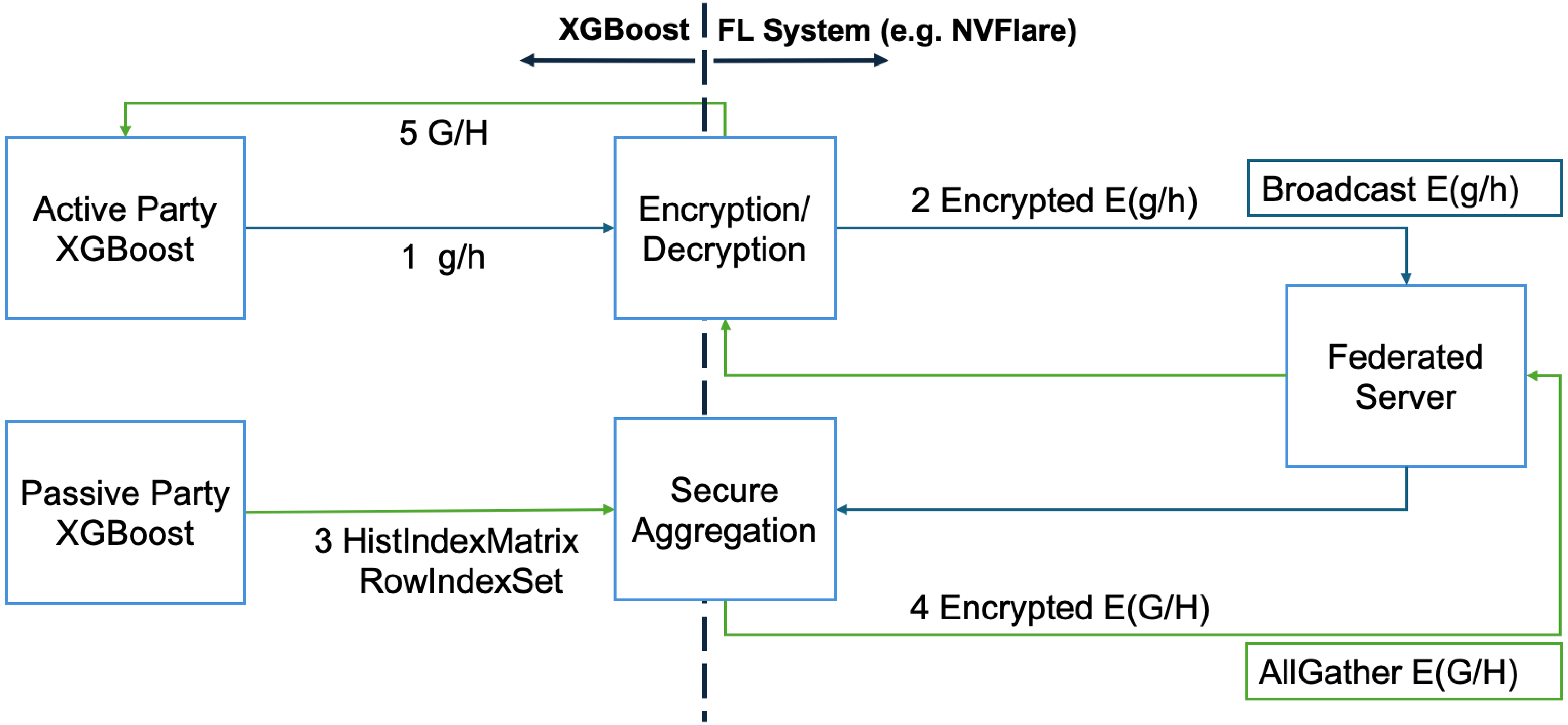
Secure Federated XGBoost

Secure Pattern and Risk Mitigation

- Vertical

  To prevent active party's label information leaking to passive parties:

  - active party computes g&h and encrypts with HE (either XGBoost-side or FL-side)
  - passive parties compute local G&H histograms (**full stats for partial feature**) within HE and send back to active party
  - active party decrypts, assemble the histograms to form a global one, and perform best split finding



NVFlare Now Features

Secure Federated XGBoost

- Information security
  - Potential key information leakage prevented by HE with strong security assurance
  - Important for application domains with high requirements over data governance
- Federated schemes for secure XGBoost:
  - Both horizontal and vertical
  - Both CPU and GPU
  - GPU acceleration on XGBoost computation enabled by new DMLC support
  - GPU acceleration on gradient encryption enabled by new plugin for performing HE
- With the secure federated XGBoost pipeline, we designed a plugin mechanism achieving flexible encryption depending on hardware environment
  - Two plugins for g&h encryption: one with IPCL library using CPU; the other with the CUDA Paillier using GPU.
  - On an experimental setting with 3 clients, each of 200k training data, GPU plugin is ~5x faster.
- Full examples covering all combinations for secure federated XGBoost
  - https://github.com/NVIDIA/NVFlare/tree/main/examples/advanced/xgboost_secure

Thank You !

Ziyue Xu, ziyuex@nvidia.com

NVIDIA FLARE and Confidential Computing

Isaac Yang

Senior Software Engineer

NVIDIA Federated Learning

NVIDIA FLARE DAY

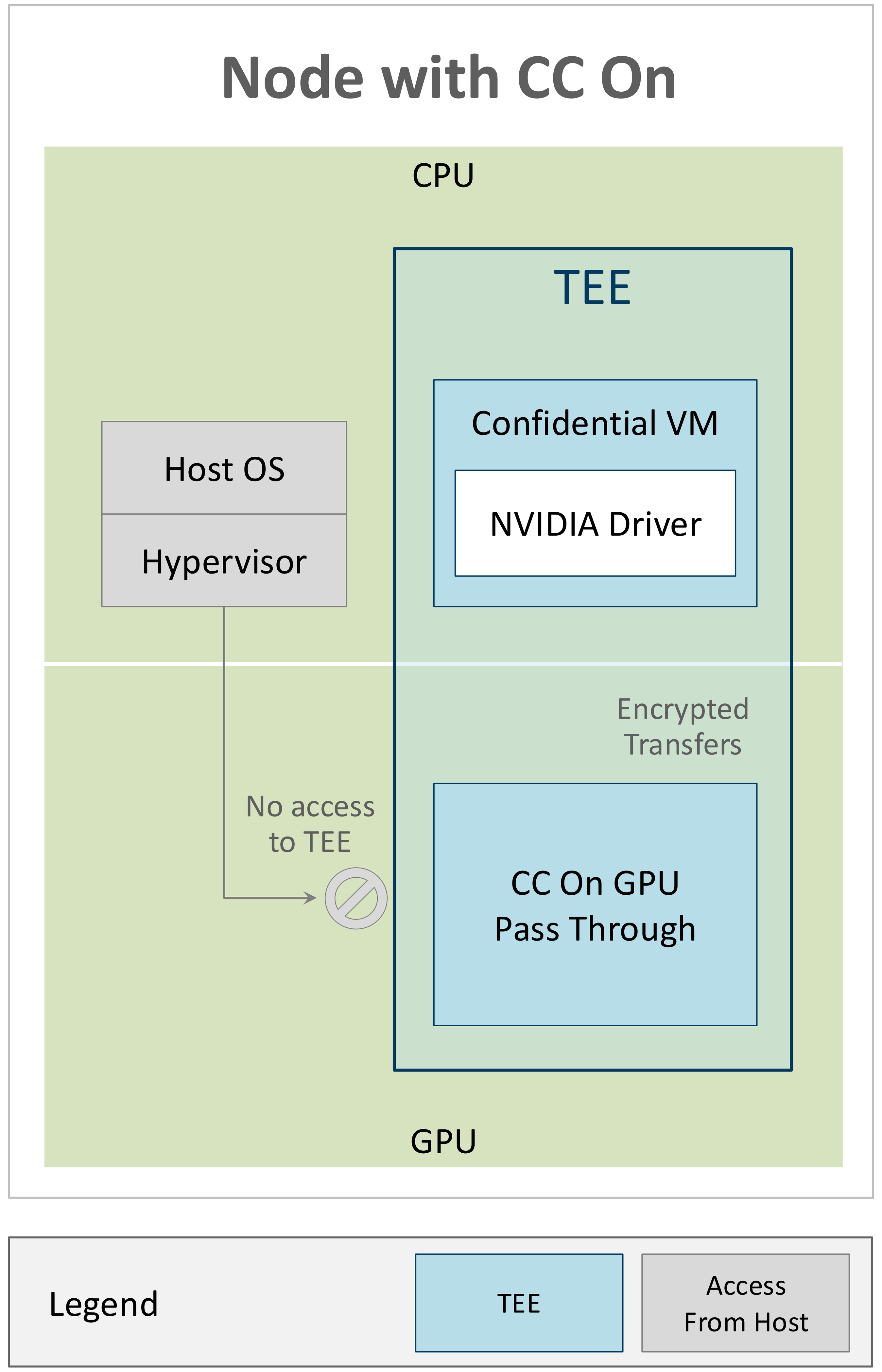
September 18, 2024

GPU Confidential Computing

Protecting Data and Code from Hypervisor and Physical Attacks

Capabilities:

- Trusted Execution Environment Isolated environment providing confidentiality & integrity
- Virtualization-based Applications can run unchanged and do not have to be partitioned
- Secure Transfers High performance HW acceleration for encrypted CPU/GPU transfers
- Hardware Root of Trust Authenticated firmware; measurement & attestation for the GPU



Confidential Computing: Use Cases

Common CC use cases across industries

Government

Security Risk

cross-agency, or cross-country multi-party collaboration

Healthcare

Patient Data Privacy

cross hospitals and corss-institutional research. Federated AI and multi-party collaboration is needed

Financial Services

Fraud Detection, AML

Multi-Party PSI, Collaborate data sharing, Federated AI training and Inference

Manufacturing

Supply Chain Analysis

Enforce Quality Control Procedures requires federated AI and multi-party collaboration

Enterprise

Multinational HR Analysis

Protect sensitive data while perform analytics. Require federated AI

Any Industry

Data Clean Room

securely share data for data cleaning, training and analytics

Federated learning Use Cases

Concerns when using federated learning

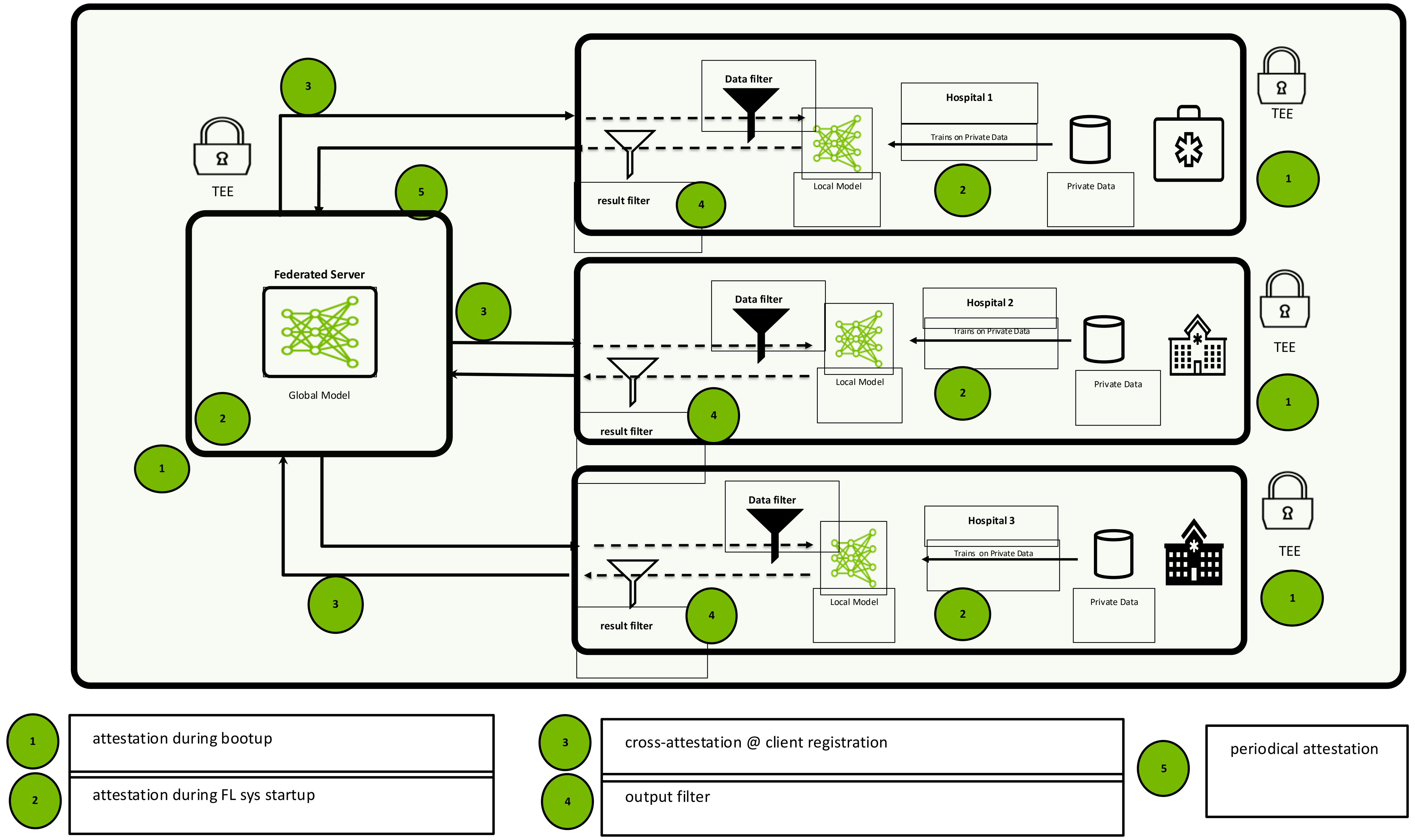
- Trust of participants
- Code tempering
- Model tampering
- Model Theft
- Model inversion attach
- Data Leak

What CC in FL can do

- Build **Explicit Trust** among participants
- **Prevent** code, model, data **tampering**
- **Secure Aggregation** at Server Node
  - Secure aggregation node
  - Aggregation code protection
- **Secure Training** at Client Node
  - Training node protection with TEE
  - Model IP protection with TEE
  - Prevent data leak
- **Federated Inference Protection**
  - Input data protection
  - Model protection

How NVIDIA FLARE Integrates with Confidential Computing

- **NVIDIA FLARE enables lift-and-shift CC features**
  - Existing application don't need to be modified to shift from non-TEE to TEE env with new hardware-based protection
- **Build Explicit Trust**
  - Attestation Service Integration
    - Different CPU/GPU attestations SDKs
    - Well-defined interfaces enabling developers to implement their own integration in the future
  - Design to verify the trust worthiness with CC attestation service
    - Self-Test at start
    - Cross-verification at client registration
    - Repeat attestation tests periodically
- **Secure Running Environment**
  - Confidential VM
    - Bare Metal CVM, CSP CVM
  - Confidential Containers (CoCo) on K8s
    - SSH lockdown
    - Require additional Trustee services features



NVIDIA FLARE with Confidential Computing in Action

provision/build → distribution → start→ submit job

Provision Stage

- **CLI: nvflare provision, Web UI: FLARE Dashboard**
  - Same command
  - output: -- startup kit with confidential computing assets (URLs for CVM, Container etc.)

Deployment Stage

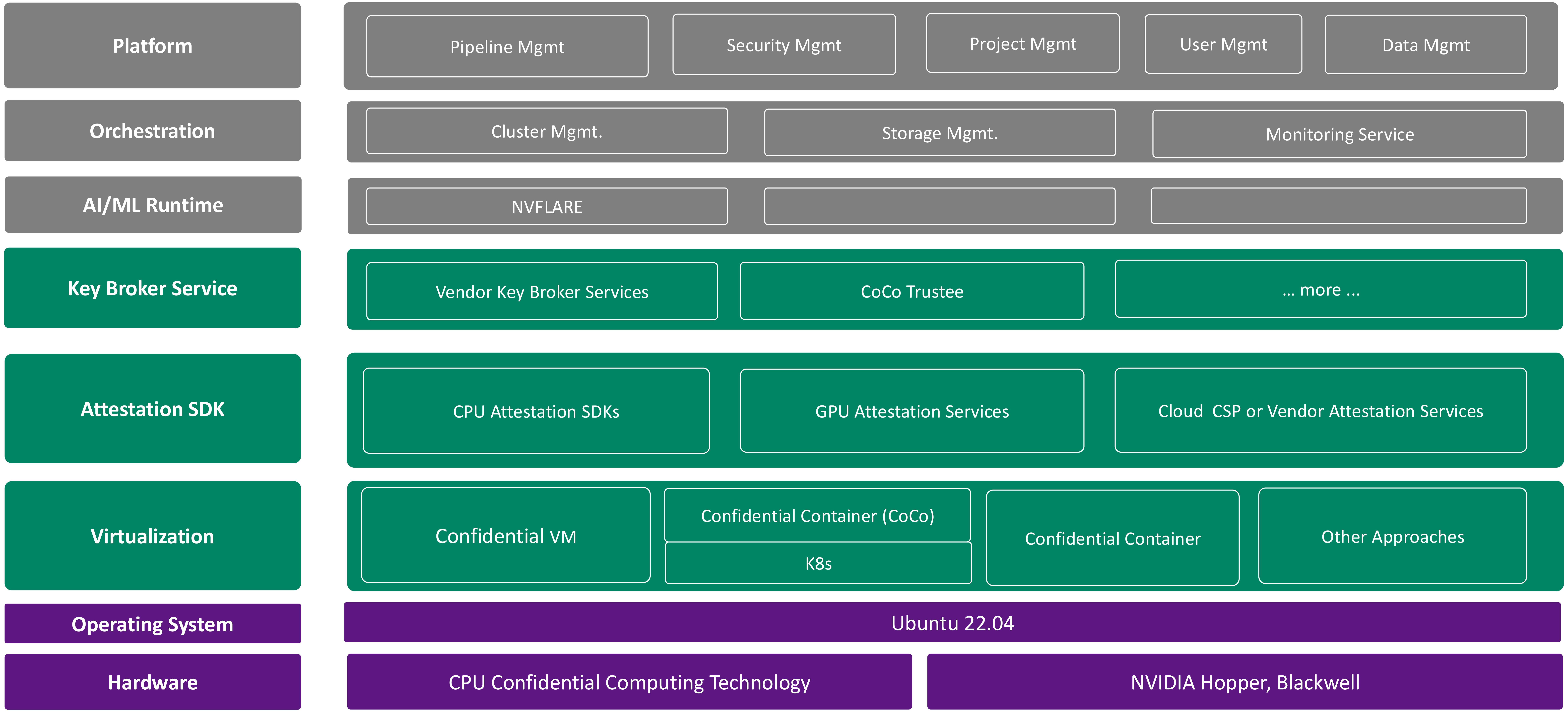
Simple command (same as existing non-CC deployment)

- **./startup/start.sh**
- Cover on-prem or in Cloud deployment

Job Submission Stage

submit_job <job folder>

Confidential Computing Tech Stack



Thank You !

Isaac Yang, isaacy@nvidia.com

Closing Remarks

Yan Cheng

Director of Engineering

Monai and Federated Learning Engineering

NVIDIA FLARE DAY

September 18, 2024

NVIDIA FLARE PRODUCT 2024-2025 Road Map

Release plan

2024 – Sept.

Release 2.5.0 (Released 9/9)

Major User Experience upgrade

Secure XGBoost

2024 – Oct.

FLARE 2.5.1

Python 3.11+ support

2025 –Q1

Release 2.6.0

Confidential FL Release

Additional LLM support