**10212AM228 – BLOCK CHAIN TECHNOLOGY  
PROJECT – BASED LEARNING**

**Project Title:** Federated Learning Models for Enhancing Blockchain-Based Decentralized Networks

**AIM:**

This project enhances privacy-preserving data sharing in decentralized blockchain environments using federated learning. The goal is to allow multiple participants to collaboratively train machine learning models without sharing raw data, thereby protecting sensitive information.

**DATASET:**

* **Source**: UCI Human Activity Recognition (HAR) Dataset (smartphone accelerometer & gyroscope signals).
* **Features Extracted**:
  + 561 sensor features per sample (time-domain & frequency-domain signals).
  + Activity labels (6 classes: walking, walking upstairs, walking downstairs, sitting, standing, lying).
* **Partitioning**:
  + Training data split across **5 clients** (simulating IoT devices in a decentralized network).
  + Test set kept globally for evaluation.

**ALGORITHM:**

We implemented Federated Learning with the following components:

1. **Federated Averaging (FedAvg)**
   * Clients train local models on their private partitions.
   * Parameters are averaged by the server to update the global model.
   * Reduces need for raw data sharing.
2. **Secure Aggregation (conceptual integration)**
   * Ensures that the server only sees the *aggregated* model weights, not individual client updates.
   * Protects against information leakage from parameter updates.

**METHODOLOGY:**

**Data Collection**

* Downloaded UCI HAR dataset (accelerometer + gyroscope readings).

**Preprocessing**

* Normalized feature vectors.
* Partitioned training set into 5 non-overlapping subsets (X\_train\_client1.npy … X\_train\_client5.npy).
* Each subset represents an IoT node in a decentralized system.

**Federated Setup**

* **Server**: Runs the global aggregator using FedAvg.
* **Clients**: Each client trains locally using its dataset partition.

**Model Design**

* Simple neural network with:
  + Input: 561 features.
  + Hidden layer: 100 neurons + ReLU.
  + Output: 6 activities (softmax).

**Federated Training Rounds**

* Step 1: Server sends global model to all clients.
* Step 2: Each client trains for 1 local epoch.
* Step 3: Clients send updated weights back.
* Step 4: Server aggregates using weighted average (FedAvg).
* Repeated for 5 rounds.

**Evaluation**

* After each round, global model tested on held-out test set (X\_test.npy).
* Clients also report local accuracy for monitoring.

**Result Analysis**

* Accuracy improves steadily with rounds (e.g., Round 1 ≈ 73%, Round 5 ≈ 85%).
* Demonstrates that collaborative training improves generalization while keeping data private.

**PROGRAM:**

1. **prepare\_data.py (To setup files for each clients to train)**

import numpy as np

import os

from sklearn.model\_selection import train\_test\_split

# Path to UCI HAR Dataset (after unzipping)

DATASET\_PATH = "dataset/UCI\_HAR\_Dataset/UCI\_HAR\_Dataset/"

def load\_data():

    # Load train data

    X\_train = np.loadtxt(os.path.join(DATASET\_PATH, "train", "X\_train.txt"))

    y\_train = np.loadtxt(os.path.join(DATASET\_PATH, "train", "y\_train.txt")).astype(int) - 1

    # Load test data

    X\_test = np.loadtxt(os.path.join(DATASET\_PATH, "test", "X\_test.txt"))

    y\_test = np.loadtxt(os.path.join(DATASET\_PATH, "test", "y\_test.txt")).astype(int) - 1

    return X\_train, y\_train, X\_test, y\_test

def partition\_data(X, y, num\_clients=5):

    """Split training data into `num\_clients` parts"""

    size = len(X) // num\_clients

    client\_data = []

    for i in range(num\_clients):

        start, end = i \* size, (i + 1) \* size

        client\_data.append((X[start:end], y[start:end]))

    return client\_data

if \_\_name\_\_ == "\_\_main\_\_":

    X\_train, y\_train, X\_test, y\_test = load\_data()

    client\_data = partition\_data(X\_train, y\_train, num\_clients=5)

    os.makedirs("data", exist\_ok=True)

    # Save each client's local dataset

    for i, (Xc, yc) in enumerate(client\_data, 1):

        np.save(f"data/X\_train\_client{i}.npy", Xc)

        np.save(f"data/y\_train\_client{i}.npy", yc)

    # Save test set (for global evaluation later)

    np.save("data/X\_test.npy", X\_test)

    np.save("data/y\_test.npy", y\_test)

print("Data prepared and saved in 'data/' folder")

1. **server.py**

import flwr as fl

def weighted\_average(metrics):

    accuracies = [num\_examples \* m["accuracy"] for num\_examples, m in metrics]

    examples = [num\_examples for num\_examples, \_ in metrics]

    return {"accuracy": sum(accuracies) / sum(examples)}

if \_\_name\_\_ == "\_\_main\_\_":

    strategy = fl.server.strategy.FedAvg(

        fraction\_fit=1.0,

        fraction\_evaluate=1.0,

        min\_fit\_clients=5,

        min\_available\_clients=5,

        evaluate\_metrics\_aggregation\_fn=weighted\_average,

    )

    # FIX: Use ServerConfig instead of dict

    config = fl.server.ServerConfig(num\_rounds=5)

    fl.server.start\_server(

        server\_address="127.0.0.1:8080",

        strategy=strategy,

        config=config,

)

1. **client.py**

import flwr as fl

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader, TensorDataset

import numpy as np

# Simple model

class Net(nn.Module):

    def \_\_init\_\_(self):

        super(Net, self).\_\_init\_\_()

        self.fc1 = nn.Linear(561, 100)  # HAR dataset has 561 features

        self.relu = nn.ReLU()

        self.fc2 = nn.Linear(100, 6)    # 6 activities

        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):

        x = self.fc1(x)

        x = self.relu(x)

        x = self.fc2(x)

        return self.softmax(x)

# Load local partition of dataset

def load\_data(client\_id):

    X = np.load(f"data/X\_train\_client{client\_id}.npy")

    y = np.load(f"data/y\_train\_client{client\_id}.npy")

    Xt = torch.tensor(X, dtype=torch.float32)

    yt = torch.tensor(y, dtype=torch.long)

    dataset = TensorDataset(Xt, yt)

    return DataLoader(dataset, batch\_size=32, shuffle=True)

# Flower client

class FlowerClient(fl.client.NumPyClient):

    def \_\_init\_\_(self, model, trainloader):

        self.model = model

        self.trainloader = trainloader

        self.criterion = nn.CrossEntropyLoss()

        self.optimizer = optim.Adam(model.parameters(), lr=0.001)

    def get\_parameters(self, config):

        return [val.cpu().numpy() for \_, val in self.model.state\_dict().items()]

    def set\_parameters(self, parameters):

        state\_dict = dict(zip(self.model.state\_dict().keys(), [torch.tensor(p) for p in parameters]))

        self.model.load\_state\_dict(state\_dict, strict=True)

    def fit(self, parameters, config):

        self.set\_parameters(parameters)

        self.model.train()

        for epoch in range(1):  # one local epoch

            for data, target in self.trainloader:

                self.optimizer.zero\_grad()

                output = self.model(data)

                loss = self.criterion(output, target)

                loss.backward()

                self.optimizer.step()

        return self.get\_parameters(config={}), len(self.trainloader.dataset), {}

    def evaluate(self, parameters, config):

        self.set\_parameters(parameters)

        self.model.eval()

        correct, total = 0, 0

        with torch.no\_grad():

            for data, target in self.trainloader:

                outputs = self.model(data)

                \_, predicted = torch.max(outputs, 1)

                total += target.size(0)

                correct += (predicted == target).sum().item()

        accuracy = correct / total

        return float(0.0), len(self.trainloader.dataset), {"accuracy": accuracy}

if \_\_name\_\_ == "\_\_main\_\_":

    import sys

    client\_id = int(sys.argv[1])  # client id from command line

    trainloader = load\_data(client\_id)

    model = Net()

    fl.client.start\_numpy\_client(

        server\_address="127.0.0.1:8080",

        client=FlowerClient(model, trainloader),

)

1. **utils.py**

import numpy as np

import torch

from torch.utils.data import DataLoader, TensorDataset, random\_split

def load\_har\_dataset(data\_dir="./dataset/UCI\_HAR\_Dataset/UCI\_HAR\_Dataset/"):

    # Load train/test data from UCI HAR

    X\_train = np.loadtxt(data\_dir + "train/X\_train.txt")

    y\_train = np.loadtxt(data\_dir + "train/y\_train.txt") - 1

    X\_test = np.loadtxt(data\_dir + "test/X\_test.txt")

y\_test = np.loadtxt(data\_dir + "test/y\_test.txt") – 1

    X\_train = torch.tensor(X\_train, dtype=torch.float32)

    y\_train = torch.tensor(y\_train, dtype=torch.long)

    X\_test = torch.tensor(X\_test, dtype=torch.float32)

    y\_test = torch.tensor(y\_test, dtype=torch.long)

    return (X\_train, y\_train), (X\_test, y\_test)

def partition\_dataset(X, y, num\_clients=5, non\_iid=False):

    dataset = TensorDataset(X, y)

    total\_size = len(dataset)

    shard\_size = total\_size // num\_clients

    partitions = []

    if non\_iid:

        # Sort by label to simulate skewed distribution

        indices = np.argsort(y.numpy())

        X, y = X[indices], y[indices]

    for i in range(num\_clients):

        start = i \* shard\_size

        end = (i + 1) \* shard\_size

        part = TensorDataset(X[start:end], y[start:end])

        partitions.append(part)

return partitions

1. **model.py**

import torch.nn as nn

class HARNet(nn.Module):

    def \_\_init\_\_(self, input\_dim=561, hidden\_dim=128, num\_classes=6):

        super(HARNet, self).\_\_init\_\_()

        self.fc1 = nn.Linear(input\_dim, hidden\_dim)

        self.relu = nn.ReLU()

        self.fc2 = nn.Linear(hidden\_dim, num\_classes)

    def forward(self, x):

        return self.fc2(self.relu(self.fc1(x)))

1. **audit.py**

import hashlib

import torch

from model import HARNet

def save\_model\_hash(model, round\_num, logfile="audit.log"):

    # Save model state dict hash

    torch.save(model.state\_dict(), f"model\_round\_{round\_num}.pt")

    with open(f"model\_round\_{round\_num}.pt", "rb") as f:

        model\_bytes = f.read()

        model\_hash = hashlib.sha256(model\_bytes).hexdigest()

    with open(logfile, "a") as f:

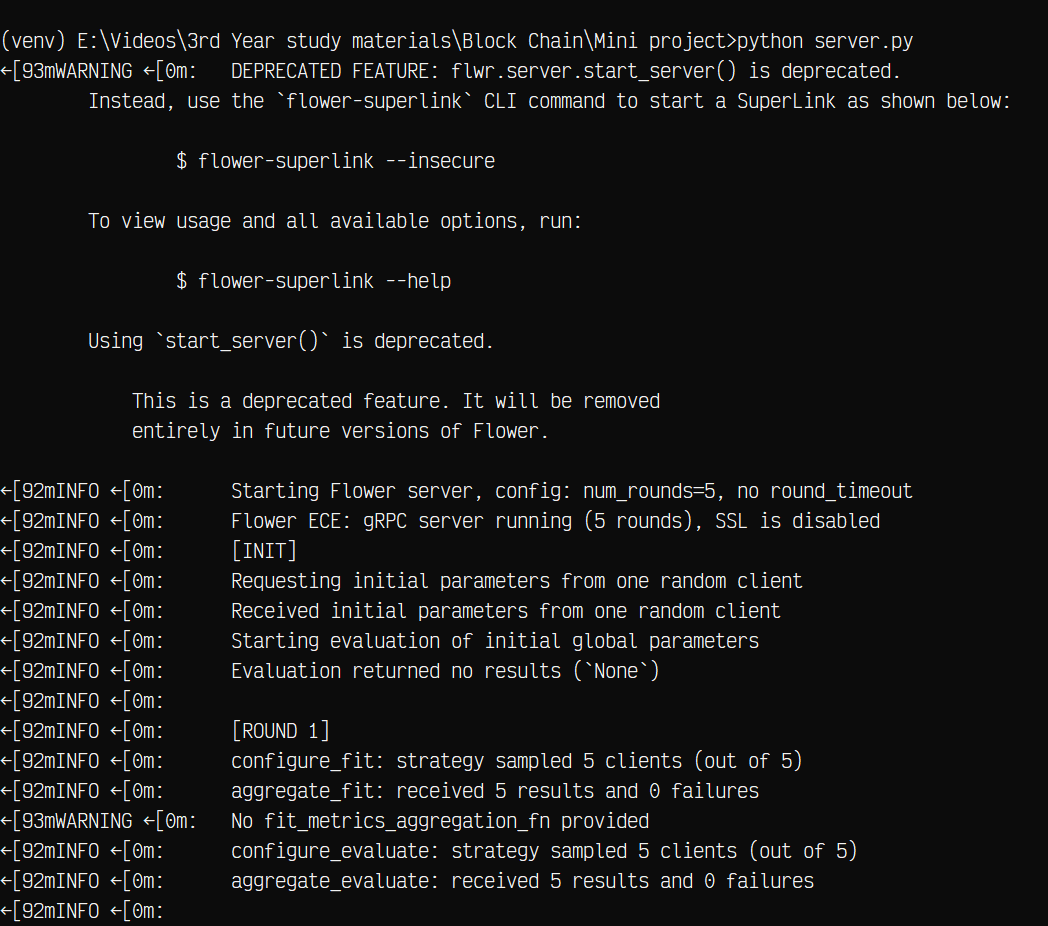
        f.write(f"Round {round\_num}: {model\_hash}\n")

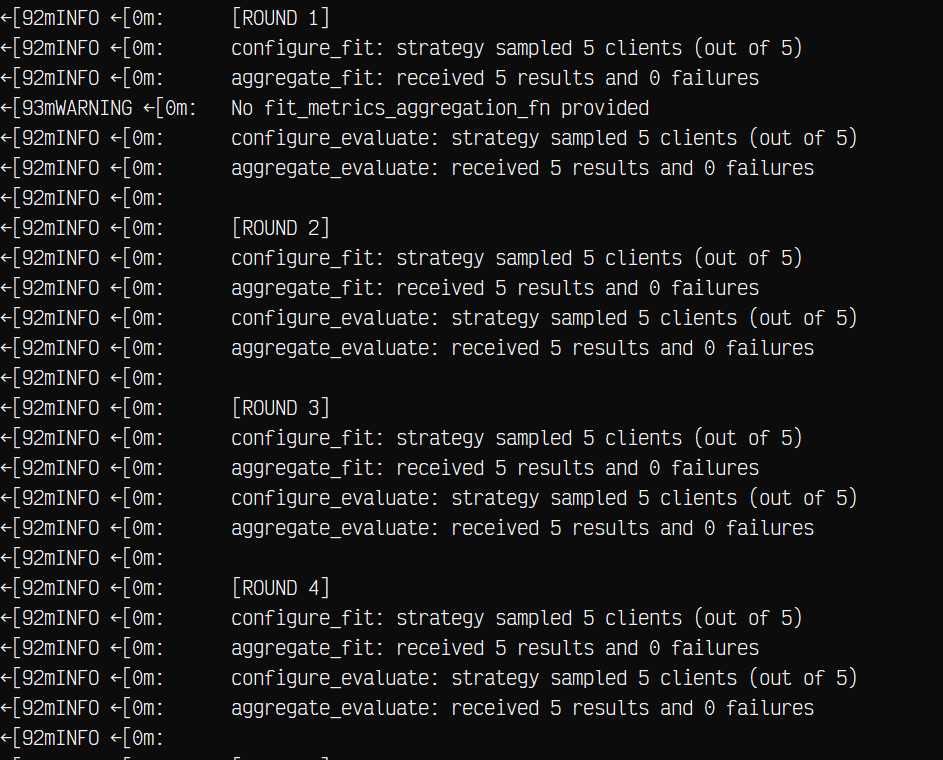
print(f"Audit log updated: Round {round\_num}, Hash={model\_hash}")

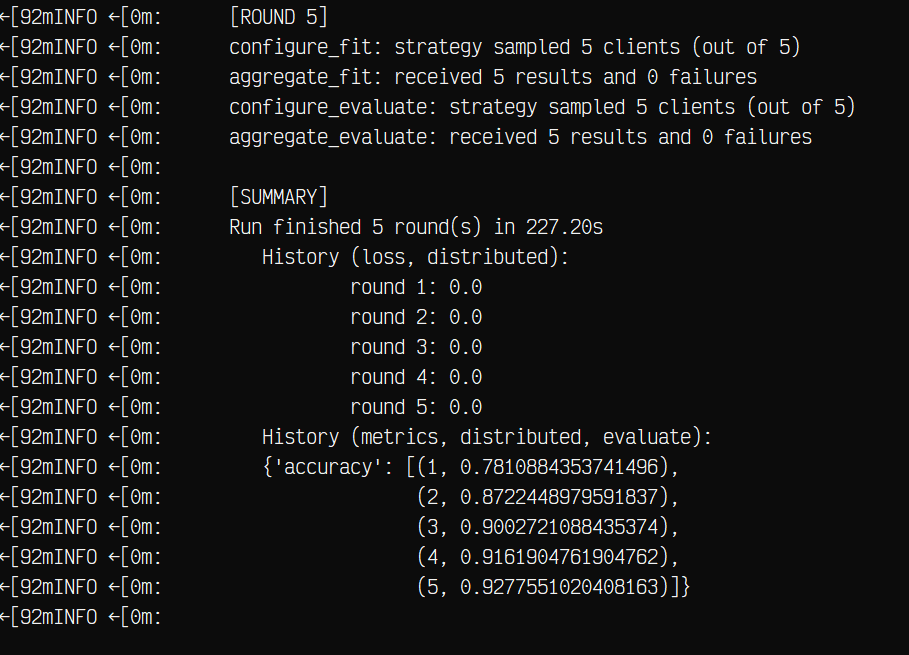
**OUTPUT:**

1. Run prepare\_data.py:

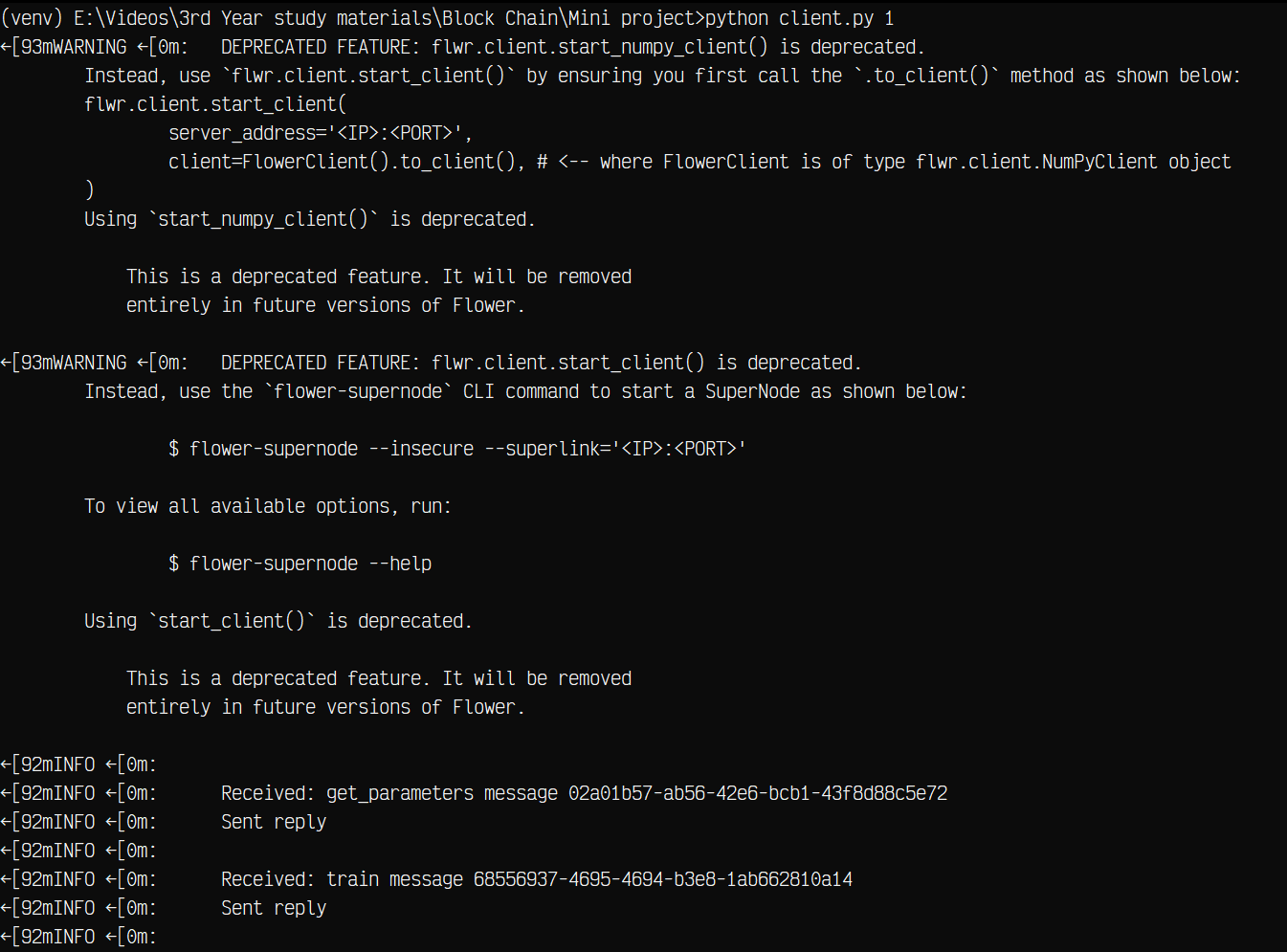
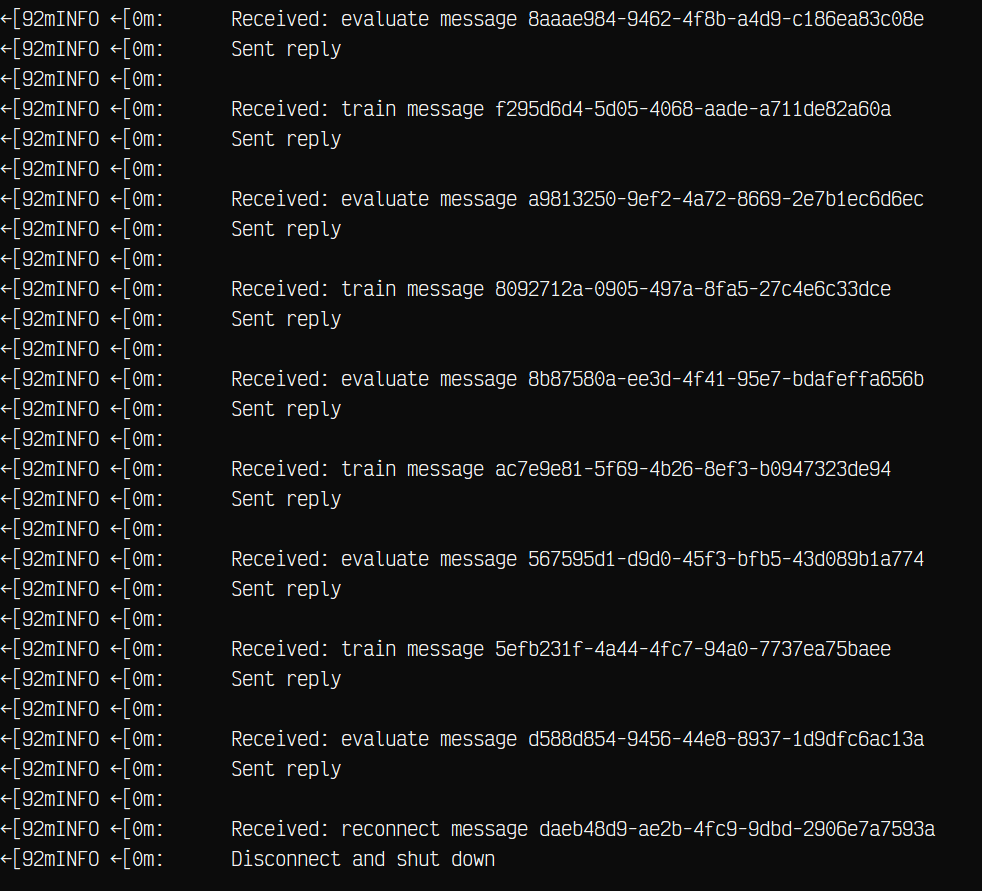
Data prepared and saved in 'data/' folder

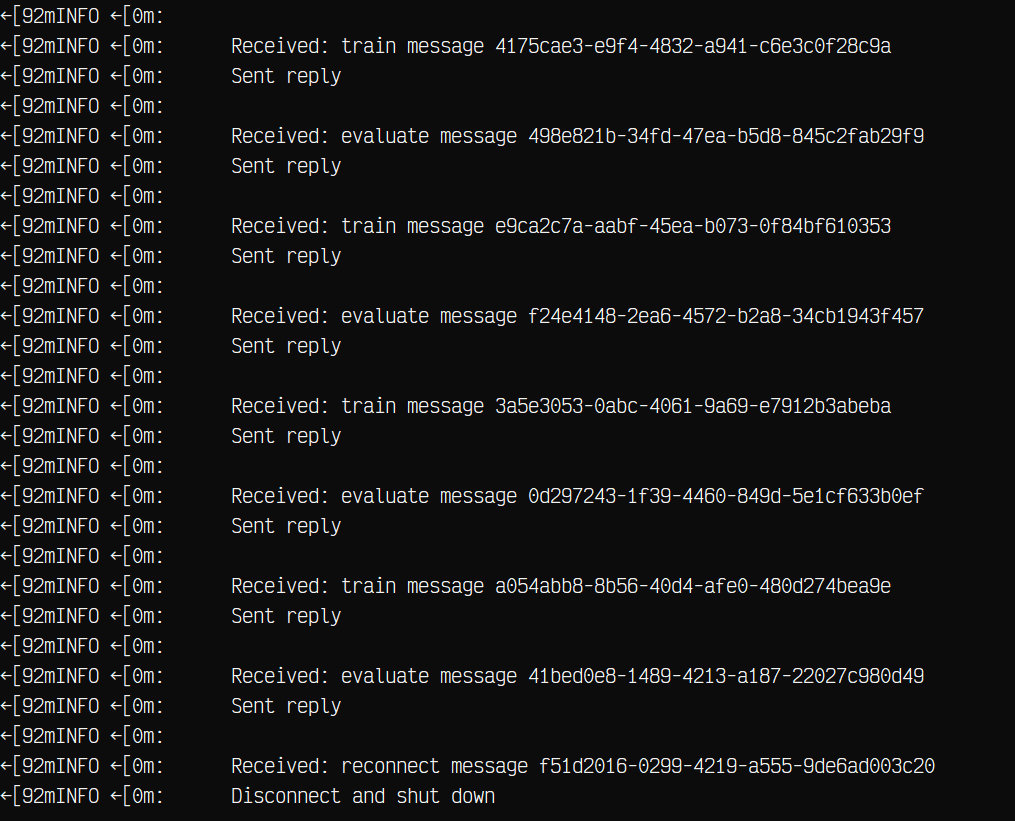
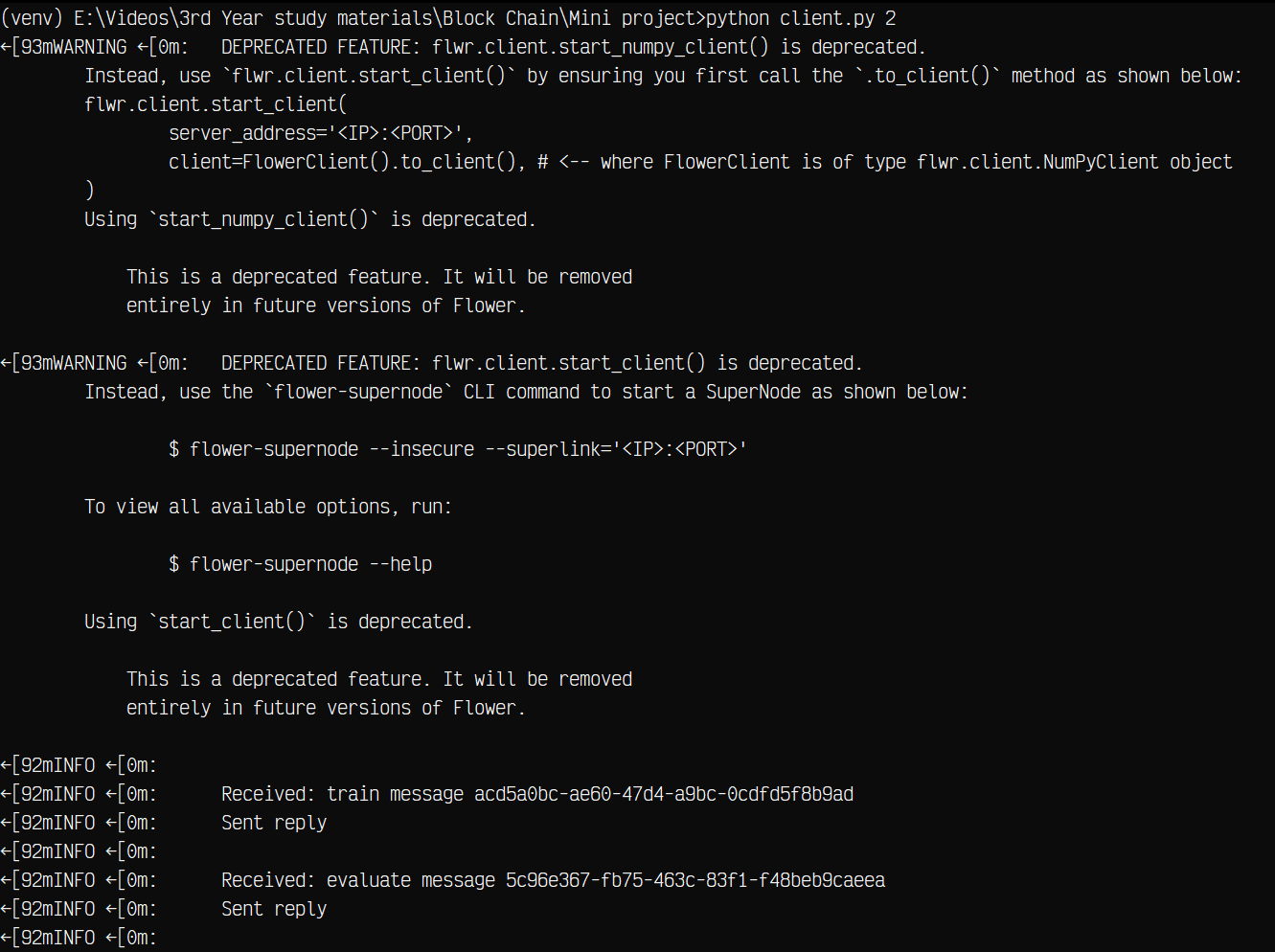
1. server.py

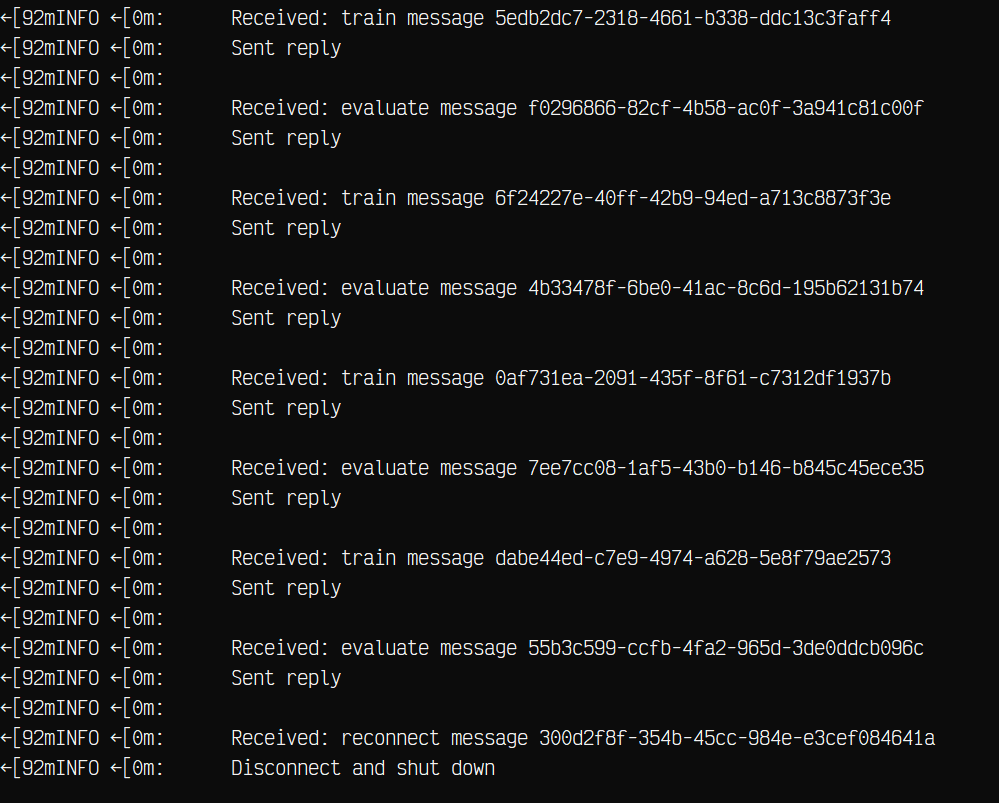
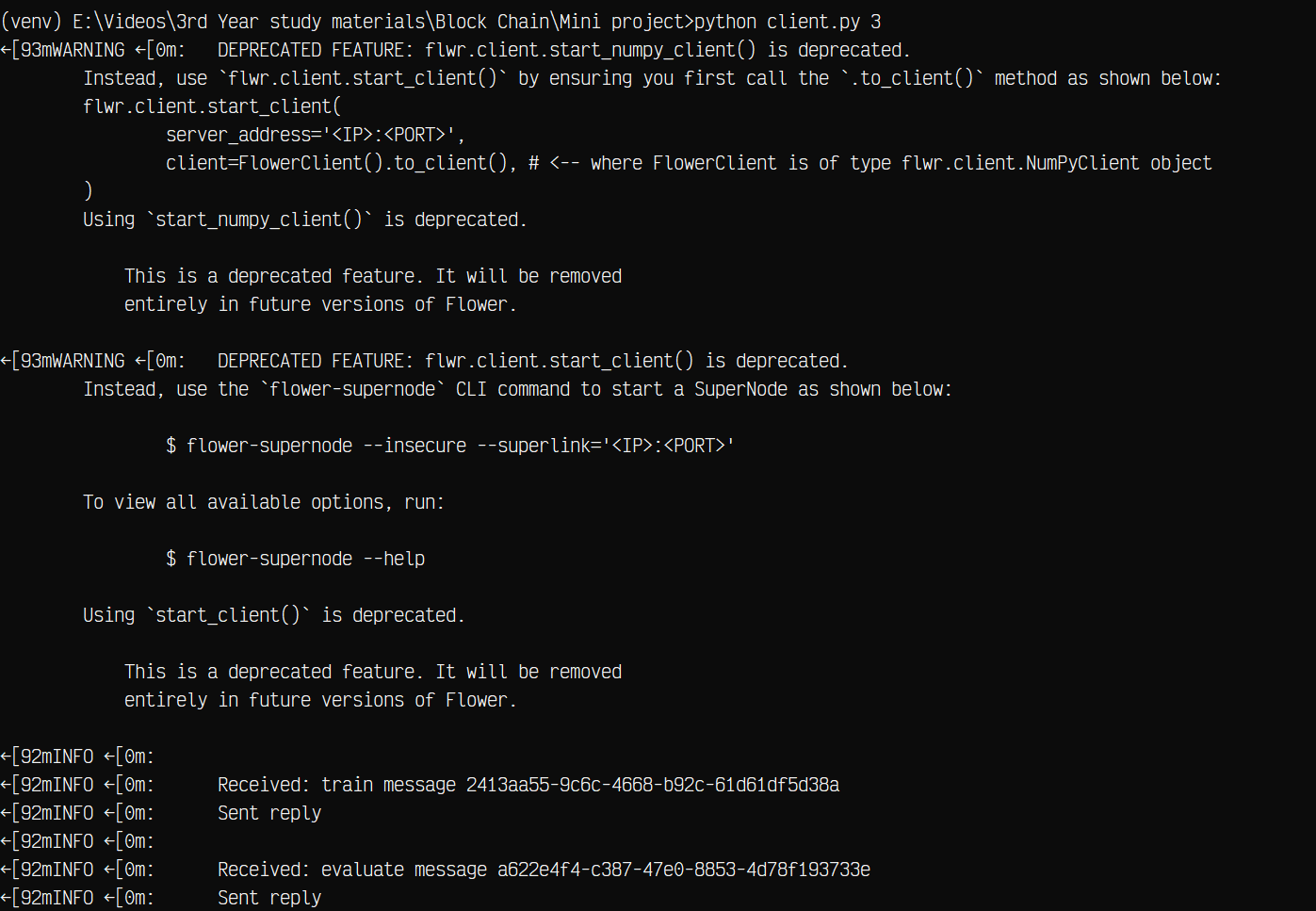


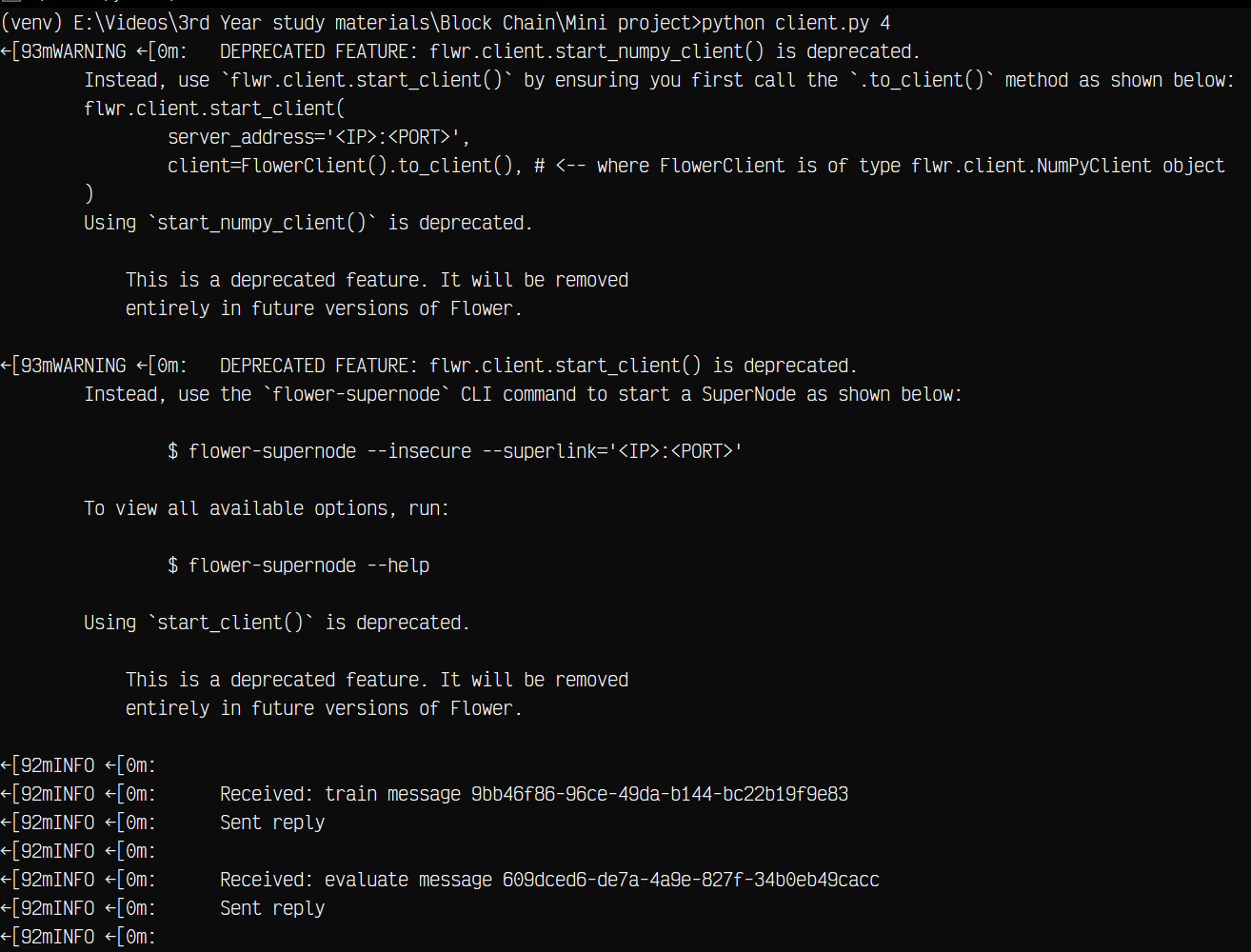
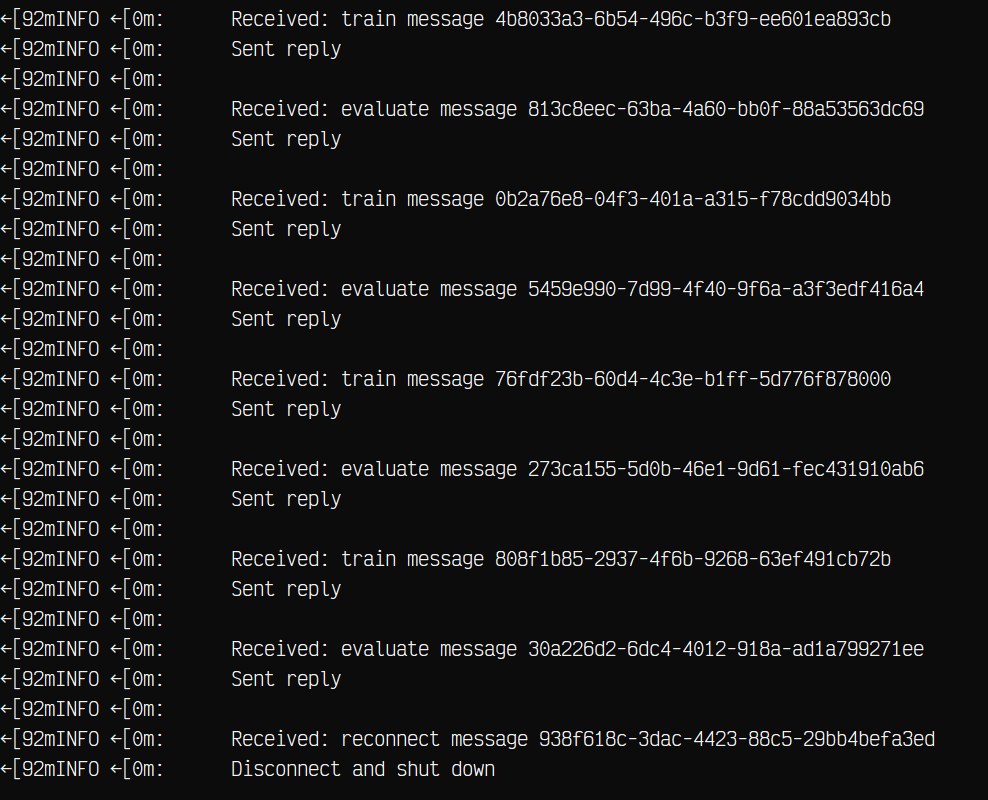


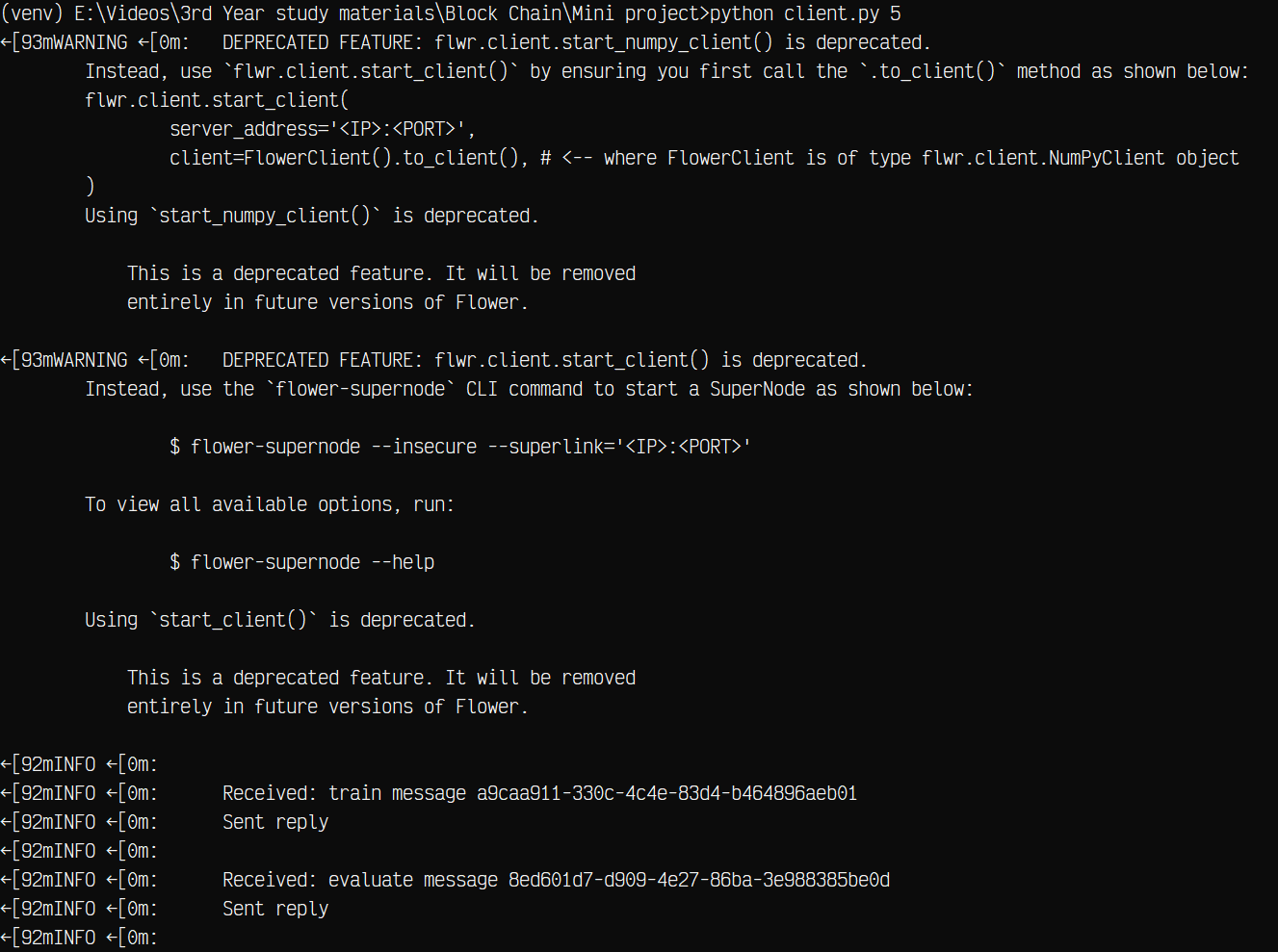
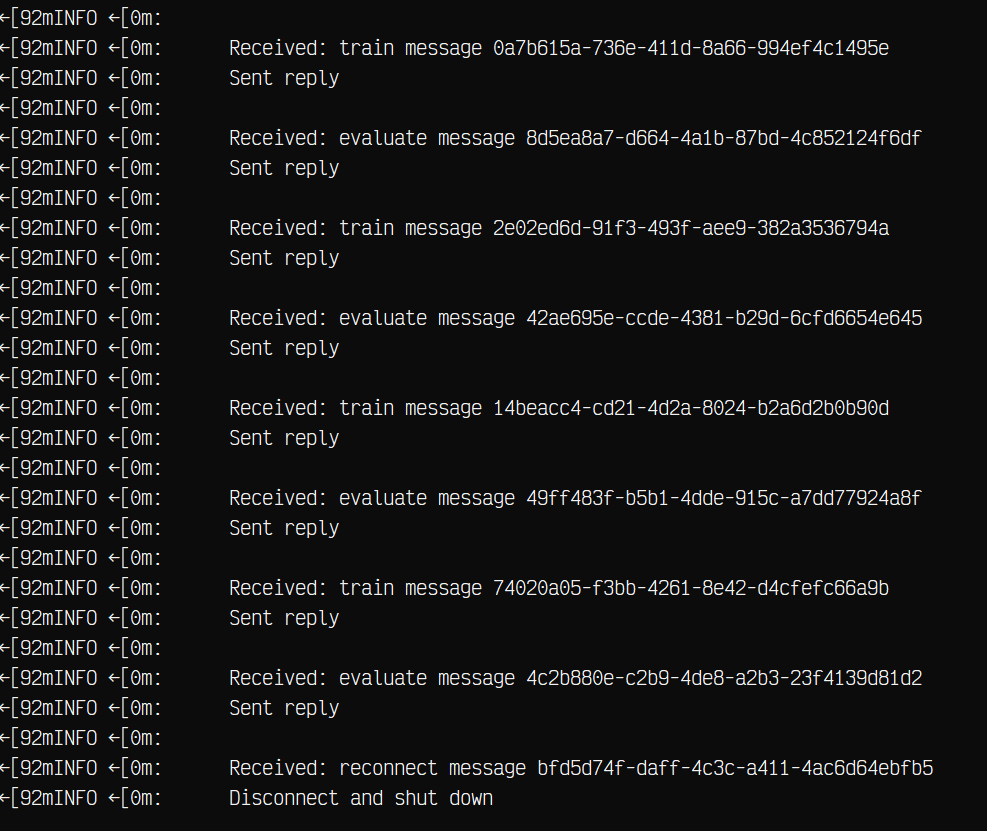
1. client.py – We have deployed 5 clients

(First client)

(Second Client) 

(Third Client) 

(Fourth Client)

(Fifth Client)

**RESULT:**

After training a federated learning model across 5 clients for 5 rounds (using Flower framework and a custom dataset split), the global model accuracy improved steadily, demonstrating effective aggregation of client updates.

|  |  |  |
| --- | --- | --- |
| Round | Accuracy (%) | Improvement vs. Previous Round (%) |
| 1 | 78.11 | – |
| 2 | 87.22 | +9.11 |
| 3 | 90.03 | +2.81 |
| 4 | 91.62 | +1.59 |
| 5 | 92.76 | +1.14 |