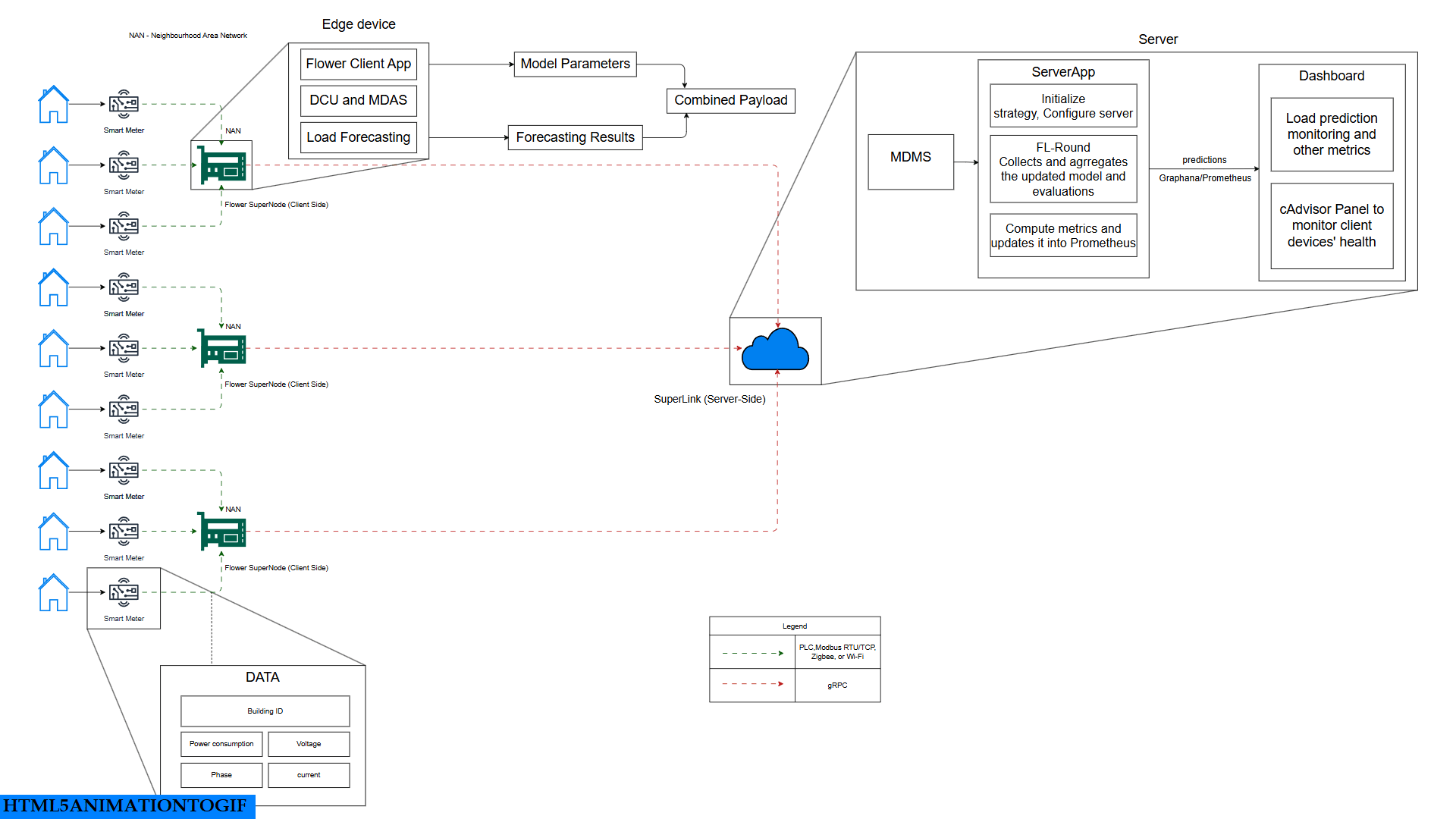
Microservice Based Architecture Design for Smart Energy Meter System Using Flower Framework

Github repo link for all the codes  
<https://github.com/bajoriya-vaibhav/SmartEnergyFL>



**What is Federated Learning?**

​Federated Learning (FL) is a machine learning paradigm in which the model is training locally at the client side and then the updated weights are then sent to the server where they get aggregated to make the final global model which solves data privacy and security issues.

Basically, FL is a decentralized approach to machine learning that enables multiple clients to collaboratively train a shared global model without exchanging their local datasets. This paradigm enhances data privacy and security by ensuring that sensitive information remains on local devices, with only model updates being communicated.

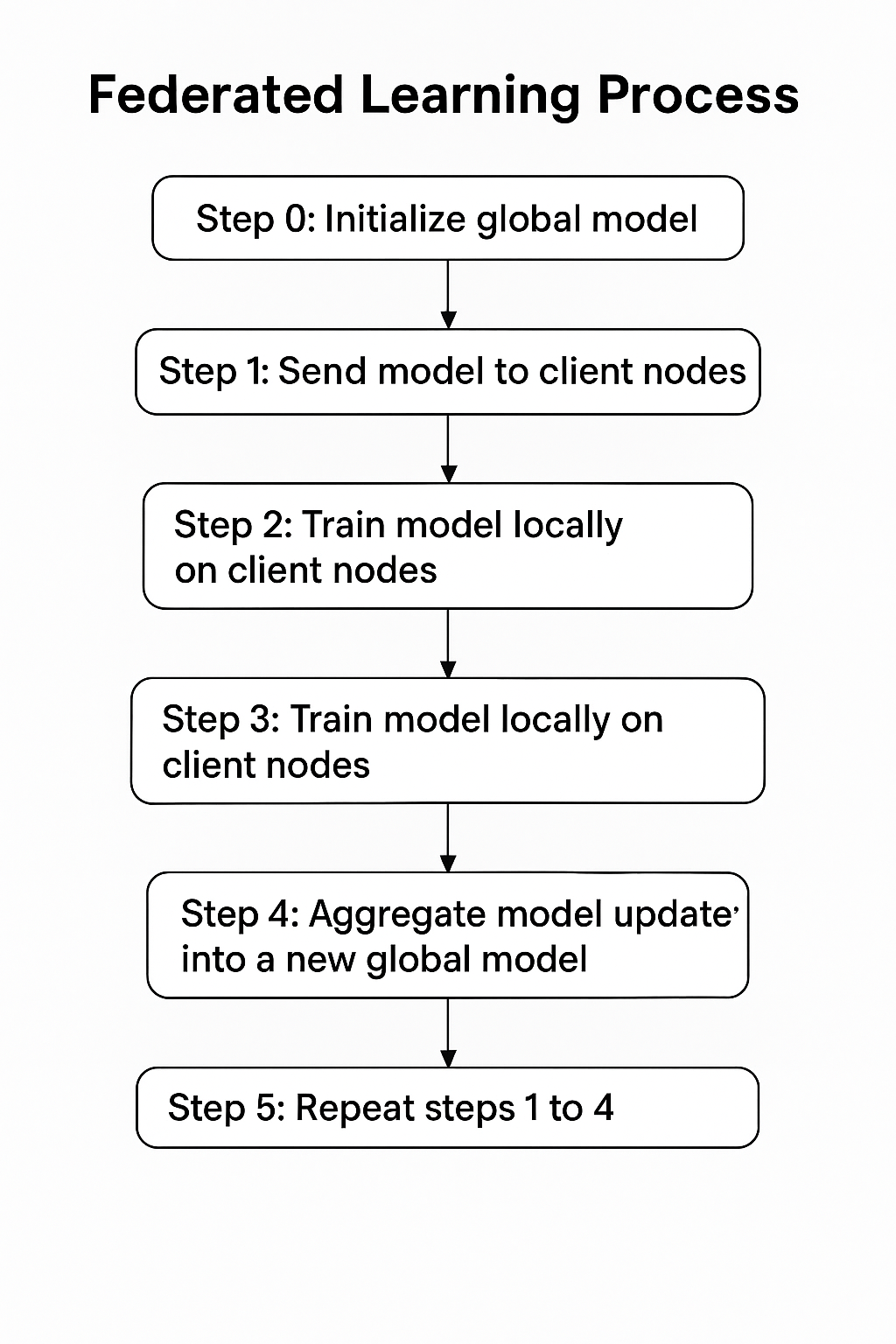
**How it is Different from the Centralized Machine Learning Approach?**

In traditional centralized machine learning, data from various sources is aggregated into a central server where the model training occurs. This approach poses significant privacy concerns. Federated Learning addresses these concerns by inverting the process: instead of moving data to the computation, it moves computation to the data. This means that model training happens locally on each client device, and only the model parameters or updates are sent back to a central server for aggregation.

**How can we implement Federated Learning?**

The FL process typically involves the following steps:

​



**Step 0**(Initialize global model): initialise the global model parameters mostly after training the global model on some previous data or training data.

**Step 1**(Send model to a number of client nodes): Now, each client devices have the global model parameters ensuring that every node is in sync. Note:-We often use only a few of the connected nodes instead of all nodes. The reason for this is that selecting more and more client nodes has diminishing returns.

**Step 2(**Train model locally on the data of each client node): Then, each client will start training the model(latest one) on the local training dataset each node has collected.

**Note**:- We don’t train the model until full convergence(overfitting may take place), but they only train for a little while. This could be as little as one epoch on the local data, or even just a few steps (mini-batches).

**Step 3**: Return model updates back to the server: After local training, each client node sends its updated weights or gradients to the server.

**Step 4**: Aggregate model updates into a new global model The server receives model updates from the selected client nodes. Then take the weighted average of the model updated by the number of samples each client used for training, this porcess is called as federated average(FedAvg).

**Step 5**: Repeat steps 1 to 4 until the model converges.

So, the process is like this (for my reference) Steps 1 to 4 are what we call a single round of federated learning - FL round. The global model parameters get sent to the participating client nodes (step 1), the client nodes train on their local data (step 2), they send their updated models to the server (step 3), and the server then aggregates the model updates to get a new version of the global model (step 4).Repeat till the model converges or give accurate results.

As we understood the workflow, now i will explain some important terminology that will be helpful when i will explain the flower framework and for code understanding.

**Terminology Alert!!🙂**

**Federated Evaluation:**

Just like we can train a model on the decentralized data of different client nodes, we can also evaluate the model on that data to receive valuable metrics.

**Why it matters**? Instead of bringing client data to a central server for evaluation (which may violate privacy), the evaluation is performed locally on each client. The results (like accuracy, precision, etc.) are aggregated, not the data itself.

**Federated Analytics:**

This involves running aggregate data queries across decentralized clients to gain insights without training a machine learning model.  
For example, we can take the example where we don’t have that much data to get clear predictions like “What’s the average blood pressure reading across all participants?”  
Here we are not getting enough data because of Privacy concerns, but for that we can get the aggregated results not individuals data hence not breaching anyones privacy.  
Therefore Federated learning provides approach like **secure aggregation** which ensure that no single client's data is visible—only the aggregate result.

In our Smart Energy Meter reading predictions also we cannot pass the individual data, therefore we have passed the aggregated predicted meter readings over the clients of a certain area so we can easily predict the consumption over that area without any breach in privacy.

**Differential Privacy:**

In this approach, we ensures individual-level data privacy by adding noise to results or updates before sharing.

This approach is used for flower internally while sending model updates or analytics from client to server to make sure individual client contributions can’t be reverse-engineered**.**

Now, we discuss about the framework we are using for our project to implement the Federated learning model for the Smart Meter System.

**What is Flower?**

Flower is an open-source framework designed to simplify the development of federated learning (FL), federated evaluation, and federated analytics systems.

It helps to orchestrate communication between a centralized server and distributed client nodes where multiple real-time data streams/datasets exist across different organizations, and a model is trained over these organizations using federated learning and sending back the updated model to server. It also allows easy integration with machine learning workflows and tools.

**Flower Architecture**

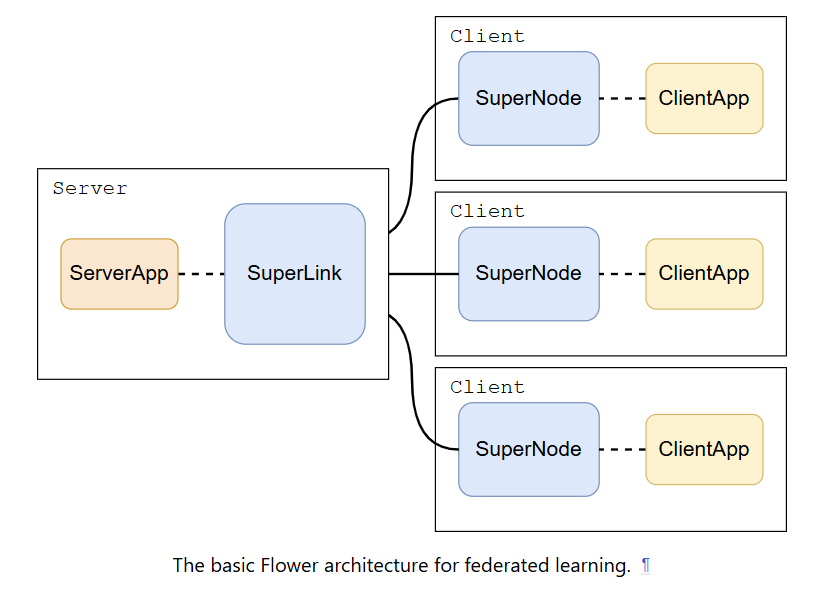
****

Figure reference link [Flower architecture design](https://flower.ai/docs/framework/explanation-flower-architecture.html)

**Flower Server design:**

The server is made up of two key parts: **SuperLink** and **ServerApp**. SuperLink is a long-running background service that acts like a central hub — it stays active all the time, sending out tasks to different devices (clients) and collecting their responses. We can think of it like a dispatcher or coordinator that keeps the communication flowing smoothly between the server and all the clients. Also as per the recent version of the Flower framework the server app is deprecated, therefore we have to use the superlink based architecture to run the application. Also it on its own it make the SSL/TLS grpc connection which we have to make by ourself earlier.

ServerApp is the main python script that all the federated learning operations like which clients to involve, how to configure them, and how to combine their results after training.

**Flower Client design:**

Each Flower client also has two parts: **SuperNode** and **ClientApp**. The SuperNode is like a helper that always stays on, waiting for instructions from the SuperLink. When a task arrives (like training a model on local data), it picks it up, runs it, and sends back the results. It’s like the persistent worker that makes sure the client is always ready.

ClientApp is the part that runs all the client-side federated learning operations like training or evaluating models on the device’s local data and then sending/loading the updated/global model weights to and from the client.

**Why we need SuperNode and SuperLink**? In federated learning,the architecture mainly involves server and client. But Superlink and Supernode orchestrates the flow in secure, reliable, efficient and organised manner. Therefore, it is named as SuperNode. The SuperLink is then responsible for acting as the missing link between all those SuperNodes.

In our project the devised architecture design is one Superlink with subprocess serverApp as server side and Supernode with subprocess clientApp as each client node.

**Federated Learning with Flower**

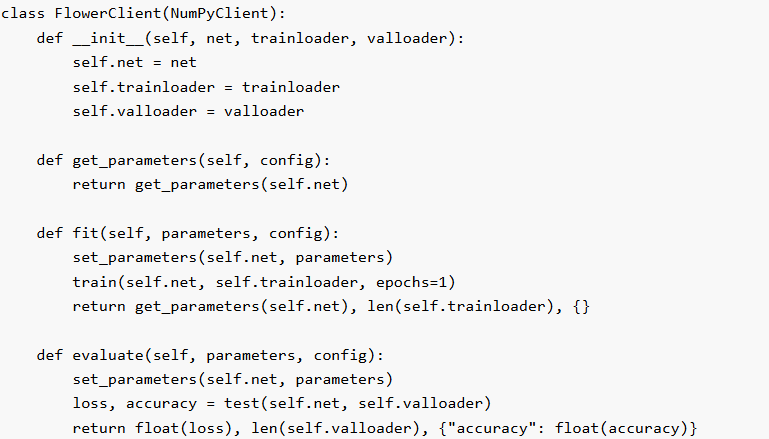
Flower handles the entire lifecycle: sending the global model to clients, locally updating the model based on client data, sending the updates back to the server, and aggregating those updates into a new global model.

**Update model parameters(set\_parameters & get\_parameters)**

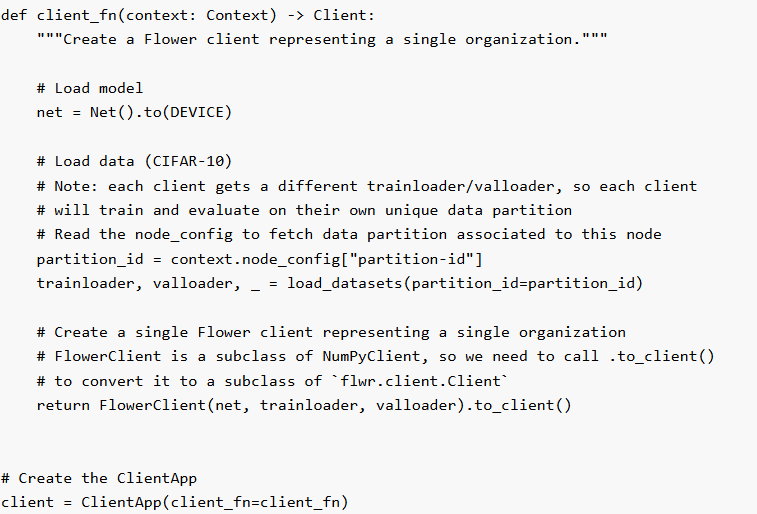
In Flower-based federated learning with PyTorch, model parameter exchange between the client and server is handled through two key helper functions: set\_parameters and get\_parameters. These functions bridge the gap between PyTorch's internal model representation and Flower's NumPy-based communication format. The get\_parameters function extracts the model’s parameters using state\_dict() and converts them into a list of NumPy arrays for transmission to the server. Conversely, set\_parameters takes the parameter list received from the server, converts them back into PyTorch tensors, and loads them into the model using load\_state\_dict(). This mechanism ensures seamless synchronization of model weights during federated learning rounds.

**Flower ClientApp**

As we discussed in the architecture design,**Flower ClientApp** represents the client-side logic in a federated learning system, where multiple clients collaboratively train a model under the coordination of a central server. In Flower, a client is implemented by subclassing either flwr.client.Client or the simpler flwr.client.NumPyClient. The NumPyClient interface requires the implementation of three methods: get\_parameters, fit, and evaluate. These methods respectively handle sending current model parameters to the server, training the model on local data and returning updated parameters, and evaluating the model on local data.



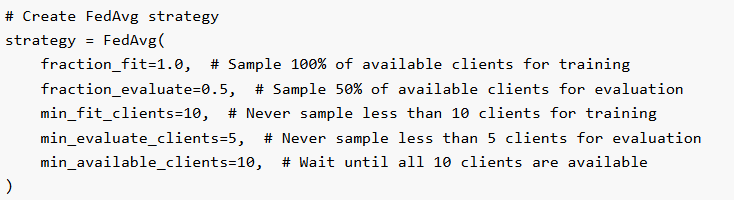
Flower provides a client\_fn function that dynamically creates FlowerClient instances when needed, reducing memory overhead. Finally, the ClientApp is instantiated with this client\_fn, acting as the entry point for executing client-side logic in Flower's federated learning rounds. This structure enables scalable, modular, and efficient client simulation in both single- and multi-machine environments.



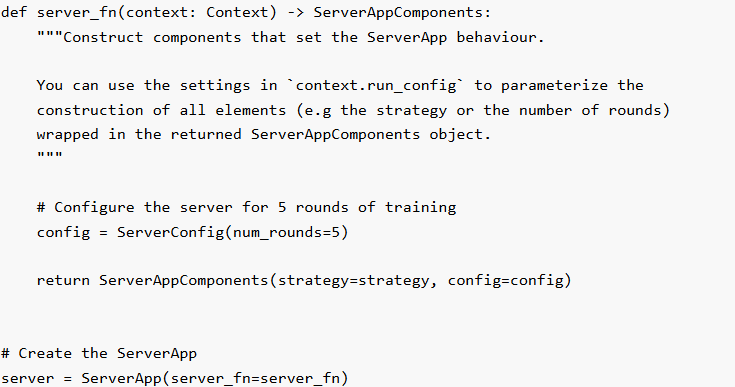
**Flower ServerApp**

The Flower ServerApp is responsible for orchestrating the federated learning process by managing communication between clients and coordinating the training strategy. The core of the server logic is the strategy, which dictates how model parameters are aggregated and how clients are sampled during training and evaluation. FedAvg (Federated Averaging) strategy — a common approach where client models are trained independently and averaged on the server. The strategy is configured with parameters such as the fractions of clients to be sampled for training and evaluation, and the minimum number of clients required to proceed.

An example FedAvg Strategy code

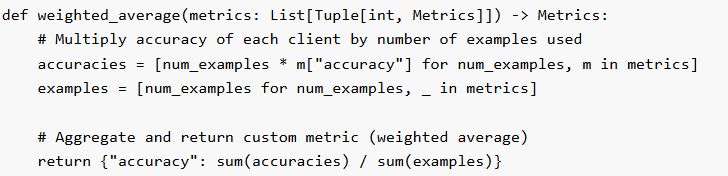


To define server behavior, a server\_fn is implemented, which returns a ServerAppComponents object including the strategy and server configuration (e.g., number of training rounds). The ServerApp is created using this function, making it the server-side entry point for Flower. Finally, run\_simulation is used to launch the training, where resources per client (CPU/GPU) are specified through backend\_config, and the simulation is started by combining the ServerApp, ClientApp, number of simulated clients, and the resource configuration. This setup enables an efficient, customizable federated learning simulation within a single or distributed environment.



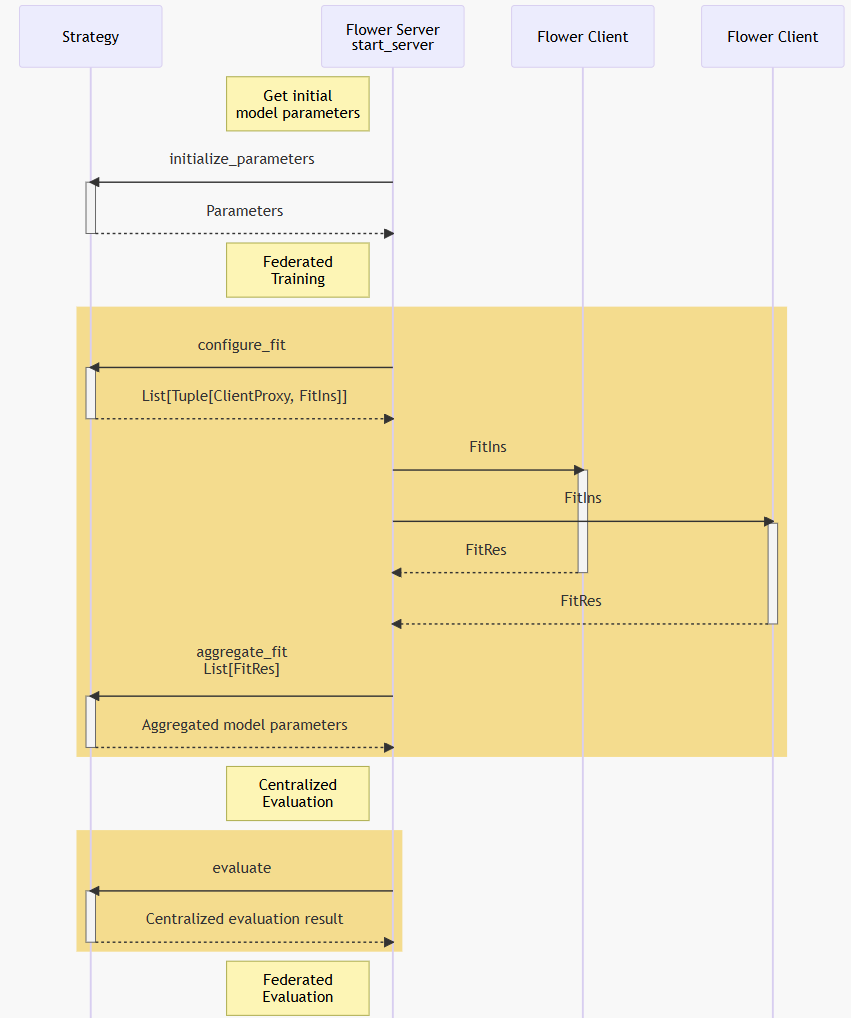
**Federated Evaluation Metrics**

Flower does not automatically aggregate custom metrics like accuracy that are returned inside a dictionary from the evaluate method. Unlike loss—which is a standard scalar value Flower knows how to handle—custom metrics can vary in type, meaning, and structure. Therefore, we must explicitly define how these custom metrics should be aggregated. To handle this, Flower allows users to pass custom aggregation functions to the strategy via the fit\_metrics\_aggregation\_fn and evaluate\_metrics\_aggregation\_fn parameters. These functions are called during training or evaluation to compute an overall result from individual client-reported metrics.



**Federated Learning Strategies:**

Federated Learning (FL) strategies define how local model updates from multiple clients (devices or nodes) are aggregated into a global model during training. These strategies are central to coordinating learning across decentralized data sources while preserving privacy. Common FL strategies include **FedAvg** (Federated Averaging), where local models are averaged based on the number of local samples; **FedProx**, which adds regularization to handle data heterogeneity; and **FedOpt**, which applies optimization techniques like Adam during aggregation.

The Flower api call flow:



Reference Figure from [Flower Api Flow](https://flower.ai/docs/framework/how-to-implement-strategies.html)

**The initialize\_parameters method**

The initialize\_parameters method is invoked once at the start of the federated learning process to provide the initial global model parameters. These parameters are serialized into a Parameters object and distributed to all clients before the first training round. In practice, initial parameters are typically derived from a server-side model (e.g., a freshly initialized or pre-trained Keras model), converted to NumPy arrays, and then serialized using fl.common.ndarrays\_to\_parameters(). This ensures all clients begin training from the same model state. If no parameters are provided (i.e., None is returned), the server falls back to requesting initial parameters from a randomly selected client—though this is primarily for prototyping and not recommended for production use. Server-side initialization is particularly useful for resuming training from saved checkpoints or fine-tuning existing models with federated learning, making it a crucial step in creating reliable and consistent training pipelines.

**The configure\_fit method**

The configure\_fit method defines how the Flower server orchestrates each training round by determining which clients participate and what instructions they receive. It takes in the current server round number, the global model parameters, and the ClientManager—which holds information about available clients. The method returns a list of tuples where each tuple contains a ClientProxy and a FitIns (fitting instructions), specifying what each client should do during that round.

In typical implementations, configure\_fit:

* Randomly samples a subset (or all) of the available clients,
* Packages the current global model and a configuration dictionary (e.g., training hyperparameters),
* Sends these instructions to the selected clients.

**The aggregate\_fit method**

The aggregate\_fit method is responsible for combining the updated model parameters received from clients after a training round. It is called by the Flower server after clients—selected via configure\_fit—have completed training and returned their results.

The method receives:

* A list of successful results, each containing a ClientProxy and corresponding FitRes (which includes the client’s updated parameters and training metrics),
* A list of failures, which includes clients that either failed to respond or encountered errors.

The method returns:

* An optional Parameters object containing the newly aggregated global model,
* A dictionary of aggregated metrics (e.g., average loss or accuracy across clients).

**The configure\_evaluate method**

The configure\_evaluate method orchestrates each evaluation round by determining which clients to evaluate and what instructions they receive. It accepts the current round number, global model parameters, and a ClientManager that tracks available clients. The method returns a list of (ClientProxy, EvaluateIns) tuples, where each EvaluateIns contains the global parameters and any evaluation-specific configuration (e.g., batch size or metrics to record). In a typical implementation, configure\_evaluate randomly samples a subset of clients and issues the same evaluation instructions to each. However, the per-client tuple structure also supports advanced use cases, such as sending different evaluation hyperparameters or even different model variants to distinct clients. By customizing client selection and instruction details, configure\_evaluate provides fine-grained control over how model performance is assessed across the federation.

**The aggregate\_evaluate method**

The aggregate\_evaluate method is responsible for combining the evaluation results received from clients after a round of federated evaluation. It takes the current round number, a list of successful evaluation results (EvaluateRes), and a list of failures (clients that did not respond or encountered errors). The method returns an optional loss value (as a float) representing the aggregated evaluation loss, and a dictionary of additional metrics such as accuracy or precision. If too many clients fail or the results are insufficient, the method can return None to indicate unreliable aggregation. This design ensures that the server can robustly summarize evaluation performance, even in the presence of client dropouts.

**The evaluate Method**

The evaluate method is designed to perform server-side evaluation of the current global model parameters. Unlike configure\_evaluate and aggregate\_evaluate, which handle client-side (federated) evaluation, this method enables the strategy to assess model performance directly on the server, typically using a centralized validation dataset. It returns an optional tuple containing the loss value and a dictionary of evaluation metrics. If evaluation is not required or fails (e.g., due to data unavailability or loading issues), it may return None. This method is useful for comparing server-side and federated evaluation results.

**Custom Strategies Implemented for Smart Energy Meter System**

The choice of strategy directly impacts model performance, convergence speed, and robustness in real-world distributed environments.

Flower provide both the approaches to use their own strategy or implement a custom strategy which will have the Flower **Strategy** base class.

To make custom Strategies use this reference guide from flower   
<https://flower.ai/docs/framework/tutorial-series-build-a-strategy-from-scratch-pytorch.html>  
  
A few example of Custom Strategies which i tried for our model training:

1. CNN based FedAvg strategy which basically take all the model weights and then average over them to get the updated global model.

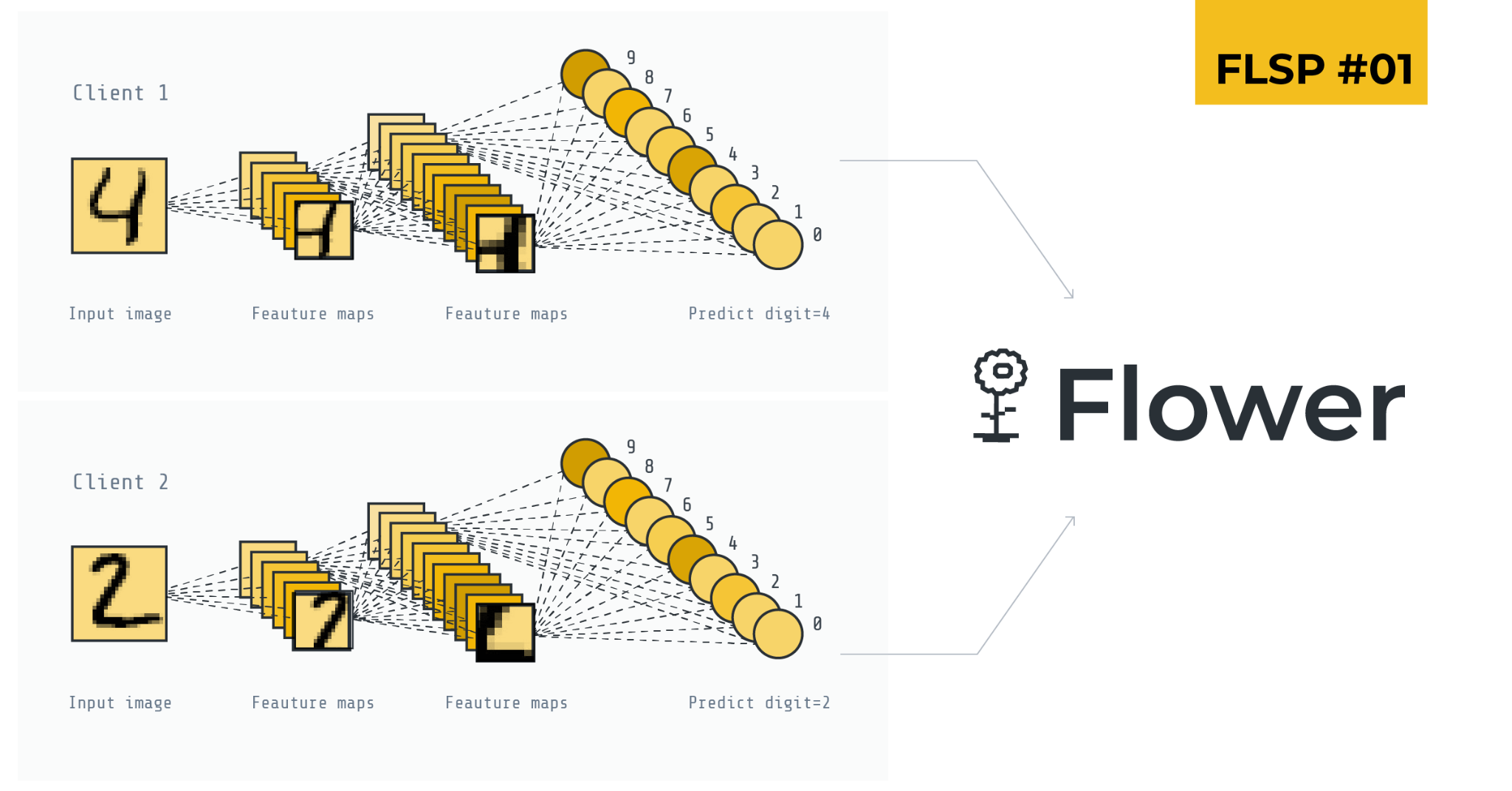
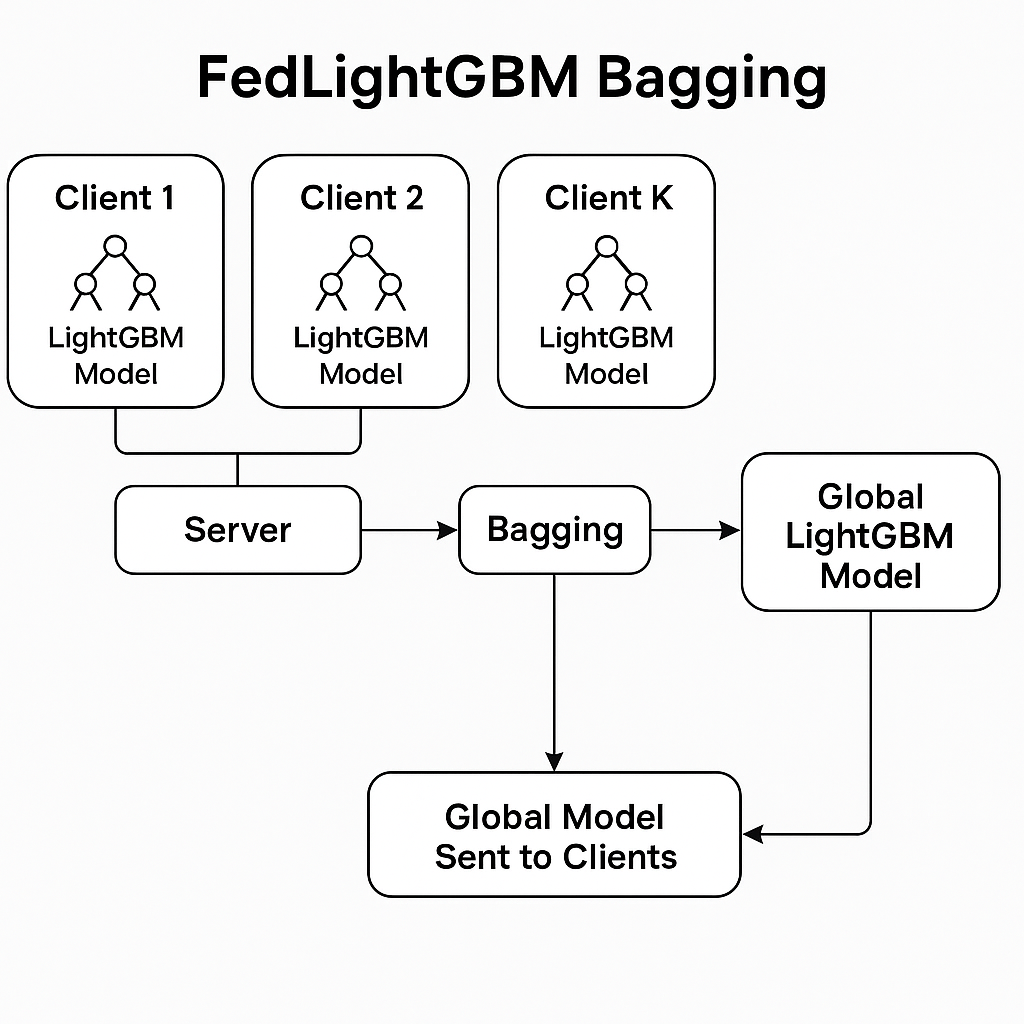


Figure reference link [CNN architecture](https://flower.ai/blog/2023-01-12-fl-starter-pack-fedavg-mnist-cnn/)

**Tree-Based Federated Learning Strategies for LightGBM:**



Tree-based Federated Learning (FL) aggregation is a hierarchical strategy where client updates are aggregated in a tree-like structure instead of a flat, centralized manner. In this approach, clients are grouped into clusters (or sub-trees), and each group elects an intermediate aggregator (e.g., edge server or SuperNode) to combine local updates before forwarding them up the hierarchy. This reduces communication overhead, enhances scalability, and allows partial model aggregation closer to data sources, making it suitable for large-scale or bandwidth-constrained federated networks. Tree-based aggregation can also improve fault tolerance and support asynchronous communication across layers.

I have implemented two custom federated learning strategies for tree-based models for the LightGBM model which we have used for our project: **FedLightGBMBagging** and **FedLightGBMCyclic**. These strategies address the challenge that traditional model parameter averaging (common in neural networks like the one i described above FedAvg for CNN) does not work well with decision trees. Instead, they define alternative server-side aggregation mechanisms for LightGBM models trained across distributed clients.

**FedLightGBMBagging Strategy**

Implements a bagging-based ensemble strategy.

(aggregate\_fit): Collects LightGBM models from all participating clients in serialized (byte array) form. Constructs an ensemble by storing all client models along with uniform weights. The server returns this ensemble as the new global model.

(aggregate\_evaluate): Aggregates metrics such as loss(RMSE and R2- score) by computing a weighted average based on each client's sample size.

**FedLightGBMCyclic Strategy**

Implements a cyclic or sequential update approach.

(aggregate\_fit): Instead of combining multiple models, only the model from the last participating client is selected as the global model. This creates a model update sequence where each round builds directly on the last selected client's result. (aggregate\_evaluate): Uses the evaluation result from the last client only, reflecting the performance of the currently adopted global model.

**Aggregate Evaluation Results**  
Flower provides full flexibility for customizing the aggregation of evaluation results from individual clients. It allows to define custom metrics and strategies based on our need and requirements. Clients can return evaluation metrics—such as accuracy or loss—in a dictionary, which the server-side strategy can then aggregate. For instance, a custom strategy can compute a weighted average of client accuracies based on the number of test samples. This enables more meaningful global performance metrics by accounting for the variability in client dataset sizes and local performance.

**Prometheus and Grafana-based Metric Monitoring**

Prometheus is an open-source monitoring and alerting toolkit designed for recording real-time metrics in a time-series database. It features a powerful query language (PromQL) and supports dynamic service discovery, making it suitable for observing distributed systems.

Grafana is a visualization and analytics platform that integrates with data sources like Prometheus to create interactive dashboards for monitoring system behavior and performance over time.

In the federated learning (FL) server application, Prometheus and Grafana can be integrated to monitor and visualize key training metrics such as RMSE, R² score, prediction mean, round duration, and other metrics based upon our need.

For it’s implementation we use prometheus\_client Python library to define and update custom metrics:

* model\_rmse, model\_r2: Evaluate global model performance.
* round\_time: Tracks the duration of each federated learning round.
* client\_count: Indicates the number of active clients per round.
* avg\_prediction: Captures the average predicted value across clients.

Metrics are updated during both the training (fit) and evaluation phases using custom aggregation functions. The save\_metrics function logs each round's RMSE and R² to both Prometheus and a CSV file for redundancy. The Prometheus server is started on port 8000 and automatically collects these metrics, which can then be visualized using Grafana dashboards for real-time insight into model training dynamics and system performance.

**Microservice-Based Docker Implementation**

Flower’s federated learning framework adopts a modular, microservice-oriented architecture that can be effectively deployed using Docker containers. This approach enhances scalability, simplifies testing, and isolates components for better maintainability.

**Architectural Overview**

The system is composed of the following microservices:

SuperLink: Acts as a central communication broker, routing messages between server and clients.

SuperNodes: Serve as intermediaries between clients and the server, facilitating scalable federation.

ServerApp: The federated learning server, orchestrating model aggregation and global training logic.

ClientApps: Simulated federated clients that train local models and communicate updates to the server via SuperNodes.

The reference guide for the same <https://flower.ai/docs/framework/docker/tutorial-quickstart-docker.html>

**Advantages of This Architecture**

Modularity: Each component is isolated and independently replaceable.

Scalability: Supports scaling to multiple clients and nodes with minimal configuration changes.

Reproducibility: Docker ensures consistent environments across machines.

Simplified Networking: Container name resolution in a custom bridge network reduces complexity.

**Secure Communication with SSL/TLS**

To ensure secure communication between federated components in a production-ready Flower deployment, SSL/TLS can be enabled for all microservices running in Docker containers. Transport Layer Security protects data exchanged between the SuperLink, SuperNodes, ServerApp, and ClientApp containers. TLS is configured using a PEM-encoded certificate authority (CA) certificate, a private key, and a certificate chain, all mounted into their respective containers using Docker’s volume binding. Additionally, the federated deployment configuration can be extended to support TLS by appending a [tool.flwr.federations.local-deployment-tls] section to the pyproject.toml, which references the trusted root certificate. All containers run under a non-root user, requiring proper ownership and permissions on the mounted certificate files. While suitable for production, self-signed certificates may also be used during testing. This TLS setup strengthens security guarantees across the microservice architecture without altering Flower's fundamental deployment logic.

**Subprocess vs. Process Isolation in Flower Docker Deployments**

Flower provides two isolation modes for launching the ServerApp and ClientApp: Subprocess and Process modes, offering a balance between resource efficiency and operational isolation. In the default Subprocess mode, the SuperLink and SuperNode containers internally spawn and manage the execution of their corresponding ServerApp and ClientApp. This mode simplifies deployment by reducing the number of containers needed and is particularly advantageous in resource-constrained environments. However, it comes with trade-offs, such as reduced fault isolation and potential security concerns, since the applications run within the same container process space. Alternatively, in Process mode, each application (ServerApp or ClientApp) runs in its own container, completely isolated from the SuperLink or SuperNode. This approach enhances modularity and fault tolerance at the cost of additional complexity and resource overhead. Flower also supports mixed-mode configurations, where subprocess and process isolation can be combined—for example, running the SuperLink in subprocess mode while deploying SuperNode in process mode. This flexibility allows to tailor the deployment architecture to the needs of their specific use case.

**Docker Compose Implementation in Flower**

The Docker Compose implementation of Flower streamlines the deployment process by allowing users to set up all core components—such as SuperLink, SuperNode, ServerApp, and ClientApp—with a single command. This quickstart approach is ideal for development and prototyping, providing a ready-to-use environment where applications can be built, run, and tested efficiently without manually managing container lifecycles. Docker Compose supports both insecure (non-TLS) and secure (TLS-enabled) configurations, enabling developers to toggle between development and production-like environments seamlessly. The setup also supports key features like local state persistence and certificate-based TLS encryption. Developers can generate self-signed certificates for testing or integrate production-grade certificates for secure deployments. Additionally, Flower projects can be customized and extended using the pyproject.toml configuration file, which controls networking, dependency management, and component interaction. This approach offers a practical and scalable way to manage federated learning infrastructure using containerized services.

I have explained all the things, that are related to the flower framework and for the deeper understanding you can read the reference guide provided by Flower official website [flower.ai](https://flower.ai/docs/framework/)

Now, I will go through the custom client ,server and strategies i have implemented for our Smart Energy Meter System. I will also go through the docker based design which i have made for the streamline implementation of the project.  
  
**Docker Compose Monitoring & Server Implementation**

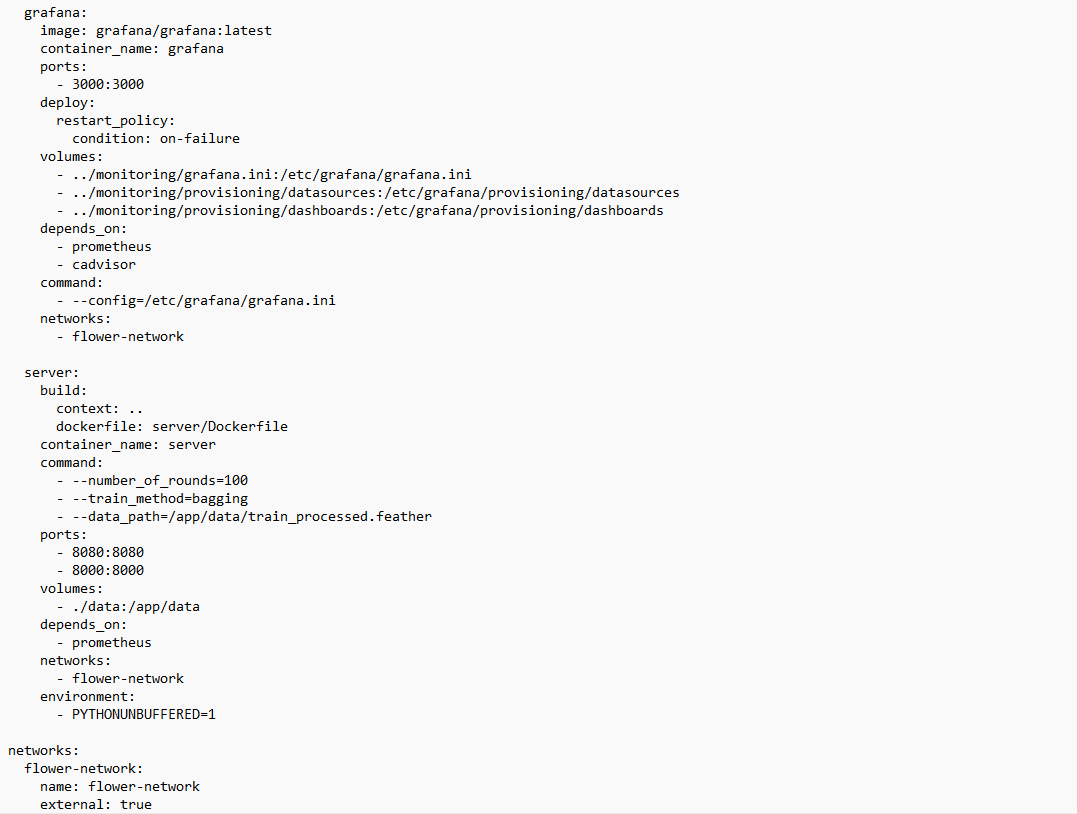
Docker Compose setup integrates a Flower federated learning server with a monitoring stack using Prometheus, cAdvisor, and Grafana.

The server container runs the training logic and also exposes ports 8080 and 8000 for application-level communication and integrates into the monitoring network to allow metric collection, while Prometheus collects metrics from cAdvisor, which monitors container performance and the Gauges i have made for the Evaluation Metrics — model\_rmse, model\_r2, round\_time, client\_count, avg\_prediction.

Grafana visualizes these metrics using pre-configured dashboards on port 3000.

And all these services run in a shared network (flower-network), enabling smooth communication and real-time monitoring of system and training performance.





**Custom Flower SeverApp**

This custom Flower server application is designed for federated learning (FL) in smart energy systems, specifically for load forecasting tasks using LightGBM. It incorporates a hybrid centralized and federated approach by first training a global baseline model on available centralized data. This model is then distributed to clients for further federated updates. The use of K-Fold cross-validation during server-side pretraining ensures model robustness and generalization.

The server supports two federated training strategies—bagging and cyclic—defined in custom strategy classes. It also integrates real-time performance monitoring through Prometheus, exposing metrics such as RMSE, R², round number, and client count. These metrics are visualized via Grafana dashboards. Client contributions during training and evaluation are aggregated using custom functions that compute weighted averages of metrics across participants. This setup provides a scalable and traceable federated learning workflow suitable for industrial energy data environments.





**Custom Federated Strategies for LightGBM in Flower**

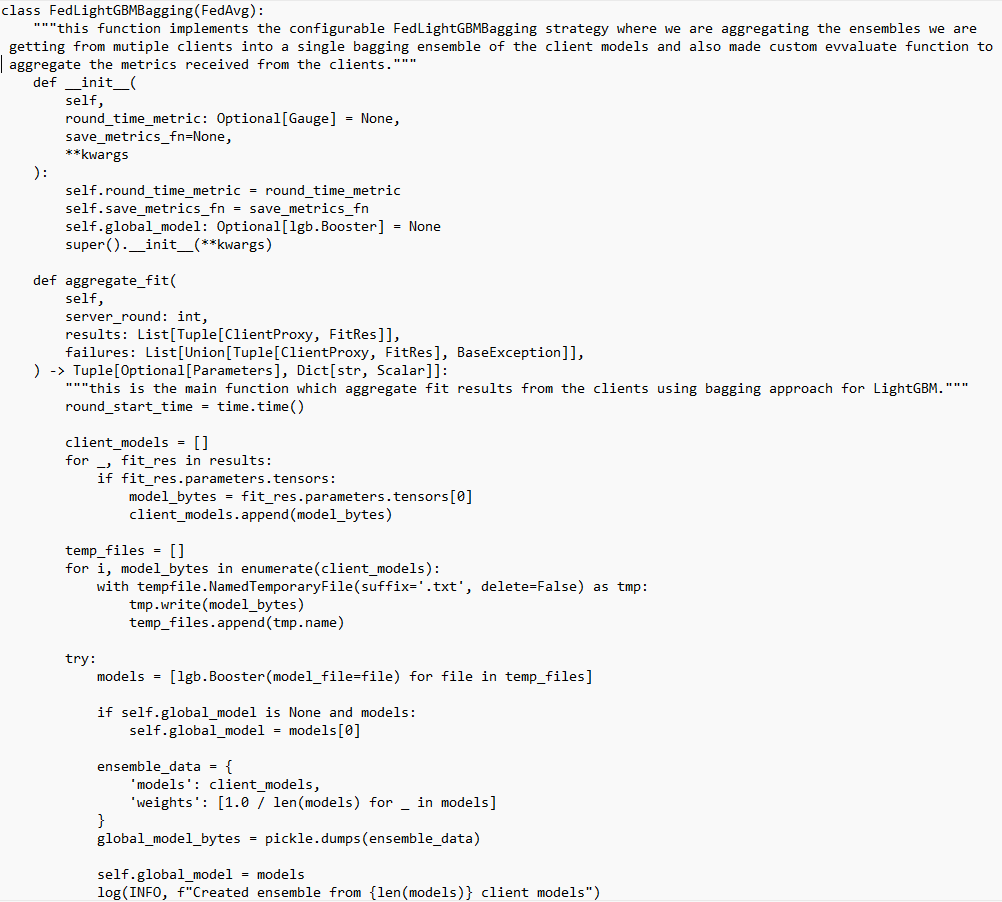
When implementing federated learning with tree-based models like LightGBM, traditional model parameter averaging techniques such as FedAvg are not directly applicable due to the non-differentiable structure of decision trees. To address this, custom strategies like FedLightGBMBagging and FedLightGBMCyclic have been designed using Flower’s extensible strategy API.

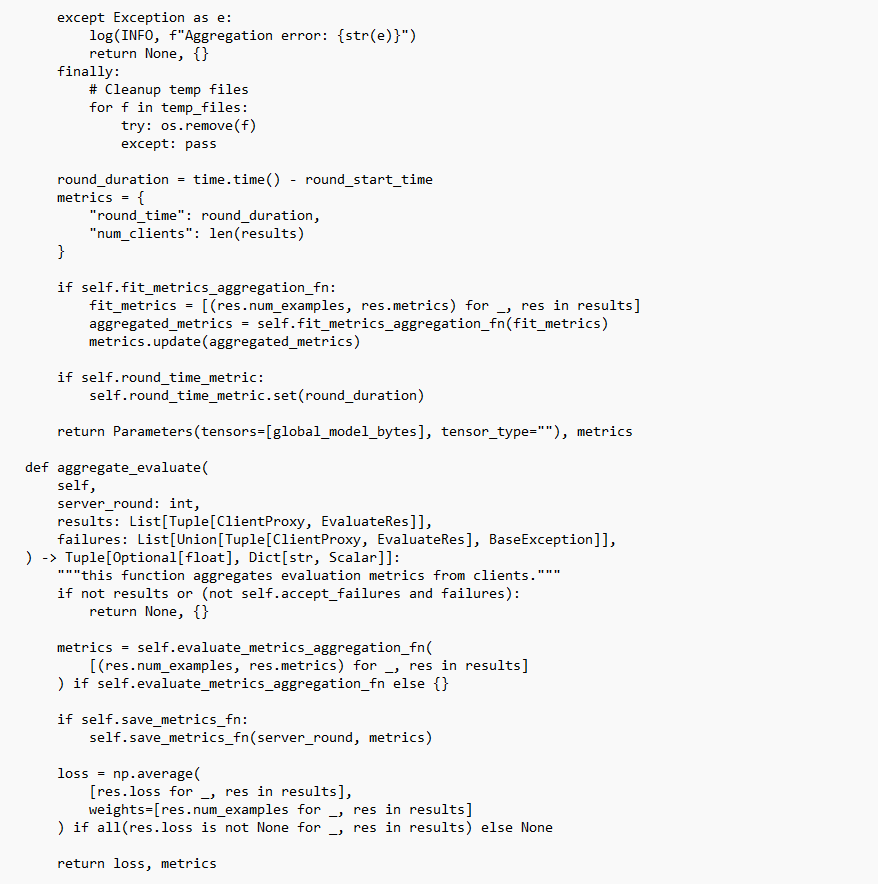
**FedLightGBMBagging**

The FedLightGBMBagging strategy implements an ensemble-based aggregation approach for LightGBM models. Instead of averaging model parameters, each client trains an independent LightGBM model locally. After training, the server collects these individual models and constructs a global ensemble by combining them using a bagging technique—effectively averaging their predictions during inference. This is achieved by: Extracting and deserializing model files from client updates. Creating a list of LightGBM Booster models. Combining them with equal weights into a single ensemble stored as a serialized object.

**FedLightGBMCyclic**

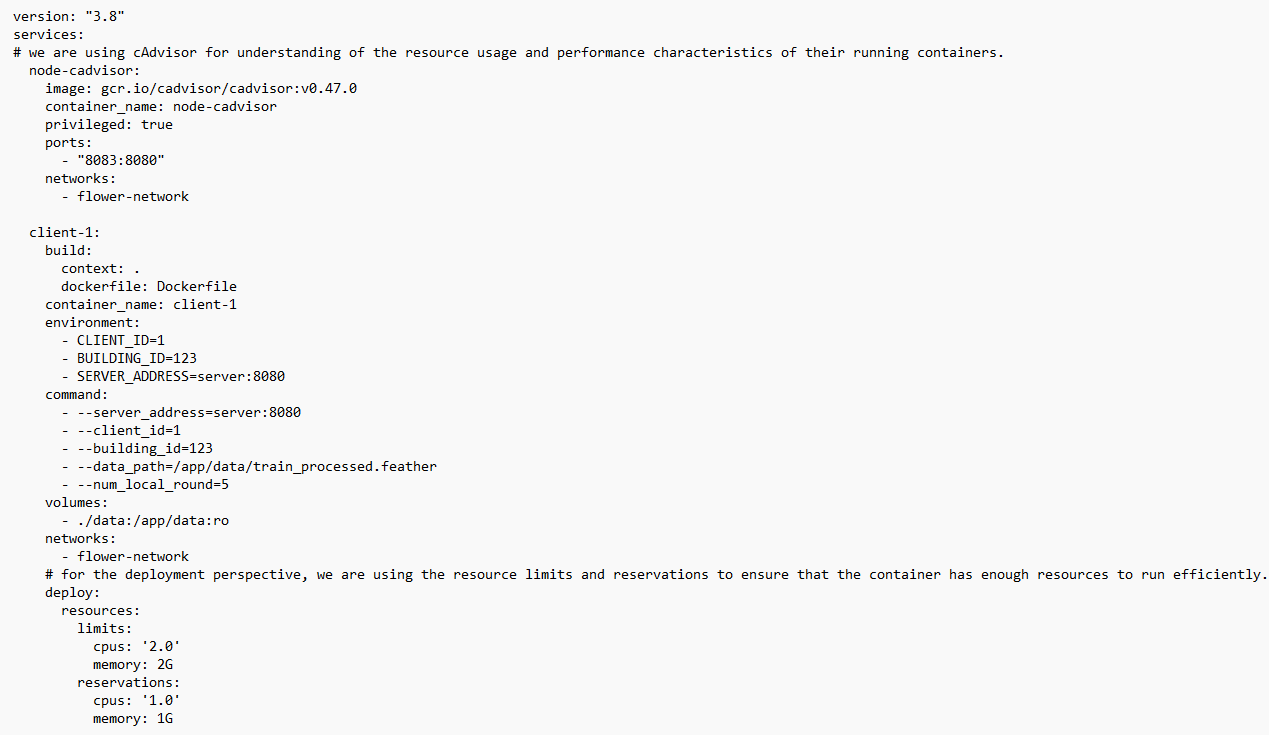
The FedLightGBMCyclic strategy follows a cyclic aggregation approach. Instead of ensemble learning, it updates the global model by selecting the latest model received from clients at the end of each round. In practice, this means the last client in the round provides the model that becomes the new global model. This method allows training to proceed in a sequential-like fashion across clients while reducing aggregation complexity. Key steps include: Using the model from the final client in the round as the new global model. Configuring all clients to participate in both training and evaluation each round. Optionally tracking round duration and evaluation metrics via Prometheus-compatible functions.





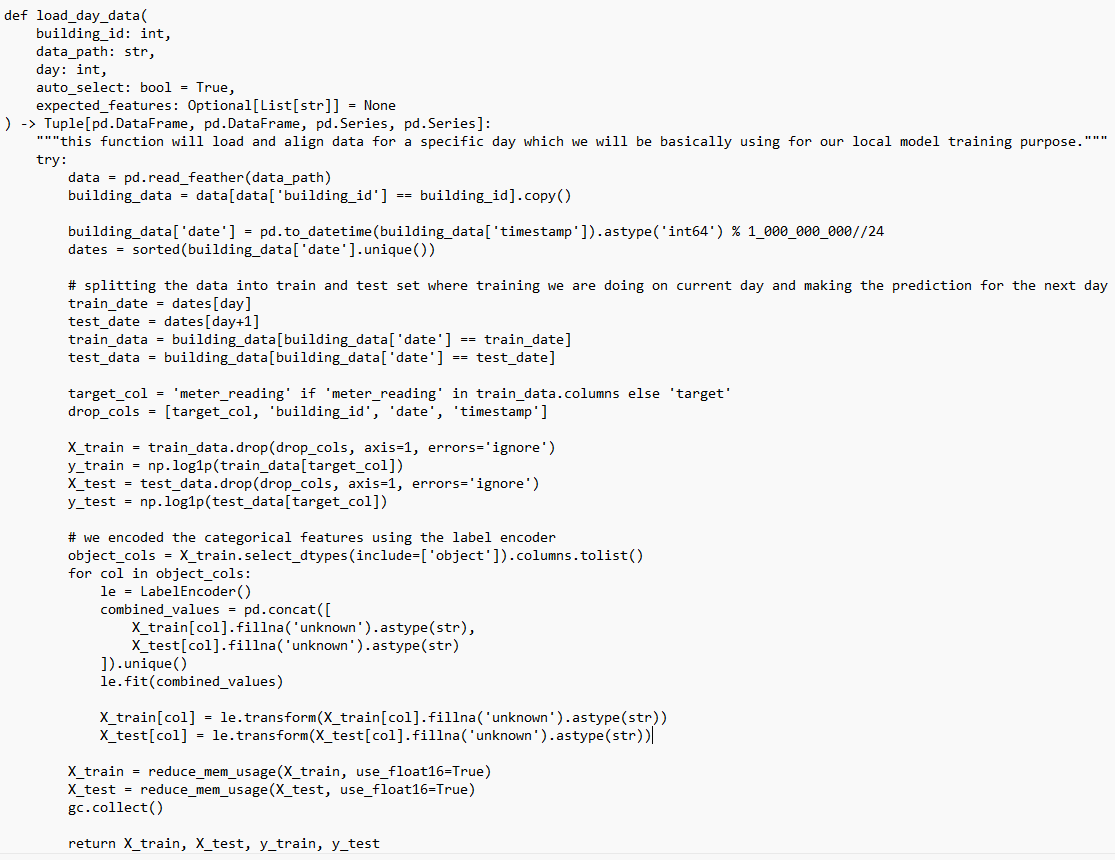
Similarly i have written the code for cyclic strategy also.

**Docker Compose Client & cAdvisor Containers**

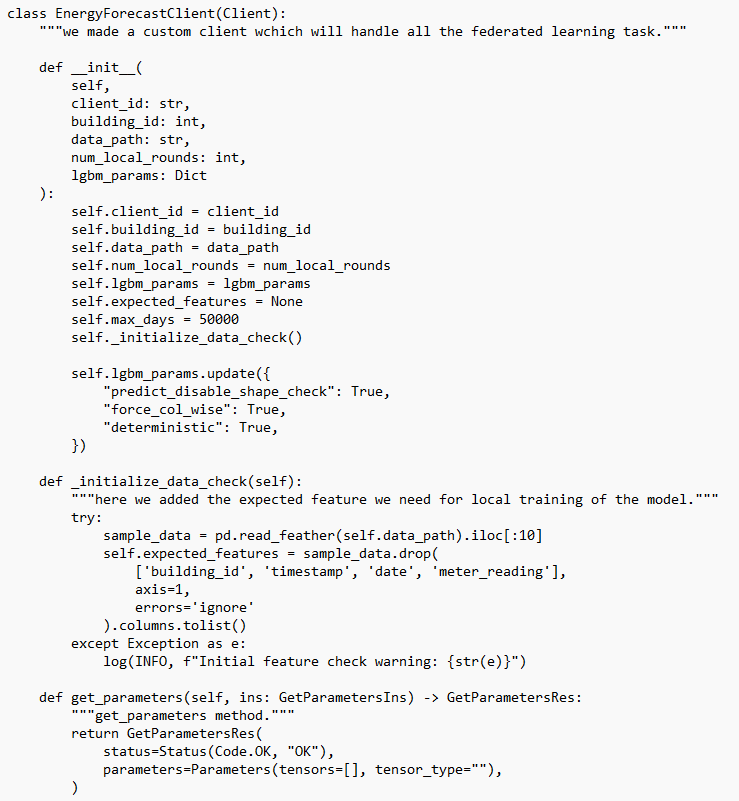
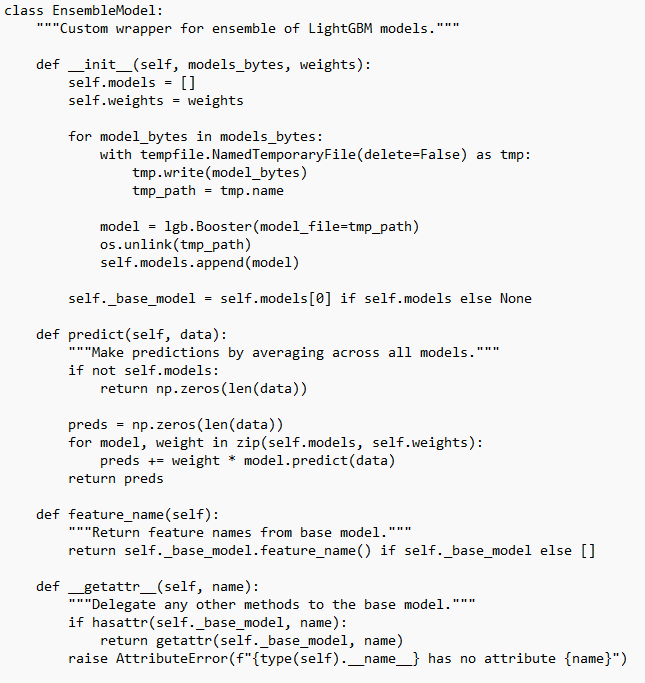
****

The Docker Compose configuration sets up a federated learning environment with resource monitoring and optimized container management. It includes a **cAdvisor service** for real-time monitoring of resource usage and performance of running containers, providing valuable insights during training. The **client service** represents a federated client, configured with specific environment variables and command-line arguments for training on local data using Flower. It also includes **resource limits and reservations** to ensure efficient and reliable execution, guaranteeing minimum CPU and memory availability while capping resource usage to prevent overconsumption. This setup supports scalable, monitored, and resource-aware deployment of federated learning clients.

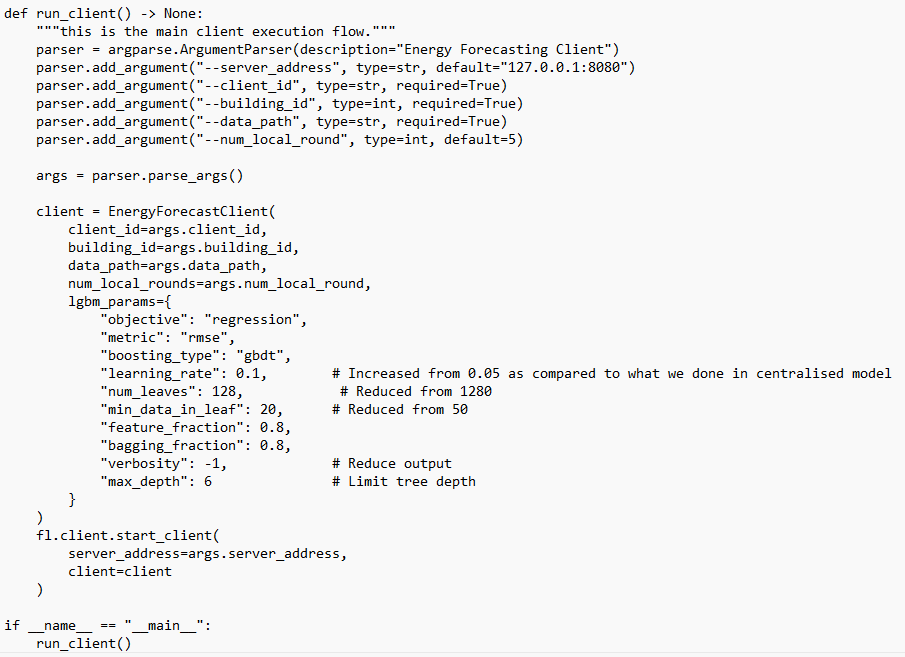
**Flower Custom ClientApp**

****

It loads and preprocesses building-specific data, aligns features, and trains a LightGBM regression model locally. During training, it uses K-Fold cross-validation to ensure robustness and selects the best-performing model. The model is then serialized and sent back to the server. For evaluation, it tests predictions on the next day’s data and returns metrics like RMSE and R². It also supports ensemble model handling and ensures memory-efficient training with data type optimization and feature alignment.

****

….



There are other functions also like fit, evaluate, load\_models,etc int the clientApp. I am not including them as those are too big for the reference you can see all the above codes in my github repo  
<https://github.com/bajoriya-vaibhav/SmartEnergyFL>

For the Implementation and demonstration purpose, we are using the ASHRAE dataset which we got from the ASHRAE Great Energy Predictor III competition. This dataset includes:

* train.csv – Training data
* test.csv – Test data
* building\_metadata.csv – Metadata about buildings
* weather\_train.csv – Weather data for training
* weather\_test.csv – Weather data for testing

**Pre-Processing Pipeline**

1. Timestamp Processing

* converted timestamps to datetime format.

2. Weather Data Processing

* Adjusts timestamps using site-specific GMT offsets.
* Fills missing values with spline interpolation.

3. Feature Engineering Lag Features

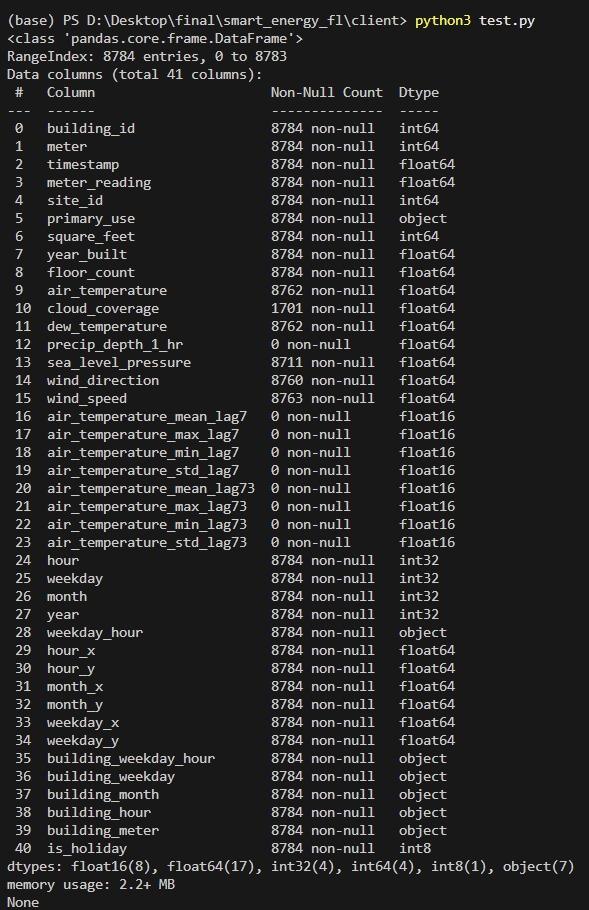
* Rolling stats (mean, min, max, std) on air temperature over 7 and 73-hour windows.
* Time-Based Features: Extracts hour, weekday, month, year + cyclical encodings using sine/cosine.
* Building/Meter Features: Interaction terms (e.g., building\_id × meter × hour); adds US holiday flags.

4. Merging Datasets Merges training/testing data with building\_metadata and weather using appropriate keys. Ensures consistent schema across datasets.

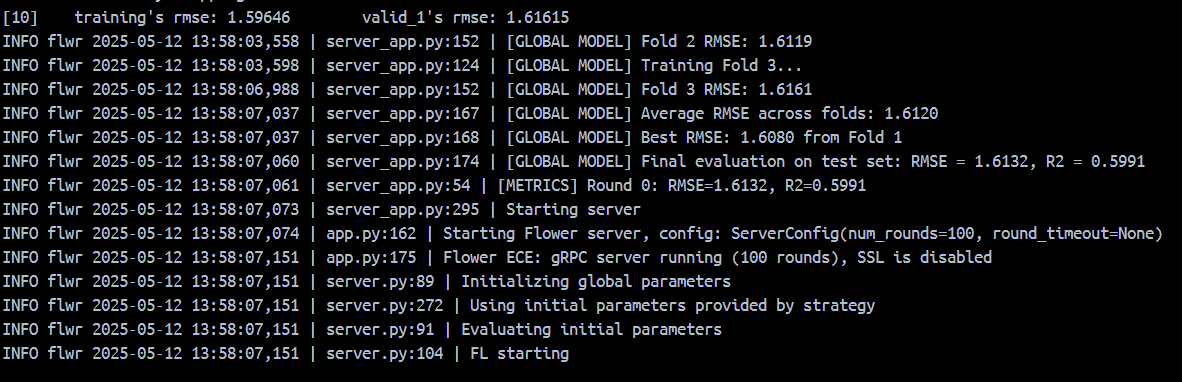
5. Memory Optimization Drops unused columns (e.g., ts). Invokes garbage collection to free up memory.

6. Saving Processed Data Saves cleaned data as Feather files:

* train\_processed.feather
* test\_processed.feather

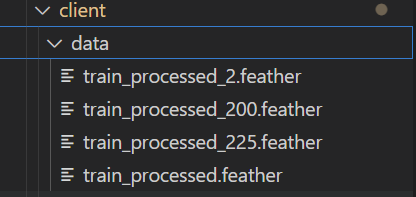


**Global Model Training**

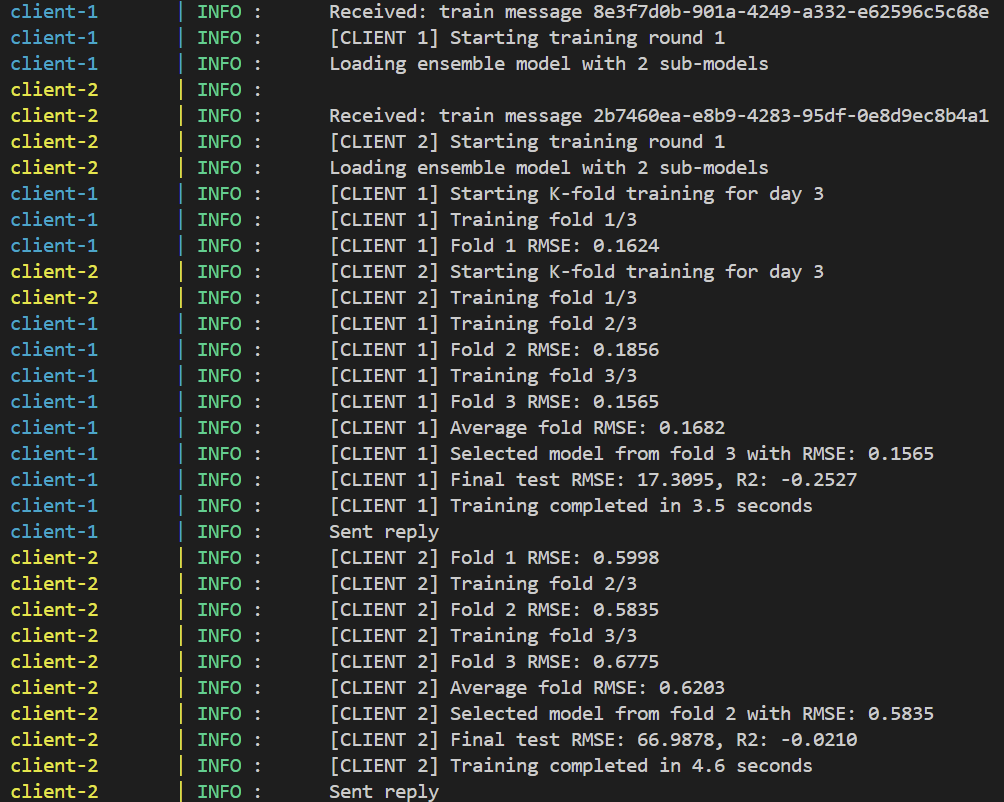
I have used the above pre-processed data for 5-building and 100 buildings, two different times i tried training the global based on the time it is taking for training where 100 building was taking a lot of time to train so i stuck with only 5-building data.  


**Local Model Training**

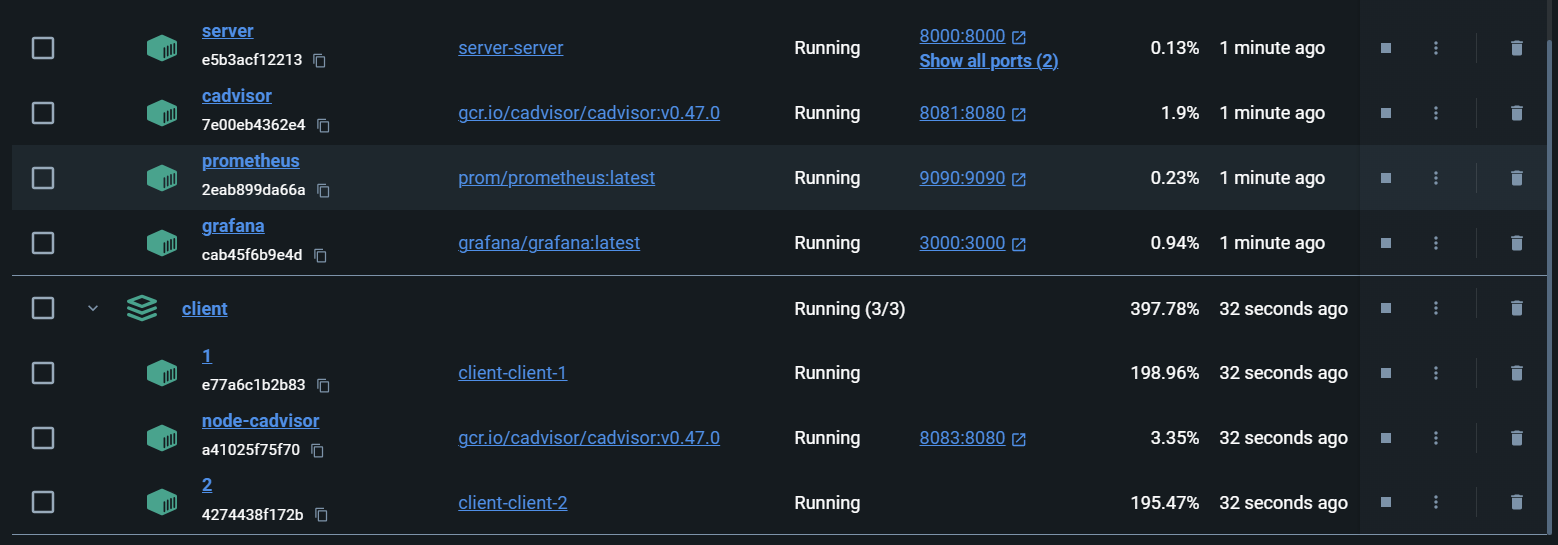
For local model training i have used the only 1 building data for each client which is having only meter reading data on hourly basis and training the received model for 1 day data which is 24 hourly reading and then testing the trained model on next 24 hour readings, then sending both updated model and evaluation metrics to the server for aggregation.



Training data for building id 100, 200, 225 and 125 respectively.



Client-Side Training of the Model in the each FL round.

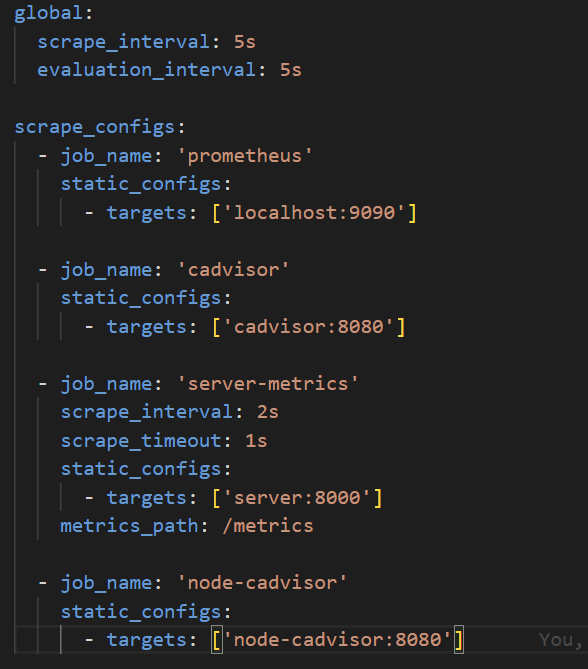


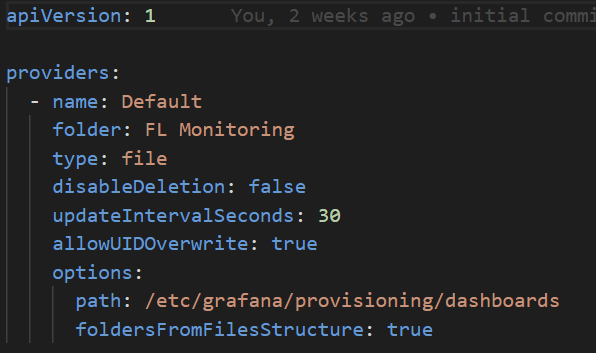
Docker Desktop Screenshot of Running Containers.

**Prometheus & Graphana Config.yml**

The Prometheus configuration file defines how metrics are scraped from different services. It sets a global scrape interval of 5 seconds, meaning Prometheus will collect metrics every 5 seconds by default. Several jobs are configured: one for Prometheus itself, one for cadvisor (which collects container-level metrics), one for server-metrics with a faster 2-second interval and 1-second timeout (useful for real-time server monitoring), and another for node-cadvisor, which monitors node-level container stats. Each job specifies the targets (host:port) and the /metrics endpoint used to fetch metrics.

The Grafana configuration file is responsible for provisioning dashboards automatically. It specifies a provider under the Default name, placing dashboards inside a folder named "FL Monitoring". The dashboards are loaded from files stored at /etc/grafana/provisioning/dashboards, and the system scans for updates every 30 seconds. The foldersFromFilesStructure setting helps organize dashboards based on the folder structure in the file system. This setup enables automated and structured dashboard management in Grafana.

****



**Graphana Dashboard Results**

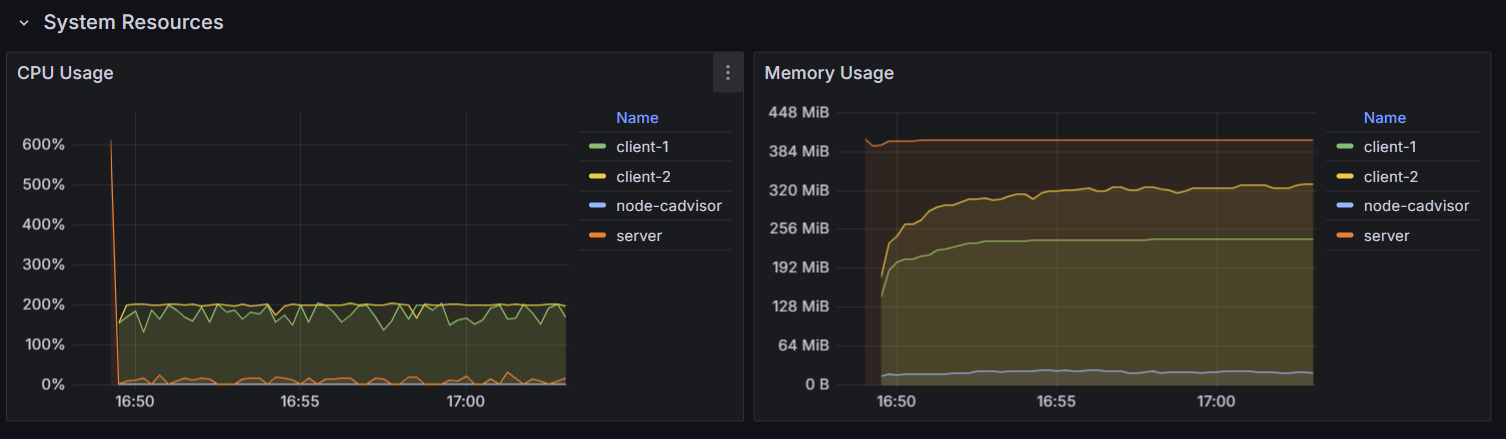
These are the evaluation metrics Results for total of 100 FL rounds with the training and testing for each round is taking on daily basis. The results are not that much good as we have used very less training to produce these results. Also there is not that much difference in the Centralised and Our FL based system.



These are the panels representing the Evaluation metrics which get updates after every FL rounds. First panel gives the average energy consumption of the 2 buildings, second panel gives the root mean square error avg. for both client predictions and similarly third panel gives the R2 score.



These Panel gives the Round trip time for each FL rounds, currently which FL round is going on, how many connected clients and the combined Perfomance metrics.



This is cAdvisor panels which show the system resource related metrics (like CPU usage and Memory Usage) for each client and server connected in the same network.

**Conclusion**

The Flower-based Federated Learning implementation successfully demonstrated decentralized model training across multiple clients while preserving data privacy. By leveraging Flower’s flexible framework, with coordinated efficient communication between the server and clients, enabling scalable and modular FL experiments. The integration with monitoring tools like Prometheus and Grafana further enhanced visibility into system performance, allowing real-time tracking of resource usage and training progress. Overall, this setup provides a practical and extensible foundation for a real time Smart Energy Meter Monitoring System.