**Low-Level Design Document**

**Privacy Preserving Quantum Secure Federated Learning Framework based on Multikey CKKS Homomorphic Encryption**

**Version :** 0.1

**Date :** 03rd Sept 2025

**Authors :** Pranali Nikam, Prabhat Karlekar, Siddhant Bopche

**Mentors :** Puneet Bakshi, Vinodh Kumar M.

**Department :** HPC I&E, C-DAC Pune

1. **Introduction**

Large-scale systems and distributed environments produce vast amounts of data from various sources such as devices, applications, and monitoring components. Directly sharing this raw information across different locations raises significant privacy, security, and regulatory challenges, making conventional centralized data processing or machine learning approaches less suitable. To address these challenges, this project integrates Federated Learning (FL), Homomorphic Encryption (HE), and High-Performance Computing (HPC) into a unified framework:

**Federated Learning (FL):** Enables multiple data centers to collaboratively train machine learning models without moving raw data outside their local environment. Each client trains locally and only shares model updates.

**Homomorphic Encryption (HE):** Ensures that all shared model updates remain encrypted end-to-end. The central server can perform aggregation on encrypted weights, preserving confidentiality while still producing a usable global model.

**High-Performance Computing (HPC):** Provides the computational backbone to scale federated training across multiple nodes, leveraging MPI and cluster resources for parallelism and efficiency.

The architecture ensures:

* Client data never leaves its source in plaintext.
* Aggregation is done on encrypted weights.
* HPC cluster resources handle the scale of computation.

Together, these technologies create a secure, scalable, and privacy-preserving federated training environment suitable for modern data centers.

1. **Purpose**

To enable privacy-preserving training of machine learning model (time-series) on distributed data using federated learning.

1. **Objectives**

* Federated Learning: Local training on client nodes, global aggregation on the server.
* Privacy via Homomorphic Encryption (OpenFHE): Encrypting model updates, enabling secure aggregation.
* Orchestration Layer: Automating workflow, round-based execution, logging.
* Kafka Integration: Streaming telemetry data ingestion.
* HPC Utilization: Running clients, server, and orchestrator processes on distributed worker nodes.

1. **Requirements**

|  |  |
| --- | --- |
| Type | Requirement |
| Functional | 1. Clients must train GRU/LSTM models on local infrastructure data. 2. Clients must encrypt model weights before transmission using homomorphic encryption. 3. Server must support model aggregation via MK-CKKS-HE /FedAvg. 4. Server must re-encrypt ciphertexts to target the client's domain using PRE. 5. Aggregated model weights must be sent back to clients for local retraining. 6. Clients compute metrics (e.g., MSE, R²) after each round. |
| Security | 1. Raw infrastructure data must never leave client premises. 2. Keys and ciphertexts must be securely exchanged and stored. |
| Non-Functional | 1. Model convergence check must be built-in to stop redundant training rounds. |

1. **High-Level Architecture**

As documented in the *High-Level Design (HLD) Document, Version 1.0*, the architecture defines the interactions between clients, server, and orchestrator. This LLD elaborates those components into detailed workflows, scripts, and configuration files.

1. **Low-Level Design**
2. **Component Details**

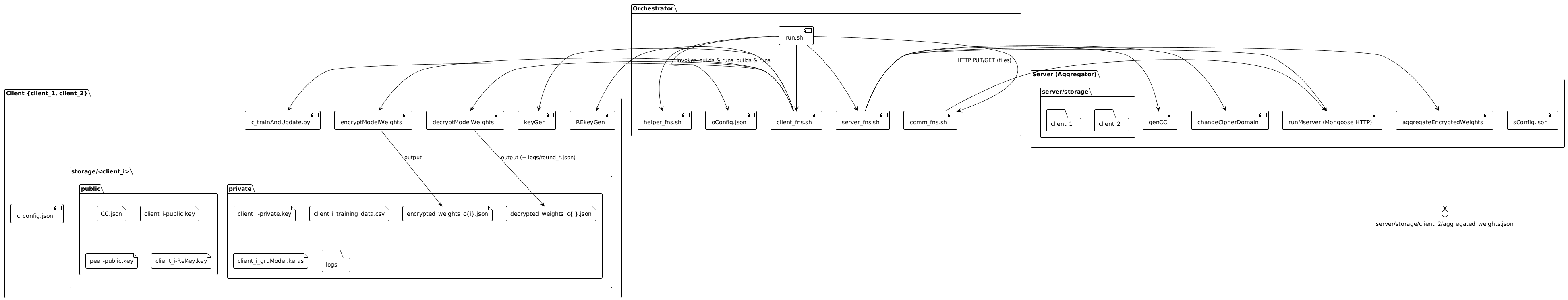


Figure 1: Component View of Client and Server

**Client Components:**

* c\_trainAndUpdate.py: Local GRU model training.
* encryptModelWeights, decryptModelWeights: C++ binaries for secure weight handling.
* keyGen, REkeyGen: Key generation and re-encryption key generation.
* storage/client\_i: Organized into public (shared keys, CC.json) and private (local keys, training data, models, logs).

**Server Components**:

* genCC: Generates crypto context.
* runMserver: Lightweight HTTP server (Mongoose).
* aggregateEncryptedWeights: Securely aggregates encrypted weights.
* changeCipherDomain: Handles re-encryption across clients.
* storage/server: Stores CC.json, keys, encrypted weights.

**Orchestrator:**

* Shell scripts (run.sh, helper\_fns.sh, client\_fns.sh, server\_fns.sh, comm\_fns.sh) control execution flow.
* Config (oConfig.json) defines round count, client/server parameters.

**Kafka:**

* producer.py: Publishes JSON telemetry.
* consumer\_client{1,2}.py: Consumes messages, stores CSV into client private dirs.

1. **Data Ingestion**

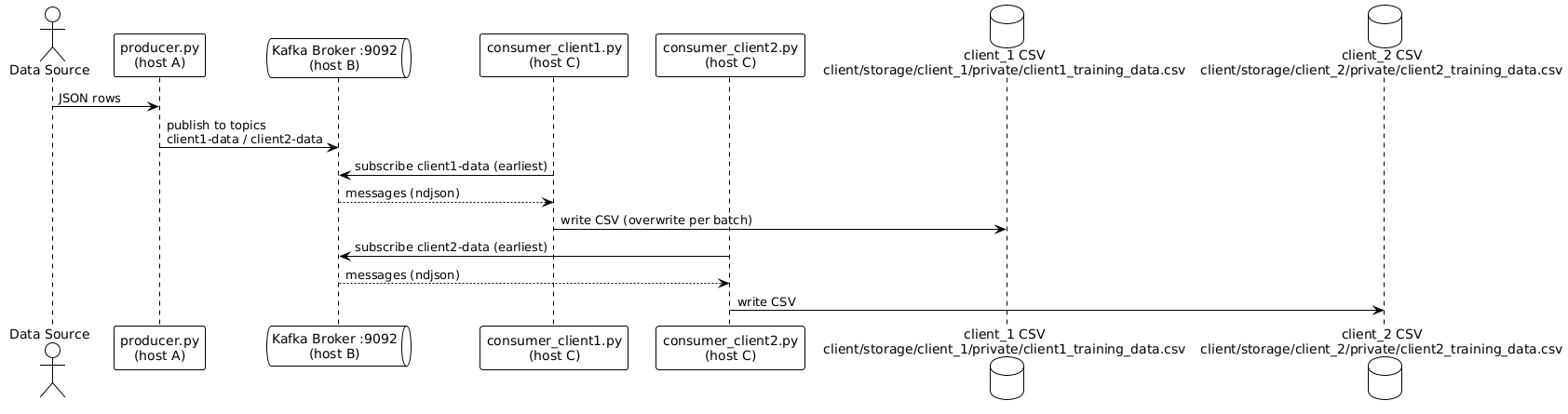
****

Figure 2: Data Ingestion-Kafka-Based Telemetry Data Flow

Kafka acts as the central publish–subscribe (pub-sub) backbone that enables reliable, scalable, and real-time data movement between the producer (data source) and multiple consumers (client-specific storage handlers). The flow can be summarized as follows:

Producer Side (Telemetry Publisher):

* A producer service collects telemetry data such as IT energy usage, non-IT energy usage, and system metrics from sensors, PDUs, or monitoring agents.
* The data is serialized into JSON messages for interoperability.
* Each JSON message is published into a designated Kafka topic (e.g., client1-data, client2-data).
* Producers are designed to be lightweight and fault-tolerant, ensuring minimal overhead on the data source nodes.

Kafka Broker (Message Hub):

* The broker acts as a centralized buffer and router, decoupling producers and consumers.
* It supports horizontal scaling with partitions and ensures durability of messages.
* In this setup, the broker is configured with separate topics for each client, ensuring proper data isolation.

Consumer Side (Client-Specific Data Ingestion):

* Consumers subscribe to client-specific topics (e.g., a consumer for client1-data will only process messages for Client 1).
* Each consumer continuously polls messages, deserializes JSON, and performs lightweight validation or preprocessing.
* The messages are persisted into CSV files in a dedicated private directory for each client. This separation guarantees that no client has access to another client’s telemetry data.
* Logging mechanisms are enabled to track ingestion progress, detect failures, and allow recovery from broker offsets.

1. **Proposed Scheme**

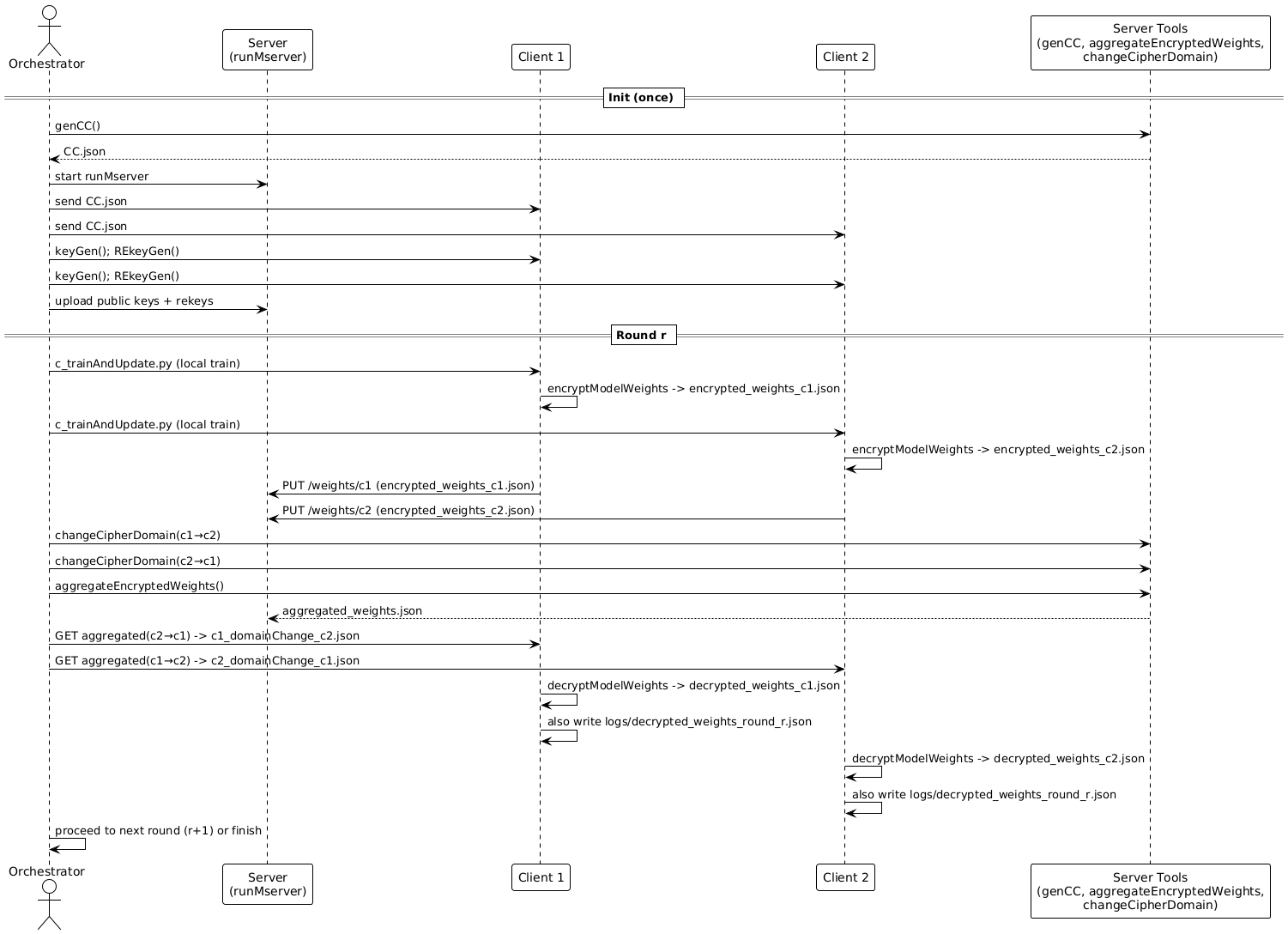


Figure 3: Sequence Diagram of Federated Learning Round

Each round follows a well-defined lifecycle, ensuring secure collaboration between clients and the orchestrator while preserving privacy through homomorphic encryption. The flow can be described as follows:

* Data Ingestion: Telemetry data from distributed sources is collected via Kafka producers and stored locally by client-specific consumers in preparation for training.
* Initialization: The orchestrator prepares the cryptographic context (CC.json) and distributes keys to clients.
* Local Training: Clients run c\_trainAndUpdate.py on their private datasets to update local models.
* Encryption & Upload: Trained weights are encrypted using OpenFHE and sent securely to the server.
* Aggregation: The server applies domain transformations if required and performs aggregation directly on encrypted weights.
* Distribution & Decryption: The aggregated model is sent back; clients decrypt and update their local models.
* Logging: Round-specific logs and artifacts are stored in client-private directories for traceability.

1. **Project Directory Structure**



Figure 4: Directory Structure

The filesystem under PPFL/ organizes code, configs, and data:

* Client Storage: public/ for CC.json + keys, private/ for sensitive files.
* Server Storage: Holds client weights, domain changes, aggregated weights.
* Orchestration: Contains scripts and configs for round control.
* Logs: Each round’s decrypted weights stored as logs/decrypted\_weights\_round\_r.json

1. **Technologies Used**

This project combines cryptography, machine learning, orchestration, and HPC resources to enable secure and scalable federated learning.

* + - 1. **OpenFHE (C++): Homomorphic encryption libraries**
* Used via binaries: encryptModelWeights, decryptModelWeights, aggregateEncryptedWeights, changeCipherDomain.
* Supports MK-CKKS scheme and proxy re-encryption.
* Linked directly with client scripts (c\_trainAndUpdate.py) and server modules.
  + - 1. **TensorFlow / Keras (Python) – Deep Learning Framework**
* Training script: c\_trainAndUpdate.py implements GRU-based models.
* Local evaluation: computes MSE, RMSE, R² per round.
* Output weights serialized before encryption.
  + - 1. **Apache Kafka (Python, Java) – Pub/Sub Data Ingestion**
* Producers (producer.py) publish telemetry JSON from sensors/BMS.
* Consumers (consumer\_client1.py, consumer\_client2.py) store data into CSV per client.
* Broker configured with multi-topic setup for data isolation.
  + - 1. **Mongoose HTTP Server (C) – Lightweight Server**
* Binary: runMserver.
* Provides /upload and /download endpoints for encrypted model weights.
* Logs stored in server/logs/
  + - 1. **Python + Shell Scripts – Orchestration Layer**
* run.sh, client\_fns.sh, server\_fns.sh, helper\_fns.sh, comm\_fns.sh coordinate training rounds.
* oConfig.json defines execution parameters (round count, clients, keys).
* Handles orchestration across MPI/SSH-enabled HPC nodes.
  + - 1. **High-Performance Computing (HPC) Cluster**
* Separate nodes host clients, server, and broker.
* Orchestrator node manages distribution of workloads.
* Logs and decrypted round weights stored centrally for auditing.

1. **Detailed Workflow**

The system workflow can be divided into phases: Initialization, Federated Learning Rounds, and Data Collection/Ingestion. Each phase ensures privacy-preserving and scalable training of models across multiple HPC-enabled data centers.

**1. Initialization Phase**

**a) Client Setup:**

* Each client node (data center) prepares its environment with local telemetry datasets and installs required libraries.
* Key generation is performed:

Public/Private key pairs (Pbᵢ, Prᵢ) for encryption/decryption.

Proxy Re-Encryption keys (PREᵢ→ⱼ) to allow domain conversion of ciphertexts.

* Clients share public keys and PRE keys with the server

**b) Server Setup:**

* Generates initial cryptographic context (genCC).
* Initializes global model weights w⁰ and broadcasts them to all clients.
* Starts the Mongoose HTTP server to handle client communication.

**c) Orchestrator Setup:**

* Reads configuration file (oConfig.json) with number of clients, rounds, and parameters.
* Prepares round-based directories for logs, configs, and encrypted weights.

**2. Federated Learning Round (Per Round Workflow)**

Each round n proceeds as follows:

1. **Local Model Training (Client-Side):**

* Clients receive the latest global model weights waggⁿ⁻¹.
* Replace their local model weights with these values.
* Retrain on local telemetry data using GRU/LSTM.
* Evaluate metrics (MSE, RMSE, R²).

1. **Encryption (Client-Side):**

* After training, clients encrypt model weights using OpenFHE:
* Encᵢ(wᵢⁿ) ← Encrypt (weights, Pbᵢ).
* Encrypted weights are sent to the server via HTTP.

1. **Proxy Re-Encryption (Server-Side):**

* For each target client, server converts ciphertexts from multiple clients into the recipient’s domain using PRE.
* Ensures that aggregation can happen without decryption.

1. **Encrypted Aggregation (Server-Side):**

* Server computes:

waggⁿ = (1/C) Σ Encᵢ(wᵢⁿ)

* The result is an encrypted aggregated model in each client’s ciphertext domain.

1. **Decryption (Client-Side):**

* Each client receives the aggregated model Encᵢ(waggⁿ) back.
* Decrypts using its private key: waggⁿ = Decᵢ (Encᵢ(waggⁿ)).
* Updates local model for the next round.

1. **Convergence Check (Orchestrator):**

* Monitors improvements in metrics across rounds.
* If convergence threshold is reached (e.g., no significant improvement over 5 rounds), training stops.

**3. Data Collection & Ingestion Workflow**

1. **Producer Stage (BMS → Kafka):**

* Telemetry data (IT load, non-IT load, UPS output, BMS readings) is extracted and streamed into Kafka. These files are then used as training input for local models
* Kafka Producers publish JSON messages to predefined topics (client1-data, client2-data, etc.).

1. **Broker Stage (Kafka):**

* Kafka broker stores and distributes messages across partitions for scalability and fault tolerance.

1. **Consumer Stage (Kafka → Client Storage):**

* Each client runs a Kafka Consumer (consumer\_clientX.py) subscribed to its topic.
* Data is persisted into CSV files in the client’s private storage directory.
* These files are then used as training input for GRU/LSTM models.

**4. HPC Orchestration Workflow**

* The orchestrator distributes client and server processes across HPC worker nodes.
* Communication happens via MPI/SSH, ensuring distributed execution.
* Logs and decrypted round weights are stored for auditing and reproducibility.