

Phase 2 Implementation Report

Blockchain Integration for Federated Learning Provenance

Muhammad Ibrahim Iqbal (27085)
Muhammad Maaz Siddiqui (27070)
Muhammad Ibrahim Farid (27098)

February 2026

1 Overview

Phase 2 introduces blockchain-based provenance logging to the federated learning system established in Phase 1. The primary objective was to create an immutable audit trail that tracks every model update contributed by each client across all training rounds, thereby addressing the accountability and transparency gaps inherent in standard federated learning architectures.

This phase demonstrates that blockchain integration can provide complete traceability with minimal computational overhead, adding less than one second per training round while maintaining the same model performance achieved in Phase 1.

2 Motivation and Problem Context

Standard federated learning systems treat the central aggregation server as a trusted black box. While this approach preserves data privacy by keeping raw data distributed, it creates significant accountability challenges. Key questions remain unanswered: Which hospitals contributed to which model version? Were all participants honest? Can we audit model lineage for regulatory compliance?

These concerns are particularly acute in healthcare, where algorithmic accountability is not merely desirable but legally mandated under frameworks like HIPAA and GDPR. A model trained through federated collaboration may be deployed in clinical decision-making, yet without provenance records, it is impossible to verify which institutions shaped its behavior or to trace performance issues back to specific training contributions.

Blockchain technology offers a solution through its core properties of immutability, transparency, and decentralization. By logging cryptographic hashes of model updates on-chain, we create a tamper-evident record that satisfies regulatory audit requirements while preserving the privacy guarantees of federated learning.

3 Technical Architecture

3.1 Blockchain Layer

We deployed a local Ethereum-compatible blockchain using Ganache, a development-focused environment that simulates blockchain behavior without the energy costs and latency of public networks. Ganache provides ten pre-funded test accounts, deterministic transaction processing, and instant block mining, making it ideal for rapid prototyping and experimentation.

The blockchain runs on localhost port 8545 using the HTTP provider protocol, allowing Python applications to interact via JSON-RPC calls. Each transaction incurs simulated gas

costs measured in the standard Ethereum unit, providing realistic estimates of on-chain storage expenses without requiring real cryptocurrency.

3.2 Smart Contract Design

The core provenance mechanism is implemented in a Solidity smart contract named `FLLogger`. The contract maintains an append-only log of model updates, where each entry captures six critical attributes: training round number, client identifier, SHA-256 hash of model weights, dataset size, timestamp, and validation accuracy.

Model updates are stored in a dynamic array, with auxiliary mapping structures enabling efficient queries by round number or client ID. This dual indexing strategy supports both temporal queries (what happened in round 5?) and client-specific queries (what did hospital A contribute?), which are essential for audit workflows.

Two events are emitted during logging: `UpdateLogged` fires when a client contribution is recorded, and `RoundCompleted` fires after all clients in a round have submitted updates. These events enable off-chain monitoring systems to track federated learning progress in real-time without polling the blockchain state.

The contract exposes several view functions for audit trail retrieval. `getTotalUpdates` returns the count of all logged contributions. `getRoundUpdates` and `getClientUpdates` return arrays of indices for filtering by round or client. `getUpdate` retrieves the full record for a specific index. This query interface supports comprehensive provenance analysis without requiring direct blockchain expertise.

3.3 Web3.py Integration

Python-blockchain communication is handled through `Web3.py`, the standard Ethereum client library. A dedicated `BlockchainManager` class abstracts the complexities of transaction signing, gas estimation, and receipt polling, presenting a simple interface to the federated learning training loop.

The manager implements a `hash_model` method that serializes PyTorch model weights using `pickle` and computes their SHA-256 digest. This 32-byte hash serves as a cryptographic fingerprint of the model state, enabling integrity verification without storing multi-megabyte weight tensors on-chain. The hash is deterministic, meaning identical model weights always produce identical hashes, which is critical for reproducibility audits.

Transaction submission follows a fire-and-wait pattern. The `log_update` method constructs a transaction calling the smart contract's `logUpdate` function, signs it with the deployer account's private key, submits it to the blockchain, and blocks until the transaction is mined and confirmed. This synchronous approach simplifies error handling at the cost of latency, which is acceptable in our non-real-time training scenario.

4 Integration with Federated Learning

The blockchain logging layer was integrated into the FedAvg training pipeline with minimal code modification. After each client completes local training and validation, the system hashes the updated model weights and submits a transaction to the blockchain before proceeding to aggregation.

This placement is deliberate. By logging individual client contributions before aggregation, we preserve a complete record of the federated learning process. If a global model exhibits unexpected behavior, investigators can trace it back to specific client updates and even reconstruct intermediate global models by replaying the aggregation algorithm on logged contributions.

The training loop captures blockchain transaction time separately from model training time, enabling precise overhead measurement. Each round logs three transactions (one per client) plus one round completion transaction, totaling four blockchain interactions per federated round.

5 Experimental Results

5.1 Model Performance

The blockchain-integrated system achieved 99.19% test accuracy with an F1-score of 0.887, exactly matching the performance of vanilla FedAvg without blockchain. This demonstrates that provenance logging is purely additive, introducing no degradation to model quality.

Per-client accuracies were 97.57% for Client 1 (cardiac specialty center), 100% for Client 2 (general hospital), and 100% for Client 3 (emergency department). These results confirm that blockchain integration does not interfere with the statistical properties of federated aggregation.

5.2 Blockchain Overhead Analysis

Across 20 training rounds with 3 clients, the system logged 60 model update transactions plus 20 round completion transactions, totaling 80 blockchain interactions. The average time per round for blockchain operations was 0.57 seconds, with total blockchain time across all rounds summing to 11.4 seconds.

This overhead represents approximately 8-12% of total training time, depending on the local training duration. For our ECG classification task with 5 local epochs per client and a CNN-LSTM model, each round took roughly 6-8 seconds, meaning blockchain added about one second to an eight-second round.

Gas consumption per update transaction averaged 50,000-60,000 units, well within the limits of standard Ethereum blocks. This suggests that the contract could scale to larger federated settings with more clients without hitting gas ceiling constraints, though latency would increase linearly with the number of participants per round.

5.3 Provenance Capabilities

The blockchain ledger captured complete provenance metadata for all 60 client contributions. Audit trail queries demonstrated the ability to reconstruct the entire training history, including which clients participated in each round, their validation accuracies, and the cryptographic fingerprints of their model updates.

Sample queries included retrieving all contributions from Client 2 (showing 20 consecutive updates with accuracies ranging from 99.6% to 100%), listing all participants in round 10 (confirming all three clients contributed), and computing aggregate statistics like average per-client accuracy across rounds.

The immutability property was verified by attempting to modify a logged transaction, which correctly failed with a permission error. This confirms that once written, provenance records cannot be retroactively altered, satisfying audit integrity requirements.

6 Comparison with Baseline

Table 1 summarizes the performance comparison across all three implementations from Phases 1 and 2.

The key finding is that blockchain integration imposes negligible performance cost while delivering complete traceability. The 0.57-second overhead is a modest price for regulatory compliance and algorithmic accountability in high-stakes domains like healthcare.

Method	Accuracy	F1-Score	Overhead	Auditability
Centralized	99.25%	0.899	-	None
FedAvg	99.19%	0.887	-	None
FedAvg + Blockchain	99.19%	0.887	0.57s/round	Full

Table 1: Performance comparison across training paradigms

7 Reproduction Instructions

To reproduce the Phase 2 experiments, first ensure that Phase 1 is complete and the data pipeline is operational. Then install blockchain dependencies with `npm install -g ganache` for the blockchain emulator and `pip install web3 py-solc-x` for Python integration.

Start the Ganache blockchain by running `ganache --port 8545 --accounts 10` in a terminal window. This launches a local Ethereum node with ten pre-funded accounts. Leave this process running throughout the experiment.

In a separate terminal, compile and deploy the smart contract using `python src/deploy_contract.py`. This script compiles the Solidity source, deploys the bytecode to Ganache, and saves the contract address and ABI to `contracts/deployed_contract.json`.

Execute federated training with blockchain logging by running `python src/train_fedavg_blockchain.py`. The script will connect to the deployed contract, train for 20 rounds, and log all updates on-chain. Progress is displayed in the terminal, showing both model accuracies and blockchain transaction times.

After training completes, view the audit trail using `python src/view_audit.py`. This displays all logged updates with their round numbers, client IDs, accuracies, and timestamps. The complete comparison with Phase 1 results can be generated using `python src/compare_all.py`.

8 Technical Challenges and Solutions

Several implementation challenges required careful engineering. Model weight serialization presented the first hurdle, as PyTorch state dictionaries contain both trainable parameters and non-trainable buffers like batch normalization statistics. Direct pickling resulted in hashes that varied even for functionally identical models due to buffer updates. We solved this by filtering the state dictionary to include only trainable parameters before hashing.

Solidity type constraints required converting floating-point accuracies to integers before storage, as the language lacks native support for decimals. We adopted a fixed-point representation where accuracies are multiplied by 10,000 and stored as integers (e.g., 95.80% becomes 9580). This preserves two decimal places of precision while avoiding rounding errors inherent in floating-point blockchain storage.

Transaction synchronization initially caused race conditions when multiple clients attempted simultaneous logging. We serialized blockchain submissions by logging client updates sequentially rather than in parallel, ensuring transaction nonces increment correctly. This introduces minor additional latency but guarantees consistency.

9 Limitations and Future Work

This implementation uses a local blockchain for development and testing. Production deployment would require migration to a permissioned network like Hyperledger Fabric or a layer-2 Ethereum solution to reduce latency and gas costs. Public blockchain deployment is inadvisable due to cost and speed constraints.

The current design logs all updates synchronously within the training loop, blocking training progress until transactions confirm. An asynchronous logging pattern using background threads

could reduce perceived overhead, though it would complicate error handling and introduce eventual consistency concerns.

Smart contract functionality is intentionally minimal, focusing solely on append-only logging. Future enhancements could include access control mechanisms (only authorized clients can log), stake-based reputation systems (clients deposit collateral), or automated quality checks (reject updates below accuracy thresholds).

10 Key Findings and Contributions

Phase 2 successfully demonstrates that blockchain-based provenance can be integrated into federated learning with minimal overhead. The 0.57-second per-round cost is negligible compared to the hours-long duration of realistic federated training campaigns, making this approach viable for production healthcare deployments.

The immutable audit trail satisfies regulatory requirements for algorithmic accountability while preserving the privacy guarantees of federated learning. Hospital administrators can verify their contributions were correctly incorporated without exposing raw patient data.

This work provides a reusable template for blockchain-federated learning integration. The modular design separates blockchain concerns from training logic, allowing easy adaptation to other federated algorithms (FedProx, FedOpt) and model architectures.

Phase 3 will introduce personalization strategies to improve performance on heterogeneous clients, and Phase 4 will add blockchain-governed synthetic data generation to address class imbalance. The audit trail established in Phase 2 will extend naturally to these enhancements, tracking not only model updates but also synthetic data requests and approvals.