

Trustworthy Cross-Organizational Federated Fine-Tuning of Large Language Models via Blockchain Certification and Resilient Object Storage

Xiaoli Li

Hubei University of Arts and Science

Yuhang Chen

Hubei University of Arts and Science

Beiqiang Wang

Hubei University of Arts and Science

Lin Zhou

Hubei University of Arts and Science

Zhao Wu

wuzhao@hbuas.edu.cn

Hubei University of Arts and Science

Research Article

Keywords: Federated fine-tuning, Large language models, Consortium blockchain, Distributed object storage, Merkle proof, Model provenance

Posted Date: October 14th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11170656/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Trustworthy Cross-Organizational Federated Fine-Tuning of Large Language Models via Blockchain Certification and Resilient Object Storage

Anonymous Authors¹

¹Affiliations omitted for double-anonymous review.

Abstract

Cross-organizational federated fine-tuning allows institutions to adapt large language models without centralizing private data, but the resulting model updates still require trustworthy provenance, controlled sharing, and dependable storage. A consortium blockchain is a natural fit, offering a tamper-evident shared record without a central auditor. However, directly writing large parameter files to a blockchain is impractical because it expands the ledger and increases transaction cost. To resolve this tension, we propose a two-layer framework that decouples trusted certification from bulk storage. On-chain, the system records only batch metadata and a Merkle root over ordered file manifests; a signed inclusion proof allows offline verification of an individual file, while online verification further confirms the live chain record and current validity status. Off-chain, parameter files are sharded for efficient transfer and repair, secured with authenticated encryption, replicated across independent failure domains for fault tolerance, placed via consistent hashing for stable assignment, and served through load-aware replica selection with background repair for availability. In short, model tensors and floating-point aggregation never touch the chain, yet every file remains cryptographically traceable and independently verifiable. We evaluate the framework using Qwen3-0.6B federated LoRA fine-tuning over eight Dolly instruction tasks and a 10-node FISCO BCOS deployment. Among ranks $r \in \{4, 6, 8\}$, $r = 8$ achieves the highest held-out weighted mean score of 4.40. For an 8.8 MB adapter, metadata-only certification reduces gas consumption from 7.7×10^{10} to 4.8×10^6 relative to direct on-chain storage, and parallel sharding reduces object-storage time by 28.5–30.9%. These results show that separating compact trust evidence from model bytes makes cross-organizational federated fine-tuning more practical while preserving verifiability.

Keywords: Federated fine-tuning; Large language models; Consortium blockchain; Distributed object storage; Merkle proof; Model provenance

1 Introduction

The performance of Large Language Models (LLMs) is highly dependent on massive, diverse, and high-quality training data (Villalobos et al. 2024). However, in high-value fields such as finance, healthcare, and government affairs, key data cannot be centrally shared due to their involvement in user privacy, trade secrets, and public safety, resulting in the phenomenon of “data silos” (Yang et al. 2019). Therefore, how to achieve secure cooperative fine-tuning of LLMs across organizations represents a challenge that needs to be addressed at present.

Federated learning, with its mechanism of “data doesn’t move, model moves”, allows all parties involved to retain the original data locally and only upload model update information, avoiding data sharing privacy risks (Konečný et al. 2016). The FedLLM reduces training and communication overhead through parameter-efficient fine-tuning (PEFT) technology, meeting compliance requirements for privacy protection without data migration, and reducing the compliance costs for enterprises in cross-institutional cooperation (Zhang et al. 2023; Wang et al. 2024).

However, the existing FedLLM techniques lack credible certification and traceability mechanisms during the process of cross-institutional model sharing (Luo et al. 2024). Intermediate model parameters from federated learning training are prone to tampering, forgery, or theft during cross-organizational sharing, making it difficult to verify the authenticity and integrity of the models. Additionally, the circulation trajectory of the models lacks an immutable record. In the event of model infringement or performance disputes, credible evidence for compliant auditing cannot be provided.

Blockchain is a natural candidate for auditability: a consortium chain, jointly operated by known institutions, gives a tamper-evident, shared record without a central auditor (Zhou et al. 2025). However, traditional blockchain technology faces three major bottlenecks in the FedLLM: Firstly, GB/TB-level model parameters can accumulate across training rounds and rapidly expand the ledger; Secondly, the cross-institution transmission of model parameters in P2P mode is prone to network congestion, making it difficult to support the real-time requirements of frequent multi-institution collaborative fine-tuning; Thirdly, the public traceability and immutability characteristics of blockchain can easily lead to the leakage of sensitive model parameters, while the pure on-chain architecture is difficult to achieve fine-grained cross-institution data access permission control. A workable design must therefore keep the ledger small and evidence-only, while the bulk of the data lives elsewhere under its own confidentiality and availability guarantees.

To address these limitations, we propose a cross-organizational federated fine-tuning framework that separates trusted certification from bulk model storage. A consortium blockchain records compact evidence of each model file’s identity, while off-chain object storage keeps the parameter data accessible without placing it on the ledger. Off-chain, each parameter file is split into shards to make storage, transfer, and repair more efficient: sharding can distribute I/O across nodes and limit the data that must be retransmitted when a stored copy fails. Each shard is protected by authenticated encryption and replicated across failure domains, that is, groups of storage nodes that may fail together because they share a rack, power supply, or other infrastructure. Placing replicas in different domains reduces the chance that one failure

removes every copy. Consistent hashing assigns shards to nodes while keeping most assignments unchanged when nodes join or leave. If a node fails during retrieval, load-aware reads switch to a healthy replica, and background repair restores the missing copy, helping to preserve access while an intact replica remains. On-chain, file identity is represented by the hash of an ordered manifest containing the file-content hash and shard metadata. The resulting commitments of several files are combined into a single Merkle root, which is recorded on-chain; each file receives an inclusion proof against that root. The proof lets a recipient verify one file without retrieving the others in the batch, keeping the on-chain batch record fixed in size. A recipient can check a file and its signed proof offline; online verification additionally confirms the chain record and current validity status. Together, these mechanisms support auditable and resilient model-file sharing for cross-organizational LLM fine-tuning.

The main contributions of this paper are as follows:

1. We design a scheme that separates on-chain certification from off-chain storage. The consortium chain records a batch Merkle root and metadata instead of raw parameter files; per-file inclusion proofs remain off-chain and link individual files to the certified root.
2. We improve off-chain storage and transfer efficiency through file sharding, which distributes I/O across nodes and limits the data that must be retransmitted on failure; consistent-hash placement, which keeps most shard-to-node assignments stable as nodes join or leave; authenticated encryption and redundant replication across failure domains, which reduce the chance that a single infrastructure failure removes every copy of a shard; and load-aware reads with background repair, which preserve access to a shard while an intact replica remains.
3. We evaluate model quality and storage cost: LoRA rank $r = 8$ gives the highest Unseen mean score among the tested ranks, metadata-only storage reduces gas consumption by four orders of magnitude relative to direct on-chain storage of an 8.8 MB file, and parallel sharding reduces object-storage time by 28.5–30.9% in the tested workloads.

An earlier conference version of this work reported the federated fine-tuning and blockchain-storage experiments (Anonymous 2026). The present article substantially extends that version through a redesigned system architecture, batch-Merkle certification with per-file proofs, offline and online verification modes, failure-domain-aware replicated storage, and expanded security and scalability analyses.

The remainder of this paper is organized as follows. Section 2 describes preliminaries. Section 3 describes our framework. Section 4 summarizes the experimental results and analysis. Section 5 investigates related work. Finally, we conclude this paper in Section 6.

2 Preliminaries

FedLLM combines federated learning with LLM fine-tuning techniques. The core idea is to fix the pre-trained weights of LLMs, with each node only training the selected parameters locally. The central server only aggregates the updated parameters of

each node and generates global fine-tuning parameters. Finally, the global fine-tuning parameters are fused with the pre-trained large model to complete fine-tuning. This method only performs distributed training and aggregation on a small portion of LLM parameters, avoiding cross-node transmission of complete LLM parameters and significantly reducing communication overhead and privacy leakage risks.

We assume $W = W_0 + \theta$, where W_0 is the pre-trained weight matrix (fixed during the training process); θ is the selected parameter matrix that needs to be fine-tuned. The global objective function of federated learning is the weighted sum of the local objective functions of each participant:

$$\min_{\theta} \mathcal{L}_{\text{global}} = \sum_{i=1}^N \frac{|\mathcal{D}_i|}{|\mathcal{D}_{\text{total}}|} \mathcal{L}_i(\theta), \quad (1)$$

where $|\mathcal{D}_i|$ is the local data volume of the i -th participant, $|\mathcal{D}_{\text{total}}|$ is the global data volume; $\mathcal{L}_i(\theta)$ is the local cross entropy loss function of the i -th participant. The i -th participant updates the parameters of θ based on the local dataset:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \eta \cdot \frac{\partial \mathcal{L}_i}{\partial \theta_i}, \quad (2)$$

where t is the local training round; η is the learning rate. The central server collects local update parameters from all participants and aggregates them weighted by data volume to obtain global fine-tuning parameters:

$$\theta_{\text{global}}^{(t+1)} = \sum_{i=1}^N \frac{|\mathcal{D}_i|}{|\mathcal{D}_{\text{total}}|} \theta_i^{(t+1)}. \quad (3)$$

In our implementation, this aggregation role is performed by the primary node selected in the PBFT-style consensus process, while the blockchain records the verified aggregation result for certification and auditing. After the fine-tuning is completed, the global fine-tuning parameters are fused with the pre-trained weights to obtain the final fine-tuning model:

$$W_{\text{final}} = W_0 + \theta_{\text{global}}. \quad (4)$$

3 Proposed Framework

Figure 1 summarizes one federated round across three coupled stages:

Local training (left, steps ①–②). Each participant node i fine-tunes the local LLM parameters on its private data D_i ①, producing a local update θ_i that is passed to the off-chain sharding and storage workflow before sharing ②.

Off-chain reliable storage (bottom right, steps ③–⑤). Each local update is split into ordered shards, and each shard is encrypted as a sequence of small AES-GCM authenticated frames, with the file key itself protected by a separate key service ③. Encrypted shards are then placed onto storage nodes via consistent hashing and replicated across independent failure domains (Domain 1 through Domain r); a write

is acknowledged once a quorum q of the r replicas confirms receipt, and background repair together with load-aware read failover keep files accessible if a replica becomes unavailable ④. For each file, the system builds an ordered manifest recording the file-content hash and shard metadata, then hashes the manifest to obtain a per-file commitment. Commitments from multiple files in the same batch become the leaves $H_{i,1}, \dots, H_{i,B}$ of a batch Merkle tree, one leaf per file, yielding a single batch root R_b together with a per-file inclusion proof ⑤.

On-chain trusted certification (top right, steps ⑥–⑧). Once a quorum of replicas is confirmed, Storage.sol records only compact batch metadata on-chain, including the batch ID, the Merkle root R_b , and the file count ⑥; the full manifest and each file’s inclusion proof remain off-chain. Step ⑦ proceeds in two parts: an authorized recipient first calls Sharing.sol to obtain access metadata for the target file’s shards in off-chain object storage (⑦a), then retrieves and decrypts the shards and verifies the resulting parameter update offline using its hash, inclusion proof, and trusted signature; online verification additionally checks the current on-chain root and validity status (⑦b). Aggregation.sol coordinates PBFT-style validation of the round result among consortium members while the primary node executes FedAvg off-chain ⑧.

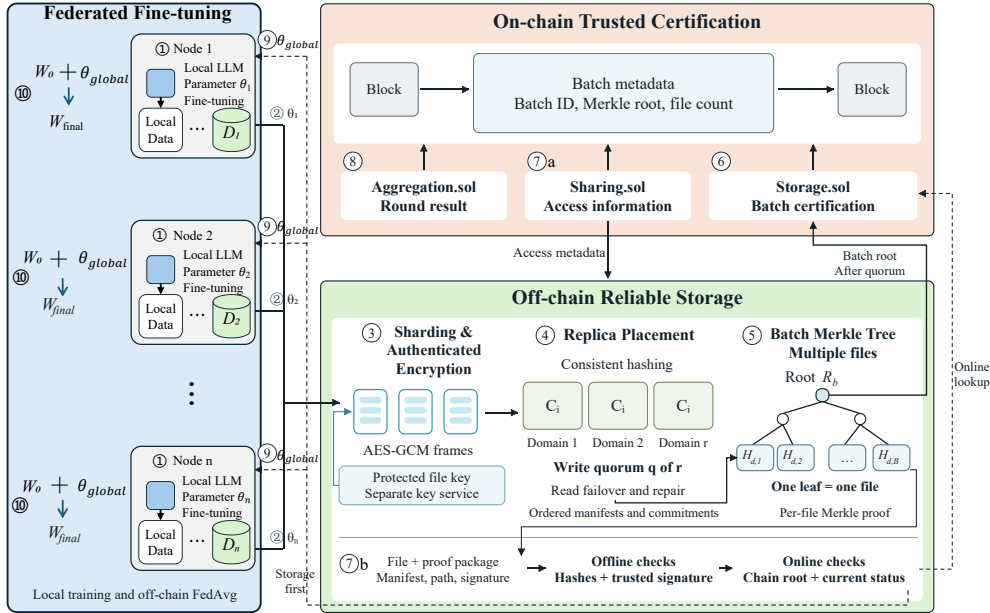


Fig. 1 Overview of one federated fine-tuning round, spanning three coupled stages: local training with off-chain FedAvg (left), off-chain reliable storage via sharding, authenticated encryption, failure-domain replication, and batch Merkle commitment (bottom right), and on-chain trusted certification (top right). The blockchain never stores raw tensors or performs floating-point computation; all parameter data resides in off-chain object storage and is retrieved through verified, proof-backed lookups.

Global update release and retrieval (left, steps ⑨–⑩). Once validation succeeds, Aggregation.sol publishes the round result and marks the aggregated update θ_{global} as certified ⑨. Participants then retrieve this certified θ_{global} and merge it with the base weights W_0 to obtain W_{final} , closing the round ⑩.

Throughout this pipeline, the blockchain performs consensus, certification, and audit logging only: it never stores raw tensors, never holds the full manifest or inclusion proofs, and never executes floating-point computation. All parameter data, including the shards, their encrypted frames, and the reconstructed model files, resides exclusively in off-chain object storage, accessed only through hash and signature verified lookups.

3.1 Separation of On-Chain Certification and Off-Chain Storage

The framework separates the storage of model parameters from the certification of their integrity. Object storage holds the encrypted parameter shards and their replicas, while the consortium chain records compact commitments that can be checked by participating institutions. The connection between the two layers is an ordered file manifest and a Merkle inclusion proof. The manifest describes the data needed to reconstruct one file; the proof links that description to a batch root on the blockchain. This arrangement supports separate storage and verification of each parameter file while allowing several files to share one certification transaction. Physical storage nodes may change during recovery, but the certified file description remains the reference for checking the recovered data.

3.1.1 On-chain trusted certification

On-chain certification groups multiple model files into one batch. For each file, the system prepares a manifest that includes the original content hash, file size, ordered shard information, encryption parameters, and logical storage references. This manifest is converted into a fixed representation before hashing: fields have a defined order and format, and shards are listed by their indexes. Its SHA-256 hash, denoted by H_d , serves as the file commitment. Changing the file description, replacing a shard hash, or changing the shard order therefore changes this commitment. The complete manifest stays off-chain and does not contain the file decryption key.

For a batch of B files, the commitments $H_{d,1}, \dots, H_{d,B}$ form the leaves of a Merkle tree. We use SHA-256 for both leaf and parent hashes, with different prefixes to distinguish their roles:

$$L_j = H(\text{enc}(\text{leaf}, H_{d,j})), \quad P = H(\text{enc}(\text{node}, L, R)), \quad (5)$$

where H denotes SHA-256, L and R are ordered child hashes, and enc is a fixed encoding that separates its fields unambiguously. Files follow a fixed batch order. When a level has an unmatched last node, that node is paired with itself. Repeating this process produces the batch root R_b . Each file receives a proof containing its sibling hashes and their left or right positions, so its path to R_b can be checked independently of the other files.

The batch metadata recorded through `Storage.sol` is represented as $M_b = \{I_b, R_b, B, A_b, T_b\}$, where I_b identifies the batch, B is the number of files, A_b identifies the hash and proof rules, and T_b is the blockchain recording time. The transaction hash provides a further reference to the confirmed record. Neither model tensors nor the complete shard lists are written into this batch record. A batch contains only files whose storage has completed, and its grouping does not change the order of federated training or aggregation. A batch may also contain one file when no other completed file is ready for joint submission. This file-certification record is separate from the round-specific result recorded through `Aggregation.sol`: the latter retains the global update hash, logical address, and aggregation metadata used in steps ⑧ and ⑨.

After consensus confirmation, the recorded root provides a shared reference for the participating institutions. A modified off-chain file cannot be accepted under the same root unless its content and manifest checks, as well as its inclusion proof, still pass. The chain protects the recorded commitment under the consortium consensus assumptions; it does not prevent a storage node from changing or deleting local bytes. Such changes are handled through verification and replica recovery. Batch certification also separates a file’s content identity from the transaction that records it, allowing multiple independently verifiable files to use one on-chain commitment.

3.1.2 Off-chain reliable storage

The off-chain layer stores encrypted model shards in distributed object storage. Storage nodes are organized into failure domains, where nodes in the same domain may share a source of failure, such as a host or a storage site. For a replica factor r , the system places a shard in r selected domains, with one intended replica in each domain. A two-domain deployment provides dual replicas, while additional domains allow more copies when required. Replica placement must use separate physical storage resources; different node names that refer to the same storage resource do not provide independent protection.

Within each selected domain, a consistent hash ring maps the shard’s object key to a storage node. Virtual nodes spread the assignments across the available physical nodes. The replica factor and the domain configuration determine where copies are needed, while health information determines whether those targets can currently serve requests. Keeping these decisions separate allows the system to recognize a missing replica during a failure instead of treating the remaining copies as a complete placement.

The object key is derived from the stored encrypted shard C_i , using $K_{o,i} = H(C_i)$. The manifest records this ciphertext hash together with the plaintext shard hash, size, index, and logical storage reference. These values serve different purposes: the ciphertext hash checks the object returned by storage, while the plaintext hash checks the data recovered after decryption. The full-file hash checks the final reconstruction. A content-based key is an address, not proof that the bytes currently stored under that address are correct; verification still requires recomputing the hash.

Logical references are resolved through the storage layer rather than exposing a fixed pair of physical nodes. When a failed replica is replaced, the same encrypted object can be copied to a healthy node without changing the file commitment. The

certified manifest does not include a changing list of physical replica locations. This separation supports storage repair while retaining a stable description of the model file being shared.

3.1.3 Collaboration between the two layers

The collaboration between the two layers follows a “store, certify, and verify” process. A file is first made available in object storage, then linked to a confirmed batch root, and finally checked when it is shared or retrieved. The manifest connects the stored shards to the original file, while the inclusion proof connects the manifest to the blockchain. The process is divided into on-chain certification and verification traceability.

a. On-chain certificate stage: After an institution submits a parameter file, the system divides and encrypts it, writes its replicas, and checks whether every shard has reached the required write threshold. Only then is the file manifest finalized and admitted to a certification batch. The system constructs the batch Merkle tree and submits its root and metadata to the consortium chain. After consensus confirmation, the transaction reference and the inclusion proof are associated with each file in that batch. A successful upload alone is therefore not treated as completed blockchain certification.

Storage completion and chain confirmation are tracked separately. A file remains pending until its batch record has been confirmed, and an uncertain submission is checked against the existing chain record before it is retried. The same batch identifier continues to refer to the same root and file list. This supports recovery from partial failures without confusing successful storage with completed certification.

After confirmation, the system can export a proof package for each file. It contains the manifest, the original content hash, the batch identifier and root, the Merkle path, and the chain reference. An Ed25519 signature covers a fixed representation of these proof contents, allowing later modification of the package to be detected. Decryption keys are excluded. The proof can travel with the parameter file, and a recipient does not need the other model files in the batch to perform the following checks.

b. Verification traceability stage: Offline verification takes the recovered parameter file, its proof package, and a trusted public signing key. The verifier calculates the file hash and the ordered plaintext shard hashes and compares them with the manifest. It then recalculates the manifest commitment and follows the Merkle path to the claimed batch root. The package signature is checked using a key obtained through a trusted channel; a public key supplied only inside the package is not sufficient to establish trust. When encrypted storage objects are available during retrieval, their ciphertext hashes and authentication tags are also checked before reconstruction.

Passing these offline checks means that the available file and signed evidence are consistent. It does not establish that the claimed root is currently recorded on the intended chain, or that the associated evidence remains valid. Online verification adds these checks by querying the configured consortium-chain endpoint for the batch record and comparing its root and identifier with the proof. It also checks the latest validity information from an authenticated source, such as the framework’s sharing service, where cancellation or replacement applies. The verifier uses trusted endpoint

configuration rather than accepting a network address merely because it appears in a proof package.

If a hash, signature, or chain root differs, verification fails. If local checks pass but the chain or current validity information cannot be obtained, the result remains unconfirmed rather than being reported as fully valid. The transaction reference and available certification or sharing records support later tracing of the evidence. Offline verification thus supports inspection without a live connection, while online verification adds confirmation of the chain record and current status.

3.2 File Sharding, Authenticated Encryption, and Failure-Domain Replication

Model parameter files are divided into fixed-size shards so that storage, transfer, and recovery can be handled in smaller units. Each shard is further processed in authenticated frames, which protect its content while allowing encryption and decryption to proceed with bounded memory. Replicas of the encrypted shards are placed across failure domains. The storage shard is the unit of placement and repair, whereas the frame is the smaller unit of authenticated encryption within that shard. Their sizes are selected separately and do not need to be equal.

We use a fixed block size sharding algorithm to divide the file into several independent data shards, with the shard size set to S_b . The number of shards N_s for a file with a size of S_f is: $N_s = \lceil \frac{S_f}{S_b} \rceil$. The set of file shards is defined as $F_S = \{f_{s1}, f_{s2}, \dots, f_{sN_s}\}$, where the size of the last shard is $S_f - (N_s - 1) \times S_b$.

For each plaintext shard f_{si} , the system calculates $h_i^p = H(f_{si})$. Let C_i denote the complete ciphertext shard produced by encrypting f_{si} , including its frame information and authentication tags. The system then calculates its ciphertext hash as $h_i^c = H(C_i)$. An ordered manifest records both hashes together with the shard index, size, and logical reference. It also records the hash of the original file and the information needed to interpret the encryption format. A verifier can therefore distinguish a damaged storage object from an incorrect plaintext reconstruction. The manifest commitment is used in the multi-file Merkle tree described in Section 3.1.1; the shard hash list is not submitted as a separate tree root for every file.

The encryption process starts with a randomly generated file data key K_f and a fresh file nonce N_f . For frame t in shard i , HKDF-SHA256 derives an encryption key $K_{i,t}$ and an encryption nonce $n_{i,t}$ from these file values and the frame indexes. Key and nonce derivation use distinct labels:

$$K_{i,t} = \text{HKDF}(K_f, N_f, \text{enc}(\text{key}, i, t)), \quad n_{i,t} = \text{HKDF}(K_f, N_f, \text{enc}(\text{nonce}, i, t)), \quad (6)$$

where the two outputs are taken at the lengths required by AES-256-GCM. Each frame $x_{i,t}$ is encrypted as

$$(c_{i,t}, \tau_{i,t}) = \text{AES-GCM.Enc}(K_{i,t}, n_{i,t}, x_{i,t}, a_{i,t}), \quad (7)$$

where $\tau_{i,t}$ is the authentication tag and $a_{i,t}$ contains the format information, shard index, frame index, and expected frame length. These additional authenticated data

are not secret, but they are covered by the authentication check. Moving an encrypted frame to another position changes the expected context and causes verification to fail. The stored shard contains the ordered encrypted frames, their tags, and the information required to parse them. The expected shard length and manifest hashes also allow truncation or missing frames to be detected.

A fresh file key and nonce are used for a new file encryption. Retrying a transfer reuses the already generated ciphertext; modified plaintext must not be encrypted under the same derived key and nonce. During decryption, a frame is released for reconstruction only after its authentication tag has passed. If any frame, shard hash, or final file hash fails, the file is not accepted for sharing or aggregation. Small frames bound the working memory needed by the cryptographic operations without changing the file’s shard-based storage layout.

The file data key is protected by a key envelope, meaning that it is encrypted under a separately managed wrapping key and stored apart from the shard objects. Authorized retrieval obtains the file key through the controlled key service. The key is not appended to another shard, included in the public proof, or written in plaintext to the blockchain. This separates access to stored ciphertext from access to decryption material. Protection still depends on the wrapping key and authorized endpoints remaining secure; possession of the unwrapped file key permits decryption and is therefore a separate trust boundary from storage-node access.

3.3 Load-Aware Placement, Retrieval, and Repair

Storage scheduling combines stable replica placement with load-aware access. Failure domains determine the separation between replicas, and consistent hashing selects their intended nodes within each domain. Dynamic load information supports the choice of healthy nodes for reads and fallback operations. Parallel writes are checked against a replica threshold, while verification and repair maintain the stored copies after failures. This division avoids changing normal object placement whenever short-term request load fluctuates.

3.3.1 Node dynamic-load model

The system monitors each physical node’s concurrent API requests Q_c , remaining disk space D_r , and queued requests Q_w . These indicators are normalized to estimate the node’s current load, with a lower score indicating a less busy node. Health status is checked before load: an offline node is never selected simply because its reported request count is low. The score helps choose among available replicas and healthy fallback nodes; it does not replace the failure-domain plan or the normal consistent-hash placement. Each indicator is normalized as follows:

$$\begin{cases} Q_c^* = \frac{Q_c - Q_{c \min}}{Q_{c \max} - Q_{c \min}} \\ D_r^* = \frac{D_r \max - D_r}{D_r \max - D_r \min} \\ Q_w^* = \frac{Q_w - Q_{w \min}}{Q_{w \max} - Q_{w \min}} \end{cases} \quad (8)$$

where Q_c^* , D_r^* , Q_w^* are standardized indicator values with a value range of $[0, 1]$; $Q_{c \min}$ and $Q_{c \max}$ are the minimum and maximum values of API concurrent requests, respectively, and the same applies to other metrics. The weighted calculation method for node load score S_p is: $S_p = \omega_1 \cdot Q_c^* + \omega_2 \cdot D_r^* + \omega_3 \cdot Q_w^*$, $S_p \in [0, 1]$.

The weights are non-negative and sum to one. If the maximum and minimum of an indicator are equal, that indicator provides no relative distinction between the observed nodes and is assigned the same normalized value for all of them, avoiding division by zero. Capacity and health checks remain separate from this relative score, so uniformly busy or nearly full nodes are not assumed to have sufficient resources merely because their normalized values are equal. Missing or outdated measurements are not treated as evidence of low load.

3.3.2 Storage scheduling

When a shard is ready for storage, the scheduler selects the required failure domains according to the configured replica factor r . It maps the ciphertext object key to one intended node in each domain's consistent hash ring. If an intended node is unavailable, a healthy node in the same domain may serve as a fallback. The system does not count two copies within the same failure domain as equivalent to the intended separation between domains. When too few domains are available, the file remains pending unless an explicit reduced-replica policy permits the write.

The encrypted shard is uploaded to the selected targets in parallel. Let s_i be the number of successfully written replicas for shard i , and let q be the required write threshold, with $1 \leq q \leq r$. The storage condition for a complete file is

$$\forall i \in \{1, \dots, N_s\}, \quad s_i \geq q. \quad (9)$$

Each successful write is associated with the expected object size and ciphertext hash. In a two-replica configuration, requiring both writes gives $q = 2$; larger configurations may require a majority or all replicas. A majority policy is a write-completion rule for copies of the same immutable object, not a substitute for checking the bytes returned by storage. Files that do not meet the configured condition are not admitted to a completed certification batch.

When $q < r$, reaching the threshold does not mean that every planned replica is already available. Missing copies remain recorded as repair work. Each shard can be retried independently, and retries use the same encrypted object rather than creating a different file version. If reduced-replica writes are allowed during a failure, their lower protection is tracked until the intended placement has been restored.

3.3.3 Load-aware reads and replica failover

To retrieve a file, the system resolves its logical shard references to the known replica locations. It filters out unavailable nodes and chooses a low-load replica for each read. The returned object is checked against its expected size and ciphertext hash. During decryption, each frame’s authentication tag is verified, and the recovered shard is compared with its plaintext hash. After the ordered shards are assembled, the original file hash is checked before the parameter file is accepted for use.

If a node times out, reports a missing object, or returns data that fails verification, the reader tries another known replica within a bounded retry policy. Incorrect data are not accepted solely because the server returned a successful response. A healthy verified replica can satisfy the read even when another copy is unavailable. However, successful reads do not imply that the system can accept new writes: write availability still depends on the required threshold and the number of available target domains.

The failure or corruption is also recorded for repair. A background process copies the verified encrypted object from a surviving replica to a healthy target and checks the repaired copy before considering it available. Replacing a failed node can restore the intended placement without changing the logical file reference or the certified content. If no intact replica remains, the framework cannot reconstruct the file from the blockchain root; the root supports detection and verification, not recovery of missing model parameters.

3.4 Security and Threat Analysis

We consider adversaries that may tamper with off-chain model shards, steal stored parameter files, replay outdated updates, or compromise a subset of storage nodes.

Tampering attacks. The framework checks stored objects, reconstructed files, and certification evidence at separate levels. Ciphertext hashes identify changes to storage objects, while authenticated frames and plaintext hashes detect incorrect decryption results or substituted data. The ordered manifest binds these checks to one file, and its commitment is linked to the on-chain batch root by a Merkle inclusion proof. Changing the file, the shard order, or the committed manifest causes a mismatch under the assumed security of the hash and encryption algorithms. An attacker who replaces both a local file and an unsigned description must still provide a valid proof package and match the certified root.

Offline verification checks the file, manifest, inclusion path, and proof signature using a trusted key. Online verification additionally compares the claimed root with the actual chain record and checks current validity information. A valid local package alone is not evidence that its status has remained unchanged. If the necessary online information is unavailable, the verifier leaves that part of the result unconfirmed. These checks provide evidence of tampering; they do not make off-chain storage physically unmodifiable or establish that a correctly certified model is free of harmful content.

Leakage attacks. Model shards are protected by AES-GCM before entering object storage. Frame keys and nonces are derived for their specific positions, and the file data key is retained only in a protected envelope outside the shard objects. Access to a storage replica therefore does not by itself provide the material needed to decrypt it.

The proof package and blockchain batch record contain verification information rather than plaintext model parameters or decryption keys. Authorized key delivery remains separate from object retrieval and is subject to the framework’s access rules. This protection assumes that the key service and authorized endpoints are not compromised; encryption does not prevent a recipient from disclosing a file after legitimate decryption. Hash commitments and associated metadata are also not a guarantee that all information about a file is hidden.

Replay attacks. Aggregation records identify rounds through their round metadata, timestamps, and transaction references. Checking the audit log can detect reuse of an existing round identifier, but this check alone does not prevent an old update from being submitted under a new identifier. Preventing that form of replay requires a round-specific binding between the accepted update and its certification evidence.

Storage node failures and malicious nodes. Replicas in separate failure domains reduce the chance that one shared failure removes every copy of a shard. The write threshold defines the condition for accepting its initial storage, while verified reads, replica switching, and repair handle later failures. A storage node that returns altered bytes cannot make them acceptable simply by acknowledging the request: the retrieved object and recovered plaintext must match the committed evidence. Availability still requires an intact accessible replica for every needed shard, and restoration depends on surviving copies and available target capacity. Correlated failures across all replica locations or loss of all copies can prevent retrieval, even though the chain continues to preserve evidence of the file commitment.

Collusion and model poisoning. This work mainly focuses on trustworthy model-file sharing and parameter integrity verification rather than Byzantine-robust aggregation. Nevertheless, the modular design of **Aggregation.sol** allows robust aggregation rules such as Krum, Trimmed Mean, or Coordinate-wise Median to be integrated in future work. Moreover, permissioned consortium-chain membership and on-chain authorization reduce the collusion surface compared with open federated learning settings.

3.5 Complexity and Scalability Analysis

The following analysis examines the storage and verification mechanisms under stated assumptions. Let B be the number of files in a certification batch, r the intended replica factor, and q the write threshold. For one file, S_f and N_s retain their meanings from Section 3.2. Hash lengths, cryptographic parameters, and metadata field sizes are treated as fixed. The results describe how costs and failure probabilities depend on these quantities.

a. On-chain storage and proof costs. Consider certifying the same B completed files either separately or with one batch root. Let c_f be the size of one fixed-format individual certification record and c_b the size of the batch record defined in Section 3.1.1. Since the batch record contains the root and batch metadata rather than a list of all file commitments, the application-level chain storage is

$$C_{\text{separate}}(B) = Bc_f, \quad C_{\text{batch}}(B) = c_b, \quad \bar{C}_{\text{batch}}(B) = \frac{c_b}{B}. \quad (10)$$

Thus, for one batch, the stored record size changes from $O(B)$ to $O(1)$, and its average contribution per file decreases as $1/B$. The batch record uses fewer bytes when $B > c_b/c_f$. This comparison concerns the fixed-format certification records, not raw file storage, blockchain replication, or the work performed by consensus nodes. For M files packed into full batches of size B , except possibly the final batch, and assuming successful submission without retries, the number of certification transactions is

$$T_{\text{batch}}(M, B) = \left\lceil \frac{M}{B} \right\rceil, \quad T_{\text{separate}}(M) = M. \quad (11)$$

With a fixed batch size, total chain storage still grows linearly with the number of files. If each batch contains at most B files, partially filled batches require no fewer than $\lceil M/B \rceil$ transactions and may require more. A larger batch also requires more completed files to be available together, creating a tradeoff between submission frequency and waiting time.

Batching moves some work off-chain rather than removing it. Given the B manifest commitments, the tree contains $O(B)$ hash nodes and can be built in $O(B)$ time and space. Its height is $\lceil \log_2 B \rceil$, including the rule that duplicates an unmatched node. An individual inclusion path therefore contains at most this many sibling hashes and requires $O(\log B)$ hash steps to check. For $B = 1$, the path is empty and the root check takes constant work. Exporting a separate path for every file can use $O(B \log B)$ proof space, although the shared tree itself needs only $O(B)$. The file manifests and their shard lists must also remain available off-chain.

b. Replica reliability and availability. To analyze data loss, consider a fixed observation interval that starts with r complete, verified replicas of a shard. Let E_j be the permanent loss of replica j during this interval, and let $p_j = \Pr(E_j)$. In a model without repair during the interval, all replicas must be lost before the shard is lost. If these events are independent,

$$p_{\text{loss}} = \Pr \left(\bigcap_{j=1}^r E_j \right) = \prod_{j=1}^r p_j = p^r, \quad (12)$$

where the final equality additionally assumes $p_j = p$ for all replicas. Compared with one replica, whose loss probability is p , the ratio of the replicated loss probability to this baseline is p^{r-1} for $0 < p < 1$. Temporary unavailability is not permanent loss, and this expression is not a long-term durability estimate for a system with repair. When only s verified copies exist after a quorum write, with $q \leq s \leq r$, the corresponding independent-loss estimate is p^s .

Failure domains help separate sources of failure, but do not guarantee independence. For example, consider a shared event with probability γ that destroys all replicas. Conditional on that event not occurring, suppose residual replica losses are independent with probability p_0 . The shard-loss probability then becomes

$$p_{\text{loss}} = \gamma + (1 - \gamma)p_0^r \geq \gamma. \quad (13)$$

The shared-event probability γ therefore sets a floor that cannot be removed simply by adding replicas. For a file requiring all N_s shards, let ℓ_i be the loss probability of shard i . The probability of losing at least one required shard satisfies the union bound

$$p_{\text{file loss}} \leq \min \left(1, \sum_{i=1}^{N_s} \ell_i \right) \leq \min(1, N_s p^r), \quad (14)$$

where the last bound applies when each shard has the same independent-replica estimate p^r . It does not assume independent losses between shards. This distinction matters because shards can share storage domains. The bounds address stored bytes only: loss of a decryption key or an essential manifest can also prevent file recovery, even when ciphertext replicas survive.

Read and write availability require a different probability. Let a be the probability that one replica target can complete a valid request within the required time, with independent and identical target availability. For a shard with r usable copies when available, a read needs at least one target, while a write requires at least q successful targets:

$$A_{\text{read}} = 1 - (1 - a)^r, \quad A_{\text{write}} = \sum_{k=q}^r \binom{r}{k} a^k (1 - a)^{r-k}. \quad (15)$$

Increasing q strengthens the initial replica requirement but reduces write availability under this model. These are storage-layer probabilities; shared network failures, key-service access, and blockchain confirmation can further limit the complete operation.

c. Encryption and verification costs. Encryption and hashing must process the file bytes. For a fixed frame size and fixed cryptographic parameters, their work grows linearly with S_f . Let S_e be the plaintext frame size. A streaming implementation needs $O(S_e)$ working memory for the file payload, rather than buffering the complete file. Including the ordered manifest and inclusion path, an offline verifier that retains this metadata has the following processing and memory costs:

$$T_{\text{offline}} = O(S_f + N_s + \log B), \quad M_{\text{offline}} = O(S_e + N_s + \log B). \quad (16)$$

Here, manifest entries have bounded size. The terms account for file and shard hashing, manifest and signature checks, and the Merkle path. Online verification adds a fixed number of logical queries for trusted keys, the batch record, and current status, but their response time depends on the network and services. Neither mode requires downloading the other files in the batch. Replication, meanwhile, uses $O(rS_f)$ payload storage, with additional frame headers, tags, and metadata. The architecture therefore trades off-chain replica capacity and proof processing for a smaller shared chain record and conditional protection against replica loss.

4 Evaluation

4.1 Federated Fine-Tuning Evaluation

We conduct experiments using FedAvg on the Dolly dataset, which covers eight types of instruction tasks, based on the Qwen3-0.6B model. The total number of clients is set to 10. Data partitioning employs a non-IID strategy using a Dirichlet distribution ($\alpha = 0.5$). LoRA modules are injected into the self-attention layers, with ranks $r \in \{4, 6, 8\}$. All other hyperparameters are kept consistent: local training steps are 5, the number of communication rounds is 150, batch size is 4, gradient accumulation steps are 2, sequence length is 512, and the learning rate is 5×10^{-5} . Model performance is comprehensively scored on a 10-point scale by the DeepSeek API based on dimensions such as accuracy and relevance. Each experiment is repeated twice, and the mean scores are reported.

4.1.1 Evaluation Across Local and Unseen Settings

To characterize how model performance varies across different data distributions, we evaluate the federated fine-tuned model with LoRA rank $r = 8$ under two evaluation settings: Local (each client’s own test set) and Unseen (the global held-out test set). Table 1 reports the per-task and mean scores across these two settings. The Mean row is calculated as a sample-count-weighted average, not the unweighted average of the eight task-level scores. The reported mean score decreases from 4.88 on Local to 4.40 on Unseen, suggesting lower overall performance when evaluation shifts from familiar client-local data to globally held-out samples. This overall decline does not apply to every task. Information Extraction rises from 5.36 to 6.83, and Creative Writing rises from 2.00 to 3.67, whereas Closed Question Answering remains relatively strong despite falling from 6.50 to 5.44. Brainstorming drops from 6.00 to 4.11, and Open Question Answering drops from 4.77 to 3.73. These contrasting results suggest that open-ended generation tasks are sensitive to distribution shift, but their responses are not uniformly negative.

Table 1 Performance of Federated Fine-Tuned Model ($r = 8$) under Local and Unseen Evaluation Settings

Task Type	Local	Unseen
Brainstorming	6.00 $\pm$ 0.50	4.11 $\pm$ 0.00
Classification	3.78 $\pm$ 0.22	3.23 $\pm$ 0.08
Closed Question Answering	6.50 $\pm$ 1.10	5.44 $\pm$ 0.00
Open Question Answering	4.77 $\pm$ 0.08	3.73 $\pm$ 0.41
Summarization	5.00 $\pm$ 0.43	4.75 $\pm$ 1.00
Information Extraction	5.36 $\pm$ 0.09	6.83 $\pm$ 0.00
General Question Answering	5.10 $\pm$ 0.10	3.45 $\pm$ 0.18
Creative Writing	2.00 $\pm$ 0.00	3.67 $\pm$ 0.00
Mean	4.88 $\pm$ 0.09	4.40 $\pm$ 0.18

4.1.2 Rank Comparison Experiment

To systematically evaluate the impact of LoRA rank on federated fine-tuning performance, we compare $r \in \{4, 6, 8\}$ under the Unseen setting defined above. For each task type, 10 questions are sampled and scored. Table 2 reports the average scores.

Table 2 Rank Comparison Results under the Unseen Setting

Task Type	Unseen		
	$r = 4$	$r = 6$	$r = 8$
Brainstorming	3.89 $\pm$ 0.11	2.89 $\pm$ 0.00	4.11 $\pm$ 0.00
Classification	4.31 $\pm$ 0.08	3.92 $\pm$ 0.15	3.23 $\pm$ 0.08
Closed Question Answering	3.67 $\pm$ 0.56	5.22 $\pm$ 0.44	5.44 $\pm$ 0.00
Open Question Answering	3.05 $\pm$ 0.14	2.50 $\pm$ 0.00	3.73 $\pm$ 0.41
Summarization	4.50 $\pm$ 0.00	3.25 $\pm$ 0.50	4.75 $\pm$ 1.00
Information Extraction	6.00 $\pm$ 0.00	5.33 $\pm$ 0.33	6.83 $\pm$ 0.00
General Question Answering	3.45 $\pm$ 0.09	2.73 $\pm$ 0.18	3.45 $\pm$ 0.18
Creative Writing	2.17 $\pm$ 0.00	2.33 $\pm$ 0.17	3.67 $\pm$ 0.00
Mean	3.88 $\pm$ 0.01	3.52 $\pm$ 0.04	4.40 $\pm$ 0.18

As shown in Table 2, $r = 8$ obtains the highest mean score on the Unseen setting (4.40), indicating the strongest overall performance among the tested LoRA rank configurations. For the Unseen setting, $r = 8$ performs best on Brainstorming, Closed Question Answering, Open Question Answering, Summarization, Information Extraction, and Creative Writing, indicating stronger generalization across most task types. Nevertheless, task-level trade-offs remain: Classification obtains the highest score at $r = 4$, and General Question Answering gives comparable results for $r = 4$ and $r = 8$. These observations suggest that LoRA rank affects different task types unevenly under non-IID data, and $r = 8$ provides the best overall balance in our experiments.

To complement the quantitative results, we select one question from the Unseen setting for case-level comparison (Table 3).

In the Unseen Information Extraction example, $r = 4$ identifies only one institution while omitting the nursery and primary school, and $r = 6$ returns the Dunkirk Community Centre instead of a school. The $r = 8$ response names the nursery, primary school, and academy, but it also includes the community centre. It is therefore more complete than the lower-rank examples, not an error-free separation of schools from other entities.

Increasing LoRA rank does not uniformly improve every task under non-IID conditions, but $r = 8$ achieves the highest Unseen mean score among the tested ranks. Accordingly, $r = 8$ is adopted as the default configuration in subsequent experiments.

Table 3 Example Responses to an Unseen Question from Models with Different LoRA Ranks

Question	Dunkirk is a residential area of Nottingham... [Reference text] Given a reference text about Dunkirk, tell me what schools are located in the area.
Model	Answer
$r = 4$	Dunkirk is home to the Nottingham University Academy of Science and Technology (NUA), a school run by the University of Nottingham.
$r = 6$	Dunkirk Community Centre
$r = 8$	Dunkirk has a nursery, primary school and the Nottingham University Academy of Science and Technology; socially, there is the Old Lenton and Dunkirk Community Centre.

4.2 Storage Cost and Transmission Evaluation

To further evaluate the practicality of the proposed on-chain/off-chain design, we conduct blockchain storage experiments on FISCO BCOS 3.x. The blockchain network consists of 10 nodes in a single group. The tested file is the LoRA parameter file generated under rank $r = 8$, whose size is 8.8 MB. We compare three storage methods: M1 directly stores the parameter file on-chain; M2 stores only about 300 B of authentication metadata on-chain and places the complete file in object storage; M3 also stores compact authentication metadata of similar size on-chain, but splits the file into nine shards with a 1 MB shard size and uploads them to object storage in parallel. Each method is tested with 50, 100, and 150 requests. We also record the blockchain data directory size before and after each run, which includes blocks, transactions, state data, and other node-generated database files.

Table 4 Blockchain Storage and Transmission Results

Method	Req.	Total Gas	Chain Time (ms)	Storage Time (ms)	Total Time (ms)	Data Before (MB)	Data After (MB)
M1	50	77,183,182,750	76,458	0	76,458	80.66	4,770.88
M1	100	154,366,365,500	220,500	0	220,502	80.73	8,553.44
M1	150	223,831,229,975	476,027	0	476,028	80.66	11,784.72
M2	50	4,785,700	23,454	1,545	25,155	80.69	88.71
M2	100	9,571,400	47,077	2,981	50,289	80.68	97.26
M2	150	14,357,100	70,608	4,385	75,316	80.64	105.78
M3	50	4,786,550	24,091	1,105	25,229	80.72	88.52
M3	100	9,573,100	48,259	2,061	50,375	80.73	97.25
M3	150	14,359,650	72,623	3,035	75,750	80.68	105.94

Table 4 shows that directly storing model files on-chain is prohibitively expensive. M1 consumes over 7.7×10^{10} gas for only 50 requests, whereas M2 and M3 consume only about 4.8×10^6 gas under the same scale. The gap remains more than four orders of magnitude as the request number increases, confirming that large model files should

not be written directly to the blockchain. This conclusion is also reflected in the chain data directory size. Although the initial data directory is already about 80 MB due to the genesis block, system tables, basic node databases, and contract deployment data, direct on-chain storage expands it to 4.77 GB, 8.55 GB, and 11.78 GB after 50, 100, and 150 requests, respectively. In contrast, M2 and M3 keep the final chain data size around 88–106 MB because only compact metadata is committed on-chain.

Comparing the two hybrid storage methods, M3 has slightly higher gas consumption and chain time than M2 because the calldata and on-chain metadata are not exactly identical after sharding. Its advantage appears in the off-chain storage stage. Parallel shard uploads reduce the storage time from 1,545 ms to 1,105 ms for 50 requests (28.5%), from 2,981 ms to 2,061 ms for 100 requests (30.9%), and from 4,385 ms to 3,035 ms for 150 requests (30.8%). The absolute saving increases from 440 ms to 1,350 ms as the number of requests grows, indicating that sharding becomes more beneficial when object storage or cross-institution transmission is the performance bottleneck.

Despite this storage-stage improvement, the total time of M3 is slightly longer than that of M2. This is because total time spans pre-chain preparation, shard generation, hash computation, transaction confirmation, storage upload, verification, and scheduling. For the tested 8.8 MB LoRA file, the additional sharding and verification overhead, together with the slightly higher chain time, offsets the storage-time saving and leads to small net increases of 74 ms, 86 ms, and 434 ms under 50, 100, and 150 requests, respectively.

M3 therefore shows a storage-stage time saving, not an end-to-end time saving, for the tested 8.8 MB file. At the design level, sharding permits independent retry and verification of affected shards, while replicas can support failover when intact copies remain.

5 Related Work

The core principle of FedLLM is to freeze most of the parameters of the pre-trained large model, and only train and communicate a small number of newly added or specified parameters. For example, LoRA based methods (Wang et al. 2024; Gao et al. 2025) approximate parameter updates by training low rank matrices, Prompt based methods (Guo et al. 2023; Su et al. 2024) guide model behavior by optimizing input prompts, and Adapter based methods (Chen et al. 2024; Kim et al. 2023) insert and train lightweight adapter modules between model layers. These methods significantly reduce the number of parameters that need to be trained, stored, and transmitted while maintaining model performance, thereby lowering the requirements for communication bandwidth, device memory, and computing resources.

However, the existing FedLLM techniques lack credible certification and traceability mechanisms. Combining blockchain with federated learning can achieve trusted and traceable transactions without the need for a third-party central institution (Cai et al. 2025). Traditional Blockchain is based on P2P architecture, where model aggregation and updates are completed through direct communication between clients, and it has strict requirements for network resources. Some Blockchain solutions reduce latency by

selecting appropriate consensus mechanisms and adjusting the basic parameters of the blockchain. For example, the BlockFL system (Kim et al. 2020) improves transmission rates through asynchronous operations and channel optimization; Zhang et al. (2020) constructed a hybrid blockchain architecture consisting of permissioned chains and local directed acyclic graphs (DAGs), which improved transmission efficiency. However, when applied to FedLLM, the above solution still faces problems such as rapid expansion of the ledger due to the surge in model parameters, difficulty in meeting real-time collaboration requirements, and easy leakage of sensitive parameters.

6 Conclusion

This paper presents a cross-organizational federated fine-tuning framework that separates model-file certification from parameter storage. A consortium blockchain records compact batch roots and round results, while signed per-file proofs and ordered manifests link recovered updates to those records without placing model tensors on-chain. Off-chain storage uses authenticated encryption, sharding, consistent-hash placement, and replicas across failure domains. The design supports offline checks of file integrity and signed evidence; online verification additionally confirms the current chain record and validity status. Federated averaging remains off-chain.

In the reported Qwen3-0.6B experiments, LoRA rank $r = 8$ obtains the highest Unseen mean score (4.40) among the three tested ranks. For the 8.8 MB LoRA file on a 10-node FISCO BCOS consortium, compact metadata storage reduces gas consumption by about four orders of magnitude relative to direct on-chain storage. Parallel sharding reduces object-storage time by 28.5–30.9%, although its total measured time is slightly higher because of additional processing and chain overhead. These results support the benefits of keeping parameter files off-chain and parallelizing their upload. Future work will measure batch-proof and encryption costs, evaluate replica repair under injected failures, and test larger files and workloads.

References

- Anonymous: Details omitted for double-anonymous review. In: Accepted Conference Paper (2026). In press; full bibliographic details supplied confidentially to the editor
- Cai, Z., Chen, J., Fan, Y., Zheng, Z., Li, K.: Blockchain-empowered federated learning: Benefits, challenges, and solutions. *IEEE Transactions on Big Data* **11**(5), 2244–2263 (2025)
- Chen, H., Zhang, Y., Krompass, D., Gu, J., Tresp, V.: Feddat: An approach for foundation model finetuning in multi-modal heterogeneous federated learning. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, pp. 11285–11293 (2024)
- Guo, T., Guo, S., Wang, J., Tang, X., Xu, W.: Promptfl: Let federated participants cooperatively learn prompts instead of models—federated learning in age

- of foundation model. *IEEE Transactions on Mobile Computing* **23**(5), 5179–5194 (2023)
- Gao, Z., Zhang, Z., Guo, Y., Gong, Y.: Federated adaptive fine-tuning of large language models with heterogeneous quantization and lora. In: *IEEE INFOCOM 2025 - IEEE Conference on Computer Communications*, pp. 1–10 (2025). <https://doi.org/10.1109/INFOCOM55648.2025.11044641>
- Kim, Y., Kim, J., Mok, W.-L., Park, J.-H., Lee, S.: Client-customized adaptation for parameter-efficient federated learning. In: *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 1159–1172 (2023)
- Konecný, J., McMahan, H.B., Ramage, D., Richtárik, P.: Federated optimization: Distributed machine learning for on-device intelligence. *CoRR* **abs/1610.02527** (2016) [arXiv:1610.02527](https://arxiv.org/abs/1610.02527)
- Kim, H., Park, J., Bennis, M., Kim, S.L.: Blockchain on-device federated learning. *IEEE Communications Letters(L-COMM)* **24**(6), 5 (2020)
- Luo, H., Luo, J., Vasilakos, A.V.: BC4LLM: A perspective of trusted artificial intelligence when blockchain meets large language models. *Neurocomputing* **599**, 128089 (2024) <https://doi.org/10.1016/J.NEUCOM.2024.128089>
- Su, S., Yang, M., Li, B., Xue, X.: Federated adaptive prompt tuning for multi-domain collaborative learning. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, pp. 15117–15125 (2024)
- Villalobos, P., Ho, A., Sevilla, J., Besiroglu, T., Heim, L., Hobbhahn, M.: Position: Will we run out of data? limits of LLM scaling based on human-generated data. In: *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024. Proceedings of Machine Learning Research*, vol. 235, pp. 49523–49544. PMLR / OpenReview.net, Vienna, Austria (2024)
- Wang, Z., Shen, Z., He, Y., Sun, G., Wang, H., Lyu, L., Li, A.: Flora: Federated fine-tuning large language models with heterogeneous low-rank adaptations. *Advances in Neural Information Processing Systems* **37**, 22513–22533 (2024)
- Yang, Q., Liu, Y., Chen, T., Tong, Y.: Federated machine learning: Concept and applications. *ACM Trans. Intell. Syst. Technol.* **10**(2), 12–11219 (2019)
- Zhang, Y., Lu, Y., Huang, X., Zhang, K., Maharjan, S.: Blockchain empowered asynchronous federated learning for secure data sharing in internet of vehicles. *IEEE Transactions on Vehicular Technology* **PP**(99), 1–1 (2020)
- Zhou, Z., Liu, H., Yuan, S., Zhang, H., Lou, J., Wu, C., Li, J.: BLOCKS: blockchain-supported cross-silo knowledge sharing for efficient LLM services. *CoRR* **abs/2506.21033** (2025) <https://doi.org/10.48550/ARXIV.2506.21033> 2506.21033

Zhang, J., Vahidian, S., Kuo, M., Li, C., Zhang, R., Wang, G., Chen, Y.: Towards building the federated GPT: federated instruction tuning. CoRR **abs/2305.05644** (2023) <https://doi.org/10.48550/ARXIV.2305.05644> 2305.05644

Statements and Declarations

Funding. Funding information is supplied on the separate title page and through the submission system to preserve double-anonymous review.

Competing interests. The authors have no relevant financial or non-financial interests to disclose.

Ethics approval and consent to participate. Not applicable.

Consent for publication. Not applicable.

Data availability. The Dolly instruction dataset analyzed in this study is publicly available at <https://huggingface.co/datasets/databricks/databricks-dolly-15k>. The numerical results supporting the conclusions are reported in the article. The implementation and experiment scripts are publicly available at <https://github.com/SoarCollab/RecordPlatform>.

Code availability. The framework implementation and experiment scripts are available at <https://github.com/SoarCollab/RecordPlatform>.

Author contributions. The complete author-contribution statement is supplied on the separate title page and through the submission system to preserve double-anonymous review.