UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SUPPORTING DECENTRALIZED FEDERATED LEARNING VIA POS BLOCKCHAINS

Diploma Thesis

Alexandros Balla

Supervisor: Spyros Lalis

September 2025

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SUPPORTING DECENTRALIZED FEDERATED LEARNING VIA POS BLOCKCHAINS

Diploma Thesis

Alexandros Balla

Supervisor: Spyros Lalis

September 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΥΠΟΣΤΗΡΙΞΗ ΑΠΟΚΕΝΤΡΩΜΕΝΗΣ ΟΜΟΣΠΟΝΔΗΣ ΜΑΘΗΣΗΣ ΜΕΣΩ POS BLOCKCHAINS

Διπλωματική Εργασία

Αλέξανδρος Μπάλλα

Επιβλέπων: Σπύρος Λάλης

Σεπτέμβριος 2025

Approved by the Examination Committee:

Supervisor **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos Antonopoulos**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Manolis Vavalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

I had the pleasure to work under the supervision of Professor Spyros Lalis, who was not only a knowledgeable and meticulous mentor, when I started my thesis, his constructive feedback, and never-ending availability a to discuss ideas was in essence the key driver for the development of my work, and has undeniably made me a better, more focused researcher, thank you Professor. And of course Professor Christos Antonopoulos, his advice helped me make a big change in my life, he listened and provided me with guidance when i needed it the most, Professor Manolis Vavalis, thank you for believing in me(and teaching me linear algebra Ax=b), supporting and encouraging my brain storming ideas. I would also like to thank my friends, Christodoulos, Thodoras, Antonis, Iris, Marios, Nilos, Kostantinos, Nikos, Dimitris, and Vaggelis for helping and supporting me. Thank you to my family; my mother, Anna, who always stood by me and supported me throughout my life with uncoditional love, and my brother, Thanos, for always believing in me. To all of you, thank you. I could not have done the speedrun without you.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Alexandros Balla

Diploma Thesis

SUPPORTING DECENTRALIZED FEDERATED LEARNING VIA POS BLOCKCHAINS

Alexandros Balla

Abstract

In the case of Federated Learning (FL) there's a problem.Most existing systems require a central coordinator or permissioned ledgers, restricting the transparency of the data and leaving the prevention of Sybil attacks in a grey area.

We have addressed these issues by using a permissionless Proof-of-Stake (PoS) blockchain to coordinate FL, and making it difficult for Sybil attacks to be carried out by putting model lists and updates directly onto the blockchain. Large amounts of data are instead stored off-chain, using InterPlanetary File System (IPFS) which takes care of the problem of storage space.

Our system has a training process that is split into rounds, with clients updating a shared model locally, sending out IPFS content identifiers to the network and a designated 'lister' pooling the updates and publishing the new global model. The idea is that the blockchain would be based on Proof of Stake with longest chain, highest stake finality, but our prototype mimics this with a tiny light-weight proof-of-stake mechanism.

Keywords: Decentralized Federated Learning, Proof-of-Stake, Permissionless Blockchain, IPFS, Sybil Resistance, Incentive Mechanisms, Web3

Διπλωματική Εργασία

ΥΠΟΣΤΗΡΙΞΗ ΑΠΟΚΕΝΤΡΩΜΕΝΗΣ ΟΜΟΣΠΟΝΔΗΣ ΜΑΘΗΣΗΣ ΜΕΣΩ POS BLOCKCHAINS

Αλέξανδρος Μπάλλα

Περίληψη

Στην περίπτωση της Ομοσπονδιακής Μάθησης (Federated Learning – FL) υπάρχει ένα πρόβλημα. Τα περισσότερα υπάρχοντα συστήματα απαιτούν έναν κεντρικό συντονιστή ή αδειοδοτημένα καθολικά, περιορίζοντας τη διαφάνεια των δεδομένων και αφήνοντας την αποτροπή των επιθέσεων Sybil σε μια γκρίζα περιοχή.

Έχουμε αντιμετωπίσει αυτά τα ζητήματα χρησιμοποιώντας ένα permissionless blockchain Proof-of-Stake (PoS) για τον συντονισμό της FL, καθιστώντας δύσκολη την εκτέλεση επιθέσεων Sybil, τοποθετώντας λίστες μοντέλων και ενημερώσεις απευθείας στο blockchain. Μεγάλες ποσότητες δεδομένων αποθηκεύονται εκτός αλυσίδας, χρησιμοποιώντας το Inter-Planetary File System (IPFS), το οποίο επιλύει το πρόβλημα του χώρου αποθήκευσης.

Το σύστημά μας έχει μια διαδικασία εκπαίδευσης που χωρίζεται σε γύρους, με τους πελάτες να ενημερώνουν ένα κοινό μοντέλο τοπικά, να στέλνουν αναγνωριστικά περιεχομένου IPFS στο δίκτυο και έναν καθορισμένο ‘lister’ να συγκεντρώνει τις ενημερώσεις και να δημοσιεύει το νέο καθολικό μοντέλο. Η ιδέα είναι ότι το blockchain θα βασίζεται σε Proof-of-Stake με τον μακρύτερο κλάδο και τελική συναίνεση υψηλότερου stake, αλλά το πρωτότυπό μας το προσομοιώνει αυτό με μια μικρή ελαφριά λειτουργία proof-of-stake που απλώς μιμείται το διάστημα παραγωγής μπλοκ.

Λέξεις-Κλειδιά: Αποκεντρωμένο Federated Learning, Proof-of-Stake, Permissionless Blockchain, IPFS, Sybil Resistance, Κίνητρα, Web3

Table of contents

List of figures

List of tables

Abbreviations

e.g	for example
i.e	that is
API	Application Program Interface
DP	Differential Privacy
AI	Artificial Intelligence (Τεχνητή Νοημοσύνη)
ML	Machine Learning (Μηχανική Μάθηση)
IID	Independent and Identically Distributed (Ανεξάρτητα και Ομοιόμορφα Κατανεμημένα)
FL	Federated Learning (Ομοσπονδιακή Μάθηση)
FedAvg	Federated Averaging (Αλγόριθμος Μέσου Όρου στην Ομοσπονδιακή Μάθηση)
BC-FL	Blockchain-Coordinated Federated Learning (Ομοσπονδιακή Μάθηση με Συντονισμό
BC	Blockchain (Αλυσίδα Μπλοκ)
PoW	Proof of Work (Απόδειξη Εργασίας)
PoS	Proof of Stake (Απόδειξη Κατοχής Συμμετοχής)
BFT	Byzantine Fault Tolerance (Ανοχή Βυζαντινών Σφαλμάτων)
Sybil	Sybil Attack (Επίθεση με Πολλαπλές Ψευδείς Ταυτότητες)
CID	Content Identifier (Αναγνωριστικό Περιεχομένου στο IPFS)
IPFS	InterPlanetary File System (Διαπλανητικό Σύστημα Αρχείων)
SHA	Secure Hash Algorithm (Αλγόριθμος Κρυπτογραφικής Σύνοψης)
API	Application Programming Interface (Διεπαφή Προγραμματισμού Εφαρμογών)
JSON	JavaScript Object Notation (Μορφότυπος Ανταλλαγής Δεδομένων)
DoS	Denial of Service (Επίθεση Άρνησης Παροχής Υπηρεσίας)
P2P	Peer-to-Peer (Δίκτυο Ομότιμων Κόμβων)

Chapter 1

Introduction

1.1 Motivation and Importance

Looking at the rapidly growing world of data from edge devices, such as smartphones, wearables, self-driving cars, and medical equipment, traditional AI models just don't cut it anymore. With this explosion of IoT gadgets, we are facing serious issues with the need for efficient and decentralized learning systems.

Regulations in healthcare and finance prevent sensitive information from being sent outside its home territory and with the proliferation of IoT devices, there's an enormous demand for AI that can learn in a way that does not require sending all of that data to a central point.

Enter Federated Learning, which allows devices to train local models on their own data, and then send the updates to a 'hub' for the overall model to be integrated. This approach spares privacy and lets training run across different types of equipment. However, it basically relies on a single server to manage the show, and if that server gets compromised, the whole thing falls apart. Plus, it doesn't have any transparency and does not motivate people to play fair.

Blockchain technology has now become a popular way to get around these problems. Its secure logbooks, clear audit trails, and resistance to being hacked [1, 2] make it the perfect partner for federated learning. Replacing the need for a single central server, blockchain gives us a distributed, verifiable and trustworthy way to send, validate, and record model updates, and lets us sort out incentives for participants with things like tokens and staking, which are part of the financial side of the blockchain [3, 4, 5].

The combination of federated learning and blockchain is basically a vision referred to as

Web3 [6, 7], a web that is completely open, decentralized, and transparent. This will allow to effectively use non-private AI that also does not require any of the traditional intermediary organizations. Anyone can kick in with ideas to train the model, and the blockchain ensures fairness and reliability.

1.2 Problem Statement

In a scenario where anyone can join, we run into new problems that neither technology can fully solve on its own, when combining federated learning and blockchain. Traditionally, federated learning is made for closed systems where the participants and the one collecting the data are known, which is not ideal for open networks. On the other hand, permissionless blockchains, designed to work for open networks where anyone can participate at any point in time, are under threat from Sybil attacks [8], where malicious users can create a lot of fake identities to manipulate the outcome of the consensus.

From two different angles, Proof-of-Authority [9] and centralized coordination don't fit with the very open nature of decentralized federated learning.

Problems that arise from this combination of technologies, which this thesis looks at, include getting rid of the middleman in federated learning, fending off Sybil attacks in a completely open system, making sure that model updates are transparent, tamper-proof, and reliable, and incentivizing people to be honest and stick around.

To tame these problems, we need to join up consensus protocols, Sybil-resistant measures and the flow of data and information in the federated learning process into a cohesive whole.

1.3 Methodology Overview

The approach followed in this work, is to break down the problem into a series of stages, starting with a thorough examination of the related literature in machine learning, federated learning, blockchain and consensus protocols, which is heavily weighted towards permissionless systems, partially synchronous models [10] and Sybil-resistance methods including Proof-of-Work and Proof-of-Stake. The proposed approach has been implemented in the form of a complete system prototype.

For its use in partially synchronous models, and Sybil-resistance, the prototype/blueprint

combines federated learning and a blockchain-based management layer. Here, we introduce a lister node, as a task owner, send training jobs as 'listings' on the blockchain, and clients respond by running local training and sending their updated models as transactions. The whole thing is secured by a consensus system that is been fortified with a Sybil-resistance mechanism. One of the key things to scale the system is storing large model parameters off-chain, for instance, in IPFS [11, 12, 13, 14], but still recording pointers and verification metadata on the blockchain. From a theoretical perspective, Proof-of-Stake is considered the best Sybil-resistance mechanism for this system because it is so much more energy efficient and scalable [15]. However, in practical terms, the system prototype uses a simplified Proof of stake, because it is a lot simpler to set up in a small-scale academic setting. Also, the implemented blockchain has customized transaction types for federated learning and is working in harmony for training jobs.

1.4 Objectives

Web3-enabled federated learning the thesis has three main goals, when developing a system for decentralized. The first is to lay out a permissionless federated learning framework that leverages the consensus mechanisms of blockchain, to ensure transparency and fairness. Secondly, Proof-of-Stake will be used to keep Sybil attacks at bay, which in turn also encourage fair participation. The third major goal of the thesis is to remove the role of the central server by storing and finalizing global updates on-chain via CIDs and model bytes stored off-chain. In addition, we have also added a monetary incentive/ticket system, powered by the blockchain's native economics, to fairly reward honest participants and shame and punish the ones who don't play by the rules.

The research provides a prototype implementation design and a simulation based analysis that looks at the trade-offs between security, scalability and performance, and demonstrates that decentralized, Web3-enabled federated learning is not only possible, but a preferred future direction.

1.5 Thesis Structure

In the case of building a robust and secure framework for machine learning, the theoretical foundation is as important as the technical components. Chapter 2 2, therefore, provides an overview of machine learning, federated learning, blockchain technology, and consensus protocols, laying down the theoretical background of the proposed framework. The conceptual design in Chapter 3 3 introduces a modular and secure approach in its architectural and functional component system. The implementation and blueprint development takes center stage in Chapter 4 4 with description of the blockchain layers, data storage systems, and integration with machine learning frameworks. Experimental results in Chapter 5 5. Chapter 6 6 gives discussion, conclusion and guidelines for further research.

Chapter 2

Theoretical Background

To develop a blockchain-based federated learning (BC-FL) system, it is vital for us to examine/analyze the theory of FL and blockchain technologies. As such, the beginning of this chapter starts with the theory of federated learning, the architecture, then the challenges that it has, the chapter then focuses on blockchain technology, describing the principles of decentralization, finality(i.e., immutability, write-once, cannot be altered), and trustless coordination(no need to trust a single coordinator), then permissioned and the permissionless setting, are introduced, and the differences between them, as this choice fundamentally dictates the openness of the system. After sybil resistance mechanisms and consensus protocols are thoroughly analyzed, due to their effectiveness on the speed and security of the system with no single owner(decentralized systems). In addition, existing work will be examined, where blockchain technologies meet FL, identifying what does not yet work and then presenting the contribution of this thesis. The chapter was arranged in this way so that we can give the reader a precise understanding of the unique components that make up the system, and also make the order easy to follow from theory to practice, assembling the fundamental concepts, and the problem we are working to solve.

2.1 Federated Learning

Federated Learning (FL) [16] is a new way to train machine learning models [17]. Instead of gathering every data in one place, FL allows many devices to ’help’ train the global model and, at the same time, keep their data private. This section explains how FL works, what problems it faces, and how blockchains can assist in solving them.

2.1.1 How Federated Learning Works

In a classic ML setup, raw data is collected and transmitted to a central server. The server then trains the preferred model using all the data, but with FL, things work differently. The two main parts of the system, are `clients` and the `aggregator`. Clients are devices such as phones or sensors that have private data. The aggregator is usually a central server that manages the training process. It starts by the aggregator nodes where they send the model version(that they want to be trained) to a group of clients. Each client model will be trained by its own data, and clients then send only their model updates back to the aggregator. The raw data never leave the client. The aggregator then combines all the local models to create a new global model, averaging all the updates into one [18, 16]. This process repeats in or epochs, each round begins with the aggregator sending the latest model to the clients, clients then train and send updates again, and this will continue until the aggregated model is good enough [19].

The process discussed here has great benefits and valuable guarantees. One of them being that it shields data privacy and saves bandwidth, because instead of sending the whole raw data to the central server, only parameters are sent, which in comparison to the data are quite small. However, it still has challenges, especially in a real-world scenario, and this is what will be delved into next.

2.1.2 Problems in Federated Learning

Even though FL has many benefits, as we simply saw, here we take a closer look at the main big problems, these issues make it hard to use FL at large scale or in open networks. One big problem is that each client may have many kinds/types of data. As an example, one phone might have mostly photos of food, and another might have pictures of cars, basically not identical. This is called `non-IID data`, where IID stands for independent and identically distributed data. In the classical ML world, the assumption that the model will have data be IID is usually convenient, but in FL where generally inside each client, the data might be dependent(or correlated), for example, a client typing pattern might be related over the day, and different across clients, a client might have data in Greek, and another in Spanish, not identically distributed. When data is different like this, the training becomes harder. The global model might fail to learn accurately [20, 21, 22, 23].

Another problem is that devices are not all the same. Some are fast, others are slow. Some

have a good internet connection, others do not. This problem is called `system heterogeneity` (i.e the lack of uniformity). When some devices take too long, they slow down the whole process. These slow devices are called `stragglers` [18]. FL also has a `communication problem`, when millions of devices take part, it takes a lot of latency to send and receive models. This in return uses up bandwidth. Some researchers have tried ways to make this better, like sending smaller update packages or by reducing the number of clients [24, 25].

Although FL does not transmit client private data, it can still have `privacy risks`. Hackers might be able to figure out private data from the model updates [26]. To stop this, methods like `differential privacy` and `secure computation` are used [27, 28, 29]. However, these methods can reduce the accuracy of the model(i.e, not correct predictions) or slow the model training, since you would have to add more security mechanisms. All these problems, data, system, communication, and privacy, make it hard to trust one central aggregator. That's why we now examine/analyze how blockchain technologies could help.

2.1.3 Summary

In summary, FL is a smart way to teach a model to make good guesses (i.e., accurate model training) while keeping data private [30, 31]. It lets people and machines work together in harmony, and provides nice security guarantees by keeping data private(i.e., without the need to send the data that you trained the model). But FL still depends on a central aggregator(usually), and this in turn creates a trust problem [32, 33]. To mitigate this problem, we need to help devices train models fairly and securely, and for this we will introduce blockchain technologies, the fundamentals, the core concept, the limitations, and see if it has any guarantees that it can deliver.

2.2 Blockchain Fundamentals

This section will explain what blockchains are, how they separate in comparison to other systems, and why they are helpful for building trust without the need for a central authority. We also discuss the critical trade-offs that blockchains face, while always keeping in mind how to bridge them by looking at FL similarities with blockchain.

2.2.1 Core Concepts

A blockchain can be defined as a special kind of append-only data structure that works like a growing list of records [1]. These records are called blocks, and a list of these blocks connected together is called a 'blockchain'. Each block contains a group of transactions, the previous block's pointer/hash, a record(i.e, a timestamp that pinpoints when a tx had happened), and a proof that it was added correctly. A transaction can be defined as an instruction that, if valid, changes the ledger state, and what makes it valid? Proper signature, enough balance/fees etc. A timestamp is utilized to say when this block was produced. The hash that references the previous block is used to prove integrity, meaning everything can be linked back to the original block using its hash(genesis block). This means that if someone tries to change one block, all later blocks will break. This design makes the blockchain secure and difficult to manipulate [34]. Formally, we can describe the i-th block in the chain, B_i, as containing the of the previous block (h_{i-1}), the set of transactions (T_i), the time it was added (t_i), the proof (p_i), and its own hash (h_i). The very first block, called the genesis block (B_0), is fixed in the system from the start.

This setup makes the blockchain more than just a data storage tool. In the next part, we explain how blockchains differ from traditional systems.

2.2.2 Properties of Blockchain Systems

Blockchains stand out from regular databases in four major ways: decentralization, immutability, transparency, and pseudonymity, each of which we will analyze. First, `decentralization` means that no single group controls the system, rules, and updates come from the protocol and economic incentives, not from one company or server. Second, `immutability` means the data cannot be easily changed, because each block includes the hash of the previous one, and changing history is almost impossible. Third, `transparency` means that anyone can check the data, transactions, and state changes can be verified by the public (since the record is available to everyone in the chain). Lastly, `pseudonymity` means users are known by their public keys, not their real names, these features help people trust the system without knowing each other. They are supported by rewards and penalties that encourage users to act honestly. But these benefits come with trade-offs; the biggest one is the issue of scalability, which we explain next.

2.2.3 Scalability and the Blockchain Trilemma

Blockchain systems often struggle to scale well(i.e., transactions per second, latency, record/ledger growth etc), this means that they can be slow or costly when many users join, and here the `blockchain trilemma`, a concept introduced by Buterin (founder of Ethereum [35]) in 2017 [36], says that blockchains can only achieve two out of three things at once: decentralization, security, and scalability, so either decentralization and security, but this would mean that the system cannot be scalable, decentralization, and scalable, without security, and finally security and scalability, without decentralization.

Public (or permission-less) blockchains focus on being decentralized and secure. But this usually makes them slower. Each participant must process data(receive, validate etc), and blocks take time to spread across the network. Private (or permissioned) blockchains can be faster. They limit who can participate, which reduces the time needed for agreement(i.e, consensus). But this reduces openness and may increase trust in a central group. But given public and private blockchains, the question still remains, can you scale them without losing something? There are ways to help blockchains scale better. One way is using `layer two systems`, like the Lightning Network [37], which handle many actions off-chain. While another method is sharding [38], which splits the blockchain into smaller pieces that can work in parallel. A third option is using `Proof-of-Stake (PoS)` instead of energy-heavy Proof-of-Work [39], since PoS is faster and uses less power, due to the fact that you do not have to solve cryptographic puzzles in order to be selected the winner of the leader election. But each of these ideas changes the way the system works, some work better for public chains, others for private ones. Understanding who takes part in consensus is key, and this leads to the next topic: the difference between permissionless and permissioned systems.

2.3 Permissionless vs. Permissioned Blockchains

This section first introduces more formally the two major types of blockchain system(e.g permissionless and permissioned), and then we talk about their differences. The difference in a nutshell lies in who is allowed to join and take part. This choice affects security, trust, and how the system can be used, and since this thesis proposes a permissionless Federated Learning system that uses Proof-of-Stake (PoS), it is important to clearly explain both models and why we chose permissionless.

2.3.1 Permissionless Blockchains

A permissionless blockchain [40] is open to everyone (i.e., downloading bitcoin and running it requires no permission). Anyone can join the network, submit transactions, and even become a validator or miner. Bitcoin and Ethereum are examples of this model [34, 35]. These networks value openness and allow anyone with enough skill or stake to help run the system, by contributing to consensus, validation, etc. Because anyone can join, the network must be ready for attacks, and permissionless systems have an attack also known as sybil attack [8], where a person creates many fake identities to gain control. To prevent this, permissionless systems use `Sybil resistance` techniques such as Proof-of-Work (PoW) or Proof-of-Stake (PoS) that add randomness to the leader election and that if you want to become a leader, you have to either work or risk something [41], which we discuss more in Section 2.5.

The best part of permissionless systems is that no one can censor or control them. They offer strong decentralization and freedom. To show the contrast, we now explain permissioned systems, where access is limited.

2.3.2 Permissioned Blockchains

In a permissioned blockchain, only selected members can participate in the network. These members are often trusted organizations. Examples include `Hyperledger Fabric` [42], `R3 Corda` [43], and other systems used in supply chains [44], healthcare [45], and finance [46].

To join a permissioned blockchain, you usually require to know something prior to the system's initialization(i.e, identity, role, etc.). Members are often tied to real identities, adding legal and business trust, this makes the system much more efficient, it can use short, proven consensus protocols like `Practical Byzantine Fault Tolerance (PBFT)` [47], `HotStuff` [48], or other `Byzantine Fault Tolerant (BFT)` systems [49]. These systems give instant finality and can process thousands of transactions per second.

Because the validators are known, permissioned chains can follow rules and fit laws more easily, but they lose some key benefits of blockchains, they are no longer fully open or decentralized. Instead, they act more like shared databases run by a few groups. This works well when members already trust each other, like in traditional banking(where more systems are permissioned). But it may not work for open systems like Federated Learning, which is

discussed next.

2.3.3 Relevance to Decentralized Federated Learning

Federated learning systems based on blockchains often assume that a fixed group of institutions will run things, but what if who drives the FL rounds is chosen by staking and randomness? Even though combining Blockchain technologies with Federated learning sounds natural, combining these two also raises a whole lot of new problems, like Sybil attacks as discussed earlier. If only a few validators are trusted, a bad actor could fake identities and take control, here a `permissionless` blockchain for with FL mechanisms is a better fit overall. It lets anyone, whether a person, a phone, or a sensor, help train the model. This vision aligns with the broader `Web3` concept of ML [50]. But it comes with a need for Sybil resistance. Proof-of-Work offers a strong defense, but it uses too much energy. Instead, this thesis chooses `Proof-of-Stake`, where influence is related to economic investment. In this way, it is hard for attackers to create fake accounts on a scale. Another benefit is that all actions are public, you can check who contributed what, which helps reward good behavior and detect cheating. Before explaining how consensus works, the next chapter introduces the network timing model, which supports `safety` and `liveness` properties [51] in the system that will be suggested.

2.4 The Partially Synchronous Model

The assumption of a partially synchronous model was introduced by Dwork [10] to offer a more realistic view of how distributed systems behave in the network setting. Blends two assumptions of the network model: the synchronous model [52], where message delivery times are known, and the asynchronous model, where delays can be unbounded. In a *synchronous system*, all participants agree on message delays and use a shared clock [52]. This setup makes it easy to reach an agreement since everyone knows when the messages will arrive, and it is also advantageous on detecting crashes on systems. In contrast, *asynchronous model* does not offer such guarantees [53]. Messages may be delayed indefinitely(i.e., no'real' crash detection), and nodes do not share a clock. Due to this, reaching consensus becomes impossible in the worst case, as proven by the FLP Impossibility result [54].

But neither of these systems really reflects reality that well. In real networks, everything

is possible; there are delays, disconnections, and then periods of stability. The *partially synchronous model* captures this behavior, by allowing for unknown delays in the beginning (the asynchronous phase), but makes promises that at some unknown future point, called the `Global Stabilization Time (GST)`, communication becomes stable and predictable. After GST, the system enters the synchronous phase, and the message delays are now bounded by a known constant, usually called Δ. This upper bound means that any message sent at time t will be received by time $t + \Delta$(the synchronous phase). Even messages sent before GST will be delivered no later than GST + Δ. The model supposes that each node can access a clock with some small drift and that after GST, all network delays are within the bound Δ. These assumptions let system designers build protocols that behave sufficiently in practice, even when early communication is unreliable. Thus, this assumption of the network model is useful because it gives clear delivery guarantees once GST is reached, and sustains efficient coordination in distributed systems, even when there are occasional failures or temporary network problems. For instance, package loss or delays induced by malicious nodes, would be seen as short-term problems. The system is expected to return to normal soon afterward(synchronous phase). Unlike fully asynchronous systems, the partially synchronous model makes it doable to build robust `Byzantine Fault Tolerant (BFT)` consensus; these protocols can still reach an agreement even when failure of nodes occurs or act maliciously; for this reason, many modern blockchain and other P2P systems(in other words, distributed systems) depend on this assumption, for example, Tendermint and HotStuff [55, 48] both work under the partially synchronous assumption. With the partially synchronous model set, the next logical step is to examine the design choices in consensus and Sybil resistance mechanisms that function reliably with this network setting.

2.5 Consensus Protocols and Sybil Resistance

The consensus ensures that nodes converge on a single, consistent ledger state even when the presence of failures occurs(malicious or not) or when adversaries exist. In open networks, two threats dominate: Byzantine behavior and Sybil attacks. Reaching a consensus in a blockchain system(or in general distributed systems) is basically the backbone of what these systems do, it ensures that all nodes have the same record of transactions, and will not let anyone double-spend or create conflicting states. The issues that blockchain systems

face in achieving consensus in open and hostile environments are Byzantine faults and Sybil attacks, which are essentially ways in which adversaries can manipulate the network.

This section separates consensus protocols, the rules for how agreement is reached, from Sybil resistance mechanisms, the systems that allocate fair influence in the consensus process, and we will be analyzing both, as they are basically the foundation of blockchain security and directly affect the design of decentralized federated learning systems. The following subsections review of protocols that deliver agreement (BFT and longest-chain) and then the Sybil resistance mechanisms that allocate influence and offer robust leader election in permissionless settings (PoW and PoS).

Byzantine Fault Tolerant Protocols

Classical BFT protocols, such as Practical Byzantine Fault Tolerance (PBFT) [47], and their successors, such as HotStuff [48], work by tolerating up to $f < n/3$ Byzantine nodes out of a total of n validators in a partially synchronous system. Validators are required to pass around signed votes and are waiting for an agreement, after they've found a consensus, the result is considered a certainty, or deterministic, and cannot be changed, when using a BFT consensus protocol. These protocols deliver near-instant validation and unwaveringly secure results, making them perfect for permissioned blockchains where the list of validators is fixed. Looking at BFT-style protocols, you will see that their communication complexity is $O(n^2)$ per round [56], making them nearly impossible to scale. In the world of thousands of validators, these kinds of protocols are practically useless.

In the industry, BFT protocols still manage to be influential and are used as 'finality gadgets' in contemporary PoS blockchains to add a deterministic seal of approval on top of the probabilistic consensus system. Although BFT offers strong safety and fast finality in small, curated validator sets, scalability constraints motivate probabilistic approaches in open settings.

Longest-Chain Consensus

The longest-chain consensus protocol [34], presents an entirely different picture. Here, nodes fight to add new blocks to the chain, and the winning chain is the one with the most energy and commitment. The economics of extending alternative chains, rather than communication between nodes, leads to agreement.

The longest chain consensus gives *probabilistic finality*, basically the farther back a transaction is buried under newer blocks, the less chance it is reversed. It sacrifices instant certainty for the sake of making networks more scalable. Has proven itself to be surprisingly resilient to attacks [57], thanks to over ten years of operation in the case of Bitcoin, but it does add in some delay. You usually have to wait for a few confirmations to feel very sure that a transaction will not be changed [58].

When it comes to decentralized federated learning, longest-chain consensus ensures that model updates that are deep enough under the surface are essentially unchangeable [59], but because the finality is not guaranteed, the training cycle is slowed down, showing the trade-off between dependability and speed. For decentralized FL, such probabilistic finality implies choosing training-round cadences that tolerate reorg risk while preserving throughput.

2.5.1 Sybil Resistance Mechanisms

Proof-of-Work (PoW)

Proof-of-Work (PoW) [60] a system first used in Bitcoin [34], when defending against Sybil attacks, it has been shown to be a viable option [61], linking block production with a huge amount of computational energy [15]. Miners have to crack very difficult hash-based functions [62], and their chances of success are based on the proportion of the total computing power they contribute. As a reference point, this thesis presents PoW, and does not, however, consider it for practical use, mainly because the energy it consumes does not go towards training. These drawbacks motivate another protocol as a lower-energy alternative.

Proof-of-Stake (PoS)

Proof-of-Stake (PoS) [39] was presented as a more eco-friendly option by replacing the requirement for energy with the stake of cryptocurrency [15]. The system stipulates that validators lock in a portion of their stake(economical term), and, based on the size of that stake, they have a higher probability of getting to propose or validate the next block.

PoS gives very low energy consumption and allows transactions to be settled much more quickly, thanks to the integration of BFT-style protocols (such as Tendermint [55] or Casper FFG [63]), and kills off any would be misbehavior with something called ’slashing’ [64], where anyone who says something they shouldn’t or tries to fork the blockchain loses a part

of their stake.

In the world of Federated Learning, PoS [4] turns out to be a natural fit, with its ability to scale, streamline and give financial incentives to contributors. It enables the rapid finalization of model updates, something that cannot be said for PoW, and helps to create a token-based system that awards users for sending in useful training updates. As a result, the Sybil resistance mechanism for the proposed framework is PoS. We summarize these trade-offs in Table 2.1.

2.5.2 Comparative Analysis and Relevance

The table 2.1 sums the key trade-offs among the consensus protocols and Sybil resistance mechanisms.

Approach	Sybil Resistance	Energy Use	Finality	Scalability
BFT	No	Low	Immediate	Limited
Longest-Chain	No	Depends on Mechanism	Eventual	Good
Proof-of-Work	Yes	Very High	Probabilistic	Moderate
Proof-of-Stake	Yes	Low	Probabilistic	High

Table 2.1: Comparison of consensus protocols and Sybil resistance mechanisms, summarizing their assumptions, energy efficiency, scalability, and finality guarantees.

In terms of building a robust and efficient blockchain-based federated learning system (BC-FL), PoS-based Sybil resistance in combination with the longest chain or hybrid consensus is the best design. This idea, this combination, brings together the benefits of openness and decentralization, without the energy waste of PoW and the communication overload of pure BFT. PoS and BFT hybrids, such as Algorand [65], use a clever system where they select committees based on PoS and then run a BFT process among them to obtain deterministic finality.

Implications for Federated Learning follow from the consensus and Sybil resistance methods we have discussed. Guaranteed finality lets us have a steady stream of updated models in real time, something that is basically necessary for synchronous training, and kicks out the possibility of malicious nodes flooding the system with junk data that could ruin the model.

PoS also helps to put a price on honest contributions, making sure that those who contribute are rewarded for their efforts, and penalizing those who don't. Based on all this, this thesis builds its framework on PoS combined with longest-chain or BFT hybrids, creating a system that is completely open and free, where Federated Learning can be both practical and secure. In order for a BC-FL system to scale, the need for an off-chain storage is required [66], given today's machine learning models [67], since saving models on-chain would be inefficient. To address this, IPFS is introduced to avoid placing large model artifacts on-chain.

2.6 InterPlanetary File System (IPFS)

When building decentralized learning frameworks, there is a need for a robust storage system that can ensure the integrity, availability, and versioning of the data, but does not bloat the blockchain. The InterPlanetary File System (IPFS) is basically a peer-to-peer (P2P) storage and distribution network that satisfies these requirements by detaching data from its location and using a cryptographic hash to address it, as first proposed by Benet in 14 [11]. When adding data to IPFS, the data are broken down into smaller chunks, then hashed, and then attached to a Merkle-directed acyclic graph (DAG) [68]. Every chunk gets a Content Identifier (CID), which is derived from its hash, and the root CID is used to sum up the whole thing. Any tiny change in the data would mean a brand new CID, which makes it impossible to tamper with the data, and also gives us the power to automatically eliminate duplicated data and sort through versions of the same thing. IPFS has a clever way to spread the word about the existence of these CIDs, through a Kademlia-based [69] distributed hash table (DHT) based on the XOR metric, so that the content can be found quickly, without any centralized indexes. Public HTTP gateways can also bridge IPFS content to regular clients, but IPFS itself is fully peer-to-peer, while CIDs are immutable, the InterPlanetary Naming System (IPNS) lets content have mutable names, where names are the hash of the public key [70]. When publishing a new model, leverage the IPFS to make a moving pointer to the latest state and then keep a verifiable record of the previous versions.

IPFS is essentially a best-effort network [13], and is dependent on nodes to pin its content, in other words, to prevent it from being deleted. By pinning the content across clusters, durability can increase, such as with IPFS Cluster [14] or employing storage markets like Filecoin [12], which includes a monetary system (a storage and retrieval marketplace) that

guarantees that the content will be around for the time needed. The security of the content is basically guaranteed by the IPFS system's inherent integrity [71], thanks to the fact that any mismatched block will not have the correct hash, but the privacy of the content must be handled at the application layer, since ipfs does not encrypt the data by default. But that leaves the encryption to the application layer, and the encryption keys have to be distributed through access control logic of the protocol. With the use of CIDs that cannot be changed once they are created, on-chain commitments, binding the training rounds to specific CIDs also gives the necessary tools for provenance and auditability. As such in BC-FL, a popular technique is to upload the full model updates to IPFS and only commit the CID, along with the number of rounds, the set of clients, and the incentives to the blockchain, which slashes down the load on the chain, yet still gives us a way to verify and re-run the model [72]. Practical things to keep an eye out for include optimizing chunk sizes, laying out DAGs for massive models, policies to cut down latency when a brand-new model is being uploaded, and optional Filecoin deals for economic insurance that the referenced materials will be available and verifiable as long as we need them. Next, situate these building blocks within prior art that combines BC-FL.

2.7 Related Work

In recent years, there has been a growing interest in combining blockchain technology and Federated Learning [3]. Both fields tackle the problems of distributed computation, data privacy, and trust coming from two different directions.

2.7.1 Blockchain-Assisted Federated Learning

Most existing research on the subject of BC-FL [5, 72, 59] has shown that blockchain can greatly enhance FL by providing decentralized coordination, incentives, and a way to verify the accuracy of the results. However, the problems of scalability, openness, and resistance to malicious attacks have driven the need for something completely new, like the permissionless Proof-of-Stake system we are going to talk about in this thesis.

Early attempts to merge blockchain and FL basically replaced the need for a centralized mediator and instead used a decentralized ledger that records and verifies the contributions made by clients. These studies, like those by Kim [4], often assume a network that is restricted to a small number of organizations, such as a consortium, and can be hacked by a single entity

that can create multiple fake identities. The work by Xu et al. in ’21 [73], that combined blockchain and FL to build a distributed aggregator, did show the ability to bring out the transparency and reliability of blockchains and the private nature of FL, but falls apart when relying on a fixed network, where a single individual can bribe the validators to go against the rules, known as Sybil attacks.

Some researchers have looked at using incentives to get people to contribute to the process, for example, Lu [5] created a system in which participants are paid tokens for sending in their updated models and, therefore, have a financial reason to behave well. But these systems still rely on a central administrator who you cannot get rid of and does not really think about things that could go wrong, like someone feeding bad information to the model.

2.7.2 Security and Privacy Enhancements

Blockchain has been studied as a way to add extra security and protection to the system, down the track as a possible solution to provide unalterable logs of changes made to the models, when using Federated Learning. This is crucial for domains such as healthcare and finance, where transparency, auditable training rounds, and accountability are mandatory [3, 74]. By linking each contribution to a verifiable on-chain record, blockchains aid the detection of poisoning and reduce opportunities for misreporting. Another area of research has looked at how blockchain can help with techniques like secure aggregation [75] and differential privacy, but it still relies on the assumption that the people involved are semi-honest and do not scale well to large, diverse networks. In addition running all the necessary encryption codes within the blockchain’s consensus mechanism, is a massive overload, particularly in systems that do not have much processing power. However, security benefits must be balanced against scalability and performance constraints.

2.7.3 Scalability and Performance Challenges

The problem is that blockchains can be either permissioned or permissionless, and each has its own limitations. Permissioned blockchains can zip through transactions faster using BFT protocols, but they do not allow everyone to join in. On the other hand, permissionless blockchains get clogged up when every single model update gets logged on the chain. Solutions such as storing only metadata on-chain and sending the model updates to decentralized storage systems (like IPFS) [76, 72] raise new problems, such as data availability and

consistency.

For the choice of consensus protocol, if you want strong Sybil resistance, Proof-of-Work systems are the way to go, but they're energy-inefficient and slow. Proof-of-Stake systems offer a more efficient and scalable option, but more research is needed to see how they can be integrated with Federated Learning. Most existing works either do not care about the consensus algorithm or are fixated on permissioned BFT, which is not the same thing.

2.7.4 Limitations of Current Approaches

In the current state of blockchain-assisted federated learning (FL) systems, several issues are apparent. Most of these systems, being conceptual frameworks, often fail to provide concrete implementations, stop at presenting architectures and incentive mechanisms, and do not go into much detail with prototype or real-world test results, leaving lots of unanswered questions about the practicality and performance trade-offs of these systems.

With a whole lot of proposals, many of these FL systems have chosen permissioned or consortium blockchains as the basis for their framework, which, although it cuts down the amount of governance work, also paves the way for centralization and does not give a full account of the dangers that come with open systems, specifically Sybil attacks. The resistance to Sybil attacks in many of these systems is taken for granted, meaning they'd be easy prey to a malicious actor in a completely open network.

Most of the time, the consensus layer in FL systems is glossed over, too. Studies tend to treat the blockchain as a black box, focusing on reward and privacy schemes, but this thesis puts consensus protocols under the microscope. Specifically, Proof-of-Stake and longest-chain rules, showing exactly how they affect the security and speed of decentralized learning, and therefore limit the generalizability of these systems and stop them from being truly open, global-scale systems that adhere to the principles of Web3. When merging blockchain technology and Federated Learning, this work wanted to sidestep the problems that have plagued previous methods, which were hampered by the threat of Sybil attacks, the neglect of explicit consensus protocols and the use of permission-based consortiums. Past these issues, this project takes the state of the art towards a fully decentralized, scalable, and secure learning environment.

The proposed system has three main ways: it explicitly uses PoS+longestchain consensus, while sipping less energy compared to the traditional Proof-of-Work method; it presents a

detailed blueprint of its workings, where model updates are saved off-chain, with the aid of simulations to show that this is practical; and it puts Federated Learning into the bigger picture of Web3, highlighting how blockchain-based incentives and complete transparency can make global, trustless collaboration in machine learning a reality.

As the literature has shown the potential of BC-FL, it is still held back by centralization, issues with scaling and incetive robustness among other things. This thesis lays out a permissionless, PoS-based system that is intellectually sound and back up by real-world design and analysis, and so gets the ball rolling towards decentralized, incentive-based Federated Learning systems for the Web3 generation.

Chapter Summary

When examining the principles of Federated Learning (FL) blockchain technologies, the major drawback of traditional Fl was that they are centralized. The permissionless blockchain has been shown to be the solution to this problem, and Proof of Stake (PoS) mechanism, as a means of preventing Sybil attacks and Leader election. This theoretical framework lays the groundwork for the system Conceptual Model that we will introduce in the next chapter.

Chapter 3

Conceptual Design

3.1 High-Level Architecture

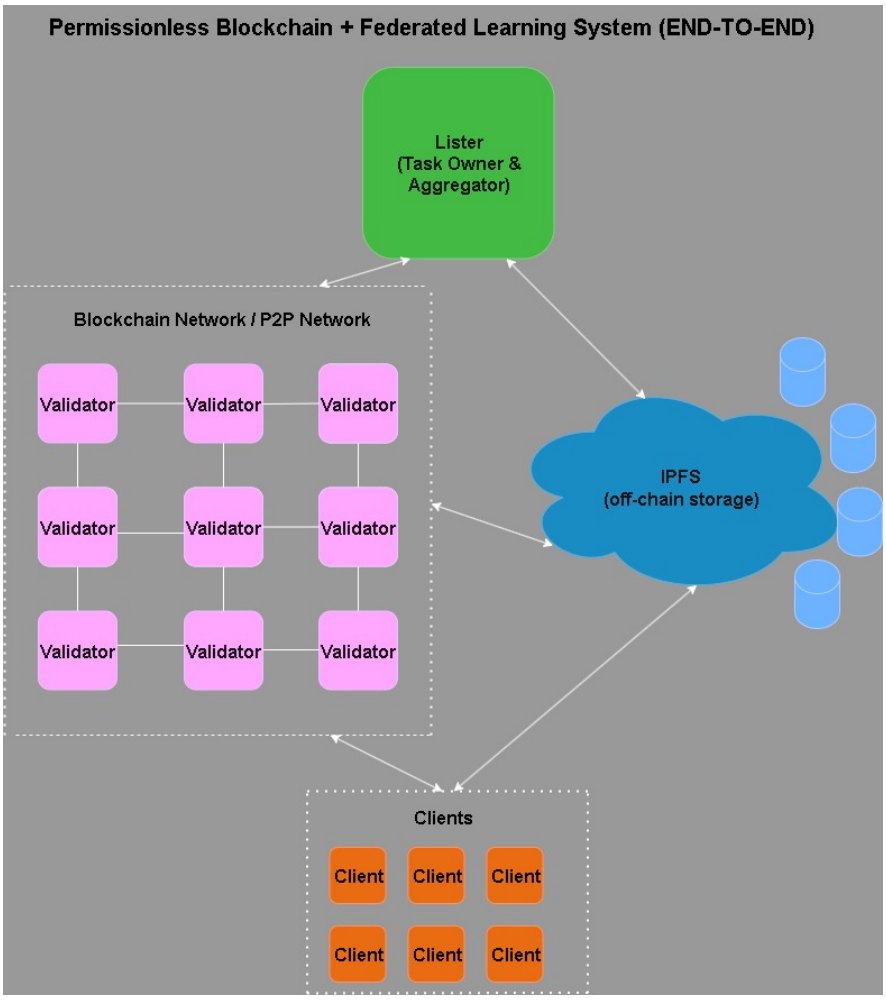


Figure 3.1: Conceptual design, roles, on-chain state, and off-chain artifacts

The high-level architecture(see Figure 3.1) of this system has three key roles, namely the Lister node (task owner/round aggregator), the Validator nodes (maintain the canonical chain using PoS and Longest-Chain, Section 2.5 and Section 3.3), and the Clients (train locally and submit updates). The Lister node takes the lead in initiating a training campaign, setting the plan for the machine learning model, posting the starting parameters, and laying out the motivation system. The validators are there to look after the ledger, verify transactions validity, and participate in the consensus/leader-election procedure using the Proof-of-Stake (PoS) method. Clients are individual participants who train the model locally and send back their improvements to the network. The interaction between these three types of entity is completely self-contained.

A lister sends a 'listing transaction' that lays out the metadata of a brand new model and sends a pointer to the initial parameters stored off-chain. Clients interested in the model download those parameters, run training on their own private data, and send an 'update transaction' that includes a reference to the new insights they've made. It is effectively finalizing the combined model and making its new parameters available to anyone in the network, when the lister or any node that is empowered to compile the data sends out a global update.

IPFS (Section 2.6) gives us content-addressable storage and decentralized file distribution, so that large model parameters do not slow down the blockchain. Instead, only the `CIDs` are stored on the blockchain, connecting each transaction to its corresponding model files, and this system ensures both the scalability and the Web3 principle of separating data availability from consensus.

State and artifacts. On the chain we keep only tiny canonical items: ModelIndex, RoundIndex, Tickets (reservation records), receipts. The off chain, large tensors sit in IPFS as `CID` blobs. No raw data ever leaves Clients.

Interaction loop

1. *Listing.* A Lister may upload initial weights to IPFS and send a `ListingTx` that binds a `modelID` to a `initCID`. (see Section 3.2.1, Fig. 4.3)

2. *Reservation.* A Client first locks a deposit by sending to the validators an `UpdateIntentTx` to receive a `ticketID` for ⟨model, round⟩. (see Section 3.2.2, Fig. 4.14)

3. *Local train and publish.* The client then trains locally, puts its update on IPFS, and then reveals the `updateCID` through an `UpdatePublishTx`. (see Section 3.2.2)

4. *Aggregation.* Lister aggregates (i.e, weighted FedAvg) the revealed updates, which he has fetched from IPFS, and posts the global checkpoint, getting a `globalCID`. (see Section 3.2.3.)

5. *Finalization.* Lister submits a `GlobalUpdateTx` carrying `globalCID`, `included[]`, `scores[]` (one-to-one with `included[]`), and `refunds[]`. Validators advance the round and *compute* entitlements deterministically from `scores[]` and model `rules`. (see Chapter 4, Alg. 7, 38, 36.)

6. *Claim.* After finalization, participants withdraw refundable deposits and round rewards by submitting a `ClaimTx`. (see Chapter 4, Alg. 17 and 22.)

3.2 Transaction Model

When structuring the federated learning workflow, we have opted to keep it very lean on-chain, and the summary in Table 3.1 shows what is needed for the basics. In Chapter 4, you will find the executable pseudocode that corresponds to the process. The following subsections include listing transactions, update transactions, and global update transaction models.

3.2.1 Listing Transactions

It starts with a listing transaction, which is performed by the lister node. When uploading their model, a Lister sends the initial weights to IPFS, generates the `initCID`, and submits the `ListingTx`, which contains `modelID`, `initCID`, and rules, all listed in the Table 3.1 (transaction schemas), to the validators. Validators are going to only let the transaction go through if the model doesn't exist, and the `initCID` is valid, as per Ch 4, Algorithm 19. The Lister can also publish an IPNS alias to make it easier to find the model, but the real authority is the chain that links the `modelID` to its `initCID`.

3.2.2 Update Transactions(Two Phase scheme)

When a client wants to upload the results of a training session, they send an `UpdateIntentTx` to the gateway interface with the current `modelID` and round number, locking a deposit and getting a `ticketID`, so spammers cannot send anything to the network. This is basically

a preauthorization/defense check, as described in Ch 4 Section 4.5. After training the local model on its own private data, serialize the new weights, upload them to IPFS to get an `updateCID`, and then send a ticketed `UpdatePublishTx` containing the ticket ID, `updateCID`, and some metrics. Validators verify that the client is the rightful owner of the ticket, check if it has expired, ensure that they are not rate-limited, and confirm that `CID` is in the correct format.

3.2.3 Global Update Transactions

The Lister aggregates the revealed updates(e.g. with weighted FedAvg for admitted tickets/updates), then uploads the global checkpoint to IPFS to get `globalCID`, and sends a `GlobalUpdateTx` with the ordered `included[]` ticket IDs, a parallel vector of **`scores[]`**, and `refunds[]` (tickets whose updates were retrievable during aggregation) to the blockchain network with the ordered list of ticket IDs.

Validators accept the transaction(if valid), they increment `RoundIndex` for the given model, and *compute* per-account entitlements from `scores[]` (per `rules`) and refundable deposit according to the lister supplied `refunds[]` list, then release those funds via subsequent `ClaimTx` calls. Clients later withdraw via `ClaimTx`, all these details can be seen in Chapter 4, Algorithms 5, 36, 7, 17.

3.2.4 Implications of the Transaction Model

When we view federated learning as a series of transactions, breaking down the logic into on-chain objects (Listings, Tickets, Publications, and Global Updates), we give ourselves a straightforward way to check in on how the system is working, hold people responsible for what they put out, and seal the record of progress forever. Returning to a traditional federated learning setup, where the aggregator is basically the only one we can trust, is no longer an option. Now, every single phase of the procedure can be verified by the community of nodes. Concepts such as token-based rewards, slashing penalties for bad contributions, and stakes for those who list, can be constructed on the back of this idea.

Object	Fields / Invariants
`ListingTx`	$\{$*modelID*, *initCID*, *rules*, *fee*$\}$. Invariants: *modelID* unused; *initCID* is `CIDv1`; *rules* fix deposit, deadlines, per-account caps.
`FundingTx`	$\{$*modelID*, *amount*$\}$. Invariants: signer is the model lister (or delegated funder per `rules`); *amount*>0; increases $R_{m,r}$ for the present round.
`UpdateIntentTx`	$\{$*modelID*, *round*, *deposit*$\}$ $\Rightarrow$ on-chain `Ticket`(*ticketID*, *owner*, *expiry*, *revealed* $=$ `false`, *deposit* $=$ d). Invariants: round open; deposit available; rate-limits respected.
`UpdatePublishTx`	$\{$*ticketID*, *updateCID*, *metrics*$\}$ sets `revealed=true`. Invariants: caller owns *ticketID*; *updateCID* valid; within expiry.
`GlobalUpdateTx`	$\{$*modelID*, *round*, *globalCID*, *included[]*, *refunds[]*, *scores[]*$\}$. Invariants: round matches `RoundIndex`; all *included* and all *refunds* are revealed tickets for $\langle m, r\rangle$; *refunds[]* is a subset of revealed tickets; *scores[]* is a nonnegative real vector aligned with *included[]* (same length); *globalCID* valid.
`ClaimTx`	$\{$*acct*, *modelID*, *amount*$\}$. Invariants: `Entitlement[acct,modelID]` $\geq$ *amount* $>$ 0; debits entitlement and credits balance atomically; idempotent via tx nonce.

Table 3.1: Canonical transaction objects and their checks, including refunds and claiming.

3.3 Consensus and Sybil Resistance

Looking at the heart of any blockchain system, you will find the issue of consensus. Getting multiple, potentially competing nodes to agree on the state of a shared ledger. In the realm of decentralized federated learning, this is an area of high importance, as the correctness of the blockchain history is paramount to the training process. If one node can manipulate the chain, the global model would be irreparably damaged. When it comes to ordering transactions in our system, we use the longest chain consensus protocols, and to make our system permissionless and safe from permissionless attacks, we choose the PoS Sybil-Resistance

mechanism for leader election and Sybil attack defense. We have already discussed in Chapter 2, specifically in Sections 2.5 and 2.5.1. Here we will explain how these two concepts are used in our system to lock in the order of transactions(`ListingTx`, `UpdateIntentTx`, `UpdatePublishTx` and `GlobalUpdateTx`) and power the rounds of federated learning.

For the concept of time, the validator set and stake are also an integral part of our system. The time is divided into epochs, each composed of a fixed number of time slots, with an active validator set($\mathcal{V}$), during each epoch. Each validator in $\mathcal{V}$ has a non-zero stake $s_i > 0$, and the sum of these stakes is S. Becoming a validator is not easy; you have to lock your stake in the blockchain, and there is a grace period [77] before you can remove it. It is this bonding mechanism and their prospective slashing that gives us Sybil resistance; people cannot just create loads of fake identities to manipulate the system, they need to put up real stakes.

The system also uses a unique block proposer election process that all consensus protocols need. Each time slot t in an epoch, a new block proposer is randomly chosen, but not entirely at random. The probability of a particular validator being chosen is proportional to its stake. What we do is run a verifiable random function (VRF) [78] on the epoch randomness R and the secret key of each validator to obtain a result $\rho_{i,t}$ that determines whether the validator i is eligible to propose in that time slot t iff $\rho_{i,t} < \tau \cdot \frac{s_i}{S}$, for a system threshold $\tau \in (0, 1]$. When multiple blocks are eligible to be chosen as the next block, the block with the lowest $\rho_{i,t}$ or smallest VRF proof is designated the canonical one, virtually guaranteeing that tiebreakers are extremely rare. This aligns with the proposed algorithm in chapter 4, where the selected block's proposer, who has fed their block into the system, signs off on it, and fellow nodes have confirmed the signature, allowing the block to be included in their fork-choice rule.

The contents for any given block are governed by ordered lists of accepted transactions and verifications, and structural validity is limited off the critical path. Signature verification, rate limitations, fee/deposit checks, and `CID` format checks, following Chapter 4(Section 4.4). Unconstrained ordering is allowed, with one exception. The consensus-level causality requires that `ListingTxs` precede any `UpdateIntentTx` or `UpdatePublishTxs` for the same `modelID`, an `UpdatePublishTx` must reference a live, unrevealed ticket created by a client with `UpdateIntentTx` in the same round, and `GlobalUpdateTx` must reference the current round index and only include tickets already revealed in this or earlier blocks of the rounds. This would be in accordance with our validator state machine and its applyTx procedure.

3.3.1 Longest-Chain Consensus

When on the blockchain, validators have to choose between multiple competing tree branches, they use the longest-chain rule. From the beginning, the branch that has the largest number of valid blocks(since the beginning) is given the top spot. Deadlocks can be avoided by applying a fixed rule to break the tie, like sorting heads of the branches by the highest stake chain of their block hashes. Under partial synchrony and the majority of honest stakeholder assumptions, this rule essentially guarantees that eventually the network will settle on one branch. Since the number of honest proposers is so high, they will all build from the same top block, and smaller branches that fall behind will be abandoned.

Finality is not guaranteed on the blockchain(probabilistic), but relies on the use of something called confirmation depth, k, to approximate finality. This parameter k is given to upper layers like Lister and Clients and is used exactly the same as in Chapter 4, where write paths `SEND_AND_WAIT` will not return until k confirmations have been made, and the read APIs will show the state as it was k blocks ago, so that applications know exactly what to expect, increasing this parameter k reduces the small probability of a reorganization of the chain that would displace a transaction.

3.3.2 Sybil Resistance through Proof-of-Stake

The stake of the validator is its weight; the greater the stake, the more chances to propose a block. Let the total stake be $S = \sum_j s_j$. The validator i holds s_i. Its stake fraction is

$$\alpha_i \;=\; \frac{s_i}{S}.$$

Per-slot probability. Every time slot runs a VRF-based lottery. The lottery uses a threshold that grows with the stake. A common form is

$$\Theta(\alpha) \;=\; 1 - (1-\tau)^{\alpha},$$

where $0 < \tau < 1$ is the *active slot coefficient*. Controls how often a slot has a leader [79, 41]. The validator i wins the slot t with probability

$$\Pr[i \text{ proposes at } t] \;=\; \Theta(\alpha_i) \;=\; 1 - (1-\tau)^{\alpha_i}.$$

When τ is small (the usual case), $(1-\tau)^{\alpha_i} \approx 1 - \tau\,\alpha_i$. Then

$$\Pr[i \text{ proposes at } t] \;\approx\; \tau \cdot \frac{s_i}{S}. \tag{3.1}$$

Tiny numeric check. Take $S = 1{,}000$ and $s_i = 50$ (so $\alpha_i = 0.05$). Let $\tau = 0.10$. Then by (3.1), $\Pr[\text{propose}] \approx 0.10 \times 0.05 = 0.005$ (about 0.5% per slot). The exact value is $1 - (1 - 0.10)^{0.05} \approx 0.00512$, very close.

Why splitting stake does not help (Sybil resistance). Suppose that an adversary splits the stake α between m identities with shares $\alpha_1, \ldots, \alpha_m$ and $\sum_j \alpha_j = \alpha$. The probability that at the very least one identity wins is

$$1 - \prod_{j=1}^{m} \big(1 - \Theta(\alpha_j)\big) \;=\; 1 - \prod_{j=1}^{m} (1 - \tau)^{\alpha_j} \;=\; 1 - (1 - \tau)^{\sum_j \alpha_j} \;=\; \Theta(\alpha).$$

It depends only on the total stake, not on the number of keys/IDs, by that making many identities (a Sybil) gives no extra edge.

Simple example. Let $\tau = 0.05$ and the total stake $S = 1000$. Example 1: one key with $s = 100$ ($\alpha = 0.1$). Win chance: $1 - (1 - 0.05)^{0.1} \approx 0.00488$. Example 2: divided into five keys with 20 each ($\alpha_j = 0.02$). Each key has a chance $1 - (1 - 0.05)^{0.02} \approx 0.000976$. The chance that at least one wins is $1 - (1 - 0.000976)^5 \approx 0.00488$, the same as in example 1.

Control by stake, not identity count. Block production follows stake, not how many identities an adversary can create [41, 39, 65]. In many slots T, the count of wins for the validator i was concentrated around $T\,\Theta(\alpha_i) \approx T\,\tau\,\alpha_i$. More stake means more blocks in expectation.

Slashable faults and "nothing-at-stake." We define two slashable faults. *Double proposal* (two blocks in the same slot). *Double voting* (conflicting votes at the same height). Both create on-chain proof (signatures). A violator loses a fraction σ of its locked stake and is removed from the active set for an epoch. With a large enough σ, the classic "nothing-at-stake" behavior is deterred.

Liveness under partial synchrony. Assume network delay is bounded by Δ (partial synchrony) [10]. Then the honest majority of stake keeps extending the longest chain in expectation. Choose a modest block interval and a small confirmation depth k. Listers and Clients then see a stable view during an FL round: forks resolve, and the chain keeps moving.

3.3.3 Consensus invoking Federated Learning Interface

When it comes to the consensus component in the framework, it is essentially a single, append-only timeline for FL that has a specific interface and is governed by four main invariants, as dictated by the changes in the state of the validators.

The existence of the model as the lister with `ListingTx` initializes `ModelIndex[modelID]` with the initial `CID` and sets the round number to zero, and no transactions referencing this model ID were valid before this point. The ticket-creating block discipline in `UpdateIntentTx` for the pair of modelID and round number locks up a deposit and generates a ticket that is owned, has a specific expiration time, and initially has its revealed flag turned off. The limits on rates and deposits make it very expensive for spammers to send intents, which is in addition to the Sybil resistance built into the validator layer.

Update binding takes place in `UpdatePublishTx`, where the revealed flag of the ticket is flipped to true, and the ticket is bound to a `CID` on IPFS along with some metrics; the blockchain never verifies the actual data. All it does is validate the structure of the `CID` and whether the ticket is in the correct state. In this way, the binding is transparent and does not overload the network.

Round progression stage in `GlobalUpdateTx` for a particular pair of `modelID` and round number, which takes a `globalCID` and a list of tickets that must already be out in the open, and after applying the transaction, automatically increases the round number $rr + 1$, gives rewards from `scores[]` per `rules`, and credits refundable deposits for `refunds[]`, and establishes the `latestGlobalCID` of the next training step.

All these invariants guarantee that the off-chain FL pipeline(downloading the global model, training it locally, uploading the updates off-chain to IPFS, and aggregating)is reproducible and that every on-chain step has a unique causal predecessor.

3.3.4 Parameters and Security Envelope

When exposing the main consensus parameters to the application, we consider the block interval(slot duration), the number of confirmations required(confirmation depth k), the length of each epoch (number of slots per epoch) and the slashing penalties. With the duration of each block can make the learning process happen faster but leads to a higher number of orphaned transactions. We have found a sweet spot for the block length that is acceptable for the network's propagation limits.

The confirmation depth, or k, is calculated to ensure that the probability of a hacker re-organizing the blockchain and displacing a `GlobalUpdateTx` during a round is basically zero, given our assumption of a majority honesty stake. The slashing penalties are tuned to make sure that the financial incentive for a hacker to try to equivocate is eliminated.

We are using two layers to combat Sybil attacks. At the consensus layer, the use of PoS gives us a direct link to the amount of stake that is bonded by a given entity. And, at the application layer, the `UpdateIntentTx` deposit system and per-account limits prevent malicious clients from sending an overwhelming number of low-cost tickets to try and flood the network, with content addressing(IPFS CIDs) and auditability working in conjunction to make this system simple, clear and defensible.

3.4 System Assumptions and Incentives/Tickets

When our system is operational, there are a few things that we are counting on to be true; these are the minimal operating assumptions. The way we get participation out of our clients, listers, and validators is through a system of incentives and tickets, and in this section, we lay out these rules. We will not repeat the background information that we have already covered in Chapter 2 and in Section 3.3, but we are going to pin down the guarantees to specific rules that these parties must follow.

3.4.1 Operating Assumptions

Network model. We consider the partial synchrony assumption as laid out in Section 2.4, under this assumption the longest-chain PoS protocol (see Section 3.3) guarantees eventual convergence for an honest majority of bonded stake.

Stake distribution and honesty. When considering the set of validators in the system $\mathcal{V}$, more than half of the stake in the bond is in the hands of parties who follow the protocol. However, adversaries are completely free to do as they wish. Except they're unable to fake Ed25519 [80] signatures, second-preimage multihashes or VRF proofs.

Storage availability. The CIDs referenced on the chain resolve with high probability through IPFS (see Section 2.6 and Chapter 4). The inaccessibility of a referenced update during ag-

gregation is handled by omission, preserving the liveness of the round.

3.4.2 Economic Objects and State

When using the protocol on-chain, each model m and per round r have its own *reward pool* $R_{m,r}$, supplied by the Lister, and a personal *deposit* balance that helps update reservation. The protocol also generates *tickets* that are created by `UpdateIntentTx` transactions, these tickets contain a unique `ticketID`, `owner`, `modelID`=m, `round`=r,`expiry`, alongside that, Refunds and rewards accrue to an `Entitlement[acct,modelID]` mapping, which clients later withdraw via `ClaimTx`. These objects correspond to the state transitions implemented by Validators in Chapter 4(Section 4.4).

3.4.3 Ticketing and Deposits: Anti-Spam and Skin-in-the-Game

When a client wants to contribute in round r of model m,they must first reserve a publication slot by submitting `UpdateIntentTx` that locks a deposit $d > 0$ that will be returned if they re-expand(see Algorithms 13, 14). Admission creates a ticket with `revealed` $=$ `false` and a deterministic `expiry` derived from the round deadline. The ticket becomes eligible for aggregation only after the client publishes a valid `UpdatePublishTx` that binds the ticket to a `updateCID` and flips `revealed` $=$ `true` (Alg. 16).

Deposits do three purposes: they slow down spammers, make empty tickets expensive, and cover the financial penalties for publishing malformed submissions. When a client wants their deposits back, they can claim it deterministically(see Algorithm 17) but only if they meet the following requirements/conditions.

Deposit refund policy. Let $T_{m,r}$ be the set of all tickets for round r of model m. Each ticket $t \in T_{m,r}$ places a deposit d_t. The rule is simple and has three cases:

1. If t *expires unrevealed*, the deposit is *forfeited*.
2. If t is *revealed* with valid `CID` and Lister can *fetch the bytes* during the aggregation window (Algorithm 5), the deposit is *refunded* even if the update is not included in the global model.
3. If t is revealed but the bound `CID` stays *unretrievable* throughout the window, the deposit is *forfeited*.

This policy covers *availability*, it does not ask Validators to judge off-chain data. Validators read only on-chain fields and apply fixed rules.
How refunds move on-chain. At the end of the round, the Lister lists all refundable `ticketID`s in `refunds[]` inside `GlobalUpdateTx`. Each Validator then, deterministically, credits those deposits to `Entitlement[owner, modelID]`. At the same time, the validators compute the splits of the rewards from `scores[]` independently of the refunds. Thus:

- Good *availability* ⇒ deposit refund.
- Good *quality* (high score) ⇒ reward share.

A ticket can earn a refund without a reward, a refund without a reward, both, or none, depending on availability and score.
What 'retrievable' means. 'Retrievable' means the Lister can pull the bytes that the `CID` names within the window, using the retry logic in Algorithm 5. A syntactically valid `CID` alone is not enough; the bytes must be reachable.
Example. Suppose that there are three tickets: t_1, t_2, t_3.

- t_1 is revealed; its bytes are retrievable; its update scores well and gets included. Result: refund d_{t_1} *and* a reward share.
- t_2 is revealed; its bytes are retrievable; its update has a low score and is not included. Result: refund d_{t_2}, no reward.
- t_3 expires unrevealed (or is revealed but never retrievable). Result: forfeit d_{t_3}.

This separation keeps incentives the rule is simple, keep your data online to get your deposit back, and submit useful updates to earn rewards. Next, we detail Scoring, Inclusion, and payouts.

3.4.4 Scoring, Inclusion, and Payouts

When a lister runs in round r it gathers all newly revealed updates $\mathcal{U}_{m,r}$ (e.g. Alg 4 and 5), performs deterministic inspections to validation/shape, and computes a *score* $s_j \geq 0$ for each client update $j \in \mathcal{U}_{m,r}$. These scores are carried on-chain with `scores[]` with the `GlobalUpdateTx` payload (via the ordered `included[]` list and optional per-update metrics), making the round reproducible.

Score definition. To establish a simple, auditable score that integrates the declared local size alongside the observed quality on a public validation subset, as evaluated by the Lister. When validating the results, we score our local samples using a clear-cut plan that regards the declared sample size and the quality of our results on a public validation set, as measured by the Lister. Formally,

$$s_j \;=\; \max\big(0,\ \delta_j\big)\cdot w_j, \quad \delta_j \;=\; \mathrm{Acc}(W_j, \mathcal{D}_{\text{val}}) - \mathrm{Acc}(W_{\text{base}}, \mathcal{D}_{\text{val}}),$$

where W_{base} is the previous global checkpoint, W_j is the candidate weights, $\mathrm{Acc}(\cdot)$ is a fixed deterministic metric, and w_j is a capped proxy for the size of the data (e.g. $w_j = \min\{n_j, \bar{n}\}$ with declared examples n_j and cap $\bar{n}$ to limit manipulation). Any candidate not improving $\delta_j \leq 0$ does not contribute a score, but can still be included if Lister's robust aggregation requires coverage. The metric, cap, and the chosen validation split are fixed by the model's `rules` in the original `ListingTx`. For on-chain determinism, the Lister also includes the per-ticket s_j values as `scores[]` inside `GlobalUpdateTx`.

Inclusion rule. In order to for the 'global update' to be made, the Lister, uses a robust deterministic aggregator, that is essentially the weighted FedAvg with outlier skipping as outlined in Alg. 5 over the subset $\mathcal{I}_{m,r} \subseteq \mathcal{U}_{m,r}$ that passed shape checks and was retrievable. The ordered list of `ticketIDs` in `included[]` commits to $\mathcal{I}_{m,r}$.

Payout function. Let $R_{m,r}$ be the reward pool available for round r of model m after fees. Rewards are allocated only to included tickets with a positive score. Writing $S = \sum_{j\in\mathcal{I}_{m,r}} s_j$, we define each payout as

$$\mathrm{pay}_j \;=\; \begin{cases} R_{m,r}\cdot \dfrac{s_j^{\gamma}}{\sum_{k\in\mathcal{I}_{m,r}} s_k^{\gamma}} & \text{if } s_j > 0 \text{ and } S > 0, \\ 0 & \text{otherwise,} \end{cases}$$

with a curvature parameter $\gamma \in [1, 2]$ fixed in `rules`. The larger γ sharpens the incentives toward higher grade while preserving any approach transparent and replayable. Payouts are carried out by the validators that deterministically are computed from `scores[]` and `rules` and create *entitlements* that Clients claim via `ClaimTx` after the `GlobalUpdateTx` is confirmed (Alg. 17).

3.4.5 Penalties and Misbehaviour

Client-side misbehaviour. The publication of syntactically invalid `CIDs` is detected at admission and rejected (see Chapter 4.4). Publishing well-formed but unavailable `CIDs` results in deposit forfeiture as above. Repeated unavailability or systematic metric misreporting can trigger protocol-level rate limit tightening for the account in subsequent rounds (encoded as per-account caps in `rules` and enforced at admission).

Lister responsibilities. The Lister must pre-fund $R_{m,r}$ before round close, and must publish the exact ordered `included[]` tickets utilized for the aggregation, together with the `globalCID`. Because Validators re-execute only the state transition and not the ML evaluation, openness, and determinism (see Alg. 5) ensure anyone can audit the outcome from on-chain references.

Validator penalties. The consensus level equivocation (double proposal or conflicting votes for the same height/slot) is slashable under the PoS rules in Section 3.3. Application-level disputes never require validators to convey or interpret model bytes.

3.4.6 Putting It Together: Round-Level Evolution

A full round-level elaboration can be seen in Figures 3.2– 3.12. The Figures follow the algorithms laid in chapter 4. At the start of round r, the chain shows the last `globalCID` and the active `rules`. Clients lock a deposit to reserve a ticket, train on their own data, reveal by posting a `CID` (a content identifier) that points to their update on IPFS. The Lister brings the updates it can only retrieve, with a fixed rule, it aggregates them, scores them on $\mathcal{D}_{\text{val}}$, uploads the global model that is newly produced on IPFS, and submits `GlobalUpdateTx` with the new `globalCID`. Validators apply this transaction needing only one step, which moves the round to $r+1$, records who gets paid, and refunds or forfeits deposits under the refund policy. After finality, clients call `ClaimTx` to withdraw rewards and any refunds. Because any action ties to on-chain state and to content-addressed files, anyone can rerun the round and audit the result.

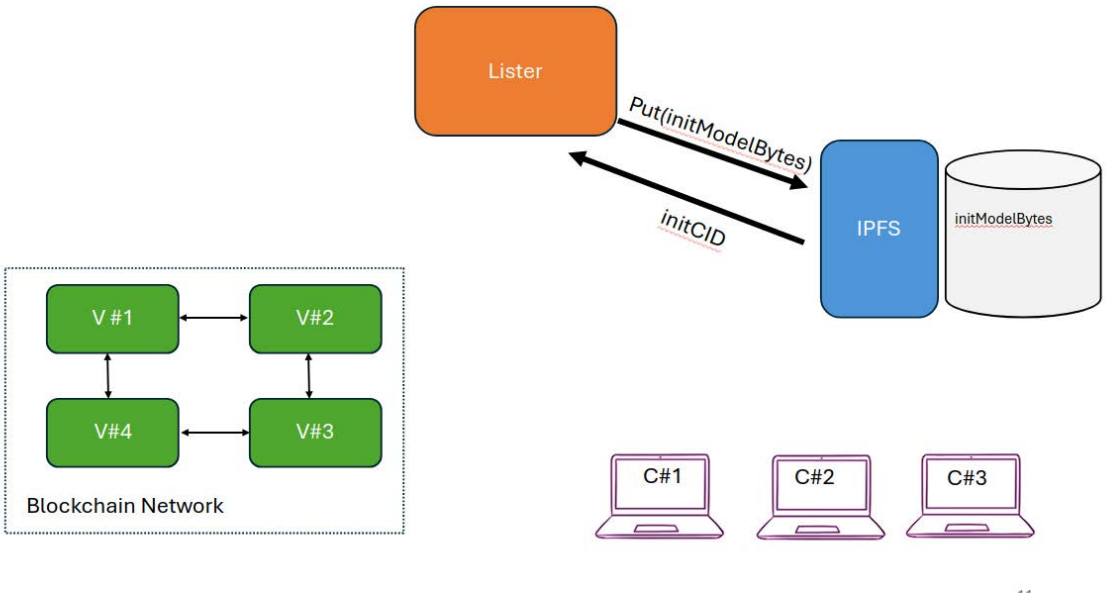


Figure 3.2: Lister uploads initmodelbytes to IPFS

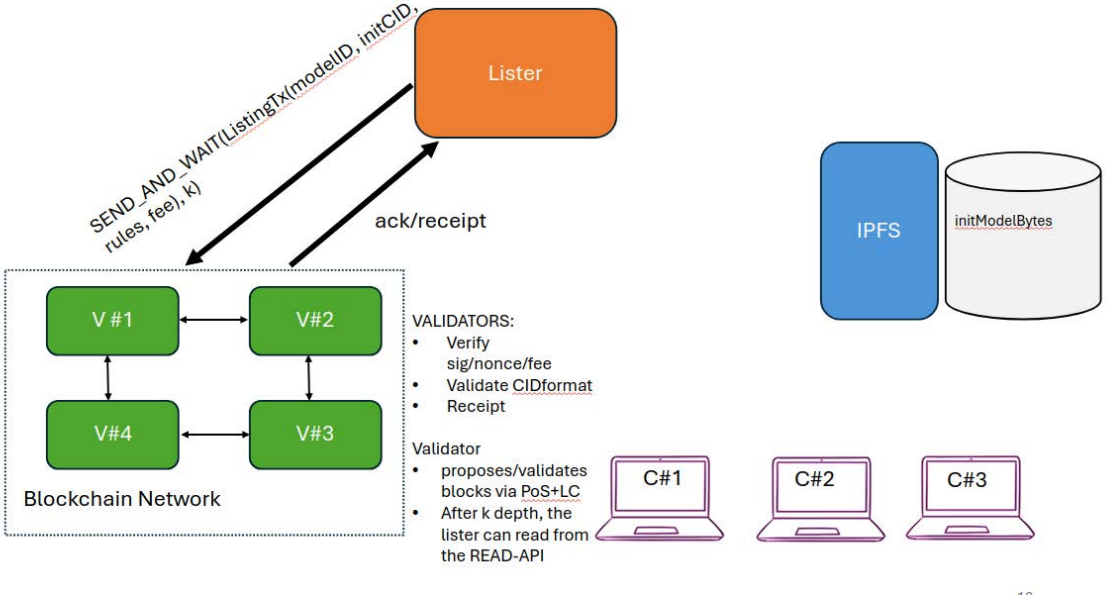


Figure 3.3: Lister performs ListingTx to the validator network

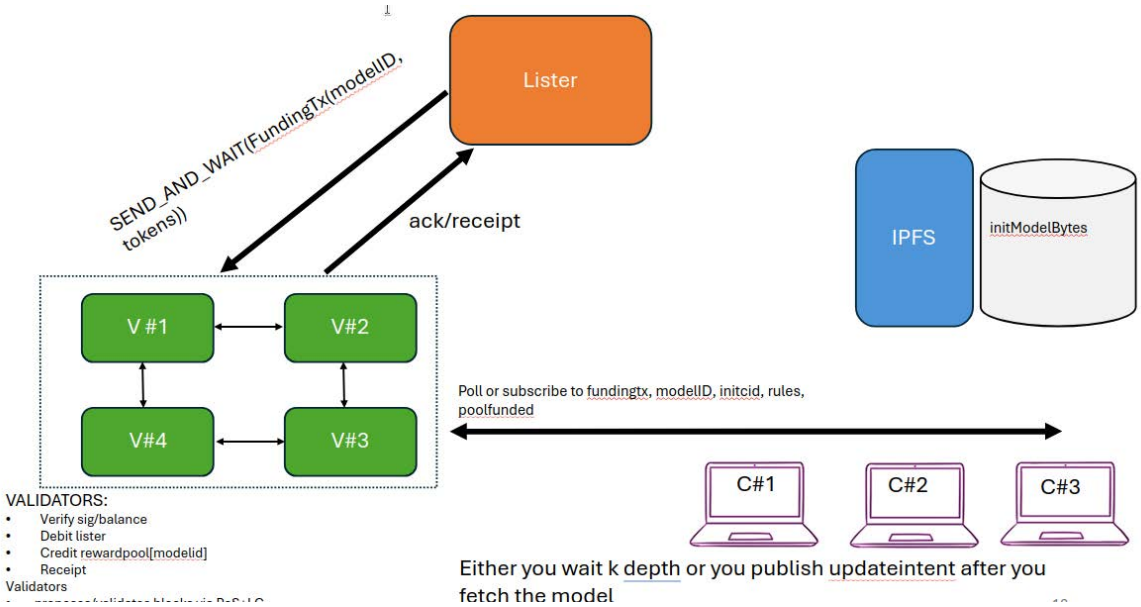


Figure 3.4: Lister does FundingTx to fund the model, and clients poll for the new FundingTx from validator network

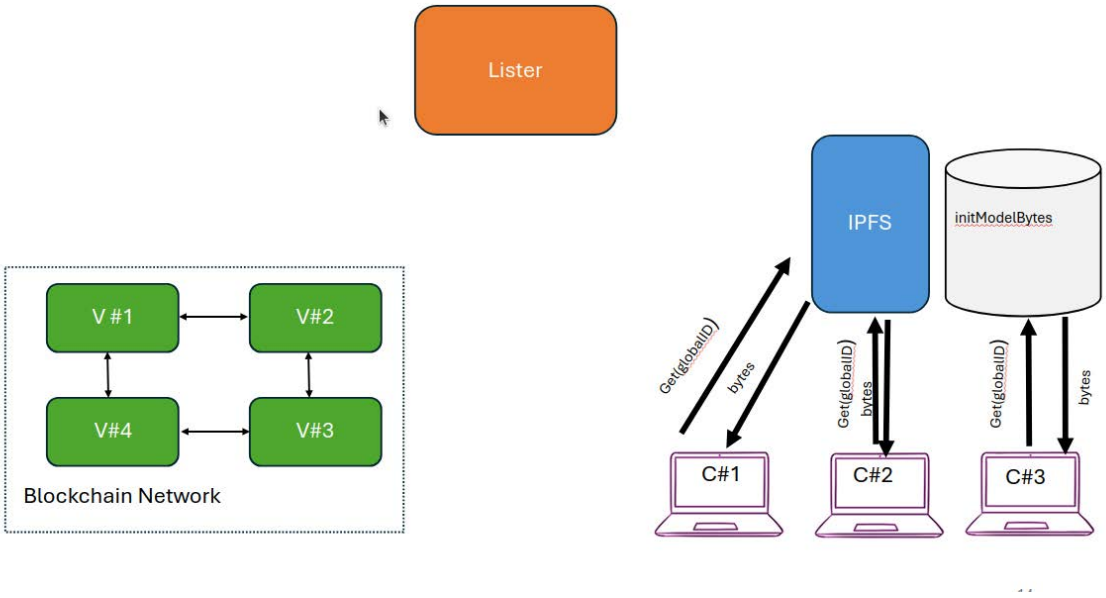


Figure 3.5: Clients yield initmodelCID from IPFS with the globalCID that they received from the validator network

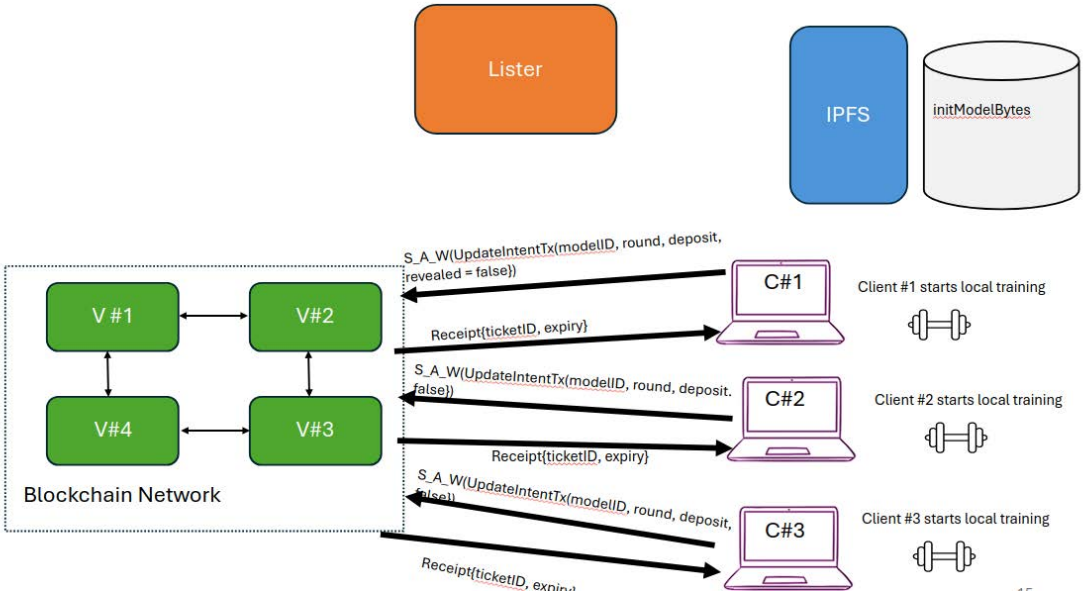


Figure 3.6: Clients publish UpdateIntent to the validator network to reserve a ticket with expiry date also the ticket currently has revealed = false parameter, then start training the model locally

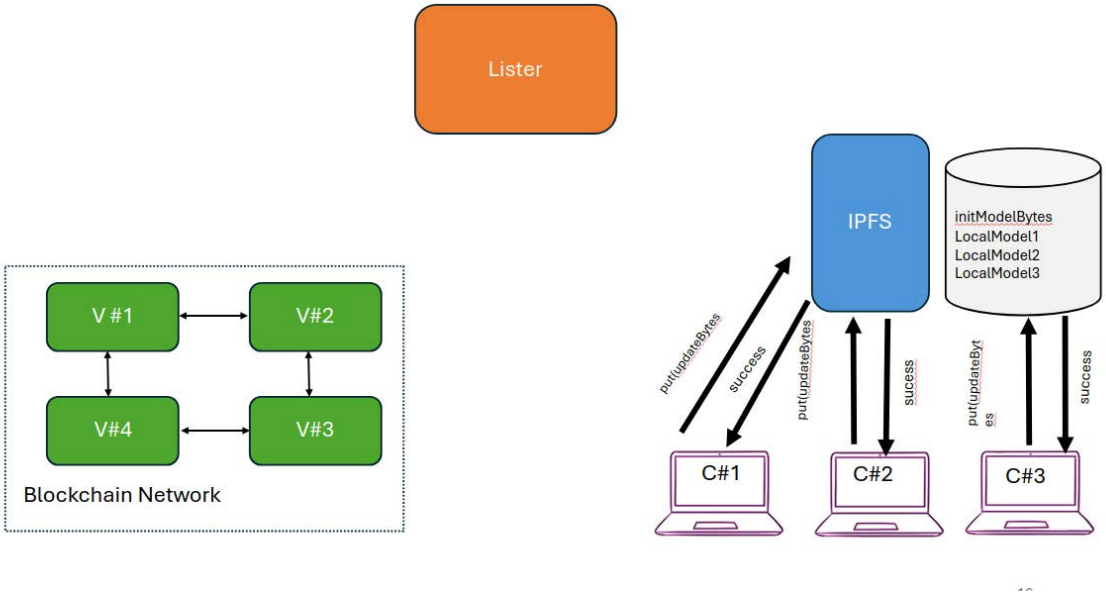


Figure 3.7: Clients put the localUpdateModel to IPFS, and receive the updateCID

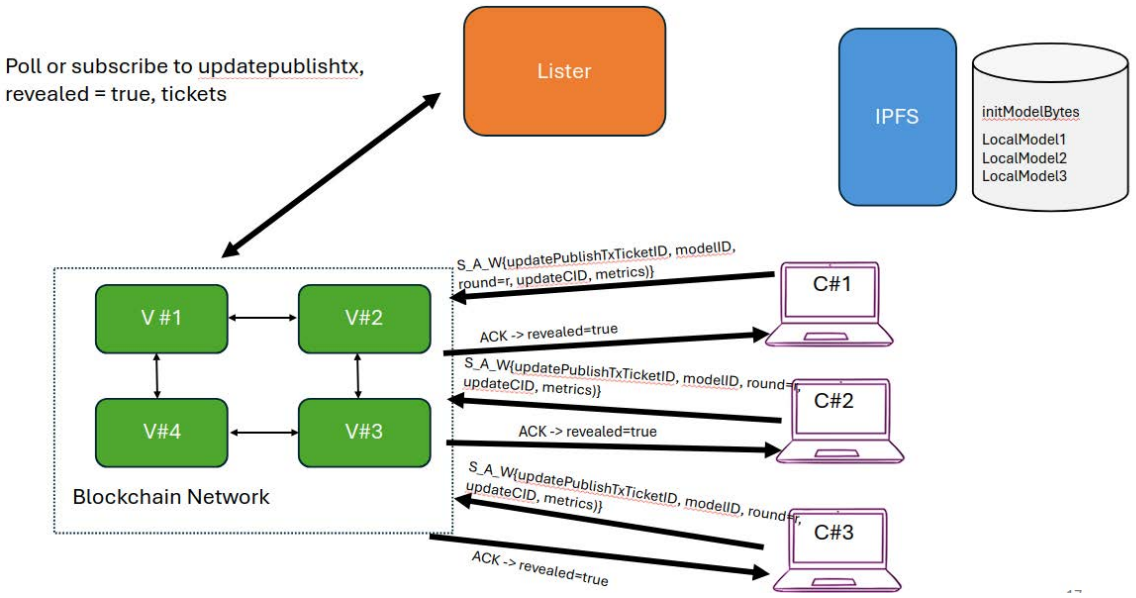


Figure 3.8: Clients perform updatePublishTx, and now the revealed parameter is turned into true, and the lister node can start yielding the updates(after k-finality)

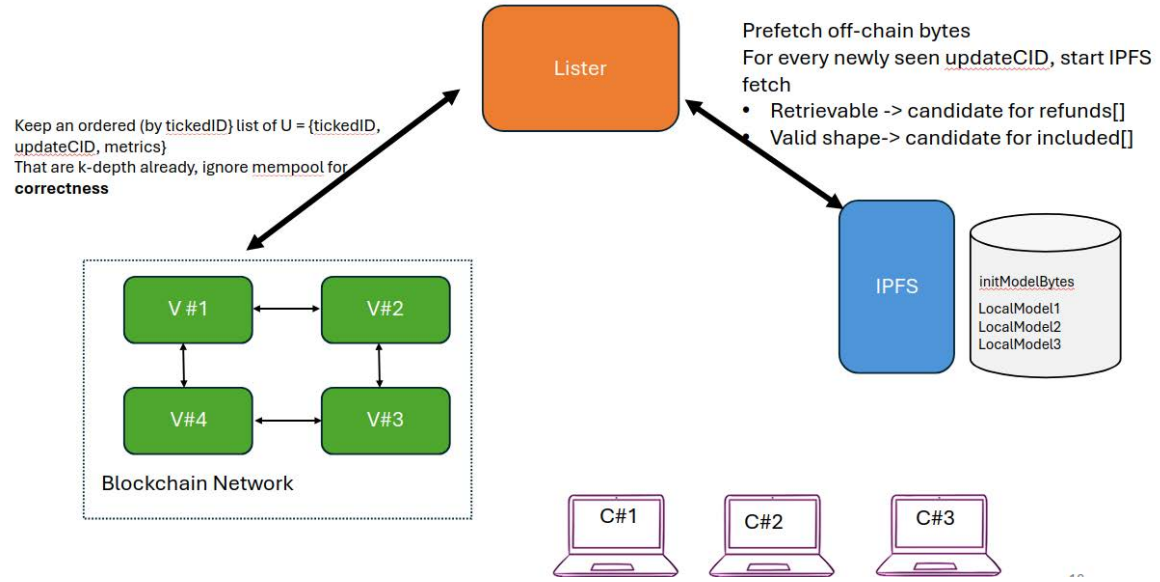


Figure 3.9: At the end of the round, the lister gets all the ticketIDs with updateCID, etc, and starts retrieving bytes from IPFS

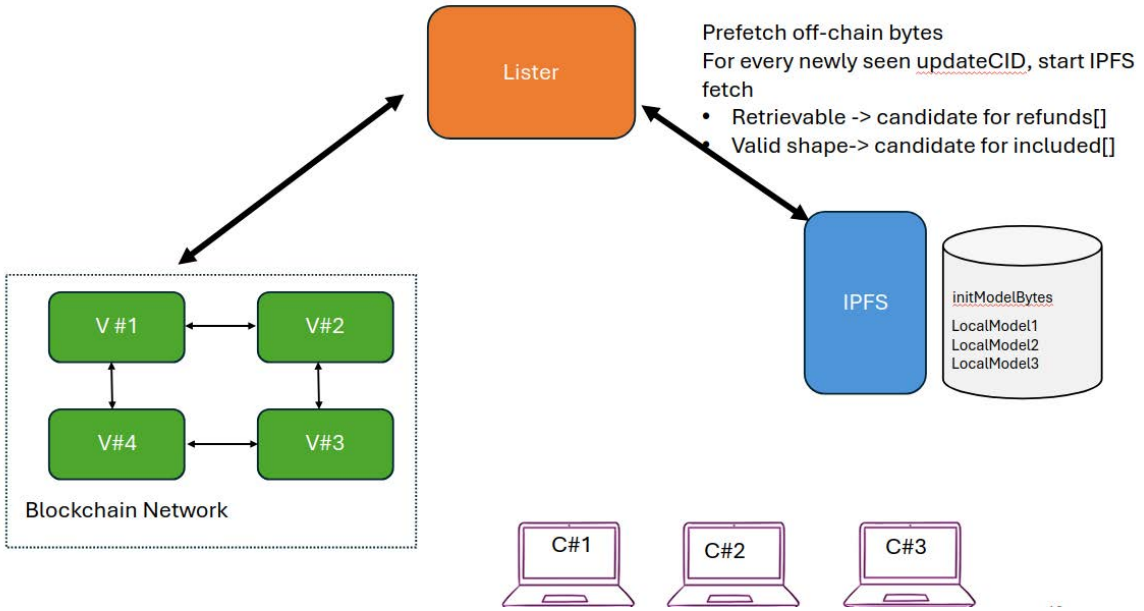


Figure 3.10: Lister fetches all the tickets, if retriavable put them in refunds, if valid shape, put them in the included list, then starts performing deterministic aggregation for all the included[] tickets

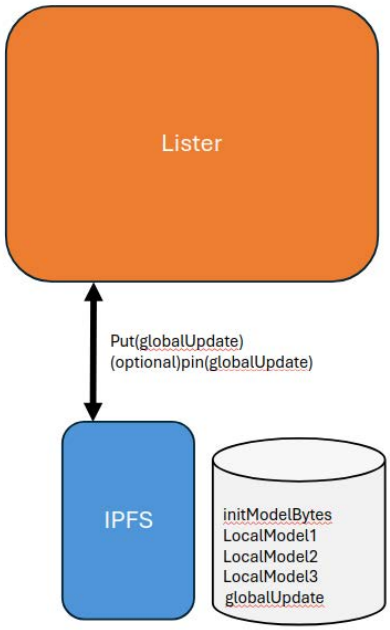


Figure 3.11: Lister uploads the aggregated GlobalUpdate model to IPFS off-chain storage, and pins the globalUpdate if needed per policy, and finally receives globalCID

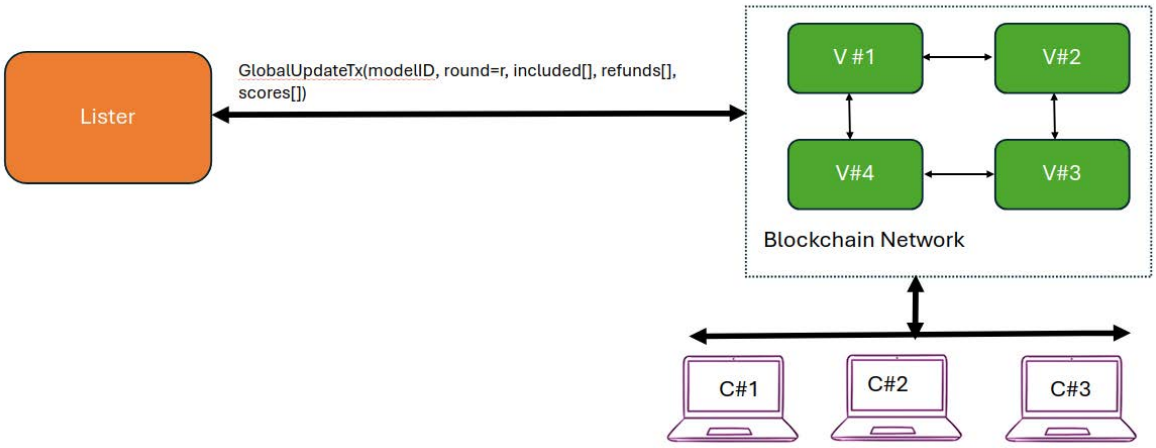


Figure 3.12: Lister performers GlobalUpdateTX with the modelID, globalCID, all the included lists, the validator network then checks the integrity of included and refunds(optional) if roundIndex higher, advance round and round continues, Clients can now ClaimTx and if they want continue with the rounds

3.4.7 Parameters of the protocol and Tuning

The application exposes four consensus-facing parameters to userspace: slot duration, confirmation depth k, epoch length, and slash fractions (see Section 3.3). The incentive layer adds: deposit size d, per-account intent caps, validation cap $\bar{n}$, curvature γ, and the round reward $R_{m,r}$. Smaller slots and k lower end-to-end latency but raise reorg risk; deposits and caps dampen Sybil pressure from clients, complementing PoS-based Sybil resistance in the validator layer(blockchain layer). All parameters are fixed in `rules` at listing time and are visible through the validator read API, so clients can decide economically whether to participate before allocating compute.

3.5 Security Analysis

This section formalizes the attacker/threat model and security goals of the proposed, permissionless, PoS & longest-chain setup, illustrating the composition of its consensus and application layer (tickets, deposits, aggregation, scoring). To unite these arguments with canonical notions of *chain growth*, *chain quality*, and *common prefix* that are underpinning current analyses of longest-chain protocols [81, 41].

3.5.1 Assets, Interfaces, and Goals

The protocol exposes a minimal on–chain interface (Table 3.1) and treats IPFS `CIDs` as unclear content addresses(with signatures each cid will be followed by on-chain binding). Primary assets include: (i) the canonical ledger state ModelIndex, RoundIndex, Tickets, entitlements(i.e, 'who, what, which round'); (ii) round reward pools $R_{m,r}$(i.e, the pot for model m in round r); (iii) client deposits(i.e, locked at ticket time; refunded or forfeited by policy); and (iv) off–chain artifacts referenced by `CIDs` (`initCID`, `updateCID`, `globalCID`), where initCID is the beginning model identifier, updateCID is a client update model identifier(trained locally), globalCID the aggregated/global model identifier, moving on, the goals are organized by layers:

Consensus-layer safety. Honest validators must probabilistically maintain a common prefix on the decided chain; conflicting histories should not be adopted after k confirmations.

Consensus-layer liveness. Under partial synchrony, new transactions from honest participants must be included within a bounded delay (modulo the fork-choice's k-deep view).

Application-layer safety. Transactions for each model m and round r must adhere to causality. Specifically: `ListingTx` initializes m; `UpdatePublishTx` can only bind a live ticket issued by `UpdateIntentTx`; `GlobalUpdateTx` must reference the current round, list only already-revealed tickets, and automatically increase $r \to r{+}1$.

Economic soundness. Rational participants should expect non-negative utility when following the rules. Detectable misbehavior will strictly decrease expected payoff via slashing (validators) or deposit forfeiture/rate caps (clients).

Privacy & integrity of learning. Raw data remains client-side; updates are content-addressed, and robust aggregation limits the influence of Byzantine updates on the global model.

3.5.2 Adversary Model

Network and stake model. We assume a *partially synchronous* network [10]. After some unknown but finite time, message delays are bounded. The set of validators $\mathcal{V}$ can change over epochs. Within an epoch it is fixed and known at the start. Each validator i bonds the stake s_i; the total is $S = \sum_{i \in \mathcal{V}} s_i$. At all times, we require an *honest stake majority*: the adversary controls strictly less than $S/2$ in any epoch.

Adversary powers. The adversary is adaptive and coordinated. It can:

- Delay, drop, or reorder messages in partial synchrony(e.g., it can hold back a client update, so it misses the current round, by dropping the publishment, or deliver blocks in a wrong order to create a short fork)
- Corrupt validators across epochs and, after corruption, use their keys and steer their behavior(e.g., but it still cannot forge signatures of validators it has not corrupted, and it cannot break standard cryptography).
- Create an unbounded number of client pseudonyms, basically flooding the system with many updates, tickets or requests; also try to sway the aggregator with volume rather than quality, but fake clients do not give it extra stake or extra validator power. In order to be a validator that wants to contribute in consensus, you still need to stake.

Cryptographic assumptions. Ed25519 signatures are unforgeable [80](widely deployed, very fast to verify). VRF outputs are unique and unpredictable [78]. Multihashes resist collisions (that is,, self-describing hash: algorithm + digest). We rely on these properties.

ML-layer threats. At the learning layer, the adversary may:

- Submit Byzantine updates (that is, garbage or malicious on purpose: arbitrary tensors, metrics, or `CIDs`).
- Launch targeted or untargeted poisoning(i.e, pushing the model toward the attacker's goal).
- Attempt reconstruction or membership inference from gradients [82].

Off-chain storage threats. At the storage layer (for example, IPFS), the adversary can deny availability for referenced `CIDs`(i.e, unpin, throttle, or refuse to server). It may do so selectively and at strategic times. The prototype design treats availability as an incentive problem, not a consensus one: validators do not judge off-chain bytes on the main path. The refund policy handles it.

3.5.3 Consensus-Layer Properties

The election of the leader in each slot t uses a VRF [83] on epoch randomness R: The validator i is eligible iff $\rho_{i,t} < \tau \cdot s_i/S$ where $\rho_{i,t}$ is its VRF output and $\tau \in (0, 1]$ controls the expected leaders per slot.

Under honest majority and partial synchrony, the resulting longest–chain protocol satisfies the standard properties [81, 41]:

Chain growth. There exists $g \in (0,1)$ such that given any window with T slots (e.g. $t_1 < t_2$ with $T = t_2 - t_1$), every honest chain C grows by at least gT blocks, *except with negligible probability*. Thus ensuring progress for Listings, Intents, Publications, and GlobalUpdates. Formally, an honest chain C satisfies.

$$|C(t_2)| - |C(t_1)| \;\geq\; g\,(t_2 - t_1)$$

Chain quality. There exists $\mu \in (0,1)$ such that given any window with ℓ consecutive blocks on an honest chain contains at most $(1-\mu)\,\ell$adversarial blocks, with overwhelming probability; hence censorship power is limited/capped unless majority stake.

Common prefix. For confirmation depth k, [84] the k-deep prefixes, given any two honest chains C_1 and C_2 are identical except with probability at most $e^{-\alpha k}$ that decays exponentially in k, given a constant $\alpha > 0$; consequently, when a tx is k-deep, the risk of reorganization is negligible. Formally, choosing k

$$k \;\geq \frac{1}{\alpha}\,\ln\!\left(\tfrac{1}{\varepsilon}\right)$$

gives the probability of reorganization $\leq \varepsilon$. The upper layers therefore gate `SEND_AND_WAIT` on k (Ch. 4, Alg. 2, 7).

Slashing and equivocation. Double proposals or conflicting votes at the same height/slot produce publicly verifiable evidence, basically the adverserial validator backs two competing histories at the same step, proposing two different blocks for slot s or voting for two different blocks at height h; the offender loses a slashing fraction $\sigma \in (0,1)$ of its bonded stake and is removed from the active set for an epoch, ensuring that 'nothing-at-stake' is strictly dominated economically [63].

Intuition. If you sign both sides of a fork at the same time, anyone can prove it with your two signatures; you lose σ of your stake , and sit out the next epoch, so for rational validators 'build on everything; is a bad deal.

3.5.4 Application-Layer Safety and Availability

Validators enforce causality via admission checks and deterministic state transitions (Ch. 4, Alg. 19, 38, 23). The following invariants hold for any honest chain:

Model existence. Any reference to a `modelID` must follow a confirmed `ListingTx` that fixed the initial `initCID` and `rules`; otherwise the tx is invalid.

Ticket discipline. Every `UpdatePublishTx` must bind a live, unrevealed ticket for $\langle m, r \rangle$ owned by the caller; publication flips `revealed=true` and is irrevocable at the state machine level.

Round progression. A valid `GlobalUpdateTx` must match `RoundIndex[m]` and list only already-revealed tickets. Applying it records `globalCID`, advances the round $r \rightarrow r{+}1$, updates entitlements, and releases or forfeits deposits according to the policy in Section 3.4.

Availability. Because validators never adjudicate off-chain bytes, availability is enforced by incentives: deposits are refunded only if the Lister can retrieve the bound `updateCID` during the aggregation window (Ch. 4, Alg. 5). Validator-side `CID` probes (Section 4.5) add observability but do not affect consensus.

3.5.5 Privacy and Integrity

Raw/Private data does not leave clients, updates(or checkpoints) are content-addressed chunks. To limit the effect of 'Byzantine' or poisoned updates, the Lister runs a deterministic robust aggregator [85] (e.g., trimmed-mean or coordinate-wise median) and a public validation score s_j (see Section 3.4) before constructing the global model. Such robust rules are known to tolerate a bounded fraction of Byzantine workers while keeping the estimator's error under control [86, 87]. Backdoor and targeted poisoning remain in scope, but the combination of shape checks, admission causality, and validation-based scoring shrinks their payoff surface. Membership inference [82] and gradient leakage [30] are mitigated at design time (no parameter server, no plaintext sharing between peers/nodes), and it can be further reduced with secure aggregation [29] or DP(i.e., by adding noise) if required [28, 27]; these are not consensus-critical and compose orthogonally with our transaction flow.

3.5.6 Economic Security and Incentives

The validator security comes from slashing, basically if found guilty of equivocating and there is public evidence(i.e., your block is not proposed and found later in the voting state of consensus): Any deviation that produces equivocation evidence yields loss $\geq \sigma s_i$, while honest participation yields the expected per-slot reward $r_{\text{blk}} \cdot s_i/S$. Setting σ and r_{blk} such

that $\sigma s_i \gg$ any plausible short-term gain from censorship or fork extension makes deviation strictly dominated by risk-neutral actors [63, 41], basically choose these two 'variables' to deter any cheating(i.e., not worth it short-term to cheat).

At the client layer, deposits and per-account intent caps *price* publication slots and blunt Sybil pressure. The payout rule is $\mathrm{pay}_j \propto s_j^{\gamma}$ with $\gamma \in [1, 2]$ (see Section 3.4) makes low-effort spam updates a losing strategy in expectation(i.e., tilts clients toward higher value contributions): unretrievable or invalid updates *forfeit* deposits; low-quality but retrievable updates are either excluded or earn *negligible* reward; high-quality updates maximize return. Because validators never retrieve or interpret tensors, all economic adjudication (refunds, rewards) depend only on the canonical state and the Lister reproducible aggregation, preserving consensus modularity(i.e, keeping consensus small and modular).

3.5.7 Denial-of-Service Vectors and Resource Depletion

Consensus-layer DoS (mempool flooding, block stuffing) is priced by fees and bounded by admission checks (signature, balance, stake). The application layer DoS through oversized model artifacts is prevented by (i) on-chain admission of *CIDs only*; (ii) IPFS gateway puts rate limitation along with size caps (token-bucket throttling) and preauthorization keyed to live tickets (see ipfs-implementation Section 4.5); and (iii) Lister-side hedged fetches and time-bounded aggregation (see Alg 5). Because `GlobalUpdateTx` can advance rounds while skipping unretrievable updates, an attacker cannot stall global progress by withholding blobs once tickets have been revealed; they only burn their own deposits.

3.5.8 Composability and Reorganization Handling

We treat pre-confirmation blocks as tentative. Every write waits for k confirmations. Every read returns a state that is given that the depth is k blocks (see Ch. 4). So applications do not treat a provisional state as final. Reorganizations can still occur. The chance decreases rapidly as k increases [81]. If a pending transaction is displaced, we retry. The client or the Lister makes the same call, while the call is idempotent. Causality checks stop replications, so no disagreeing state appears. The chain commits only to `CIDs` and ticket identifiers. A reorg cannot corrupt off-chain artifacts. It can only shift timing. At worst, the aggregation window moves by the reorg depth. For example, a two-block reorg may delay a publication by two blocks. It does not change the content. The best k depends on the latency tolerance. Some

deployments accept $k{=}2$. Others prefer, such as Bitcoin accepting $k{=}6$ blocks, meaning once they reach that depth, the chain becomes final [34].

3.5.9 Summary of Guarantees

Assume an honest majority of the bonded stake and partial synchrony. Under these conditions, the consensus layer provides safety and liveness. The probability is high [81, 41]. The application layer adds causality and availability, uses tickets, deposits, and deterministic state transitions. As far as learning, the use of aggregation rules and validation-based scoring is involved. These limit the influence of Byzantine updates on the global model [86, 87]. These defenses do not rule out attacks. They make them less influential plus less profitable. Incentives point in the same direction. Slashing and forfeits penalize misconduct. Fees, deposits, and caps price participation. The rational deviations then look unattractive. Privacy follows from the data path. Raw data never enters the chain. The on-chain entries are content addresses, not payloads. When stronger guarantees are required, add standard tools, such as differential privacy, and light cryptographic checks fit the transaction model [28, 27]. The performance and quality of the model still depend on context and parameters. Some examples include network partitions, stake turnover/reallocation, and skewed client sets. In such cases, detailed tuning and further study may be required.

Chapter 4

Implementation

4.1 Introduction

We gave a bird's eye view of the system, its essential conceptual architecture, and why we made the decisions we did, when we designed the conceptual framework in Chapter 3. The Implementation of the prototype is broken down into separable parts, algorithms, and pseudocode that show the flow of our system. We have opted for this abstraction to give readers the ability to see the logic of the software, without getting bogged down in the syntax of a particular programming language.

The prototype system is made up of four layers. Figure 4.1 shows the workflow of our system. At its base is the blockchain that oversees the chain, verifies transactions, and implements the consensus process. On top of that, the transaction management layer lays out the regulations and operations, like listing, updating, and global update transactions, the storage layer interfaces with the IPFS to store variables and updates outside the blockchain, and finally client layer orders training of local models, submits updates, and gets involved in the FL cycle.

The following sections go into detail about each of these separable parts with help from pseudo-code that shows the main steps of each process.

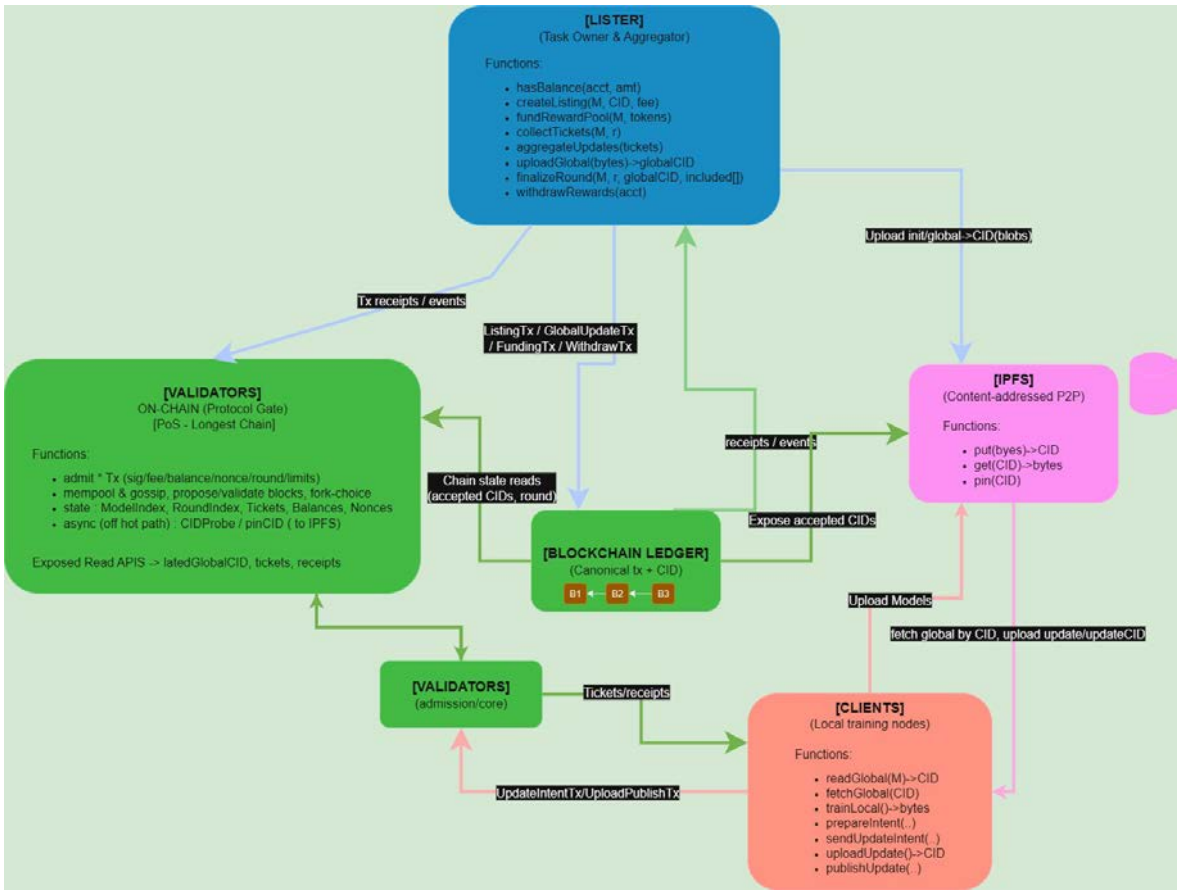


Figure 4.1: END-TO-END implementation view: nodes, state, and off-chain storage

4.2 Lister

Lister role. The Lister is the task owner and the round aggregator. It is a long-running contribution with many procedures, it creates listings and funds reward pools, collects client updates, runs secure aggregation, finalizes rounds, talks only to canonical chain state via validator gateways (read/write APIs), stores large files in IPFS as content-addressed artifacts (CIDv1 with SHA-256, `raw` codec), never will it bypass consensus on its own. Figures 4.2–4.9 show each operation. Algorithms 1 and 8 give exact steps. Ed25519 [80]is assumed for signatures, SHA-256 digests [88], CIDv1 with `raw`, and a validator read-API that discloses state after k confirmations.

Execution model. Each public call is synchronous from the Lister's point of view. Inside, it may fire asynchronous calls to IPFS or to validator gateways. A call returns only after the matching transaction is seen with k confirmations. Retries use 'EB' with jitter. Network

errors surface as typed exceptions. Reorgs are handled by idempotent resubmits. The goal is simple: caller logic and strong end-to-end guarantees.

Example flow. Create a listing and fund the pool. Collect revealed updates by `updateCID`. Aggregate with a fixed rule. Upload the new model off-chain to IPFS and get `globalCID`. Submit `GlobalUpdateTx`. Wait for k confirmations. Start round $r+1$.

4.2.1 hasBalance(*acct*, *amt*)

Figure 4.2 shows the simplest guard. Before any transaction, the Lister asks the validator read API for the account's balance. The given API yields the k-*confirmed* canonical value. The UML shows a one-way request and a one-way reply. No state changes on this path. If the balance is missing or below the set threshold, the call stops. No tx is built. No fees are spent. Algorithm 1 encodes this check. It makes sure that all value-moving steps begin with enough stake. Use it before funding pools, creating listings, or paying out.

Example. The Lister wants to fund $R_{m,r}$ with 100 currency coins/tokens. It calls `GetBalance(accour` The reply is 87 tokens at k-deep. This is below 100. The process yields an error message and halts. The Lister does not submit a funding tx. Later, after a top-up, the same call returns 140. The guard passes. The funding tx is built and sent.

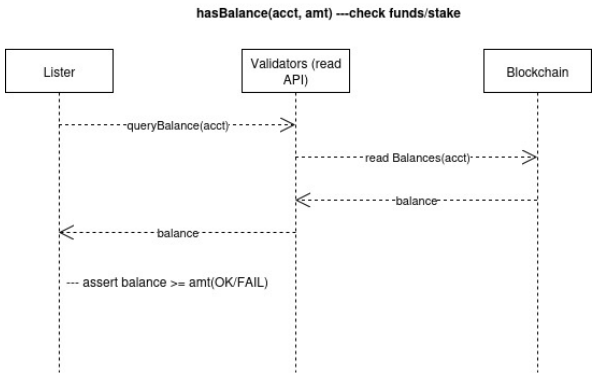


Figure 4.2: hasBalance(*acct*, *amt*) — checking funds against canonical state.

Algorithm 1 Lister.hasBalance(*acct*, *amt*)

function hasBalance($acct, amt$)
  $req \leftarrow$ {path: "/state/balances/"$\|acct$}
  $bal \leftarrow$ READ_STATE(req)
  if $bal = \perp$ **then**
    return false
  return $(bal \geq amt)$

4.2.2 createListing(modelID, initBytes, fee)

Figure 4.3 shows a two-phase flow. First, the Lister uploads the initial model parameters on IPFS, the gateway returns an initCID, a client confirms the CID locally, such as CID_CHECK, the node pins the listing bytes so that they stay available. Second, the Lister sends a signed ListingTx, the tx binds modelID to initCID, it also carries the owner account, a signature, and the initial rules (e.g., deposit, deadlines, rate caps). Validators run fast checks only, verify the signature and nonce, confirm the sender may create this modelID (not already taken), check the balance covers fees (and any minimum stake if required), validate the CID format (CIDv1, base32, raw codec, SHA-256 multihash), run schema checks and replay guards (idempotent: no second listing for the same modelID). No IPFS fetch is done on the consensus path. Then finally, if everything was correct, the tx is enqueued and later included in a block. On success, the chain initializes ModelIndex[modelID] (owner, initCID, rules) and RoundIndex[modelID] (start round). The algorithm 2 encodes these steps.

Example. Alice uploads the seed model. She gets `initCID = bafy...`, she sends `ListingTxmodel` `initCID=bafy..., rules, account=Alice, sig`. Validators check that `sentiment` is free, the signature is correct, the balance covers fees, and the CID parses as CIDv1. The tx lands in a block. After k, confirmations, the chain shows `ModelIndex[sentiment]` and `RoundIndex[sentiment]=0`. Alice can now fund the pool and start round 1.

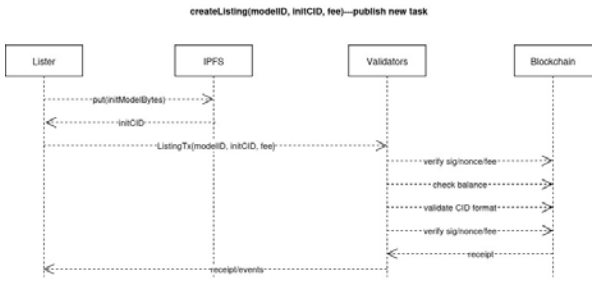


Figure 4.3: createListing(*modelID*, *initCID*, *fee*) broadcast a new model task.

Algorithm 2 Lister.createListing(*modelID*, *initBytes*, *fee*)

function createListing($modelID, initBytes, fee$)
  assert validateModelSchema(initBytes)
  $initCID \leftarrow$ IPFS_PUT($initBytes$)
  assert CID_CHECK($initCID, initBytes$)
  require hasBalance(self, fee)
  $exists \leftarrow$ READ_STATE({path: "/state/model/"$\|modelID$})
  if $exists \neq \perp$ **then**
    raise ModelAlreadyExists
  $tx \leftarrow$ {type: ListingTx, $modelID, initCID$, fee : fee}
  $rcpt \leftarrow$ SEND_AND_WAIT(tx, k)
  if $rcpt.status \neq$ Accepted **then**
    raise TxRejected
  return $rcpt$

4.2.3 fundRewardPool(*modelID*, *tokens*)

Incentives are locked in a pool that is scoped to one model. The Lister funds this pool by sending a signed `FundingTx`. Validators run fast assessments, verify the signature and nonce, assure the sender is the Lister's account, and load the current balances at the k-deep state. If the funds are sufficient, they debit the Lister and credit the model's pool. No IPFS bytes are read. Algorithm 3 adds two guards. First, it calls `hasBalance` before building the tx (preflight). Second, after confirmation, it re-reads the chain state and asserts that the post-state matches the expected amounts (allowing for fees). If the advanced round or balances do not match, the method aborts and reports an error. The effect is atomic: either the pool is funded, or nothing changes.

Example. The Lister wants to fund $R_{m,r}$ with 100 currency coins/tokens. Preflight `hasBalance` returns 140 tokens at k-deep, so it proceeds, Lister sends `FundingTxto=pool(m), amount=100`. Validators verify and include it, after k confirmations, the Lister re-reads balances: the pool rose by 100; the Lister's balance dropped by $100+$fee. The post-state check passes. If preflight had shown only 90 tokens, the call would have halted and no tx would be sent.

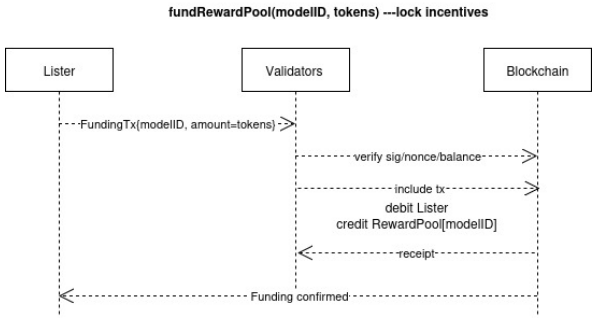


Figure 4.4: fundRewardPool(*modelID*, *tokens*) — locking incentives.

Algorithm 3 Lister.fundRewardPool(*modelID*, *tokens*)

function fundRewardPool($modelID, tokens$)
  require hasBalance(self, tokens)
  $before \leftarrow$ READ_STATE($/state/pool/modelID$)
  $rcpt \leftarrow$ SEND_AND_WAIT($\{FundingTx, modelID, tokens\}, k$)
  $after \leftarrow$ READ_STATE($/state/pool/modelID$)
  assert $(after \geq before + tokens)$
  return $rcpt$

4.2.4 collectTickets(*modelID*, *round*)

Before aggregation, the Lister must discover all revealed update tickets. Figure 4.5 shows how the Lister queries the validator read API, which scans the current blockchain state for `ticketID`, `CID`, and metrics. Algorithm 4 formalises this: only tickets with `revealed=true` are returned, and ordering by ticketID ensures reproducibility.

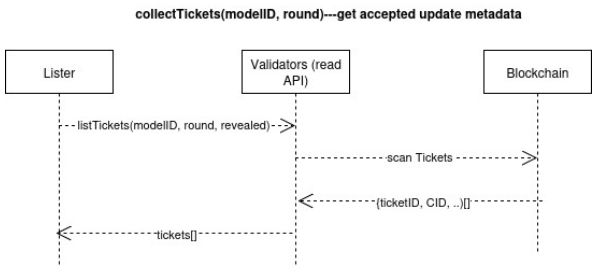


Figure 4.5: collectTickets(*modelID*, *round*) — retrieving revealed update metadata.

Algorithm 4 Lister.collectTickets(*modelID*, *round*)

1: **function** collectTickets($modelID, round$)
2: $tickets \leftarrow$ READ_STATE($/state/tickets, \{modelID, round, revealed : true\}$)
3: **return** sortBy($tickets$, key=`ticketID`)

4.2.5 aggregateUpdates(*tickets*)

Figure 4.6 shows the off-chain aggregation loop. For each instance of a ticket, the Lister fetches update bytes from IPFS, validates their structure, and feeds them into an accumulator (e.g., weighted FedAvg). Algorithm 5 makes this deterministic: retries are bounded, invalid shapes are skipped, while numerical soundness is implemented via fixed dtypes [89].

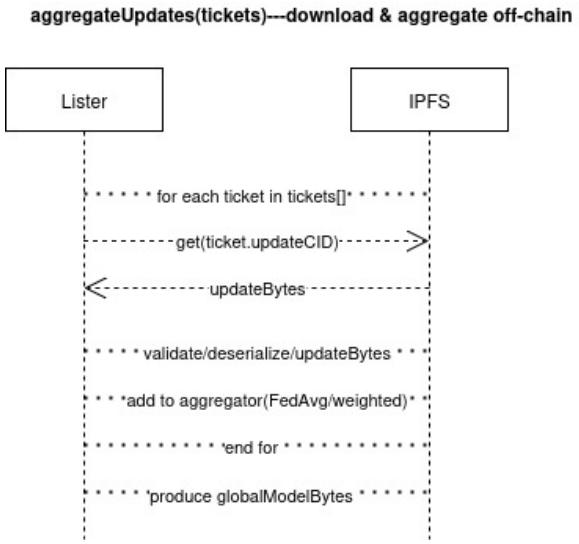


Figure 4.6: aggregateUpdates(*tickets*) — download and aggregate client updates.

Algorithm 5 Lister.aggregateUpdates(*tickets*)

function aggregateUpdates(tickets)
  $acc \leftarrow$ initAccumulator(`fp32`); $w \leftarrow 0$; $included \leftarrow [\,]$; $refunds \leftarrow [\,]$; $scores \leftarrow [\,]$
  for $t \in tickets$ **do**
    $bytes \leftarrow$ retryIPFS$(t.CID, B)$
    if $bytes = \perp$ **then**
      continue ▷ not retrievable ⇒ not refund-eligible
    append(refunds, t.ticketID) ▷ retrievable ⇒ refund-eligible
    $u \leftarrow$ DESERIALIZE_PARAMS$(bytes)$
    if not conformsToListing(u) **then**
      continue
    $s \leftarrow$ WEIGHT$(t.metrics, t.size)$ ▷ e.g. $w_j = \min\{n_j, \bar{n}\}$ from metrics
    accumulate(acc, u, s); $w \leftarrow w + s$
    append(included, t.ticketID)
    $q \leftarrow$ QUALITY_DELTA$(t.metrics)$ ▷ e.g. δ_j from public validation
    append(scores, $\max(0, q) \cdot s$)
  if $w = 0$ **then**
    raise NoEligibleUpdates
  $W^* \leftarrow$ finalize(acc, w)
  return $\langle$SERIALIZE(W^*), $included$, $refunds$, $scores\rangle$

4.2.6 uploadGlobal(*modelBytes*)

After aggregation, Figure 4.7 shows the simple course of actions, Lister uploads the global model bytes off-chain to IPFS, obtains a CID, and pins it locally. The algorithm shows the procedure 6 ensures integrity by recalculating the given hash.

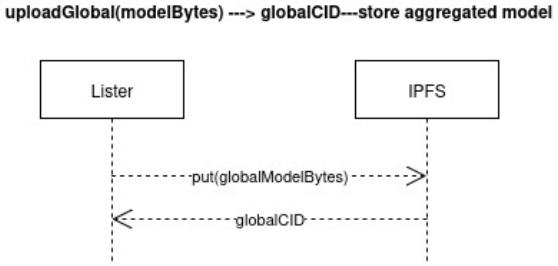


Figure 4.7: uploadGlobal(*modelBytes*) — storing the aggregated model.

Algorithm 6 Lister.uploadGlobal(*modelBytes*)

```
1: function uploadGlobal(modelBytes)
2:     cid ← IPFS_PUT(modelBytes)
3:     assert CID_CHECK(cid, modelBytes)
4:     IPFS_PIN(cid)
5:     return cid
```

4.2.7 finalizeRound(*modelID*, *round*, *globalCID*, *included[]*)

Figure 4.8 shows the finalization. The Lister sends a `GlobalUpdateTx`, the tx contains the name of the model m, the round r, the new `globalCID`, and the list of included `ticketIDs` (with scores/weights). It is signed by the Lister account, then the validators run fast, deterministic checks. They verify the signature, check that the round in the tx has the recent/current *RoundIndex* on-chain, check each ticket is for $\langle m, r \rangle$, is revealed, is unexpired, and is not already utilized, review that the 'list' has no replications and follows the rule (ordering and weights), check the `globalCID` is well formed (CIDv1, valid multicodec/multihash). They recompute entitlements the same way and confirm the reward pool can pay. Given that if any statement fails, reject. Given that everything succeeds, apply the state change *atomically* (all-or-nothing). They record entitlements, mark tickets as included, resolve deposits by procedure/rules, revise the model's `globalCID`, advance $r! \rightarrow! r{+}1$. Algorithm 7 encodes this. If the round has already advanced by the time the tx arrives, the Lister aborts and re-runs for the new round to avoid a race.

Example. The Lister prepares a `GlobalUpdateTx` for $r{=}7$. While it waits for a block, another valid finalize for $r{=}7$ lands first. The chain now shows $r{=}8$. Validators reject the stale tx. The Lister sees the 'round advanced' error, aborts, and aggregates for $r{=}8$ instead. No double finalization occurs. State stays clean.

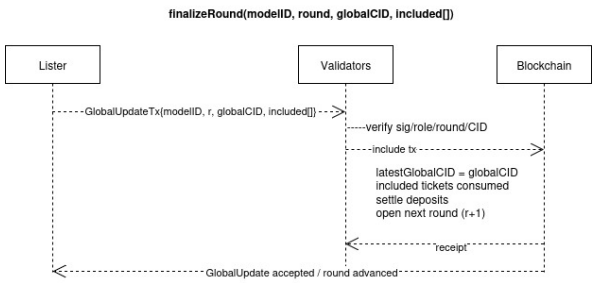


Figure 4.8: finalizeRound(*modelID*, *round*, *globalCID*, *included[]*) — binding and advancing a round.

Algorithm 7 Lister.finalizeRound(*modelID*, *round*, *globalCID*, *included*, *refunds*, *scores*)

function finalizeRound($modelID, round, globalCID, included, refunds, scores$)
  $r^\star \leftarrow$ READ_STATE($/state/round/modelID$)
  if $r^\star \neq round$ **then**
    raise RoundMismatch
  assert $|included| = |scores|$
  $tx \leftarrow \{GlobalUpdateTx, modelID, round, globalCID, included, refunds, scores\}$
  $rcpt \leftarrow$ SEND_AND_WAIT(tx, k)
  if $rcpt.status \neq$ Accepted **then**
    raise TxRejected
  return $rcpt$

4.2.8 withdrawRewards(*acct*, *modelID*, *amount*)

Finally, Figure 4.9 shows how the Lister or beneficiary reclaims funds. A `WithdrawTx` is admitted only if entitlement exists, transferring tokens deterministically. Algorithm 8 enforces entitlement and verifies post-state.

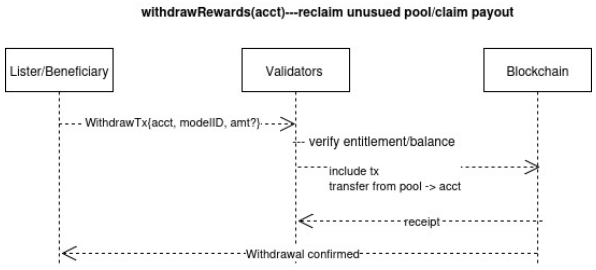


Figure 4.9: reclaiming funds via ClaimTx

Algorithm 8 Lister.withdrawRewards(*acct*, *modelID*, *amount*)

function withdrawRewards($acct, modelID, amount$)
    $elig \leftarrow$ READ_STATE($/state/entitlement, \{acct, modelID\}$)
    if $elig < amount$ **then**
        raise InsufficientEntitlement
    $before \leftarrow$ READ_STATE($/state/balances/acct$)
    $rcpt \leftarrow$ SEND_AND_WAIT($\{ClaimTx, acct, modelID, amount\}, k$)
    $after \leftarrow$ READ_STATE($/state/balances/acct$)
    assert $(after \geq before + amount)$
    return $rcpt$

Determinism and auditability. Every operation references canonical chain state, and Algorithms 4–5 guarantee reproducible aggregation. Anyone can re-run the pipeline by fetching the same tickets and IPFS CIDs.

Failure semantics. Figures 4.6 and 4.8 highlight error points: IPFS unavailability is bounded by retries, while race conditions are caught by re-checking the round index.

Security posture. As seen in Figures 4.3 and 4.4, the Lister relies only on validator-confirmed state. Strict validation prevents malformed models, and deposit locks ensure 'skin-in-the-game' for clients. Combined, this makes the Lister a transparent and accountable round aggregator.

4.3 Client

In the suggested setup, the *Client* node is the distributed worker that turns private data into verifiable, on-chain contributions. A Client talks to three subsystems: the validator gateway (read API for canonical state and write API for transactions), the IPFS storage network (to provide storage for extensive model artifacts), a local training setup (for stochastic optimization on personal data). Below subsections follow the life cycle of a contribution and, for each given stage, a corresponding UML sequence diagram and then followed by a concrete, implementation-oriented algorithm. The algorithms are written to be deterministic, auditable, and compatible with the validator and lister procedures defined elsewhere in this chapter.

4.3.1 queryRoundState(*modelID*)

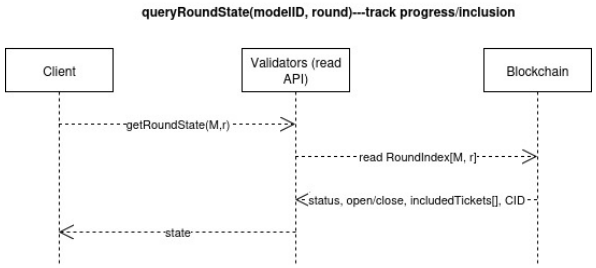


Figure 4.10: Querying the canonical round state before participating.

Figure 4.10 shows the first mandatory interaction. Before a Client allocates compute, it must anchor itself to the canonical chain by asking the validator through read-API currents round metadata of a model. The reply has among many things, round number, deadlines, deposit requirements, and rate boundaries. Given that any of these changes while the client is preparing an update, the Client aborts and re-queries to avoid producing work for the wrong round. Algorithm`queryRoundState` 9 enforces this guard and returns a compact view of the round state used throughout the remaining procedures.

Algorithm 9 Client.queryRoundState(*modelID*)

1: **function** queryRoundState($modelID$)
2: $req \leftarrow$ {path: "/state/round/"$\|modelID$}
3: $round \leftarrow$ READ_STATE(req)
4: **if** $round = \perp$ **then**
5: **raise** ModelNotFound
6: $rules \leftarrow$ READ_STATE({path: "/state/rules/"$\|modelID$})
7: **return** $\langle round, rules \rangle$

Round snapshot and re-check. The pair $\langle round, rules \rangle$ from Algorithm 9 shows that this is the source of validity for this contribution. Treat it like a snapshot that has procedures such as, guides tickets, reveals, and claims, but know the chain can move, blocks may even advance $r! \rightarrow !r{+}1$. A brief reorg can also turn the tide. So any client must re-check right before any tx that depends on r or on `rules`. If r has been altered, update the plan. If `rules` changed, adjust deposit and deadlines/expiry variable. Guaranteeing and prevents stale submits.

Example. You query and see $r{=}7$. You prepare an `UpdatePublishTx` for $r{=}7$. Before send, query again. If the chain now shows $r{=}8$, do not send the old tx. Reserve a fresh ticket

for $r{=}8$ and reveal there.

4.3.2 fetchGlobal(*modelID*) and readGlobal(*cid*)

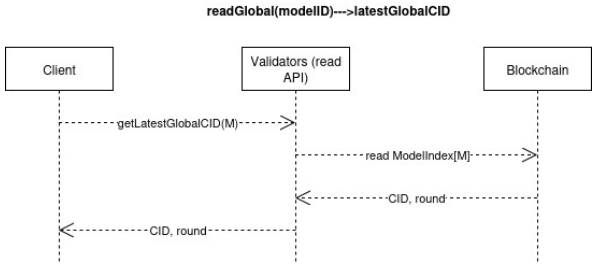


Figure 4.11: Two-step model retrieval: resolve latest CID from validators, then fetch bytes from IPFS.

Training begins by downloading the latest global checkpoint. As illustrated in Figure 4.11, this is intentionally divided into two phases. First any client must first resolve the latest `CID` for the model from the validator read-API; that step is captured in Algorithm 10. Then it recovers the corresponding model off-chain from IPFS and verifies that the bytes re-hash to the same `CID`; that step is captured in Algorithm 11. This distinguishment makes the process auditable: the chain only ever commits to a content identifier, even though the bulk payload remains off-chain.

Algorithm 10 Client.fetchGlobal(*modelID*)

function fetchGlobal($modelID$)
    $cid \leftarrow$ READ_STATE({path: "/state/latestGlobal"$\|modelID$})
    if $cid = \perp$ **then**
        raise NoGlobalModel
    return cid

Algorithm 11 Client.readGlobal(*cid*)

function readGlobal(cid)
    $bytes \leftarrow$ IPFS_GET(cid)
    if $bytes = \perp$ **then**
        raise CIDUnavailable
    assert CID_CHECK($cid, bytes$) ▷ re-hash to the same CID
    $model \leftarrow$ DESERIALIZE($bytes$) ▷ strict schema and dtype checks
    return $model$

If either step falls or produces stale data, the Client does not proceed to training the model. Thus, with assurance, this avoids wasted compute and controls unexpected publication against an outdated checkpoint.

4.3.3 trainLocal(*dataset*)

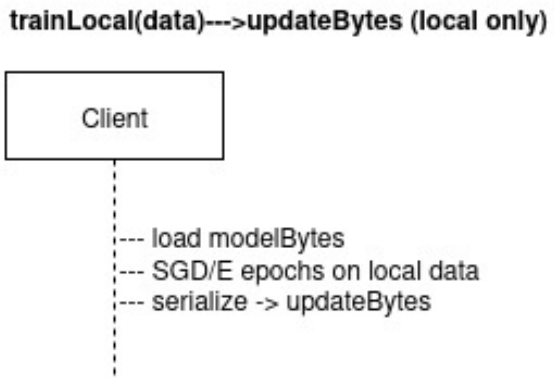


Figure 4.12: Local training on secret data; guaranteeing that the newly produced global_model is never exposed externally.

Figure 4.12 depicts the purely local phase in which the Client initialises the model to the global checkpoint and executes E epochs of training on private/secret data. With this in mind, this procedure must be reproducible across platforms: we enforce fixed random seeds, explicit dtypes and deterministic serialisation. The routine also computes validation metrics that will later accompany the update and may be used by the lister's aggregation policy (see Alg. 5).

Algorithm 12 Client.trainLocal(*modelID*, *dataset*)

function trainLocal($modelID, dataset$)
  $cid \leftarrow$ fetchGlobal($modelID$)
  $model \leftarrow$ readGlobal(cid)
  for $epoch \leftarrow 1$ **to** E **do**
    for all $batch \in$ split($dataset$) **do**
      $model \leftarrow$ UPDATE($model, batch$) ▷ SGD/Adam; fixed lr schedule
  $metrics \leftarrow$ EVAL($model,\ dataset.validation$)
  return $\langle model,\ metrics \rangle$

The procedure result of Algorithm 12 shows the training inside each client, obtaining new

weights(i.e., the slope and the biases) of the model, and validation metrics summarizing their effect. The Client does not publish either until it has secured an on-chain reservation.

4.3.4 prepareIntent and sendUpdateIntent

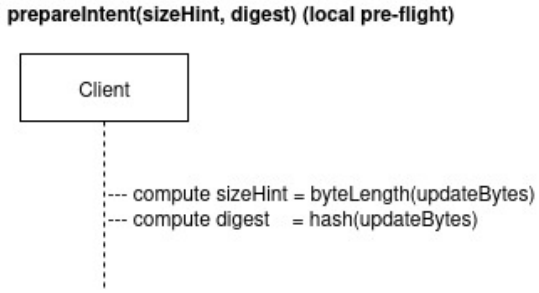


Figure 4.13: Preparing an intent and deposit to reserve a publication slot at present round.

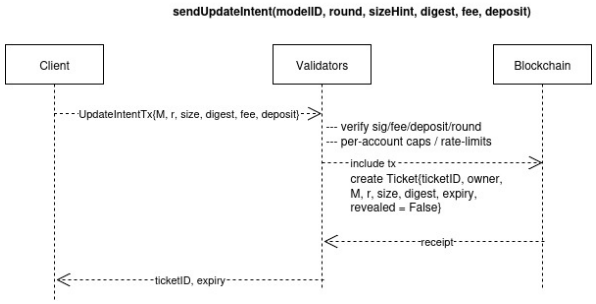


Figure 4.14: Broadcasting the intent; validators create a ticket if limits and deposits are satisfied.

Figures 4.13 and 4.14 capture the anti-spam mechanism. A Client must first reserve a slot for the round by submitting an intent transaction that locks a deposit and is subject to per-account rate limitations. Reservation is represented on-chain as a ticket with an owner, expiry and a `revealed=false` flag. Algorithms 13 and 14 implement both actions. Know that the client does not upload any bytes to IPFS before the receipt confirms inclusion with k confirmations.

Algorithm 13 Client.prepareIntent(*modelID*, *round*)

```
1: function prepareIntent(modelID, round)
2:     ⟨r, rules⟩ ← queryRoundState(modelID)
3:     if r ≠ round or now > rules.deadline then
4:         raise RoundClosed
5:     deposit ← rules.deposit
6:     tx ← {type: UpdateIntentTx, modelID, round, deposit}
7:     return tx
```

Algorithm 14 Client.sendUpdateIntent(*tx*)

```
1: function sendUpdateIntent(tx)
2:     h ← SEND_TX(tx)
3:     rcpt ← WAIT_RECEIPT(h, k)
4:     if rcpt.status ≠ Accepted then
5:         raise TxRejected
6:     return rcpt                          ▷ contains ticketID and expiry
```

4.3.5 uploadUpdate(*localModel*)

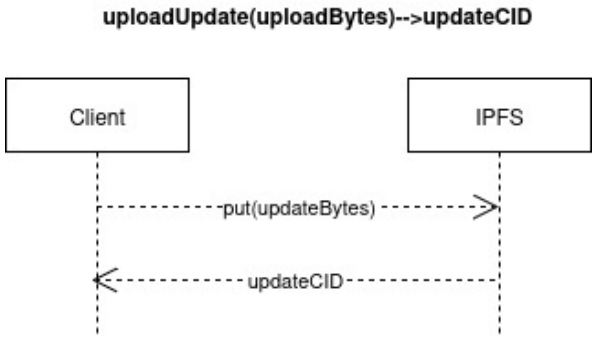


Figure 4.15: Uploading the local update to IPFS and pinning it for availability.

With a confirmed ticket, every client must serialize his trained model in a canonical byte format and upload it off-chain on IPFS as shown in Figure 4.15. The resulting `updateCID` uniquely identifies the payload. The client pins the object locally to improve availability, but the protocol only requires that the `CID` is valid and retrievable.

Algorithm 15 Client.uploadUpdate(*localModel*)

```
1: function uploadUpdate(localModel)
2:     bytes ← SERIALIZE(localModel)          ▷ deterministic order and dtype
3:     cid ← IPFS_PUT(bytes)
4:     assert CID_CHECK(cid, bytes)           ▷ recompute multihash locally
5:     IPFS_PIN(cid)
6:     return cid
```

4.3.6 publishUpdate(*modelID*, *round*, *cid*, *metrics*)

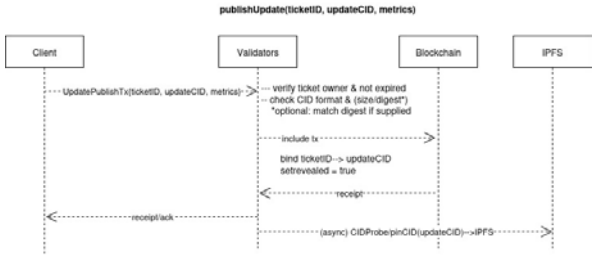


Figure 4.16: Publishing the update: bind `ticketID` to `updateCID` and expose metrics.

Client reveal. Figure 4.16 shows the reveal step. The client calls the write API and then transmits an `UpdatePublishTx`. The tx binds its `ticketID` to the `updateCID`. It also includes the account, a signature, and simple validation metrics. Validators run fast checks only. They verify the signature, then check ticket ownership, next check the ticket is live (confirmed, unused, unexpired), clients then check the model and round match, then check the `CID` format (CIDv1, base32, valid multicodec and multihash length), although they do not fetch bytes. Finally, If everything matches out, they mark the ticket `revealed=true` and admit the tx to the mempool. It will be included in a block. Algorithm 16 shows the client path. Send the tx. Wait for k confirmations. Only then treat the update as visible to the Lister's collector (Algorithm 4). If finality is not reached, retry the same tx idempotently.

Example. Alice has a live ticket for $\langle m{=}sentiment, r{=}7\rangle$. She uploads her update and gets an `updateCID`. She sends `UpdatePublishTxticketID, updateCID, metrics, account, sig`. The validator sees the ticket belongs to Alice, confirms that it is live, and matches the same model and round. The `CID` parses as CIDv1. The tx enters the mempool and lands in a block. After k confirmations, Alice understands that the Lister will see her update now.

Algorithm 16 Client.publishUpdate(*modelID*, *round*, *ticketID*, *cid*, *metrics*)

function publishUpdate($modelID, round, ticketID, cid, metrics$)
  $tx \leftarrow$ {type: UpdatePublishTx, $modelID$, $round$, $ticketID$, cid, $metrics$}
  $h \leftarrow$ SEND_TX(tx)
  $rcpt \leftarrow$ WAIT_RECEIPT(h, k)
  if $rcpt.status \neq$ Accepted **then**
    raise TxRejected
  return $rcpt$

4.3.7 claimDeposit(*acct*, *modelID*)

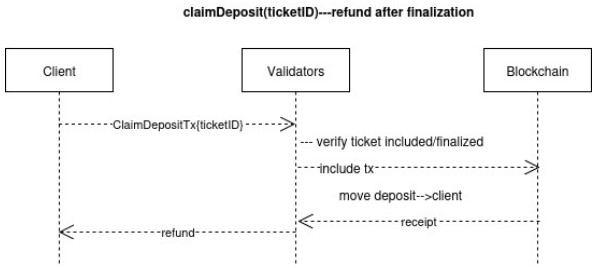


Figure 4.17: Claiming deposits or rewards after the round is finalised.

After the lister finalises the round and the corresponding `GlobalUpdateTx` is accepted, deposits are released and rewards are credited according to policy. As depicted in Figure 4.17, the Client first queries its entitlement from the read API; if non-zero, it submits a claim transaction and waits for a receipt. Algorithm 17 describes the complete flow.

Algorithm 17 Client.claimDeposit(*acct*, *modelID*)

function claimDeposit($acct, modelID$)
  $elig \leftarrow$ READ_STATE({path: "/state/entitlement", query : {$acct, modelID$}})
  if $elig = 0$ **then**
    raise NoEntitlement
  $tx \leftarrow$ {type: ClaimTx, $acct$, $modelID$, amount : $elig$}
  $h \leftarrow$ SEND_TX(tx)
  $rcpt \leftarrow$ WAIT_RECEIPT(h, k)
  if $rcpt.status \neq$ Accepted **then**
    raise TxRejected
  return $rcpt$

Determinism and accountability. Every step above is tied to canonical state. A Client trains only against the latest global `CID` (Algorithms 10–11); it reserves a slot and proves skin-in-the-game via an intent and deposit (Algorithms 13–14); it publishes a content-addressed update that anyone can retrieve and verify (Algorithm 15); and it binds that update on-chain with metrics for transparent aggregation (Algorithm 16). The outcome refunds or rewards is then deterministically claimable (Algorithm 17).

Failure semantics. The Client treats validator state as the single origin of truth and performs bounded tries that have logarithmic wait-times (i.e., exponential back-off) for IPFS reads and writes. If an `updateCID` cannot be resolved by validators within the lister's aggregation window, the ticket may be skipped without deposit refund. Chain reorgs are handled by waiting k confirmations for all write paths and by re-querying round state prior to any transaction that depends on it.

Security posture. Deposits and per-account caps reduce spam and Sybil pressure. Deterministic serialization prevents equivocation across platforms. Metrics are validated for shape and range before inclusion, and no raw data ever leaves the Client. The Client never trusts off-chain announcements and only advances through states that validators expose as canonical, which aligns its behaviour with the lister and validator algorithms defined earlier in this chapter.

4.4 Validator (Blockchain Layer)

When it comes to the blockchain, validators are the ones responsible for checking the rules, basically acting as fair judges that make sure all transactions are in order, and using the PoS + LC consensus protocol, they create and verify blocks, unlike listers and clients, validators do not own the models or train them. They just manage the master ledger that keeps a record of all the listings, updates, and bonuses. This section of the document goes over the whole validation process in a step-by-step manner, drawing on UML diagrams, and gives you concrete pseudocode for each stage, and explains how consensus guarantees are applied.

4.4.1 VRF-Based Leader Election and Block Proposal

We turn to a verifiable random function (VRF) that is driven by epoch-based randomness, denoted R_e, when electing a leader. Each of our validators, i, with its bonded stake s_i generates a unique, per-slot output $\rho_{i,t}$ and a corresponding proof $\pi_{i,t}$ from a domain-seperated message that zeroes in on the epoch e, the slot t and the chain tip (i.e $(\text{epoch } e, \text{slot } t, \text{chain tip})$). Eligibility formally is defined as a *stake-proportional threshold* test as follows,

$$\rho_{i,t}\ tau \cdot \frac{s, i}{S},$$

where $S = \sum_{j \in \mathcal{V}} s_j$ and $\tau \in (0, 1]$ is the target expeced leaders per slot A validator is considered eligible to be a leader if their VRF output $\rho_{i,t}$ falls below a stake-proportional threshold, τ times the ratio of its stake, s_i to the sum of the stakes of all the validators, S. Should multiple blocks show up that meet this eligibility criteria, we go by the block with the lowest normalized VRF score, to be the canonical block for that particular slot.

The proposed blocks will have a specific structure that allows for easy verification and fork-choice: it contains the parent block's hash, slot number, epoch number, VRF output, VRF proof, the number of the proposing validator, a state root and a signature, which makes it possible for peers to check a block's ownership and fork-choice with utmost certainty. Formally, a proposed block header carries $\langle \texttt{parentHash},\ \texttt{slot} = t,\ \texttt{epoch} = e,\ \texttt{vrfOutput} = \rho,\ \texttt{vrfProof} = \pi,\ \texttt{proposer} = i,\ \texttt{stateRoot},\ \texttt{sig} \rangle$

The epoch-based randomness is updated at the end of each epochR_{e+1}. And what we do is hash the last epoch's accepted blocks into the previous random seed. The randomness produced R this way will effectively remain unbiased under the condition that the validators hold a majority.

Algorithm 18 proposerEligible$(i, t, \mathsf{R}_e, s_i, S, \tau)$

function proposerEligible$(i, t, \mathsf{R}_e, s_i, S, \tau)$
  $m \leftarrow \mathsf{domainSep} \parallel \mathsf{epoch}(e) \parallel \mathsf{slot}(t) \parallel \mathsf{parentHash}$
  $(y, \pi) \leftarrow \mathsf{VRF.Prove}(sk_i, m)$ ▷ y is VRF output bytes; π is proof
  $\rho \leftarrow \mathsf{Normalize}(H(y))$ ▷ map $H(y)$ to $[0, 1)$ by 2^{-256} scaling
  if $\rho < \tau \cdot \frac{s_i}{S}$ **then**
    return $\langle \text{true}, \rho, \pi \rangle$
  else
    return $\langle \text{false}, \rho, \pi \rangle$

The algorithms proposerEligible 18 makes the leadership test, while the other functions and procedures have been moved to Appendix A for clarity such as, `updateEpochRandomness` (see Alg. 34) for epoch randomness update, Function `verifyIncomingBlock`(see Alg. 32), procedure `onSlotTick`(see Alg. 31), Function `forkChoice`(see Alg. 33)for proposal,validation,and tie-break. With these we have added the most important VRF leader election algorithms with concise descriptions, and we are ready to move on to how the validators interact with other layers.

4.4.2 Admitting Listings

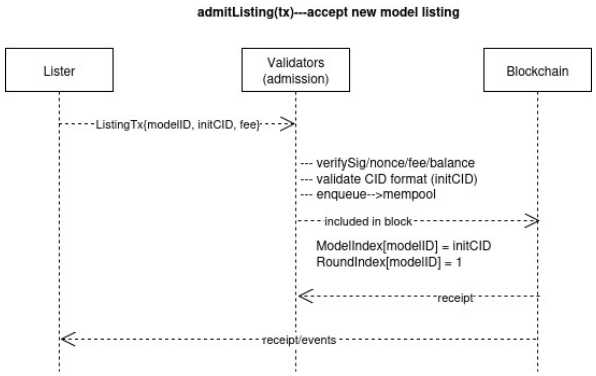


Figure 4.18: Admission of a new model listing transaction.

Figure 4.18 shows how a validator admits a `ListingTx`. The tx comes from the Lister's account. It names a `modelID` and an `initCID`. It also carries a signature and fee fields. The validator runs fast checks only. It verifies the signature (right key; right bytes). It checks the nonce and fee budget. It confirms the account balance covers the fee at the k-deep view. It validates the `CID` syntax (CIDv1, base32, `raw` codec, SHA-256 multihash length). It may run simple schema checks (required fields present; sizes within limits). It does not fetch IPFS bytes. If everything succeeds, the tx is put inside the mempool for later inclusion. Given that if any statement fails, it is rejected with a clear reason. Final uniqueness and state updates happen at block execution, not at mempool admission.

Example. Alice sends `ListingTxmodelID=sentiment, initCID=bafy…, rules, account=Alice, sig`. The validator verifies the signature, sees the nonce is next, and that Alice's balance covers the fee. The `initCID` parses as CIDv1 and fits size limits. The tx is accepted into the mempool. Later, when a block includes it, the chain initializes `ModelIndex[sentiment]` and starts round 0.

Algorithm 19 Validator.admitListing(tx)

```
1: function admitListing(tx)
2:     if not VERIFY_SIG(tx.lister, tx.sig) then
3:         reject                                   ▷ invalid signature
4:     if not CHECK_BALANCE(tx.lister, tx.fee) then
5:         reject                                   ▷ insufficient funds
6:     if not VALIDATE_CID(tx.initCID) then
7:         reject                                   ▷ malformed or unverifiable CID
8:     ENQUEUE_TX(mempool, tx)
```

The algorithm 19 formalizes the admission checks and the helper function `admitFunding(tx)` can be found in Appendix(see Alg. 35). At this stage the transaction is not yet canonical; only once included in a confirmed block does the listing become a piece inside the ledger. Validators thereby filter out malformed or malicious attempts before consensus.

4.4.3 Admitting Update Intents

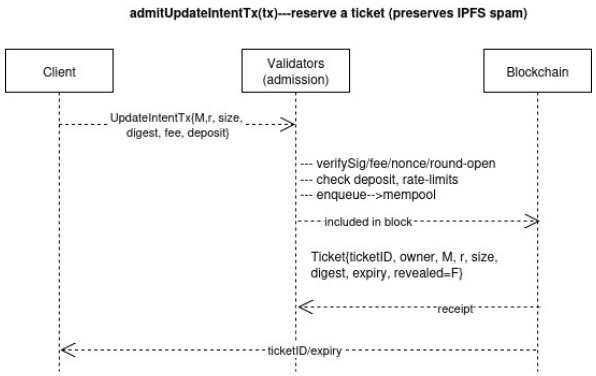


Figure 4.19: Admission of an update intent transaction from a client.

Figure 4.19 shows how a client reserves a slot, each client must send an `UpdateIntentTx`, to ask for a ticket for $\langle m, r \rangle$, tx carries the account, a signature, a deposit, and a nonce. The validator runs fast checks, verifies the signature, checks the fee and the nonce, checks that round (denoted as r) is open for model (denoted as m), checks the account balance can lock the deposit and pay the fee, enforces per-account rate limitations and size caps, denies duplicates and obvious spam. No IPFS fetch happens on this path, if everything fits/pass, the tx enters the mempool. When included in a block, the deposit is locked and a `ticketID` is created, ticket is now live (confirmed, unused, unexpired). This puts real money at stake. It

makes flooding costly and slows Sybil attacks. A client can reveal later by binding this ticket to an `updateCID`.

Example. Alice wants a ticket for $\langle sentiment, 7 \rangle$. She sends `UpdateIntentTxdeposit=10, account=Alice, sig`. The validator sees round 7 is open. Alice's balance covers the deposit and fee. She has not hit her per-round cap. The tx is admitted. It lands in a block. The chain locks 10 tickets and then yields `ticketID=42`. Alice can now upload her update and reveal with `ticketID=42`. If she had no funds or had too many tickets already, the intent would be rejected.

Algorithm 20 Validator.admitUpdateIntent(*tx*)

```
1:  function admitUpdateIntent(tx)
2:      if not VERIFY_SIG(tx.client, tx.sig) then
3:          reject
4:      if not CHECK_ROUND(tx.modelID, tx.round) then
5:          reject                                  ▷ round closed or mismatched
6:      if not CHECK_DEPOSIT(tx.client, tx.deposit) then
7:          reject                                  ▷ insufficient deposit
8:      if not ENFORCE_RATELIMITS(tx.client) then
9:          reject                                  ▷ too many intents
10:     ENQUEUE_TX(mempool, tx)
```

The algorithm 20 enforces both economic and temporal constraints, confirming that only suitable clients can publish updates.

4.4.4 Admitting Update Publications

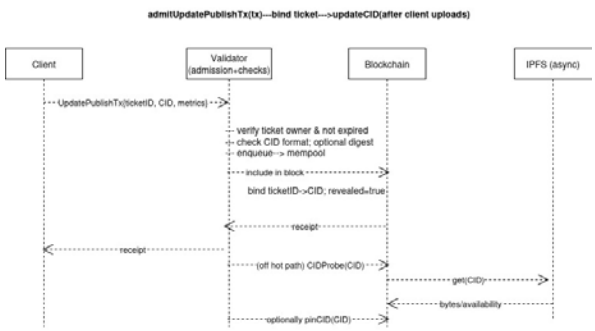


Figure 4.20: Admission of an update publication transaction.

Reveal stage. Figure 4.20 shows the reveal stage. The client holds a live ticket for $\langle m, r \rangle$. It now publishes an `UpdatePublishTx`. The tx binds that `ticketID` to an `updateCID`. It also carries the client account and a signature. The validator runs fast checks only. It verifies the signature (the account signed this tx). Scans that the given ticket belongs to that member. It scans if the given ticket is live (i.e., confirmed, unused, unexpired). It checks the model and round match. It enforces one ticket $\rightarrow$ one CID (no double bind). It checks the `CID` is well formed (CIDv1, base32, valid multicodec and multihash length). The validator does not fetch IPFS bytes. If everything is well put, the tx is enqueued in the mempool. Given that if any statement fails, then the tx is rejected. These checks keep the path simple and safe. They also make later aggregation clean and repeatable.

Example. Alice has `ticketID=42` for $\langle m{=}sentiment, r{=}7 \rangle$. She uploads her update to IPFS and gets `updateCID`. She sends `UpdatePublishTxticketID, updateCID, account=Alice, sig`. The validator sees that `ticketID=42` belongs to Alice, confirm it is still live, and matches $\langle sentiment, 7 \rangle$. The `CID` parses as CIDv1. All good. The tx enters the mempool, later, if the Lister cannot fetch the bytes, that ticket is cut for this round by policy, but the chain does not stall.

Algorithm 21 Validator.admitUpdatePublish(*tx*)

```
1:  function admitUpdatePublish(tx)
2:      if not VERIFY_SIG(tx.client, tx.sig) then
3:          reject
4:      if not CHECK_TICKET(tx.ticketID, tx.client) then
5:          reject
6:      if not WITHIN_EXPIRY(tx.ticketID) then
7:          reject
8:      if not VALIDATE_CID(tx.updateCID) then
9:          reject
10:     ENQUEUE_TX(mempool, tx)
```

Algorithm 22 Validator.admitClaim(*tx*)

```
1: function admitClaim(tx)
2:     if not VERIFY_SIG(tx.acct, tx.sig) then
3:         reject
4:     if tx.amount ≤ 0 then
5:         reject
6:     ent ← LOOKUP_ENTITLEMENT(tx.acct, tx.modelID)
7:     if ent < tx.amount then
8:         reject
9:     ENQUEUE_TX(mempool, tx)
```

Algorithm 21 ensures that no update can enter aggregation without a valid ticket and verifiable CID, protecting listers against unavailable or forged content.

4.4.5 Admitting Global Updates

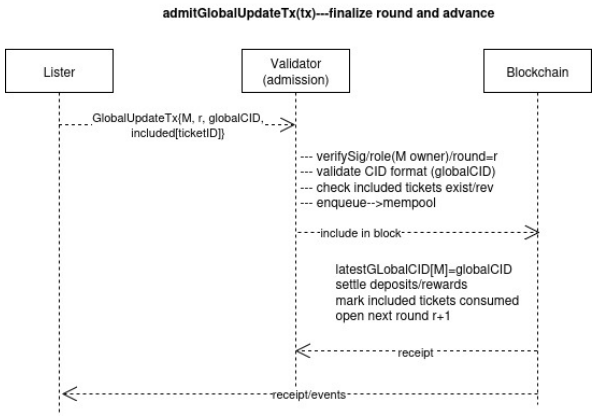


Figure 4.21: Admission of a global update transaction from a lister.

As shown in Figure 4.21, listers submit `GlobalUpdateTx` to finalize a round. Any given validator must correspond to lister signatures, verifies that the round matches the canonical index, and guarantees that all the contained tickets were properly revealed. The `globalCID` is validated and queued. The algorithm`admitGlobalUpdate(tx)` (see appendix Alg. 38) protects against premature or inconsistent finalization. Only once this tx is included in a final chain block are rewards and round increments applied.

4.4.6 Block Production and Validation

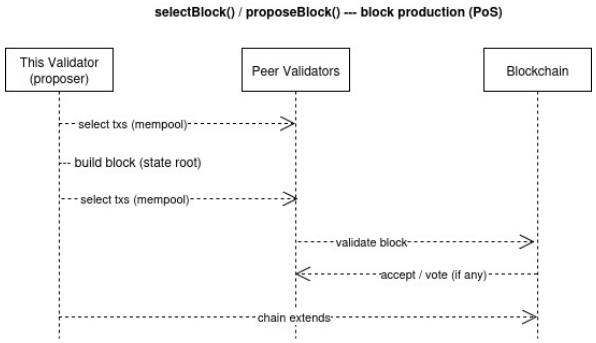


Figure 4.22: Block selection and proposal under Proof-of-Stake.

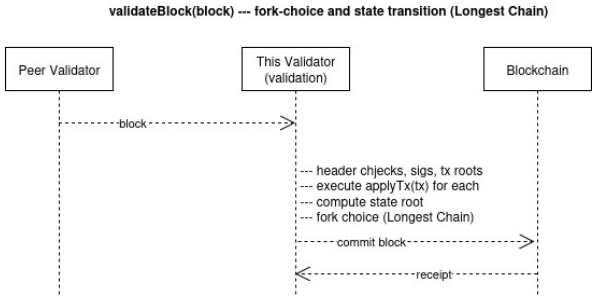


Figure 4.23: Block validation and fork-choice rule.

Figures 4.22 and 4.23 illustrate the consensus-critical steps. A validator is selected via PoS lottery to propose a block, which contains an ordered set of admitted transactions. Peers validate the proposed block by re-executing all transactions using `applyTx`, recomputing the state seed, and inspecting proposer signatures. If correct, proposed block is appended. Forks are resolved by the longest-chain rule.

Algorithm 23 Validator.applyTx(tx)

```
1:  function applyTx(tx)
2:      if tx.type = ListingTx then
3:          MODEL_INDEX_ADD(tx.modelID, tx.initCID)
4:      else if tx.type = FundingTx then
5:          POOL_ADD(tx.modelID, tx.amount)
6:      else if tx.type = UpdateIntentTx then
7:          TICKET_CREATE(tx.modelID, tx.client, tx.round, tx.deposit)
8:      else if tx.type = UpdatePublishTx then
9:          TICKET_REVEAL(tx.ticketID, tx.updateCID)
10:     else if tx.type = GlobalUpdateTx then
11:         ADVANCE_ROUND(tx.modelID, tx.round, tx.globalCID, tx.included, tx.scores, tx.ref
12:     else if tx.type = ClaimTx then
13:         ENTITLEMENT_TRANSFER(tx.acct, tx.modelID, tx.amount)
14:     else
15:         raise UnknownTxType
```

Algorithm 23 represents the consensus-critical execution path. All validators must achieve the same reasoning deterministically to ensure that state roots match, any difference would cause block rejection. In Appendix you can find algorithms of `advanceRound` 36and helper function `entitlementTransfer` 37.

4.4.7 Gossip and CID Probing

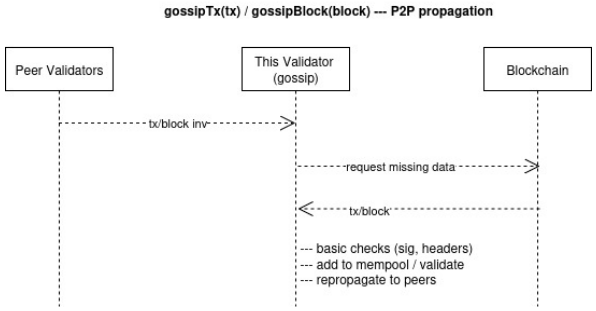


Figure 4.24: Transaction and block gossip across the peer-to-peer network.

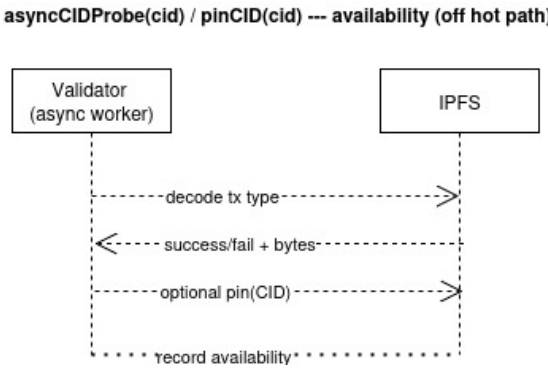


Figure 4.25: Asynchronous CID probing and optional validator pinning.

Supporting services. Figures 4.24 and 4.25 show two helper services. One is *gossip*. One is the *CID probe*. Gossip procedure spreads transactions and proposed blocks fast between peers (peer-to-peer relay). This keeps mempools in sync and cuts time to inclusion. The CID probe runs on a timer and is asynchronous (off the consensus path), it tries to fetch each referenced `CID` to see reachability. Given a failure outcome it tests again, and waits even longer (i.e., exponential backoff, waits longer after each miss). It records success, failure, and latency in logs and dashboards. Thus, these results can trigger optional selective pinning. Probe outcomes never affect admission or consensus, blocks are made even if a fetch fails. The goal is visibility and availability, not gating consensus.

Example. Alice posts an `UpdatePublishTx`. Gossip relays it to validators in under a second. The probe tries the `CID`, fails once, waits, and tries again. The chain still includes the tx and keeps producing blocks. Ops can pin the `CID` if it stays slow or missing.

4.4.8 Read APIs

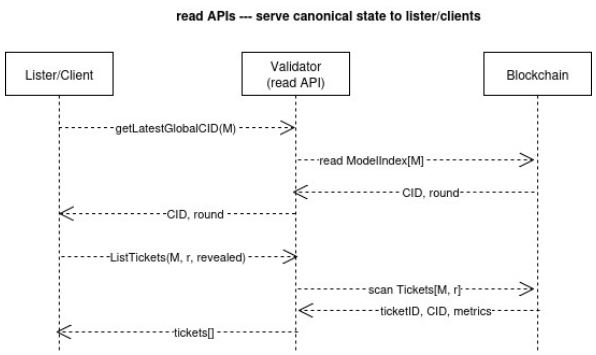


Figure 4.26: Read APIs serving canonical state to listers and clients.

As shown in Figure 4.26, validators expose read-only APIs to clients and listers. These include queries for balances, latest global CIDs, ticket status, and entitlements. APIs are versioned to avoid ambiguity during chain reorgs.

Consensus anchoring. Transaction admission (Algorithms 19–38) occurs before consensus and only becomes canonical once the enclosing block is finalized under PoS. Block production and validation (Figures 4.22–4.23, Algorithm 23) are the consensus-critical stages where validators agree on the chain state. Forks are resolved by the longest-chain rule, guaranteeing convergence.

Security posture. In relation to the security of a system, validators serve as its firewall. They are responsible for the verification of signatures, balance checks, deposits, CID validation, and meeting the deadline for rounds. By isolating admission, execution, gossip, and read APIs, the validator is able to keep the core of the system as streamlined and auditable as possible. The combination of PoS finality and CID availability checks delivers a strong definition of robustness and soundness to economic and network-level attacks.

4.5 IPFS

IPFS role and access. The IPFS holds large model files off-chain. This keeps the ledger small and fast. All parties read and write through an authenticated gateway that sits in front of IPFS. The gateway enforces light *pre-authorization* and per-account rate limits, while integrity comes from content addressing (CIDv1 with a SHA-256 multihash, basically cidv1 is self-describing label for some bytes it tells you how to analyze stored bytes and proves what the bytes are) Validators never decide consensus on off-chain bytes(i.e., storage, ledger stores only hash pointers to the updates). They only admit transactions that reference `CIDs`. Availability is checked off the critical path by probes that try to fetch bytes. This section explains the eight IPFS interactions we rely on (Figures 4.27–4.34) and the robust routines used by the Lister, Clients and Validators.

The eight interactions in short.

- **Put init model** (`initCID`): add and pin initial model bytes; return `CID`.

- **Put client update** (`updateCID`): add and pin a client's update; return `CID`.
- **Put global model** (`globalCID`): add and pin aggregated model; return `CID`.
- **Get update by `CID`**: yield an update's bytes for scoring or audit.
- **Get global by `CID`**: yield the present (or past) global model for training.
- **`CID` probe**: retrying availability check used by off-path monitors(no yields on the consensus path).
- **Read current global and then yield**: read `globalCID` from chain state, then fetch from IPFS.
- **Rate/size enforcement**: gateway rejects oversized or over-quota uploads(i.e, size caps rule and per-client rate limit).

Common helpers. We use the following primitives consistently: `SHA256(bytes)` to compute a digest, they turn byte arrays into hash pointers; `MakeCID(digest)` to build a CIDv1 with `raw` codec, wraps the hash pointer into a CIDv1 with the raw codex, that basically gives you the content ID you can share(self-describing, bas32); `CID_CHECK(cid, bytes)` to recompute and match, then verifies theintegrity, it recalculates hash from the given bytes and inspects if it is equal/match to the cid; `GatewayPut(bytes, meta)`(i.e., upload to the write-side IPFS gateway; runs, limits, then adds+pins, and returns the cid, admission before storage) and `GatewayGet(cid, timeout)` to talk to the local IPFS gateway, where (i.e., gatewayget it fetches bytes by CID from the gateway, and stops if it takes too long) ; `Pin(cid)` to persist, tell the node to keep the data, and pinned blocks do not get garbage-collected(stay fetchable and stay seedable); `TokenBucket.tryConsume(key, n)` to enforce rate/size budgets, if the bucket for key has tokens, consume n and allow, else deny or delay, `PreAuthAllowed(op, ctx)` to ask the validator read API whether the uploader is entitled (e.g., holds an active `UpdateIntent` ticket or is the owner of a model). To keep availability high, `MultiGet` concurrently accesses multiple gateways with hedged requests and then returns the first verified stream.

4.5.1 Lister uploads initial model: `put(initModelBytes)` → `initCID`

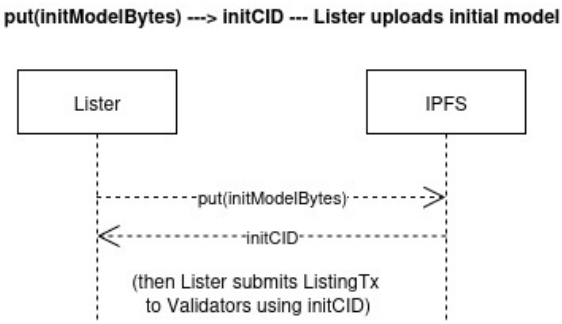


Figure 4.27: The Lister uploads the initial checkpoint, receives `initCID`, then references it in `ListingTx`.

Figure 4.27 shows a two-step pattern. First, the Lister computes the CID locally and uploads the bytes to IPFS. Second, it utilizes the yielded `initCID` inside `ListingTx` (handled in Section 4.2). The gateway applies per-account budgets and a basic pre authorization: only addresses that can subsequently submit a valid `ListingTx` for `modelID` are allowed to upload the initial model.

Algorithm 24 IPFS.PutInitModel(modelID, initBytes, uploader)

1: **function** PutInitModel($modelID, initBytes, uploader$)
2: $digest \leftarrow$ SHA256($initBytes$); $cid \leftarrow$ MakeCID($digest$)
3: **require** PreAuthAllowed(`InitModel`, $\{modelID, uploader\}$)
4: **require** TokenBucket.tryConsume($\langle uploader, \texttt{init}\rangle, |initBytes|$)
5: $ack \leftarrow$ GatewayPut($initBytes$, $\{\texttt{cidHint} : cid, \texttt{role} : \texttt{Lister}\}$)
6: **assert** CID_CHECK($ack.cid$, $initBytes$) ▷ defends gateway bugs
7: Pin($ack.cid$)
8: **return** $ack.cid$

4.5.2 Client uploads update: `put(updateBytes)` → `updateCID`

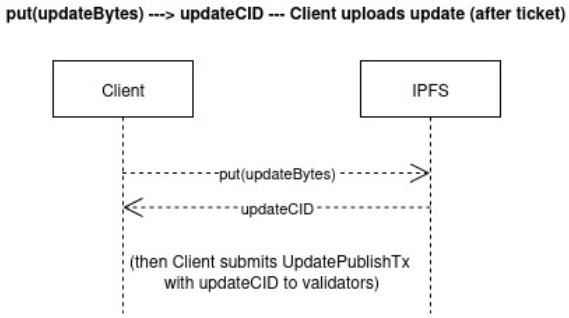


Figure 4.28: A Client uploads its locally trained update after reserving a ticket; the returned `updateCID` is later bound on-chain.

Gateway admission. As shown in Figure 4.28, the gateway(i.e., a write-side IPFS, that does admission checks, basically a controlled point) accepts an update only if the uploader shows a live reservation for $\langle m, r \rangle$. The proof is a confirmed `UpdateIntentTx` that yields a `ticketID` for that pair. The uploader(i.e., client or whoever else) presents `ticketID`, account, signature. The gateway checks that the ticket is live (such as, confirmed, unused, unexpired), matches the same model and round, and meets the size cap, given that if any statement fails, then the upload is rejected. This light *pre-authorization* cuts IPFS spam: Each upload now carries a locked deposit and faces a rate cap, so flooding is costly. Also it aligns itself with the flow on-chain: Only reserved tickets can later link an `updateCID`, and deposits are resolved under the refund policy.

Algorithm 25 IPFS.PutClientUpdate(modelID, round, ticketID, updateBytes, uploader)

function PutClientUpdate($modelID, round, ticketID, updateBytes, uploader$)
  $digest \leftarrow$ SHA256($updateBytes$); $cid \leftarrow$ MakeCID($digest$)
  require PreAuthAllowed(`UpdateUpload`, $\{modelID, round, ticketID, uploader\}$)
  require TokenBucket.tryConsume($\langle uploader,$ `update`$\rangle$, $|updateBytes|$)
  $ack \leftarrow$ GatewayPut($updateBytes$, {`cidHint` : cid, `ticketID` : $ticketID$})
  assert CID_CHECK($ack.cid$, $updateBytes$)
  Pin($ack.cid$) ▷ short-term *pin*; long-term persistence is optional
  return $ack.cid$

4.5.3 Lister uploads aggregated model: `put(globalModelBytes)` → `globalCID`

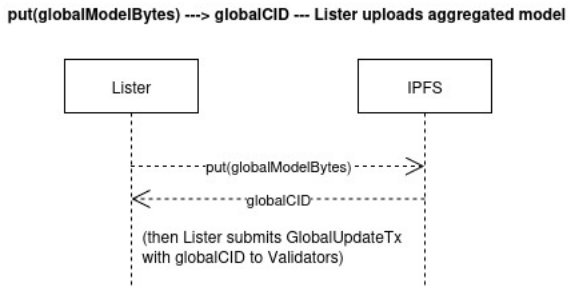


Figure 4.29: After aggregation off-chain, the Lister uploads the round's global model and obtains `globalCID`.

Figure 4.29 reflects the initial upload but this is tied to a specific round. The Lister must be the task owner (or a commissioned aggregator) for the corresponding `modelID`.

Algorithm 26 IPFS.PutGlobalModel(modelID, round, globalBytes, uploader)

function PutGlobalModel($modelID, round, globalBytes, uploader$)
  $digest \leftarrow$ SHA256($globalBytes$); $cid \leftarrow$ MakeCID($digest$)
  require PreAuthAllowed(`GlobalUpload`, $\{modelID, round, uploader\}$)
  require TokenBucket.tryConsume($\langle uploader, \texttt{global} \rangle$, $|globalBytes|$)
  $ack \leftarrow$ GatewayPut($globalBytes$, $\{\texttt{cidHint} : cid\}$)
  assert CID_CHECK($ack.cid$, $globalBytes$)
  Pin($ack.cid$)
  return $ack.cid$

4.5.4 Clients fetch global models: `get(globalCID)` → model bytes

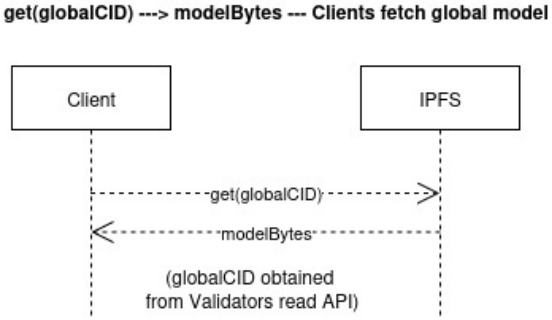


Figure 4.30: Clients resolve the latest `globalCID` from validators (i.e., read API) and bring bytes from IPFS.

In Figure 4.30, CIDs come from the canonical chain; IPFS only supplies content for already decided references. Our client wrapper streams the file, verifies the multihash on the fly, and returns a byte array buffer only if the computed digest matches the CID.

Algorithm 27 IPFS.MultiGetVerified(cid, gateways, timeout)

function MultiGetVerified($cid, gateways, timeout$)
    launch up to k parallel GatewayGet($cid, timeout/k$) calls with jitter
    as soon as a stream arrives, *tee* it to `SHA256` and a byte sink
    upon stream end, **if** `CID_CHECK` holds **then** cancel slower calls/messages and **return** bytes
    otherwise, mark that gateway faulty and continue waiting for the next fastest
    return `None` if all the shots taken fail within $timeout$

4.5.5 Lister fetches accepted updates: `get(updateCID)` → update bytes

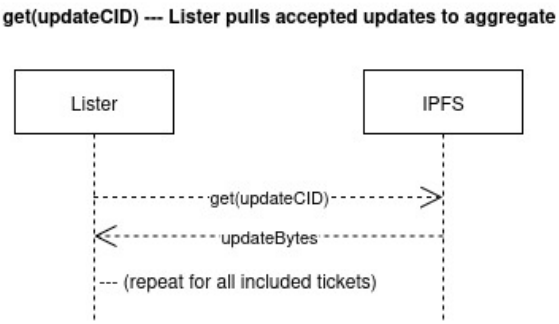


Figure 4.31: The Lister downloads each update referenced by accepted tickets before aggregating.

The same MultiGetVerified routine (Algorithm 27) is used by the Lister to materialize each updateCID listed in the ticket set. Tickets are chain objects; the Lister never fetches arbitrary CIDs.

4.5.6 Optional persistence policy: pin(CID)

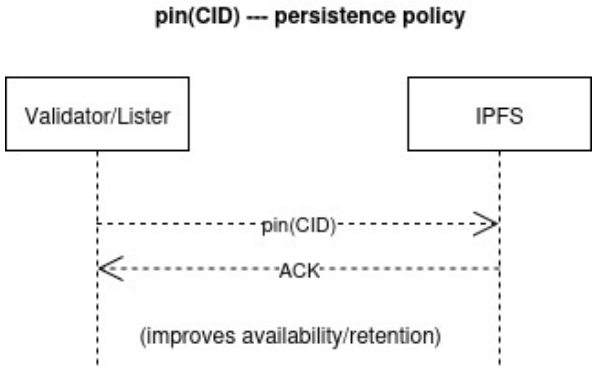


Figure 4.32: Pinning increases retention probability; it is not consensus-critical.

As illustrated in Figure 4.32, both the Lister and (optionally) Validators can pin hot CIDs to improve availability during a round and for audit. Pins are reference-counted and subject to local retention quotas.

Algorithm 28 IPFS.PinManaged(cid, class)

```
1: function PinManaged(cid, class)
2:     if class=hot then ttl ← 14 days else ttl ← 72 hours
3:     if HasCapacity(ttl) then Pin(cid); Remember(cid, ttl, class)
4:     else EvictLRU(until capacity); Pin(cid)
```

4.5.7 Validator availability probing: CIDProbe(CID)

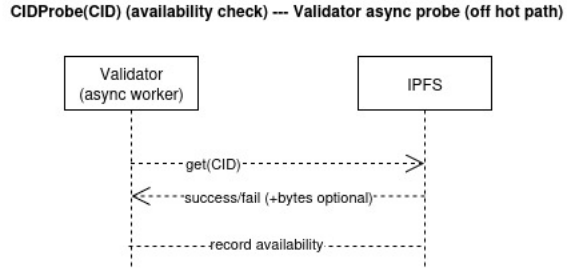


Figure 4.33: A background worker probes newly referenced CIDs and records availability; it runs off the admission hot path.

Figure 4.33 depicts a validator worker as activated whenever mempool receives/admits an `UpdatePublishTx` or a `GlobalUpdateTx`, in other words when a new `UpdatePublishTx` enters the mempool(i.e., the waiting room for transactions), a validator worker tries to fetch the CID to see if the file is obtainable, know that this action does not block the chain, we still accept the transaction and keep making blocks even if fetch falls, then the probe just records stats(like success/fail and speed) for our dashboards and, if needed, trigger selective pinning to keep important files available.

Algorithm 29 Validator.CIDProbeWorker(queue)

```
1: function CIDProbeWorker(queue)
2:     while true do
3:         cid ← queue.popWait
4:         bytes ← MultiGetVerified(cid, G, T)        ▷ G: gateway set, T: SLA timeout
5:         if bytes ≠ None then MarkAvailable(cid); MaybePin(cid)
6:         else MarkUnavailable(cid); ScheduleRetry(cid)
```

4.5.8 Edge controls: rate limit and size caps

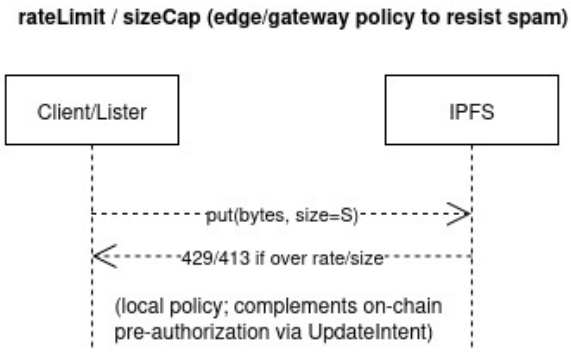


Figure 4.34: Gateway-level throttling complements on-chain pre-authorization to resist storage abuse.

Gateway throttling (Figure 4.34) ensures a single misbehaving node cannot overwhelm the storage cluster. Our policy is a token-bucket per account and operation class (init/update/global) with burst and daily refill; a hard size ceiling prevents pathological uploads.

Algorithm 30 TokenBucket.tryConsume(key, size)

1: **function** tryConsume($key, size$)
2: $(tokens, last) \leftarrow$ Load(key); $now \leftarrow$ Clock
3: $tokens \leftarrow \min(\texttt{cap},\ tokens + (now - last) \cdot \texttt{rate})$
4: **if** $size >$ `maxObjectSize` **then return false**
5: **if** $tokens < size$ **then** Store($key, (tokens, last)$); **return false**
6: Store($key, (tokens - size,\ now)$); **return true**

End-to-end guarantees. Placing these parts together, the storage path provides (i) *integrity*: every fetch re-hashes the payload and matches the CID; (ii) *auditability*: the chain only references CIDs that were admitted by validators, and anyone can reconstruct a round by re-fetching those CIDs; (iii) *abuse resistance*: uploads are gated by on-chain pre-authorization and throttled by the gateway; and (iv) *liveness under failures*: all fetches employ hedged requests and retries without blocking consensus.

Where validators fit. Validators do not judge update bytes (no parsing or scoring). They check only the *syntax* of each `CID` during mempool checks (form, length, multibase/multicodec), and they commit transactions that reference those `CIDs`. The chain does not fetch IPFS data on the consensus path (no network pulls in block validation). Availability is checked *off the critical path* by Algorithm 29 (a separate, retrying fetch test). If an `updateCID` cannot be conveyed when the Lister aggregates, Algorithm 5 (see Section 4.2) *skips* that ticket at current round. This keeps the round moving (progress over completeness) and does not weaken safety, because the choice rule is deterministic and uses only revealed, retrievable updates. The deposit outcome follows the refund policy for that round's window (refund if revealed and retrievable; forfeit if unrevealed or unretrievable).

4.6 Evaluation Metrics and Security Outlook

To demonstrate engineering completeness, we outline how the prototype could be evaluated(tested) and where its main security defenses lie.

Benchmark plan. A systematic evaluation plan(how we measure performance) would focus on three axes:

- **Throughput:** transactions per second and updates per round that validators can admit and finalize(i.e., how much work per unit of time?). For example, our chain admits 40 tx/s and a lister can include 120 updates per round.

- **Latency:** measures how long a round takes. It breaks down into interval, finality depth k, and aggregation delay(i.e., how long a round takes, broken into smaller components). Given a block interval rate is 2 seconds, finality depth $k = 6$, thus, this implies it will take 12 seconds to treat a tx as final, plus aggregation delay(IPFS fetch + scoring) 8 seconds. Total round span would equal 20 seconds!

- **Storage overhead:** bytes committed on-chain (tickets, indices, receipts) versus off-chain (IPFS artifacts). For example, on-chain we store 100 KB per round, Off-chain (IPFS) stores big files(updates, global model)50-200MB per round.

With the proposed evaluation metrics, the prototype will allow the existence of bottlenecks to be identified at the blockchain layer (consensus finality) versus the storage layer (IPFS fetch and pinning). Meaning that if throughput(tx/s) is high, but rounds still take long, the delay is likely off-chain(slow IPFS fetches, big models, or slow scoring). If IPFS is fast but few tx make it into blocks or finality takes long, the limit is consensus/finality.

Outlook. The system as implemented, is auditable and reproducible: every artifact is content-addressed and every transition references a canonical chain state. Thus, this sets a baseline against which experimental results in Chapter 5 can be compared.

Chapter 5

Experiments and Results

This chapter validates the end-to-end design specified in Chapters 3– 4. We exercise the full pipeline—from `ListingTx` and ticketed publications to deterministic aggregation and `GlobalUpdateTx`—and measure the consensus-facing metrics uncovered by our validator read-API: per-block throughput (txs/block), forks observed, updates applied, fees charged, and a conservation check via the *fees–distribution ratio*. We organize results along the same control knobs that the implementation exposes (Ch. 4): number of clients/validators, fee-sink policy, `intent_cap`, IPFS availability, VRF knobs (τ, randomness period), and round-level ML improvement.

5.1 Experimental Setup

Environment. Experiments were conducted on a single workstation and a small LAN cluster of validator/client processes. Table 5.1 lists the host used for the figures and CSV summaries and the github repository of this implementation included below Section 5.7.

Table 5.1: Hardware & software environment.

Component	**Specification**
Operating System	Ubuntu 24.04.3 LTS, Kernel 6.14.0-29-generic
Processor	AMD-Ryzen-7-5800X (8 cores / 16 threads), max boost 5.36 GHz
Memory	32 GiB DDR4 RAM
GPU	NVIDIA-GeForce-GTX-1660-Ti (6 GiB VRAM)
Python	3.12.3

Method. Each scenario runs the complete loop specified in Chapter 4: the Lister publishes a model (`ListingTx`), clients reserve *tickets* via `UpdateIntentTx`, train locally and upload updates to IPFS, reveal via `UpdatePublishTx`, and the Lister finalizes the round with `GlobalUpdateTx` carrying `included[]`, `scores[]`, and `refunds[]`. Validators apply state transitions deterministically (Alg. 23) and expose round statistics through the read API. CSV summaries (one per sweep) capture the per-scenario means returned by that API.

5.2 Baselines and Sanity Checks

Table 5.2 records the baseline means (no stressors). Throughput is 1.571 txs/block. Forks are 14. Applied updates are 26. Fees charged are 25. The fees–distribution ratio is 1.0 (money is conserved). All other tables should be read in comparison to the just stated row metrics, since we will introduce 'ablations'(i.e., the removal of a component in a ML setup).

Table 5.2: Baseline summary (means over runs).

avg_txs_per_block _mean	forks_observed _mean	txs_applied _mean	fees_charged _mean	fees_distribution _ratio
1.571	14.000	26.000	25.000	1.000

Two immediate consistency checks hold across all sweeps: (i) the *fees–distribution ratio* stays at 1.0 (value conservation in Section 3.4), and (ii) the ticket/aggregation discipline of Section 3.4 ensures liveness even when some IPFS fetches fail; unretrievable updates are simply omitted (Alg. 5) and deposits forfeited.

5.3 Results by Control Knob

The Table 5.16 supplies the reader with necessary guidance on how to interpret the results that will be followed throughout this chapter.

5.3.1 Plain FL vs. Our System (BC - FL), Simple Summary

Setup. The setup runs the same task in both modes, meaning FL vs BC-FL assumed the use of 5 clients, for BC-FL 3 validators set assumed equal stake across validators, $\tau = 0.8$ VRF-leader election, each slot is 0.4 s, finality $k = 0$ for the headline result. IPFS is emulated (e.g., 30 ms put/get, jitter 5ms, -10Mb/s), payloads are tiny, so compute/storage are minor vs. the consensus timing. The run lasts 20 s, seed is 1, model is a small logistic regressor with D=32 and a validation set of 512 points.

Result. Plain FL needs **1.6 s** per round (median). BC-FL needs ∼**2.034 s** (assuming $k = 0$) per round (median), i.e., gap is **+27.12%**. If $k = 3$ confirmations, raises median to 4.3 s, i.e., gap is $+ \sim 170\%$ The extra time comes from placing intent, publish, and finalize into blocks.

Side-Note. BC-FL in the lab setting, which consists of a single host, in-process networking and 0.4 second time slots, the results found that it takes roughly one more time slot for BC-FL to reach consensus compared to a plain coordinator. Coming from a baseline that wraps up in about four time slots, this adds a 27.12% increase in median latency. The bulk of this delay is due to the consensus process, not compute power or storage. For deployments in distributed/WAN settings and larger-scale models, it is to be expected to see a higher degree of delay caused by communication and contention, which we will be looking at this more closely in the future.

Table 5.3: Round latency (median) and overhead.

Mode	Latency (s)	Overhead vs. Plain
Plain FL (in-memory)	1.600	0.0%
BC-FL (ours)	2.034	**+27.12%**

Block-production probability and latency model. For clarity, with equal stake assumption across n validators and VRF target $\tau \in (0, 1]$, we get that the per-slot probability that at least one proposer is eligible is

$$p(n, \tau) \;=\; 1 - \left(1 - \tfrac{\tau}{n}\right)^n \;\xrightarrow[n\to\infty]{}\; 1 - e^{-\tau}. \tag{5.1}$$

Given this, the expected slots per block should be

$$\mathbb{E}[\text{slots/block}] \;=\; \frac{1}{p(n, \tau)} \tag{5.2}$$

If a round needs S blocks (Intent/publish/finalize, so our s=3 and thus we require k confirmation blocks for finality, and a simple latency model is

$$L_{\text{BC-FL}} \approx (S+k)\,\frac{1}{p(n,\tau)}\,\Delta, \tag{5.3}$$

where Δ is the slot duration. For the assumed $\tau = 0.8$ the expected probability is$p(n,\tau) \to 1 - e^{-0.8} \approx 0.551$, so improvements from increasing n flatten quickly beyond small n, in the experiments as you can see in Table 5.4 after $n = 10$ for validators the chance gets lower, as expected.

Table 5.4: BC–FL latency vs. Plain (no extra finality, k=0). τ=0.8, Δ=0.4s, Plain= 1.6s.

Validators n	$L_{\text{BC-FL-EXPECTED}}$ (s)	$L_{\text{BC-FL-ACTUAL}}$ (s)	Overhead (%)
1	1.500	1.710	+6.8%
3	1.981	2.034	+27.12%
5	2.063	2.131	+33.18%
10	2.122	2.232	+39.5%
15	2.141	2.299	+43.68%
20	2.151	2.355	+47.18%
25	2.156	2.456	+53.5%
30	2.160	2.495	+55.94%
35	2.163	2.631	+64.43%
40	2.165	2.688	+68%
45	2.166	2.750	+71.86%
50	2.168	2.831	+76.94%

BC - FL breakdown. With 0.4 s slots, the median stage times are:Intent: **0.4 s**, Publish: **0.4 s**, Finalize: **0.8 s**, Scheduling slack (between stages): **0.4 s**.

Takeaway. BC-FL adds a small, clear cost from consensus timing. In return, it gives on-chain finality, public traces of each round, and fair access control. The compute cost is small in this setup.

5.3.2 Scaling the number of clients

Table 5.5: Client scaling (means).

clients	avg_txs_per_block_mean	forks_observed_mean	txs_applied_mean
4	3.118	16.000	53.000
8	3.694	20.000	66.000
12	4.071	24.000	72.000

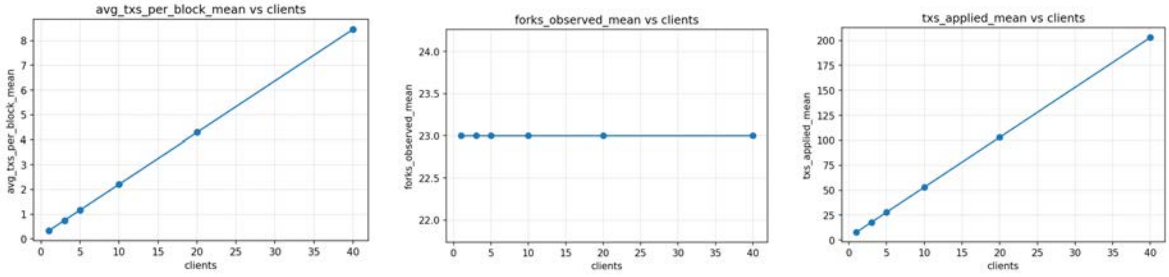
More clients give more txs per block and more applied updates. Forks rise only a bit. This is the expected shape. Look for the “knee”: the point where gains slow. If the forks line stays flat, the system is still safe.

Throughput and applied updates increase near-linearly with the number of clients, about *Throughput*, defined as $tx/block$ when the number of transactions increases and the validators remain the same, we should expect that the throughput should increase, and the sanity/confirmation can be seen in Table 5.5. For *fork_growth*,

$$\text{fork rate} \;\approx\; \Pr[K \geq 2] \times \phi\left(\frac{\delta}{T_{\text{slot}}}\right),$$

Where $\Pr[K \geq 2]$ (multiple eligible proposers),and δ is the effective block spread time, while the overlap factor $\phi(\cdot)$ grows as blocks and gossip get slower under higher client load, even if the validator numbers remain the same, increasing the number of clients in the network, increases the block size, which in turn makes it harder to verify/pick/relay the block, consistent with longest-chain dynamics under higher load (Section 3.3).

Figure 5.1: Batch 1 (segment ’A’): scaling clients.



(α′) txs/block vs. clients (β′) Forks vs. clients (γ′) Applied updates vs. clients

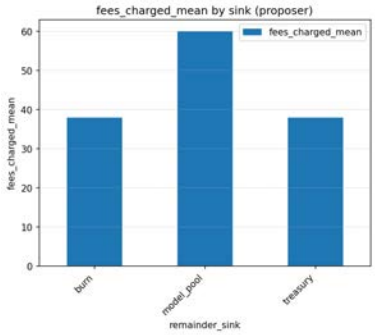
5.3.3 Fee sink policy (remainder destinations)

Table 5.6: Fee policy vs. remainder sink (means).

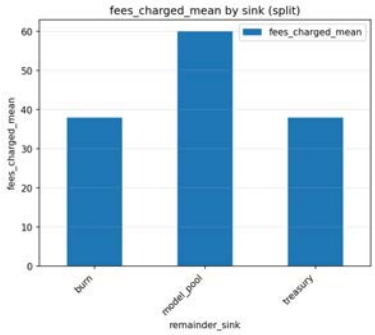
fee_policy	remainder_sink	fees_charged_mean	forks_observed_mean	txs_applied_mean
proposer	burn	38.000	20.000	28.000
proposer	model_pool	60.000	20.000	44.000
proposer	treasury	38.000	20.000	28.000
split	burn	38.000	20.000	28.000
split	model_pool	60.000	20.000	44.000
split	treasury	38.000	20.000	28.000
treasury	burn	38.000	20.000	28.000
treasury	model_pool	60.000	20.000	44.000
treasury	treasury	38.000	20.000	28.000

Executing remainder payments/fees to a *model pool* increases both fees charged and updates applied, with no effect on forks (Table 5.6). Assuring alignment with our rules-based incentives (previously discussed in Section 3.4): more immediate value routed to the task stimulates contribution without disturbing consensus, Main take-away with a better output, same safety guarantees.

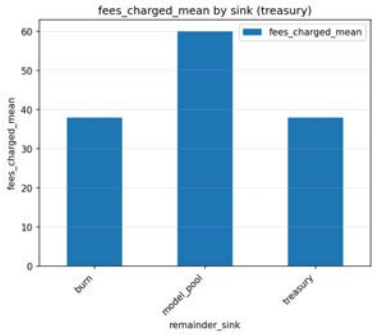
Figure 5.2: Batch 1 (segment ’B’): fee policy × sink destination.



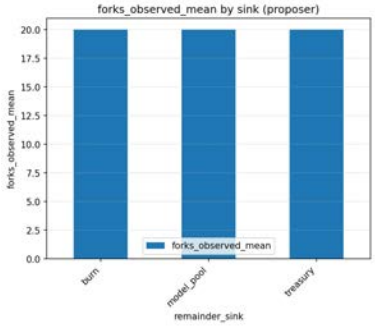
(α′) Fees (proposer)



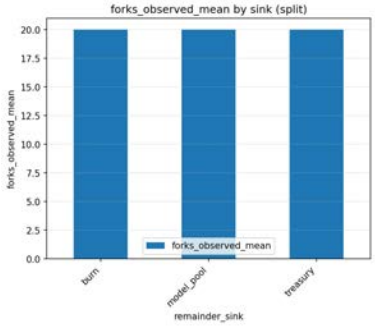
(β′) Fees (split)



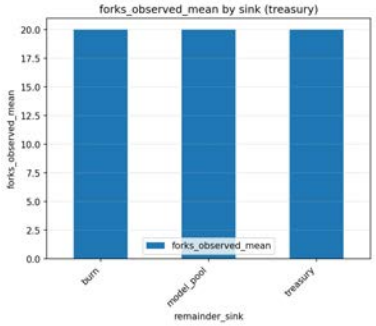
(γ′) Fees (treasury)



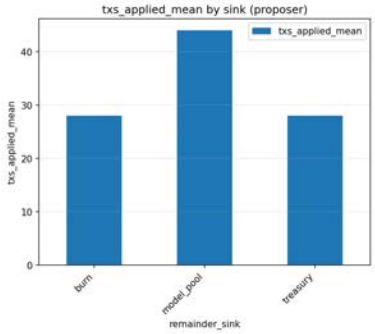
(δ′) Forks (proposer)



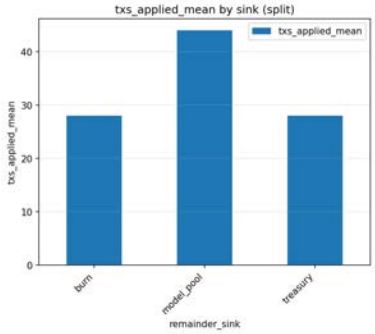
(ε′) Forks (split)



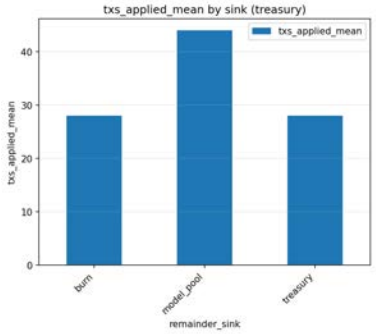
(στ′) Forks (treasury)



(ζ′) Applied (proposer)



(η′) Applied (split)



(θ′) Applied (treasury)

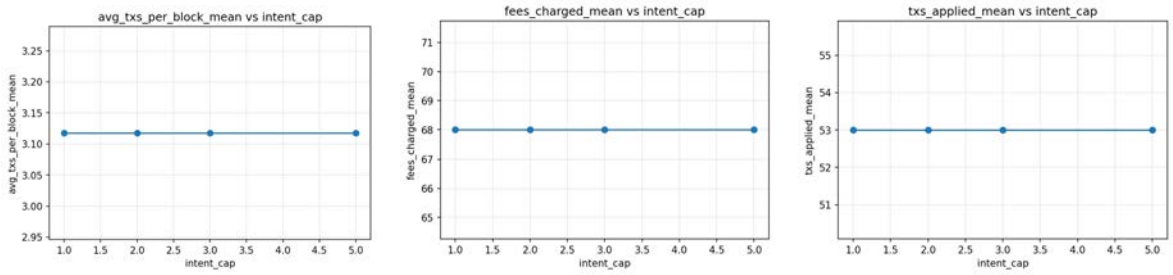
5.3.4 Intent-cap sensitivity

Table 5.7: `intent_cap` sweep (means).

intent_cap	avg_txs_per_block_mean	fees_charged_mean	txs_applied_mean
1	3.118	68.000	53.000
2	3.118	68.000	53.000
3	3.118	68.000	53.000
5	3.118	68.000	53.000

Within the tested range, per-account `intent_cap` had negligible to no effect (Table 5.7), suggesting our ticket deposit alongside rate boundaries holds, (see Section 3.4) and that they are already sufficient to deter spam even at the current workload, in other words all the tested caps gave the same means, this translates to that the current load is well with the safe bounds of the setup, No further tuning is required for this knob at this scale.

Figure 5.3: Batch 2 (segment 'A'): `intent_cap` sweep.



(α′) txs/block (β′) Fees charged (γ′) Applied updates

5.3.5 IPFS availability stress

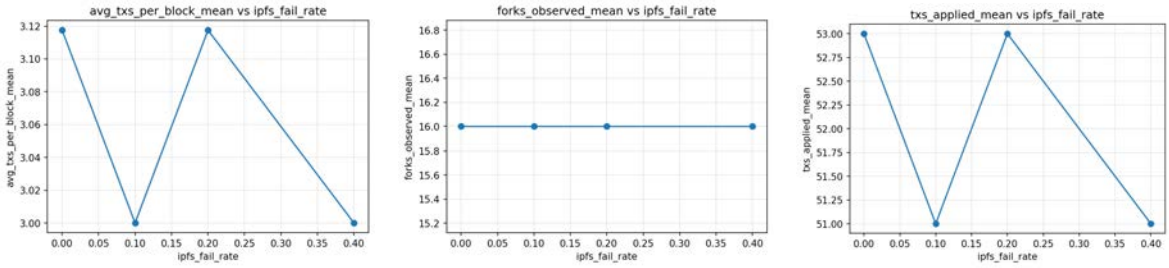
By increasing the ipfs_fail_rate, we should expect that all forks aren't increased since consensus doesn't depend on IPFS to continue, we should also expect that some transactions will be lost due to IPFS failure, so txs_applied_mean would slightly decrease, but for avg_txs_per_block to not drop by much since, intents, publish, refunds, listing still fill the blocks so throughput should remain the same in a small experiment.

Table 5.8: IPFS failure rate sweep (means).

ipfs_fail_rate	avg_txs_per_block_mean	forks_observed_mean	txs_applied_mean
0.00	3.118	16.000	53.000
0.10	3.000	16.000	51.000
0.20	3.118	16.000	53.000
0.40	3.000	16.000	51.000

Even with 40% update fetch failures, the forks remain flat and only a small drop in applied updates is observed (Table 5.8). This is exactly the availability behaviour designed in Chapter 4: validators never fetch model bytes on-path; the Lister excludes unretrievable CIDs and advances the round.

Figure 5.4: Batch 2 (segment ’B’): IPFS sensitivity.



(α′) txs/block (β′) Forks (γ′) Applied

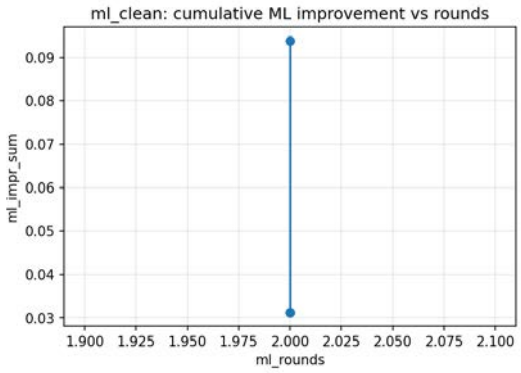
5.3.6 Round-level ML improvement

Table 5.9: Cumulative ML improvement (two-round runs).

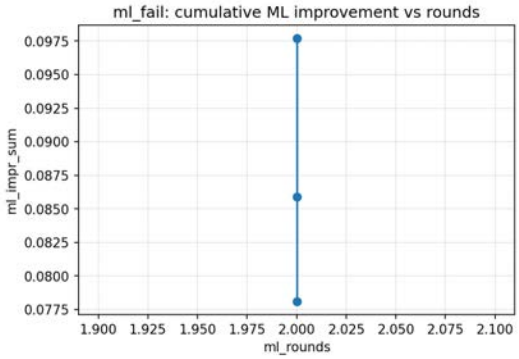
ml_rounds_mean	ml_impr_sum_mean_clean	ml_impr_sum_mean_fail
2.000	0.031	0.086

Both “`ml_clean`” and “`ml_fail`” configurations produce positive cumulative improvement over two rounds (Table 5.9), defending the assertion that the deterministic aggregator plus validation-based scoring (see Section 3.4) can advance the model even under some adverse conditions.

Figure 5.5: Batch 2 (segment 'C'): cumulative ML improvement across two rounds.



(α′) `ml_clean`



(β′) `ml_fail`

5.3.7 Randomness period in VRF leader election

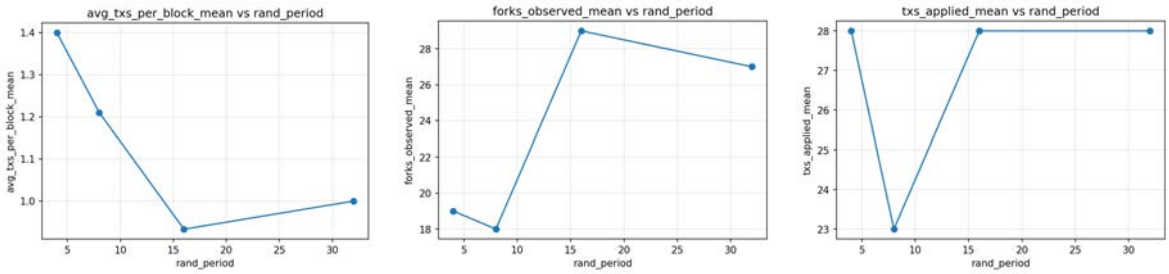
By increasing the rand_period, you have more time with the given VRF origin/seed for more time slots. Thus this makes any proposer node, have his eligibility stay for more and be more predictable, but sticky eligibility translates to more forks, and since the round is longer, forks last longer even if the system makes the same number of blocks. The table that follows proves our hypothesis.

Table 5.10: Randomness period sweep (means).

rand_period	avg_txs_per_block_mean	forks_observed_mean	txs_applied_mean
4	1.400	19.000	28.000
8	1.211	18.000	23.000
16	0.933	29.000	28.000
32	1.000	27.000	28.000

Longer periods (slower entropy refresh) increase forks with a tiny margin in throughput (Table 5.10), which is consistent with the stake-proportional VRF analysis in Section 4.4.1.

Figure 5.6: Batch 3 (segment 'A'): effect of randomness period.



(α') txs/block (β') Forks (γ') Applied

5.3.8 Eligibility threshold τ

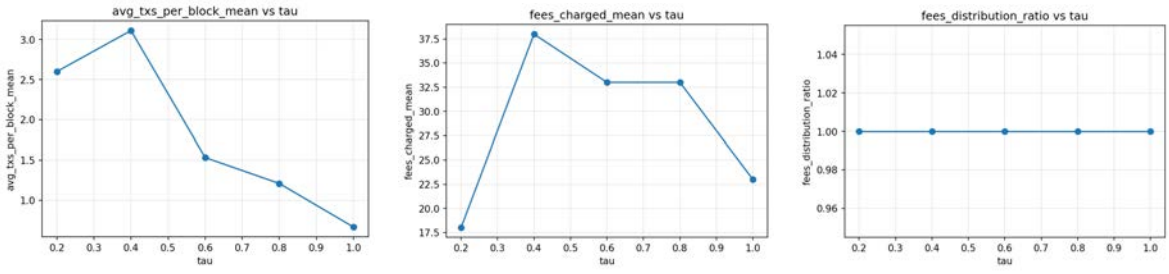
In this setup, we increase the τ parameter(i.e., the expected number of eligible proposers per slot goes up), the lower τ the less proposers you have(i.e., not a lot of forks). So we should expect in these evaluated experiment that if we start increasing this parameter the number of forks will rise alongside it.

Table 5.11: τ sweep (means; fees–distribution remains 1).

tau	avg_txs_per _block _mean	fees _charged _mean	fees _distri-bution _ratio	forks _ob-served _mean	txs _applied _mean
0.20	2.600	18.000	1.000	4.000	13.000
0.40	3.083	38.000	1.000	8.000	28.000
0.60	1.533	33.000	1.000	14.000	23.000
0.80	1.200	33.000	1.000	18.000	23.000
1.00	0.667	23.000	1.000	26.000	18.000

Moderate τ increases proposer eligibility (hence throughput and applied updates) while very high τ raises contention and forks (Table 5.11). Notably, glimpse that fees–distribution proportion holds at 1 across all settings, confirming correctness-guarantee of validator accounting (see Section 4.4).

Figure 5.7: Batch 3 (segment ’B’): τ sensitivity.



(α′) txs/block) (β′) Fees charged (γ′) Fees–distribution ratio

(δ′) Forks (ε′) Applied

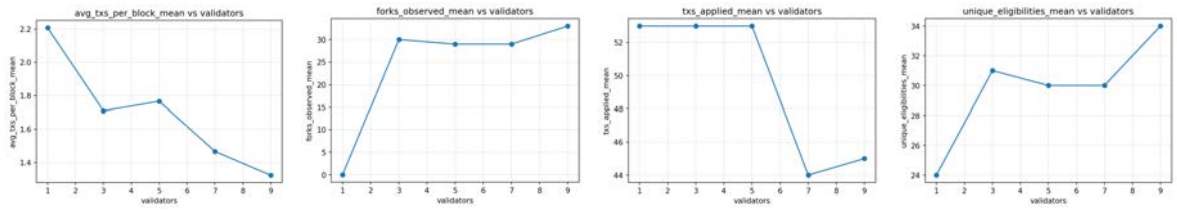
5.3.9 Scaling the validator set

Table 5.12: Validator scaling (means).

validators	avg_txs_per_block_mean	forks_observed_mean	txs_applied_mean	unique_eligibilities_mean
1	2.211	0.000	53.000	24.000
3	1.706	30.000	53.000	31.000
5	1.765	29.000	53.000	30.000
7	1.471	29.000	44.000	30.000
9	1.324	33.000	45.000	34.000

As the validator set grows, proposer uniqueness increases (healthier leader dispersion) at the cost of more forks and slightly lower throughput—canonical behaviour for longest-chain systems under higher concurrency.

Figure 5.8: Batch 3 (segment ’C’) and Batch 4: validator scaling.



(α′) txs/block (β′) Forks (γ′) Applied (δ′) Unique eligibilities

5.4 What the CSVs Say (Side-by-Side)

For completeness, we collate all sweep summaries here. Each table comes directly from the validator read API over confirmed blocks; means are across runs.

Table 5.13: All summaries at a glance part a(per-sweep tables).

(a) Baseline

avg_txs_per_block _mean	forks_observed _mean	txs_applied _mean	fees_charged _mean	fees_distribution _ratio
1.571	14.000	26.000	25.000	1.000

(b) Clients

clients	avg_txs_per_block_mean	forks_observed_mean	txs_applied_mean
4	3.118	16.000	53.000
8	3.694	20.000	66.000
12	4.071	24.000	72.000

Table 5.14: All summaries at a glance part b(per-sweep tables).

(c) Fee policy × sink

fee_policy	remainder_sink	fees_charged_mean	forks_observed_mean	txs_applied_mean
proposer	burn	38.000	20.000	28.000
proposer	model_pool	60.000	20.000	44.000
proposer	treasury	38.000	20.000	28.000
split	burn	38.000	20.000	28.000
split	model_pool	60.000	20.000	44.000
split	treasury	38.000	20.000	28.000
treasury	burn	38.000	20.000	28.000
treasury	model_pool	60.000	20.000	44.000
treasury	treasury	38.000	20.000	28.000

(d) Intent cap

intent_cap	avg_txs_per_block_mean	fees_charged_mean	txs_applied_mean
1	3.118	68.000	53.000
2	3.118	68.000	53.000
3	3.118	68.000	53.000
5	3.118	68.000	53.000

(e) IPFS fail rate

ipfs_fail_rate	avg_txs_per_block_mean	forks_observed_mean	txs_applied_mean
0.00	3.118	16.000	53.000
0.10	3.000	16.000	51.000
0.20	3.118	16.000	53.000
0.40	3.000	16.000	51.000

Table 5.15: All summaries at a glance part c(per-sweep tables).

(f) Randomness period

rand_period	avg_txs_per_block_mean	forks_observed_mean	txs_applied_mean
4	1.400	19.000	28.000
8	1.211	18.000	23.000
16	0.933	29.000	28.000
32	1.000	27.000	28.000

(g) τ threshold

tau	avg_txs_per _block _mean	fees_charged _mean	fees _distribution _ratio	forks _observed _mean	txs_applied _mean
0.20	2.600	18.000	1.000	4.000	13.000
0.40	3.083	38.000	1.000	8.000	28.000
0.60	1.533	33.000	1.000	14.000	23.000
0.80	1.200	33.000	1.000	18.000	23.000
1.00	0.667	23.000	1.000	26.000	18.000

(h) Validators

validators	avg_txs_per_block_mean	forks_observed_mean	txs_applied_mean	unique_eligibilities_mean
1	2.211	0.000	53.000	24.000
3	1.706	30.000	53.000	31.000
5	1.765	29.000	53.000	30.000
7	1.471	29.000	44.000	30.000
9	1.324	33.000	45.000	34.000

(i) ML improvement (two rounds)

ml_rounds_mean	ml_impr_sum_mean_clean	ml_impr_sum_mean_fail
2.000	0.031	0.086

5.5 Batch Mapping and Figure Organization

For traceability to the raw images and tables provided, we group visuals by submission batch: Batch 1 contains client-scaling and fee–sink ablations (Figs. 5.1–5.2); Batch 2 includes `intent_cap`, IPFS sensitivity, and the ML-improvement curves (Figs. 5.3–5.5); Batch 3 covers randomness period, τ sensitivity, and validator scaling (Figs. 5.6–5.8); Batch 4 contributes the additional validator metrics included in Fig. 5.8. All figure file names are used verbatim as sent.

Table 5.16: Metrics quick reference (short, plain meanings).

Name	Unit	What it means / how we get it
`avg_txs_per_block_mean`	txs/block	Average transactions per block. Mean over blocks, then over runs.
`forks_observed_mean`	events/run	Fork events per run. We count when we see 2+ heads at one height. Lower is better.
`txs_applied_mean`	txs/run	Transactions that change state (e.g., `UpdateTx`, `GlobalUpdateTx`). Mean per run.
`unique_eligibilities_mean`	count/run	Distinct validator–slot wins in a run. Higher means wider spread.
`ml_rounds_mean`	rounds/run	Finished FL rounds per run (list → tickets → aggregate → global).
`ml_impr_sum_mean_clean`	score	Sum of model gains in clean runs. Same public test each round.
`ml_impr_sum_mean_fail`	score	Same as above, but with IPFS faults. Shows robustness.
Fees–distribution ratio	unitless	Fees charged ÷ fees paid out. Should be 1.0 (money is conserved).
Proposer count (per validator)	blocks	Blocks proposed by each validator in the run.
Proposer dispersion (Gini/var)	unitless	Spread of proposer counts. Lower Gini means fairer.
Time between proposals	slots	For each validator, slots between its proposals. Shorter is better.
Refunds	txs or currency	Tickets we refund. CID could be fetched in the window.
Forfeits	txs or currency	Tickets we burn. Unrevealed, or CID could not be fetched in the window.

5.6 End-to-End Guarantees (Observed)

Causality and replayability. Every experiment replays deterministically from on-chain references (`included[]`, `scores[]`, `refunds[]`, CIDs), matching the consensus/application

invariants in Section 3.4 and validator execution (Alg. 23).

Availability without consensus coupling. Under induced IPFS failures (see Section 5.3.5), rounds progress and deposits are forfeited or refunded strictly by policy (see Section 3.4); consensus never stalls, confirming the setup partition of storage and chain (see Section 4.5).

Value conservation. The *fees–distribution ratio* stays at 1.0 across sweeps (Table 5.11), evidencing correct accounting and fee splitting in the validator state machine.

PoS dynamics. Varying τ, randomness period, and validator set size produces the expected chain-growth/quality trade-offs (see Section 3.3, Section 4.4.1): moderate τ boosts throughput; higher τ and slower entropy increase forks; a larger validator set improves proposer dispersion at some cost to throughput.

Incentives. Routing remainder fees to a model pool (Section 5.3.3) measurably increases applied updates without harming equilibrium, exemplifying that the on-chain economic objects (See background at Section 3.4) meaningfully drive client behaviour.

5.7 Discussion

Overall, the empirical picture matches the formal story in Chapters 3 and 4. The ticketed publication flow with deposits and refunds maintains liveness under adverse storage conditions; the validator-only causality checks and PoS fork choice maintain safety; and the lister's deterministic aggregation with public scoring produces reproducible model progress. Importantly, all of these properties are visible and auditable through a minimal on-chain interface, with large artifacts referenced by CIDs, as designed.

Reproducibility note. The full implementation code can be found at GITHUB LINK with instructions on how to run and reproduce, the CSV summaries utilized by the Tables 5.2–5.13 correspond to: `baseline_summary.csv`, `fees_vs_sink_summary.csv`, `intent_cap_summary.csv`, `ipfs_fail_summary.csv`, `ml_clean_summary.csv`, `ml_fail_summary.csv`, `rand_period_summary.csv`, `scale_clients_summary.csv`, `scale_validators_summary.csv`, and

`tau_summary.csv`.

Chapter 6

Conclusion

This chapter concludes the thesis by reflecting on what was built, the empirical evidence, and the potential future of the design. The previous chapters detailed a setup that utilized listings, ticketed intents, publications anchored to IPFS CIDs, and deterministic GlobalUpdateTxs to close out the process. Chapter 3showed the conceptual design and assumptions of the system, Chapter 4 showed a reproducible and verifiable prototype, and Chapter 5 examined the functionality of the system in different operational states and failures.

6.1 Summary of Contributions

The work contributes a complete end-to-end architecture that includes consensus, incentives, and off-chain storage without entangling the blockchain with large tensors or model logic. Privacy and tolerance to attacks. Tickets and deposit prices influence participation and help resist Sybil pressure. IPFS provides content addressing and availability without burdening the ledger; validators only commit to CIDs and causal state, which keeps consensus minimal. The lister's aggregation is deterministic and transparent, so anyone can audit a round from on-chain references alone.

The prototype embodies these ideas with precise interfaces and executable pseudocode for listers, clients, and validators, together with a VRF-based leader-election(PoS) and the help of a longest-chain(LC)fork choice. The evaluation demonstrated that the proposed system remained sound across common operating points, that it degrades gracefully under storage faults, and that its consensus parameters give a clear performance–safety trade-off surface.

6.2 Empirical Takeaways

Across the fee-distribution sinks, we found that the setups/pipeline's health was preserved, and the throughput and additive metrics remained stable, following the set distribution ratio. From the opposite direction, we found that a variable intent cap did not have much of an impact on the core workings of the chain, showing that tickets are successfully throttling spam without suppressing legitimate throughput. Also injecting off-chain IPFS failures did not affect the consensus safety of the network and only moderately reduced the number of updates that were applied. This is basically as we wanted it and shows that we have made the right choice in putting the burden of keeping the listing on the lister layer rather than the whole network. The randomness period and the VRF threshold τ showed the expected trade-offs. When making these settings longer and higher, we get more dense eligibility, and therefore a higher chance of short-lived contention, which our data confirmed with higher fork observations and a slight reduction in the size of the average block. Increasing the number of validators resulted in more significant variety of proposers and an increased number of unique/individual, eligible nodes, with a minor consequence being a reduction in per-block payload. Lastly, the FL pipeline showed that it could improve the quality of its work, per round and that in both our clean and failure tests, the chain continued to work, even in the instance that some updates were missing. Deposit/refunds were calculated precisely as per the deterministic policy that was fixed on the chain.

6.3 Limitations

The prototype intentionally fixes several degrees of freedom. The robust aggregator is deterministic but simple; stronger Byzantine resilience (e.g., coordinate-wise median) would increase tolerance but introduce heavier computation and tuning. The PoS layer is backbone-style and does not include a finality gadget; while confirmation depth k sufficed for our threat model, applications with stricter latency/rollback constraints may need clear finality guarantee. Storage availability remains an economic contract between the lister and clients; if huge models or malicious or not attacks are expected, wider pinning or erasure coding would be prudent.

6.4 Future Work

Three directions are immediate. First, integrate a configurable family of robust aggregators and public validation tasks with verifiable metrics, so that `scores[]` can be verified by third parties beyond replay. Second, add an optional finality layer on top of the longest-chain backbone to reduce time-to-confidence for `GlobalUpdateTx`. Third, extend the incentive market: dynamic deposits and per-account caps that react to observed behavior, as well as auctioned publication slots to clear demand at peak load. Orthogonally, secure aggregation and DP can be offered as model-specific `rules` without touching th consensus.

6.5 Closing Remarks

This thesis argues and demonstrates in a working prototype that decentralized coordination of FL can be driven more functional without collapsing the BC-Network into a compute or storage substrate. By dividing problems, and including the consensus on causality and value, on the chain, bytes and models off-chain, and economics to encourage participation, the design keeps each layer simple, verifiable, and robust. The executed evaluation showed that the resulting system is not only conceptually sound but operationally stable across a realistic parameter envelope. The path forward is incremental: stronger robustness, faster confirmations, and richer markets, all while maintaining the essence invariants that make the system effortless to assert and to trust.

Bibliography

[1] Jesse Yli-Huumo, Deokyoon Ko, Sujin Choi, Sooyong Park, and Kari Smolander. Where is current research on blockchain technology?—a systematic review. *PLOS ONE*, 11, 10 2016.

[2] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, and Vignesh Kalyanaraman. Blockchain technology: Beyond bitcoin. *Applied Innovation Review*, (2):6–19, 2016.

[3] Zhilin Wang, Qin Hu, Minghui Xu, Yan Zhuang, Yawei Wang, and Xiuzhen Cheng. A systematic survey of blockchained federated learning, 2024.

[4] Hyesung Kim, Jihong Park, Mehdi Bennis, and Seong-Lyun Kim. Blockchained on-device federated learning. *IEEE Communications Letters*, 24, 06 2019.

[5] Yunlong Lu, Xiaohong Huang, Yueyue Dai, Sabita Maharjan, and Yan Zhang. Blockchain and federated learning for privacy-preserved data sharing in industrial iot. *IEEE Transactions on Industrial Informatics*, PP:1–1, 09 2019.

[6] Shicheng Wan, Hong Lin, Wensheng Gan, Jiahui Chen, and Philip S. Yu. Web3: The next internet revolution, 2023.

[7] Gavin Wood. Why we need web 3.0. Medium, 2018. Accessed Sept. 2025.

[8] John R. Douceur. The sybil attack. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, pages 251–260, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[9] Ethereum.org. Proof-of-authority (poa), 2024. Explains PoA as reputation-based with authorized signers; typically used in private chains and testnets.

[10] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[11] Juan Benet. Ipfs - content addressed, versioned, p2p file system, 2014.

[12] Protocol Labs. Filecoin: A decentralized storage network. White paper, 2017.

[13] Dennis Trautwein, Yiluo Wei, Yiannis Psaras, Moritz Schubotz, Ignacio Castro, Bela Gipp, and Gareth Tyson. Ipfs in the fast lane: Accelerating record storage with optimistic provide. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*, pages 1920–1929, 2024.

[14] Sid Lamichhane and Patrick Herbke. Verifiable decentralized ipfs cluster: Unlocking trustworthy data permanency for off-chain storage, 2024.

[15] Rameez Asif and Syed Hassan. Shaping the future of ethereum: exploring energy consumption in proof-of-work and proof-of-stake consensus. *Frontiers in Blockchain*, 6, 08 2023.

[16] H. B. McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *International Conference on Artificial Intelligence and Statistics*, 2016.

[17] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. A survey on distributed machine learning. *ACM Computing Surveys*, 53(2):1–33, March 2020.

[18] Keith Bonawitz et al. Towards federated learning at scale: System design, 2019.

[19] Peter Kairouz, H. Brendan McMahan, Brendan Avent, et al. Advances and open problems in federated learning, 2021.

[20] Zili Lu, Heng Pan, Yueyue Dai, Xueming Si, and Yan Zhang. Federated learning with non-iid data: A survey. *IEEE Internet of Things Journal*, 11(11):19188–19209, June 2024.

[21] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks, 2020.

[22] Mang Ye, Xiuwen Fang, Bo Du, Pong C. Yuen, and Dacheng Tao. Heterogeneous federated learning: State-of-the-art and research challenges, 2023.

[23] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank J. Reddi, Sebastian U. Stich, and Ananda Theertha Suresh. Scaffold: Stochastic controlled averaging for federated learning, 2021.

[24] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding, 2017.

[25] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[26] Truc Nguyen and My T. Thai. Preserving privacy and security in federated learning. *IEEE/ACM Transactions on Networking*, 32(1):833–843, February 2024.

[27] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, page 308–318. ACM, October 2016.

[28] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. volume Vol. 3876, pages 265–284, 01 2006.

[29] Bonawitz. Practical secure aggregation for privacy-preserving machine learning. pages 1175–1191, 10 2017.

[30] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients, 2019.

[31] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep models under the gan: Information leakage from collaborative deep learning, 2017.

[32] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning, 2019.

[33] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing federated learning through an adversarial lens, 2019.

[34] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[35] Vitalik Buterin. Ethereum: A next-generation smart contract & decentralized application platform. White paper, 2014.

[36] Vitalik Buterin. On public and private blockchains (the ”scalability trilemma”). Blog post, 2017.

[37] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. White paper, 2016.

[38] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform, 2017.

[39] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. 2012.

[40] Christine V. Helliar, Louise Crawford, Laura Rocca, Claudio Teodori, and Monica Veneziani. Permissionless and permissioned blockchain diffusion. *International Journal of Information Management*, 54:102136, 2020.

[41] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. pages 357–388, 07 2017.

[42] Elli Androulaki, Barger, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys ’18, page 1–15. ACM, April 2018.

[43] Richard Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: An introduction, 09 2016.

[44] Aritri Saha and Ujjwal Guin. Optimizing supply chain management using permissioned blockchains. pages 1–7, 04 2025.

[45] Katru Rao and s Naganjaneyulu. Permissioned healthcare blockchain system for securing the ehrs with privacy preservation. *Ingénierie des systèmes d information*, 26:393–402, 08 2021.

[46] Andreas Schaad, Jonathan Moffett, and Jeremy Jacob. The role-based access control system of a european bank: a case study and discussion. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, SACMAT ’01, page 3–9, New York, NY, USA, 2001. Association for Computing Machinery.

[47] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, pages 173–186, 1999.

[48] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain, 2019.

[49] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[50] Meng Shen, Zhehui Tan, Dusit Niyato, Yuzhi Liu, Jiawen Kang, Zehui Xiong, Liehuang Zhu, Wei Wang, Xuemin, and Shen. Artificial intelligence for web 3.0: A comprehensive survey, 2023.

[51] Thomas A. Henzinger, Nicolas Mazzocchi, and N. Ege Saraç. Quantitative safety and liveness, 2023.

[52] Pieter van Goor and Robert Mahony. Synchronous models and fundamental systems in observer design, 2025.

[53] Victor Kozyakin. A short introduction to asynchronous systems. pages 153–165, 01 2004.

[54] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[55] Jae Kwon. Tendermint : Consensus without mining. 2014.

[56] Weiyu Zhong, Ce Yang, Wei Liang, Jiahong Cai, Lin Chen, Jing Liao, and Naixue Xiong. Byzantine fault-tolerant consensus algorithms: A survey. *Electronics*, 12:3801, 09 2023.

[57] Fredy Andres Aponte-Novoa, Ana Lucila Sandoval Orozco, Ricardo Villanueva-Polanco, and Pedro Wightman. The 51 *IEEE Access*, 9:140549–140564, 2021.

[58] Joachim Neu, Srivatsan Sridhar, Lei Yang, David Tse, and Mohammad Alizadeh. Longest chain consensus under bandwidth constraint, 2022.

[59] Kangde Liu, Zheng Yan, Xueqin Liang, Raimo Kantola, and Chuangyue Hu. A survey on blockchain-enabled federated learning and its prospects with digital twin. *Digital Communications and Networks*, 10(2):248–264, 2024.

[60] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Proceedings of the IFIP TC6/TC11 Joint Working Conference on Secure Information Networks: Communications and Multimedia Security*, CMS '99, page 258–272, NLD, 1999. Kluwer, B.V.

[61] Mohamed Baza, Mahmoud Nabil, Mohamed Mahmoud, Niclas Bewermeier, Kemal Fidan, Waleed Alasmary, and Mohamed Abdallah. Detecting sybil attacks using proofs of work and location in vanets. *IEEE Transactions on Dependable and Secure Computing*, PP:1–1, 05 2020.

[62] Felix Hoffmann. Challenges of proof-of-useful-work (pouw), 2022.

[63] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget, 2019.

[64] Zhiguo He, Jiasun Li, and Zhengxun Wu. Don't trust, verify: The case of slashing from a popular ethereum explorer. In *Companion Proceedings of the ACM Web Conference 2023*, WWW '23 Companion, page 1078–1084, New York, NY, USA, 2023. Association for Computing Machinery.

[65] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. Cryptology ePrint Archive, Paper 2017/454, 2017.

[66] Anton Wahrstätter, Sajjad Khan, and Davor Svetinovic. Openfl: A scalable and secure decentralized federated learning system on the ethereum blockchain. *Internet of Things*, 26:101174, 2024.

[67] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap, 2022.

[68] A. Athreya, Ashwin Kumar, S. Nagarajath, Gururaj H L, V. Kumar, Sachin D N, and Rakesh K R. *Peer-to-Peer Distributed Storage Using InterPlanetary File System*, pages 711–721. 01 2021.

[69] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, page 53–65, Berlin, Heidelberg, 2002. Springer-Verlag.

[70] Hoon Ko, Juhee Oh, and Sung Kim. Digital content management using non-fungible tokens and the interplanetary file system. *Applied Sciences*, 14:315, 12 2023.

[71] Peng Kang, Wenzhong Yang, and Jiong Zheng. Blockchain private file storage-sharing method based on ipfs. *Sensors*, 22:5100, 07 2022.

[72] Jing Li, Jufeng Sun, Fengyong Li, Xingkai Yang, and Yuwei Wang. Biflc: A blockchain and ipfs-based multi-consensus federated learning framework. In *2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 1409–1414, 2023.

[73] Yajing Xu, Zhihui Lu, Keke Gai, Qiang Duan, Junxiong Lin, Jie Wu, and Kim-Kwang Raymond Choo. Besifl: Blockchain-empowered secure and incentive federated learning paradigm in iot. *IEEE Internet of Things Journal*, 10(8):6561–6573, 2023.

[74] Yuzheng Li, Chuan Chen, Nan Liu, Huawei Huang, Zibin Zheng, and Qiang Yan. A blockchain-based decentralized federated learning framework with committee consensus. *IEEE Network*, 35(1):234–241, January 2021.

[75] Washington Enyinna Mbonu, Carsten Maple, and Gregory Epiphaniou. An end-process blockchain-based secure aggregation mechanism using federated machine learning. *Electronics*, 12(21), 2023.

[76] Christodoulos Pappas, Dimitris Chatzopoulos, Spyros Lalis, and Manolis Vavalis. Ipls: A framework for decentralized federated learning. pages 1–6, 06 2021.

[77] Stefanos Leonardos, Daniël Reijsbergen, and Georgios Piliouras. Weighted voting on the blockchain: Improving consensus in proof of stake protocols. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 376–384, 2019.

[78] S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 120–130, 1999.

[79] James Hsin yu Chiang, Bernardo David, Ittay Eyal, and Tiantian Gong. FairPoS: Input fairness in permissionless consensus. Cryptology ePrint Archive, Paper 2022/1442, 2022.

[80] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. The provable security of ed25519: Theory and practice. pages 1659–1676, 05 2021.

[81] Aggelos Kiayias and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. pages 281–310, 04 2015.

[82] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models, 2017.

[83] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 66–98, Cham, 2018. Springer International Publishing.

[84] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 913–930, New York, NY, USA, 2018. Association for Computing Machinery.

[85] Krishna Pillutla, Sham M. Kakade, and Zaid Harchaoui. Robust aggregation for federated learning. *IEEE Transactions on Signal Processing*, 70:1142–1154, 2022.

[86] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Byzantine-tolerant machine learning, 2017.

[87] Dong Yin, Yudong Chen, Kannan Ramchandran, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates, 2021.

[88] Arul Selvakumar and C. Ganadhas. The evaluation report of sha-256 crypt analysis hash function. *Communication Software and Networks, International Conference on*, 0:588–592, 01 2009.

[89] Numpy. Python data type objects: dtype.

APPENDICES

Appendix A′

Algorithms

A′.1 Validator Leader Election and Block proposal

Algorithm 31 proposeAndValidateBlock on slot tick (run by every validator)

procedure onSlotTick(t)
  $\langle ok, \rho, \pi \rangle \leftarrow$ proposerEligible$(i, t, \mathsf{R}_e, s_i, S, \tau)$
  if ok **then**
    $txs \leftarrow$ SELECT_TXS(mempool) ▷ local policy; all txs must pass admission
    $hdr \leftarrow \langle \texttt{parentHash}, t, e, \rho, \pi, i \rangle$
    $st' \leftarrow$ REPLAY(txs, st); $hdr.\texttt{stateRoot} \leftarrow$ ROOT(st')
    $hdr.\texttt{sig} \leftarrow$ SIGN$(sk_i,$ HASH$(hdr))$
    GOSSIP_BLOCK$(\langle hdr, txs \rangle)$

A′.2 Validator Helper Algorithms

Algorithm 32 verifyIncomingBlock function

function verifyIncomingBlock($B = \langle hdr, txs \rangle$)

$m \leftarrow \mathsf{domainSep} \parallel \mathsf{epoch}(e) \parallel \mathsf{slot}(hdr.t) \parallel \mathsf{parentHash}$

if not $\mathsf{VRF.Verify}(pk_{hdr.proposer}, m, hdr.\texttt{vrfProof})$ **then**

return reject

$\rho \leftarrow \mathsf{Normalize}(H(\mathsf{VRF.Out}(hdr.\texttt{vrfProof})))$

if $\rho \geq \tau \cdot \frac{s_{hdr.proposer}}{S}$ **then**

return reject

if not VERIFY_SIG($pk_{hdr.proposer}$, HASH(hdr), $hdr.\texttt{sig}$) **then**

return reject

$st' \leftarrow \text{REPLAY}(txs, st_{\texttt{parent}})$; **require** ROOT($st'$) $= hdr.\texttt{stateRoot}$

return accept_candidate(B, ρ)

Algorithm 33 forkChoice function

function forkChoice(C) ▷ C = set of competing tips at same or greater height

return argmax$_{B \in C}$ { $\texttt{chainHeight}(B)$, then **min** ρ per-slot }

Algorithm 34 updateEpochRandomness($\mathsf{R}_e, \mathcal{B}_e$) $\rightarrow \mathsf{R}_{e+1}$

function updateEpochRandomness($\mathsf{R}_e, \mathcal{B}_e$)

$\Pi \leftarrow [\,]$; **for each** accepted block $B \in \mathcal{B}_e$ **do** append(Π, $B.hdr.\texttt{vrfProof}$)

$\Pi' \leftarrow \text{sortLex}(\Pi)$ ▷ deterministic ordering

return $H(\mathsf{R}_e \parallel \Pi')$

Algorithm 35 Validator.admitFunding(*tx*)

function admitFunding(tx)

if not VERIFY_SIG($tx.lister,\ tx.sig$) **then**

reject

if $tx.amount \leq 0$ **then**

reject

if not IS_LISTER_OR_DELEGATE($tx.lister, tx.modelID$) **then**

reject

if not CHECK_BALANCE($tx.lister,\ tx.amount$) **then**

reject

ENQUEUE_TX(mempool, tx)

Algorithm 36 Validator.advanceRound(*modelID*, *round*, *globalCID*, *included*, ***scores***, *refunds*) — robust

function ADVANCE_ROUND($modelID, round, globalCID, included,$ **scores**$, refunds$)
  require ROUND_INDEX(modelID) = round
  require $|$*included*$| = |$**scores**$|$
  require IS_CID_VALID(globalCID)
  require IS_STRICTLY_UNIQUE(included) **and** IS_STRICTLY_UNIQUE(refunds)
  for all $t \in included$ **do**
    require IS_TICKET_OF($t, \langle modelID, round \rangle$)
    require IS_REVEALED(t)
  for all $t \in refunds$ **do**
    require IS_TICKET_OF($t, \langle modelID, round \rangle$)
    require IS_REVEALED(t)
  SET_GLOBAL_CID($modelID, globalCID$)
  $\langle \gamma, d \rangle \leftarrow$ RULES($modelID$) ▷ curvature and deposit size
  $R \leftarrow$ POOL_CURRENT_ROUND_AMOUNT($modelID$)
  $S \leftarrow \sum_{j=1}^{|included|} \max(0,$ **scores**$[j])^{\gamma}$ ▷ ignore negative inputs defensively
  $paid \leftarrow 0$
  for $j \leftarrow 1 \ldots |$*included*$|$ **do**
    $acct \leftarrow$ TICKET_OWNER(*included*$[j]$)
    $s \leftarrow \max(0,$ **scores**$[j])$
    if $S > 0$ **and** $s > 0$ **then**
      $pay \leftarrow$ ROUNDING_SAFE$(R \cdot \frac{s^{\gamma}}{S})$
      $paid \leftarrow paid + pay$
      ENTITLEMENT_ADD($acct, modelID, pay$)
    MARK_TICKET_FINALIZED(*included*$[j]$)
  for all $t \in refunds$ **do** ▷ deposit refunds are separate from rewards
    $acct \leftarrow$ TICKET_OWNER(t)
    ENTITLEMENT_ADD($acct, modelID,$ DEPOSIT_AMOUNT(t))
    MARK_TICKET_SETTLED(t)
  POOL_CONSUME($modelID, paid$) ▷ consume only what was actually paid
  ROUND_INDEX_INCR($modelID$)

Algorithm 37 Validator.entitlementTransfer(*acct*, *modelID*, *amount*)

1: **function** ENTITLEMENT_TRANSFER($acct, modelID, amount$)
2: $e \leftarrow$ ENTITLEMENT_GET($acct, modelID$)
3: **require** $e \geq amount > 0$
4: ENTITLEMENT_SET($acct, modelID, e - amount$)
5: BALANCE_CREDIT($acct, amount$)

Algorithm 38 Validator.admitGlobalUpdate(*tx*)

function admitGlobalUpdate(tx)
  if not VERIFY_SIG($tx.lister,\ tx.sig$) **then**
    reject
  if not CHECK_ROUND($tx.modelID,\ tx.round$) **then**
    reject
  if not VALIDATE_CID($tx.globalCID$) **then**
    reject
  if $\|tx.scores\| \neq \|tx.included\|$ **then**
    reject
  if not IS_STRICTLY_UNIQUE(tx.included) **or not** IS_STRICTLY_UNIQUE(tx.refunds) **then**
    reject
  for all $t \in tx.included$ **do**
    if not ENSURE_REVEALED(t) **or not** IS_TICKET_OF($t, \langle tx.modelID, tx.round \rangle$) **then**
      reject
  for all $t \in tx.refunds$ **do**
    if not ENSURE_REVEALED(t) **or not** IS_TICKET_OF($t, \langle tx.modelID, tx.round \rangle$) **then**
      reject
  if not VALIDATE_CID($tx.globalCID$) **then**
    reject
  if $\|tx.scores\| \neq \|tx.included\|$ **then**
    reject
  for all $v \in tx.scores$ **do**
    if $v < 0$ **then**
      reject
  ENQUEUE_TX(mempool, tx)