SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER SCIENCE

FedEthML

BLOCKCHAIN AND DISTRIBUTED LEDGER TECHNOLOGIES

Professors:
Claudio Di Ciccio
Massimo La Morgia

Students:
Pietro D'Amico
Francesco Bonanni
Michele Granatiero
Seweryn Kaniowski

Academic Year 2023/2024

Contents

1 Preface **2**
1.1 Presentation of the Dapp . . . 2
1.2 Team members and main responsibilities . . . 2
1.3 Abstract . . . 2

2 Background **3**
2.1 Blockchain Technology . . . 3
2.1.1 Bitcoin . . . 3
2.1.2 Ethereum 1.0 . . . 3
2.1.3 Ethereum 2.0 . . . 4
2.2 Smart Contracts . . . 4
2.3 Types of Blockchain . . . 4
2.4 Application Domain . . . 5

3 Context **6**

4 Architecture **7**
4.1 Actors . . . 7
4.2 Procedure . . . 8

5 Implementation **15**
5.1 Smart Contract . . . 15
5.2 Frontend . . . 19

6 Known Issues and Limitations **21**

7 Conclusions and Future Remarks **21**

References **22**

1 Preface

1.1 Presentation of the Dapp

Our Dapp, FedEthML, shows how the blockchain technology can be adopted to address two of the main issues of Federated Learning. The first is centralization: the central server is replaced by the blockchain, and any adversarial behavior of the central node is therefore prevented. The second issue is represented by the lazy/adversarial behavior of training nodes. This problem is (partially) solved by introducing an incentive mechanism based on the proof-of-stake upon which the Ethereum environment relies.

1.2 Team members and main responsibilities

- Pietro D'Amico: Solidity development, oracle implementation, testing;
- Francesco Bonanni: Solidity development, Machine Learning implementation, testing;
- Michele Granatiero: Solidity development, Frontend logic and UI, Web3.js integration;
- Seweryn Kaniowski: Solidity development, gas optimization, MetaMask integration, UML diagrams;

1.3 Abstract

Federated Learning (FL) enables a pool of users to join forces (and data) in order to train a global model in a distributed fashion. Standard FL involves a central server as the coordinator of the training task: at each iteration, users train their local models and send updated weights to the central server, which will then aggregate those updates into the new global model. However, as shown in [1], a malicious server can launch inference attacks against selected users, gathering information about the datasets used during the training phase. Our proposal to prevent these threats is to adopt a decentralized approach toward Federated Learning, entrusting the blockchain with the role of coordinator of the training task. Another issue we will be facing when dealing with FL are the "lazy updates" sent from users that are submitting bogus updates or copies of other workers' updates. This will be mitigated with the introduction of an incentive mechanism, that aims at rewarding the workers based on the quality of the submitted updates. Finally, we will point out some issues with the proposed solutions together with some possible future developments and improvements.

2 Background

2.1 Blockchain Technology

Blockchain technology has emerged as a significant innovation in recent years, initially popularized by its use in Bitcoin. Its applications now span various sectors, including finance, supply chain, and more. In essence, a blockchain is an open and distributed ledger, consisting of a chain of blocks forming a connected, acyclic graph with a single end point. Each block can contain multiple transactions, representing the information within the ledger. Both the transactions and the blockchain's structure are immutable.

2.1.1 Bitcoin

Bitcoin operates on the foundational concept of blockchain technology, which serves as a secure and decentralized ledger. This technology was initially developed to manage digital currency transactions but has since expanded to facilitate secure, immutable transactions of various kinds.

Blocks in Bitcoin are interconnected using cryptographic hash functions. When a user initiates a transaction, it is sent to miners who validate and include it in a block. Miners compete to solve a computational puzzle by hashing the entire block until they find a nonce that results in a hash with a specific number of leading zeros. This process, known as Proof of Work, ensures that blocks are added to the blockchain at a controlled rate, preventing multiple blocks from being created simultaneously.

Each block in Bitcoin consists of:

- **Header Part**: Includes the hash of the previous block, a timestamp indicating when the block was mined (though this timestamp isn't precise due to the lack of a global clock in the decentralized network), a nonce (adjustable by miners to achieve the desired hash), and a Merkle root hash of the block's transactions.
- **Body of Block**: Contains transactions, each identified by a unique transaction ID. Each transaction specifies the sender, recipient, and amount transferred. It typically includes two outputs: the first for the recipient and the second returned to the sender if the amount sent exceeds what's needed. If the second output (change) is less than the difference between the input and the first output, the remaining part goes to the miner as a fee or tip for processing the transaction.

Bitcoin's PoW mechanism utilizes ASICs (Application-Specific Integrated Circuits) optimized for hashing functions to accelerate computation speed. To adjust the difficulty of PoW, the system recalibrates every 2016 blocks (approximately every two weeks), ensuring that the average time to mine a block remains around 10 minutes by adjusting the target hash value based on recent mining times.

In essence, Bitcoin's design ensures transaction security and decentralization through a robust cryptographic framework, allowing for a reliable and globally accessible financial system independent of centralized authorities.

2.1.2 Ethereum 1.0

Ethereum, introduced by Vitalik Buterin in 2015, expanded the capabilities of blockchain technology by enabling smart contracts and decentralized applications (DApps). Ethereum 1.0 (Proof of Work) operates on an Account Model, not Bitcoin's Transaction Model, exchanging values rather than individual transactions. Unlike Bitcoin, which primarily

serves as a digital currency, Ethereum provides a platform for executing code on the blockchain. This allows for the creation of programmable contracts that automatically execute terms encoded within them. It employs the Ethereum Virtual Machine (EVM) for standardized smart contract execution across nodes, each transaction incurring gas fees based on computational complexity. Gas limits prevent infinite loops, ensuring transaction execution within defined parameters. Miners validate and record transactions, earning rewards and fees. Ethereum's PoW difficulty increases every 100,000 blocks, challenging scalability due to resource-intensive mining. Rewards include system rewards, transaction fees, and occasional rewards for fork resolution. Nodes vary from full archival to simple, maintaining blockchain integrity through chain and state data. Unlike Bitcoin, Ethereum's Ethash algorithm deters ASIC use, promoting fair competition but requiring significant node memory.

2.1.3 Ethereum 2.0

In September 2022, Ethereum transitioned from a proof-of-work (PoW) to a proof-of-stake (PoS) consensus mechanism, significantly reducing its energy consumption. Gas fees are now computed based on gasLimit, baseFee, and priorityFee. baseFee adjusts dynamically to network demand, combating inflation by burning a portion of transaction fees. PoS in Ethereum 2.0 features Proposers who propose and validate blocks, and Validators who stake ETH to bet on future blocks. Validators earn rewards for accurate bets but face penalties for misbehavior, managed by the Gasper Algorithm. Ethereum plans to transition fully to PoS, merging PoW and PoS chains, aiming for a decentralized network with roll-up shards.

2.2 Smart Contracts

Smart contracts were introduced by Ethereum, functioning as decentralized applications executed through blockchain transactions. Their code is translated into EVM bytecode, with each instruction incurring specific fees. Costs include deployment and execution expenses, varying with code complexity. Smart contracts manage tokens, categorized as fungible (exchangeable), semi-fungible (like time-sensitive tickets), or non-fungible (unique, non-replicable). Tokens serve various functions: currency, commodity, utility, or security. When deployed, contracts use compiled code by default, with the option to include the source code. Each contract has a nonce indicating its instances, while Externally Owned Accounts (EOAs) use nonce to track transactions. Smart contracts, unsigned, can be invoked locally for simulation or globally through transactions in the blockchain, incurring costs without modifications.

2.3 Types of Blockchain

Blockchain technology can be categorized into several types, each serving different purposes and use cases:

- **Public Blockchain:** Open to anyone, allowing any participant to join the network, validate transactions, and execute smart contracts. Bitcoin and Ethereum are prime examples.
- **Private Blockchain:** Restricted access, usually employed by enterprises for managing sensitive data. Only selected participants can join and validate transactions.

Further distinction can be made based on the permissions to participate in consensus processes:

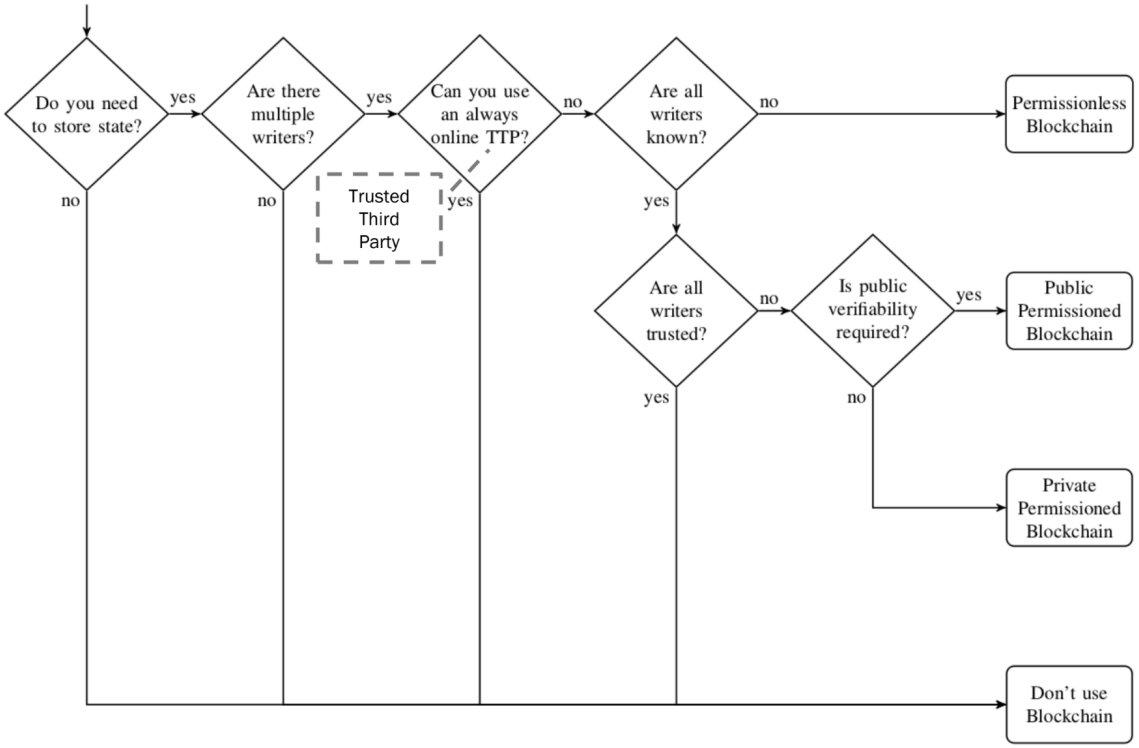
- **Permissionless Blockchains:** Open networks where anyone can participate in the consensus process, ensuring full decentralization and transparency.
- **Permissioned Blockchains:** Closed networks with designated participants, providing controlled transparency and security tailored to the needs of participating organizations.

2.4 Application Domain

Blockchain technology has vast applications beyond cryptocurrency, impacting areas such as smart contracts, logistics, and supply chain management. Smart contracts automate and enforce contractual agreements without intermediaries, enhancing efficiency and trust. In logistics, blockchain ensures transparency and security throughout the supply chain, preventing fraud and enabling traceability.

Despite its potential, blockchain technology faces technical and regulatory challenges that need to be addressed for widespread adoption. Its benefits in security, transparency, and automation continue to drive interest from various industries and organizations.

3 Context



The Dapp aims to enable distributed and decentralized training of neural networks while defending users from a class of inference attacks called "model inconsistency attacks". These attacks are possible in a centralized scenario since they consist in the adversary crafting a malicious update to be sent to the target user. The user then train its model using the forgery and communicate its updates back to the adversary. By analyzing the updates received, the adversary is now able to infer some information about the dataset of the target user. As the name implies, these attacks are possible as long as the target receives a particular version of the parameters to be trained. With the use of the blockchain - and through consensus - it is impossible to trick a user into using a bogus set of parameters.

We opt for a permissionless blockchain to avoid having a TTP - a potential adversary - while allowing anyone to participate in training tasks. Indeed, it suffices to have a wallet to issue a training task or participate in a task issued by someone else.

4 Architecture

The protocol involves several actors, which play specific roles in its operation, as shown in Figure 1.

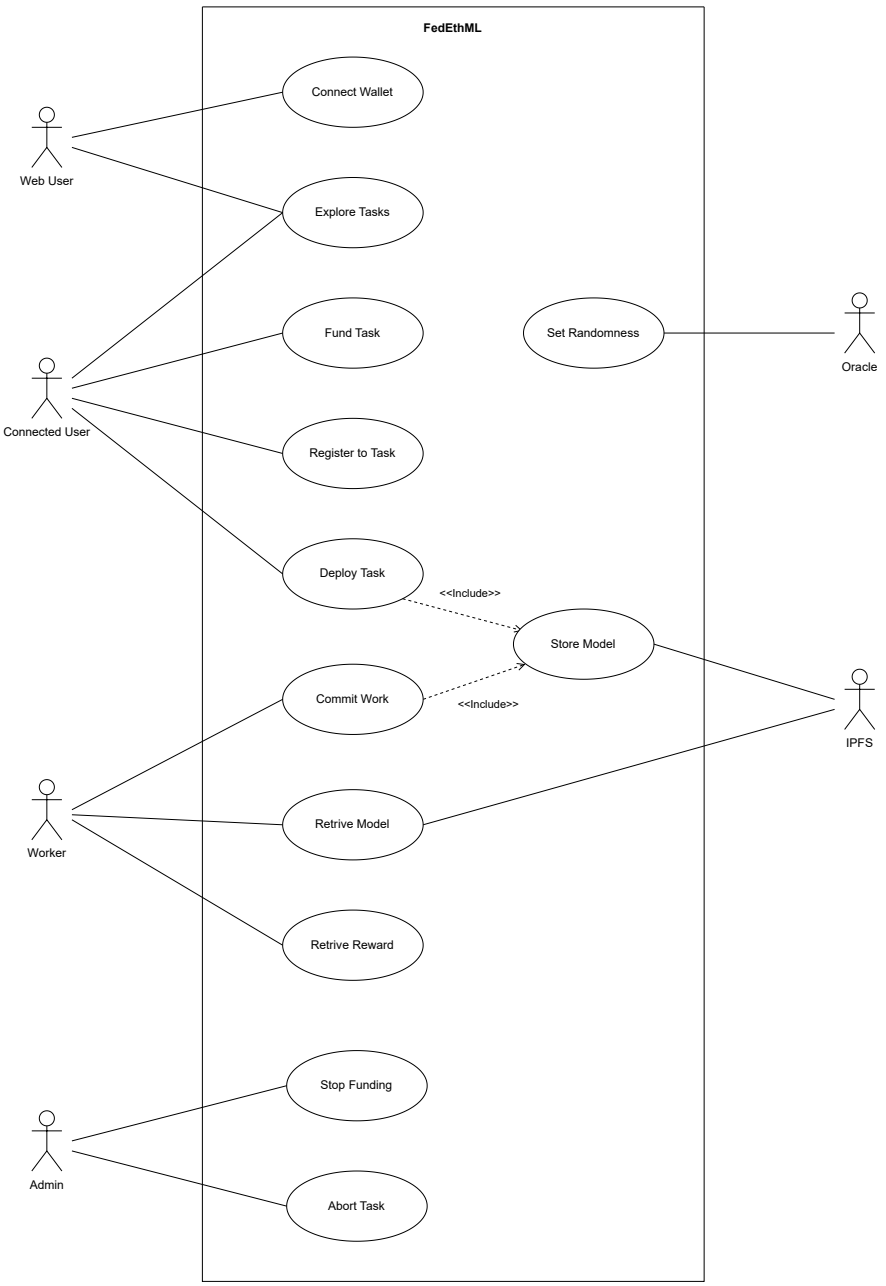


Figure 1: UML Use Case diagram

4.1 Actors

In particular, we have the following list of actors:

- **Admin**. Publishes a new Federated Machine Learning task. The protocol is

task-agnostic but requires participants to understand the task, which is publicly available in the SC and on the IPFS.

- **Funders**. Provide funds for the tasks using the blockchain's native cryptocurrency or tokens.
- **Workers**. Train the model and earn rewards for their efforts.
- **Oracles**. Fetch external data necessary for the smart contract tasks, like randomness.
- **Decentralized Storage**. Store data across multiple locations in an overlay network using the IPFS. Each resource in IPFS is identified by a unique content identifier.
- **FML Smart Contract**. Manages and coordinates the FML tasks and store necessary data.

4.2 Procedure

1. **Task Creation**: Admin creates the task with its details (e.g., input data format, loss function, learning rate, starting model) and setting parameters (e.g., registration fee, number of workers per round, number of rounds, task name and description).
2. **Funding**: Stakeholders fund the task, increasing the bounty for workers. No restrictions on who can fund; future workers and the admin can also be funders.
3. **Registration**: Interested workers register via the SC before the task starts, paying an entrance fee decided ahead by the admin, which adds to the bounty and deters malicious behavior.
4. **Task Start**: The task begins when the required number of registered workers is reached and the funding is stopped by the admin, after that the oracle is triggered by an event for setting the randomness. For each communication round, a subset of workers is randomly selected using the oracle. Each worker joins only one round.
5. **Worker Job**: Every worker follows the procedure shown by the sequence diagram in Figure 10. The workers perform the following steps:
   (a) **Retrieving Models**: Download previous round's models from decentralized storage.
   (b) **Local Evaluation**: Evaluate models using their local dataset, rank them, and vote for the top ones.
   (c) **Local Training**: Train the averaged model using their local dataset.
   (d) **Publishing Model**: Upload the updated model to the decentralized storage.
   (e) **Commitment**: Submit votes and model CID to the SC.
6. **Communication Rounds**: After each round the process repeats. Votes from each round determine worker rankings and rewards.
7. **Reward Distribution**: Workers' rewards are based on the votes received for their models. The SC tallies votes and assigns rewards accordingly.

8. **Last Round**: Workers only provide votes for the previous round's models in a two-phase commitment scheme to prevent laziness.

9. **Task Closure**: Upon completing all communication rounds, the task concludes. The final output includes all models from the second-last round.

Figure 2: High Level Architecture

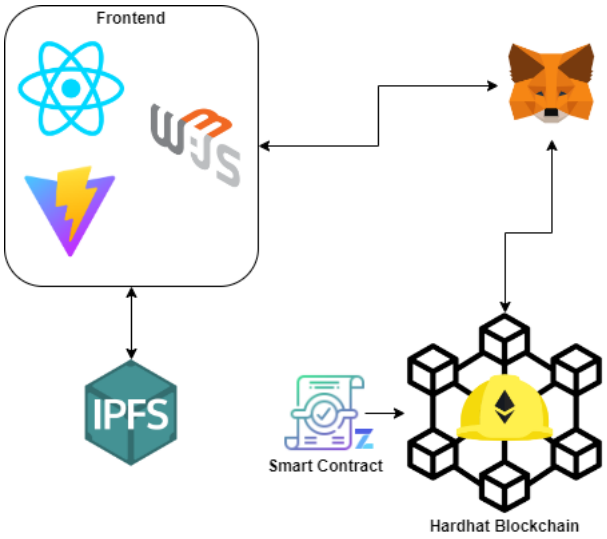
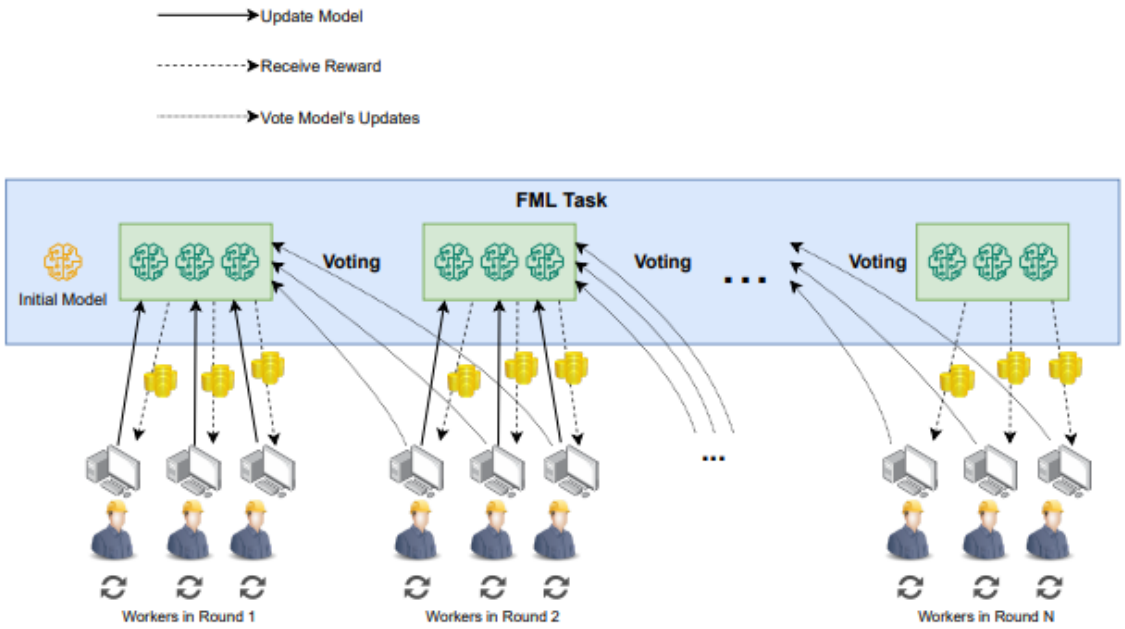


Figure 3: Task Workflow



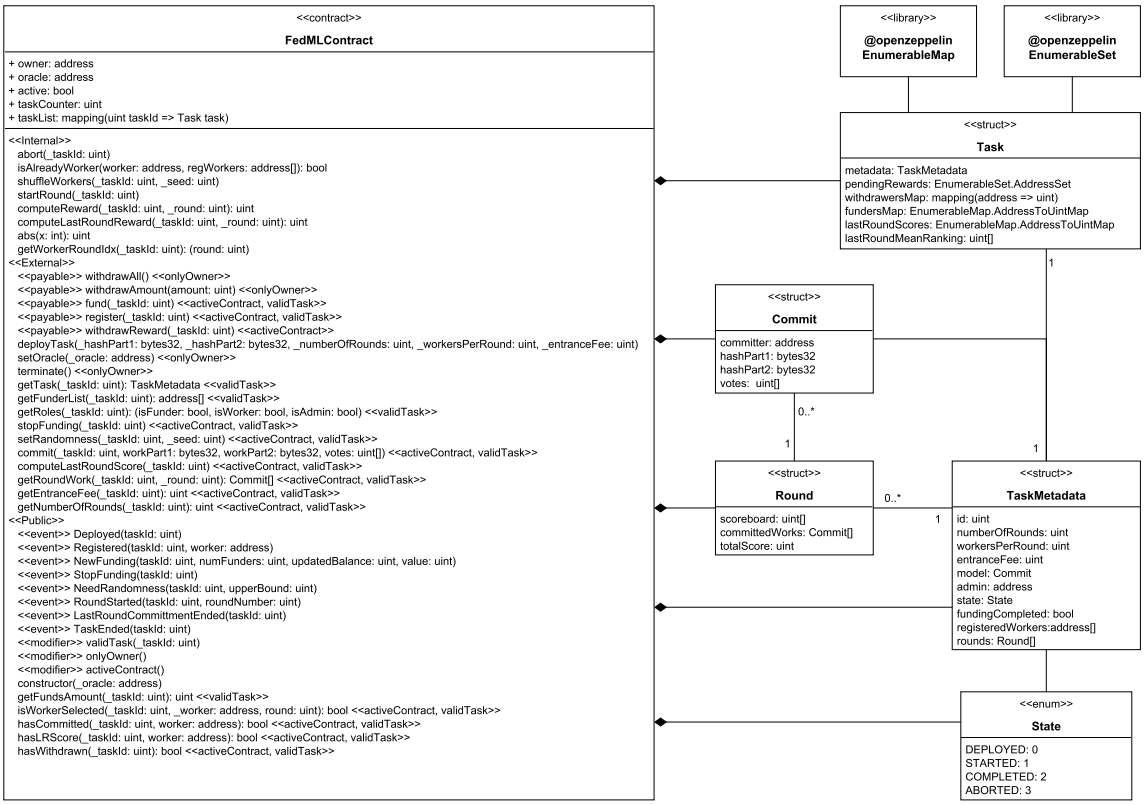


Figure 4: class diagram of the smart contract

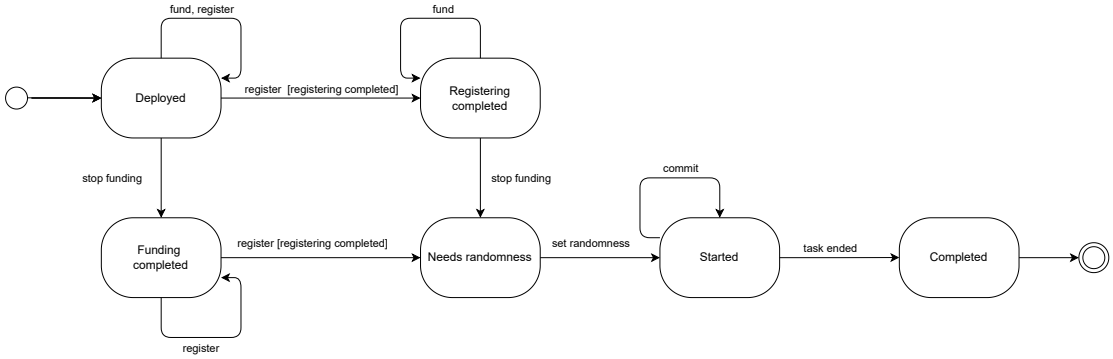


Figure 5: state machine diagram of the lifecycle of a task

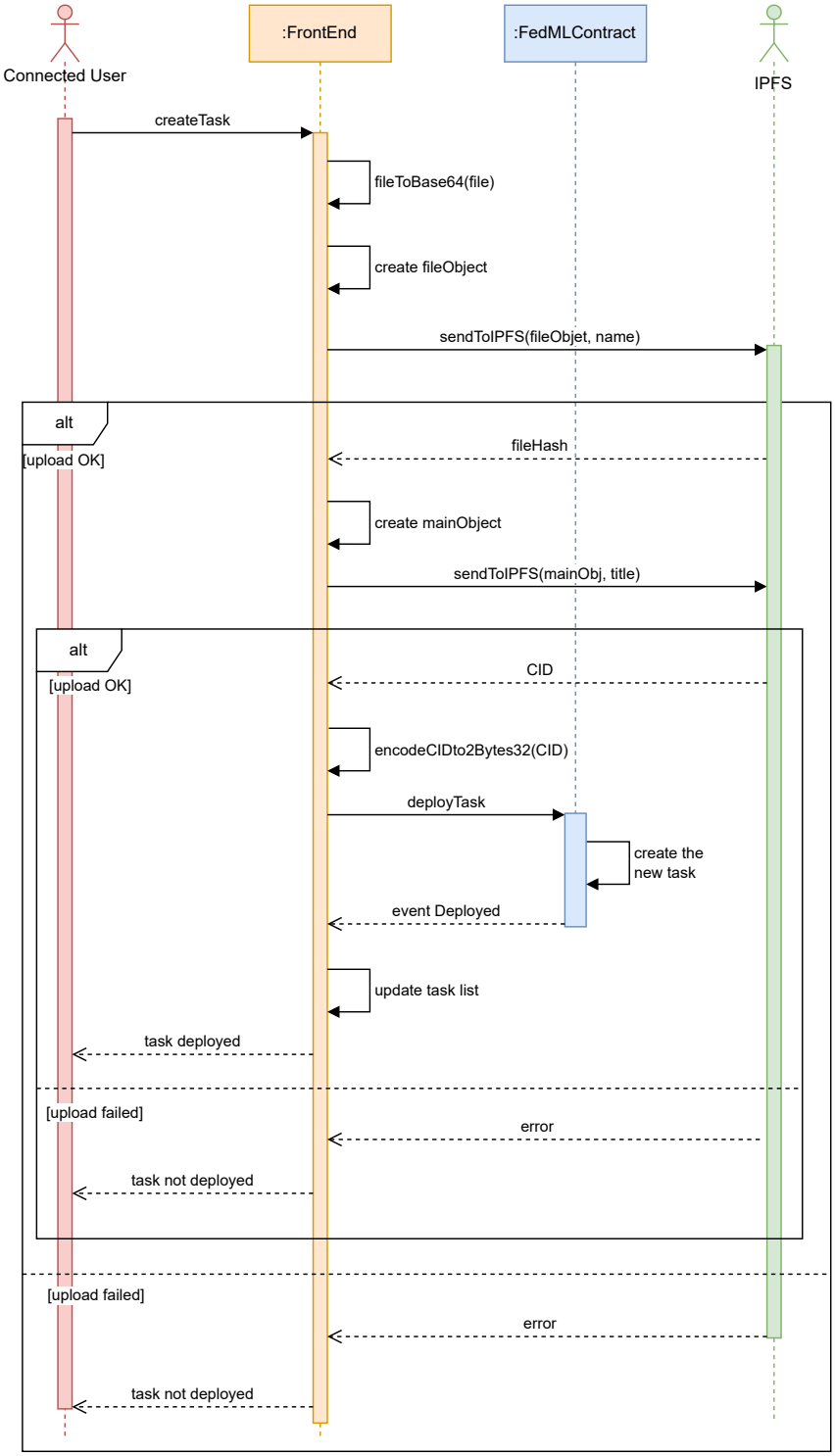


Figure 6: sequence diagram of the creation of a new task

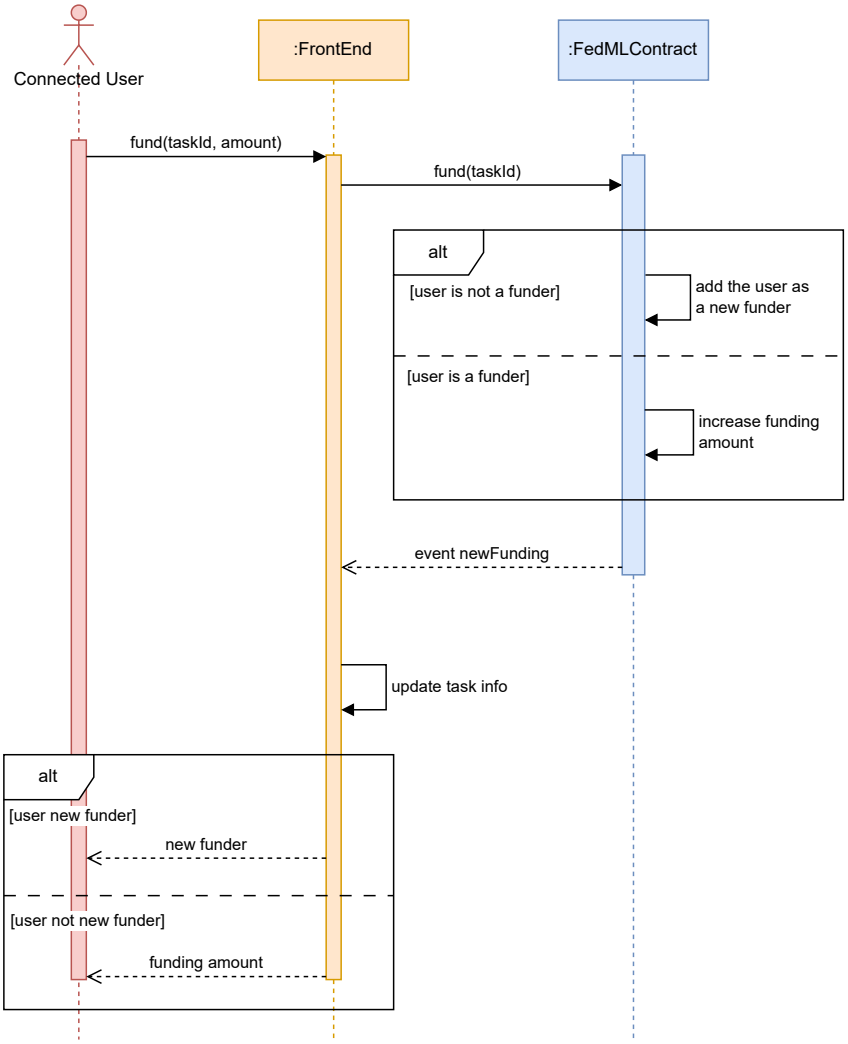


Figure 7: sequence diagram of the funding of a task

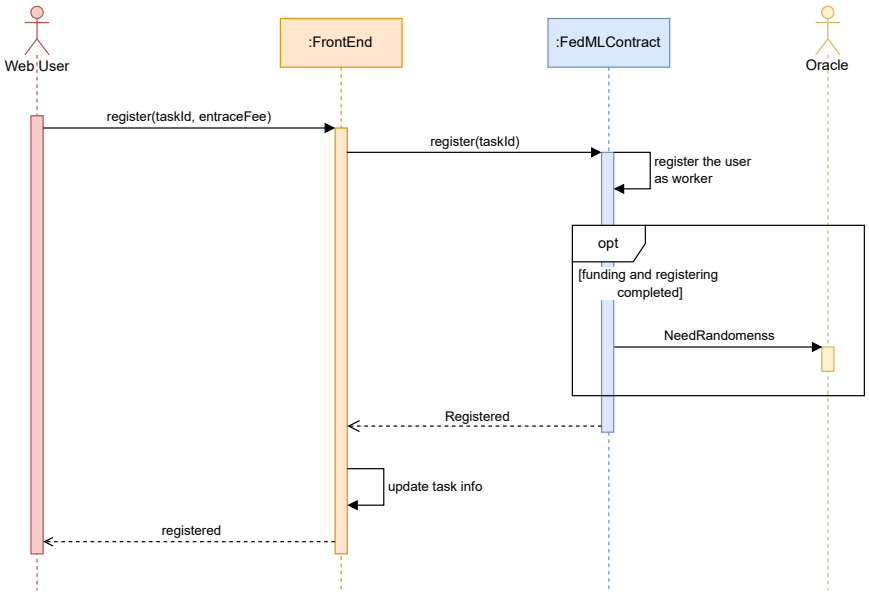


Figure 8: sequence diagram of the registration of a user as a worker to a task

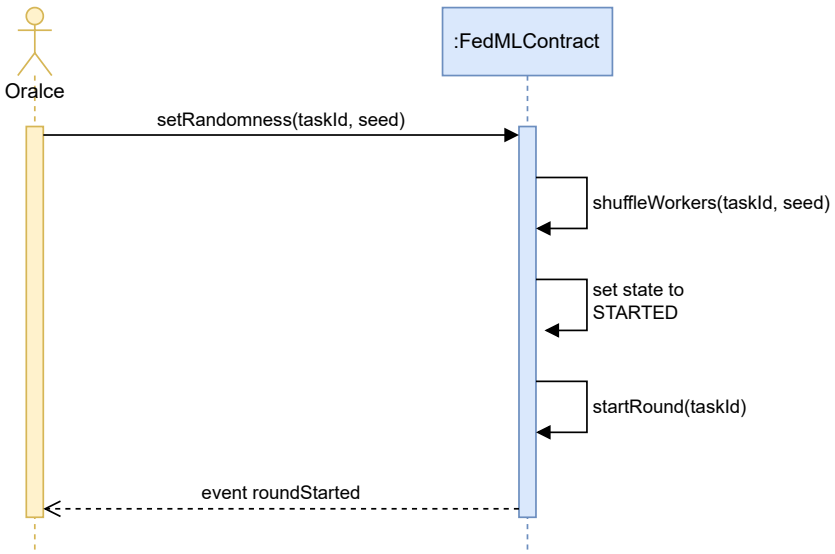


Figure 9: sequence diagram for the setting of the randomness of a task by the oracle

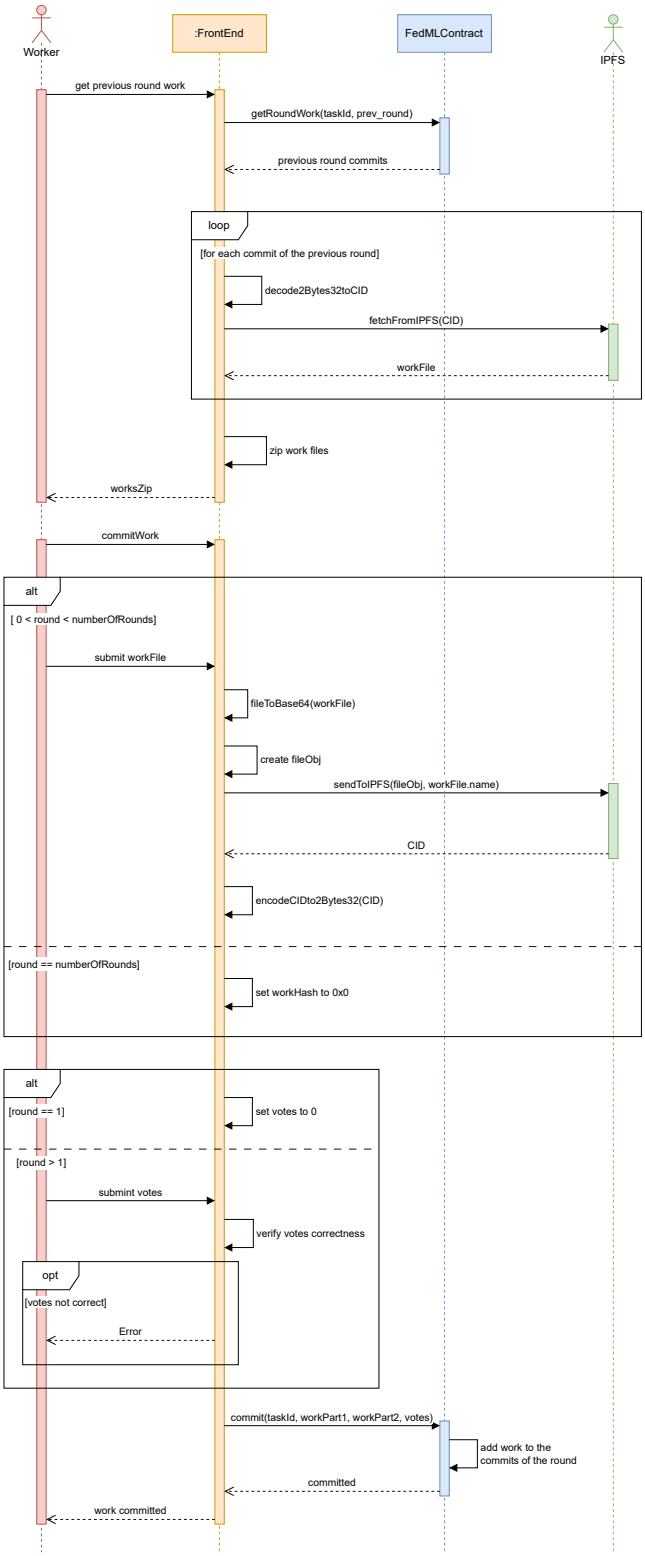


Figure 10: sequence diagram of the worker's job (retrieve previous round work and commit)

5 Implementation

5.1 Smart Contract

The smart contract of our decentralized application was written using the Solidity programming language and the OpenZeppelin library, to make use of the EnumerableMap and EnumerableSet data structures.

The deployTask function, which code is reported in Listing 1, creates a new task accordingly to the parameters passed by the user that is deploying the task. The two parameters `_hashPart1` and `_hashPart2` represent the CID of the initial model, that was uploaded by the admin on the IPFS. The original CID hash string is transformed into two `bytes32` objects in order to avoid the use of strings, which are very costly data structures.

```
function deployTask(
    bytes32 _hashPart1,
    bytes32 _hashPart2,
    uint _numberOfRounds,
    uint _workersPerRound,
    uint _entranceFee
    ) external {
    require(_workersPerRound > 1);
    require(_numberOfRounds > 1);
    Task storage task = taskList[taskCounter];
    TaskMetadata storage taskMetadata = task.metadata;
    taskMetadata.id = taskCounter++;
    taskMetadata.admin = msg.sender;
    taskMetadata.model = Commit(msg.sender , _hashPart1, _hashPart2, new uint[](0));
    taskMetadata.numberOfRounds = _numberOfRounds;
    taskMetadata.workersPerRound = _workersPerRound;
    taskMetadata.fundingCompleted;
    taskMetadata.state = State.DEPLOYED;
    taskMetadata.entranceFee = _entranceFee;
    task.lastRoundMeanRanking = new uint[](_workersPerRound);
    emit Deployed(taskMetadata.id);
}
```

Listing 1: Task deployment

The Listing 2 shows the code that allows a user to register as a worker for a task, the registration requires the payment of a registration fee decided in advance by the admin of the task. Once the local training is finished then the worker submits his model and his votes by means of the function reported in the Listing 3.

A different behavior it triggered if the worker belongs to the last round. Every round, except the last one, terminates when all the related workers have committed their work, i.e. they have sent the votes for the models of the previous round and updated the model. Their votes are used for the computation of the score of the previous round, which will be used for assigning the rewards. Clearly, the reward for the workers of the last round cannot be computed in this way. That's the reason why the last round follows a two-phase commitment scheme: in the first phase the workers will only evaluate the models of the second-last round and send their votes. Once all votes are gathered the last round scores are computed based on the distance between the votes of the single workers and the mean of the votes sent by all the workers. Listing 4 shows how the scores are computed during the second phase of the last round. Once a worker's job has been evaluated, the worker can retrieve the reward through the function `withdrawReward` in Listing 5. Depending on whether he belongs to the last round or not, the corresponding reward will be computed using the code in Listing 6 or in Listing 7.

```
function register(uint _taskId) payable activeContract validTask(_taskId) external {
    TaskMetadata storage taskMetadata = taskList[_taskId].metadata;
    //check if the entrance fee is enough
    require(msg.value == taskMetadata.entranceFee, "Insufficient funds!");
    uint numRegWorkers = taskMetadata.registeredWorkers.length;
    uint workersPerRound = taskMetadata.workersPerRound;
    uint numberOfRounds = taskMetadata.numberOfRounds;
    uint workersRequired = workersPerRound*numberOfRounds;
    require(numRegWorkers < workersRequired, "Impossible to register!");
    //check if address is already registered
    require(!isAlreadyWorker(msg.sender, taskMetadata.registeredWorkers));
    console.log("Registered worker %d",numRegWorkers);
    taskMetadata.registeredWorkers.push();
    taskMetadata.registeredWorkers[numRegWorkers] = msg.sender;
    if (numRegWorkers+1 == workersRequired) {
        console.log("All workers registered!");
        if (taskMetadata.fundingCompleted) {
            console.log("Requesting randomness...");
            emit NeedRandomness(_taskId, workersPerRound*10);
        }
    }
    emit Registered(_taskId, msg.sender);
}
```

Listing 2: register function

```
///@notice Allow a worker to commit his work
///@param _taskId The numerical identifier of the task
///@param workPart1 The first half of the hashed CID of the work
///@param workPart2 The sencond half of the hashed CID of the work
///@param votes The votes given to the works of the previous round
function commit(uint _taskId, bytes32 workPart1, bytes32 workPart2,
uint[] calldata votes) activeContract validTask(_taskId) external {

    Task storage task = taskList[_taskId];
    TaskMetadata storage taskMetadata = task.metadata;
    //task started and not completed yet
    require(taskMetadata.state == State.STARTED, "Task not started!");
    //the worker is selected for the current round, could prevent registering!
    require(isWorkerSelected(_taskId, msg.sender, taskMetadata.rounds.length-1),
        "You are not selected!");
    //the worker has not committed yet
    require(!hasCommitted(_taskId, msg.sender), "You have already committed!");
    uint currentRound = taskMetadata.rounds.length-1;

    //set the sender eligible for rewards
    EnumerableSet.add(task.pendingRewards, msg.sender);

    if (currentRound > 0) {
        require(votes.length == taskMetadata.workersPerRound);
        //assign votes
        for (uint i = 0; i < votes.length; i++) {
            taskMetadata.rounds[currentRound-1].scoreboard[i] += votes[i];
            taskMetadata.rounds[currentRound-1].totalScore += votes[i];
            console.log(taskMetadata.rounds[currentRound-1].totalScore);
        }
    }

    Commit memory work;
    work.committer = msg.sender;
    work.votes = votes;

    //if the current round is the last round
    if (currentRound == taskMetadata.numberOfRounds-1) {
        taskMetadata.rounds[currentRound].committedWorks.push(work);
        console.log("Updating the last round mean ranking...");
        //update the last round mean ranking as a running average
        for (uint i = 0; i < votes.length; i++) {
            console.log("Vote: %s", votes[i]);
            console.log("Last round mean ranking: %s", task.lastRoundMeanRanking[i]);
            task.lastRoundMeanRanking[i] = uint(int256(task.lastRoundMeanRanking[i])+
            int256(int256(votes[i]) - int256(task.lastRoundMeanRanking[i]))
            /int256(taskMetadata.rounds[currentRound].committedWorks.length));
        }

        /*If it was the last commitment for the round emit the event
        LastRoundCommittmentEnded*/
```

```
        if (taskMetadata.rounds[currentRound].committedWorks.length == taskMetadata
        .workersPerRound) {
            console.log("All workers submitted their work for round %s (last round)",
                currentRound);
            emit LastRoundCommittmentEnded(_taskId);
        }
    } else { //if not last round
        taskMetadata.rounds[currentRound].scoreboard.push();
        work.hashPart1 = workPart1;
        work.hashPart2 = workPart2;
        taskMetadata.rounds[currentRound].committedWorks.push(work);

        //If it was the last commitment for the current round, end the round

        if (taskMetadata.rounds[currentRound].committedWorks.length == taskMetadata
        .workersPerRound) {
            console.log("All workers submitted their work for round %s",
            currentRound);
            startRound(_taskId);
            }
        }

    }
```

Listing 3: commit function

```
function computeLastRoundScore(uint _taskId) activeContract validTask(_taskId)
external {
    Task storage task = taskList[_taskId];
    /*require the worker has not already computed the score + other things inside
    hasLRScore */
    require(!hasLRScore(_taskId, msg.sender), "You have already computed the score!");
    /*compute the sender's score as the inverse of the sum of the distances between
    the sender's votes and the mean ranking of the last round */

    //retrieve the sender's votes
    uint[] memory senderVotes;
    for (uint i = 0; i < task.metadata.workersPerRound; i++) {
        if (task.metadata.rounds[task.metadata.numberOfRounds-1].committedWorks[i]
        .committer == msg.sender) {
            senderVotes = task.metadata.rounds[task.metadata.numberOfRounds-1]
            .committedWorks[i].votes;
            break;
        }
    }

    /*sum of the distances between the sender's votes and the mean ranking of the last
    round */
    uint score;
    for (uint i = 0; i < senderVotes.length; i++) {
        score += abs(int(senderVotes[i]) - int(task.lastRoundMeanRanking[i]));
    }

    //save the sender score in the last round ranking
    task.lastRoundScores.set(msg.sender, 100000/(1 + score));

    //update the last round total score
    task.metadata.rounds[task.metadata.numberOfRounds-1].totalScore += task
    .lastRoundScores.get(msg.sender);

    //check all workers have computed their score in last round
    if (task.lastRoundScores.length() == task.metadata.workersPerRound) {
        console.log("Finished calculating last round scores, ending task...");
        task.metadata.state = State.COMPLETED;
        emit TaskEnded(_taskId);
    }

}
```

Listing 4: Last round score

```
function withdrawReward(uint _taskId) activeContract external payable {
    Task storage task  = taskList[_taskId];
    require(!hasWithdrawn(_taskId), "You have already withdrawn the reward");
    EnumerableSet.remove(task.pendingRewards, msg.sender);
    uint roundIdx = getWorkerRoundIdx(_taskId);
    uint reward;
    if (roundIdx == task.metadata.numberOfRounds-1) {
        reward = computeLastRoundReward(_taskId, roundIdx);
    } else {
        reward = computeReward(_taskId, roundIdx);
    }
    //add to withdrawers map
    console.log("reward: %s", reward);
    task.withdrawersMap[msg.sender] = reward;

    payable(msg.sender).transfer(reward);
}
```

Listing 5: Withdraw reward

```
///@notice Computes the reward for the worker
///@param _taskId The numerical identifier of the task
//@param _round the round number
function computeReward(uint _taskId, uint _round) internal view returns (uint) {

    TaskMetadata storage taskMetadata  = taskList[_taskId].metadata;
    Round storage round = taskMetadata.rounds[_round];

    uint fundedAmount = getFundsAmount(_taskId);
    uint roundBounty = fundedAmount / taskMetadata.numberOfRounds;
    uint totalScore = round.totalScore;

    console.log("Round bounty is: %s", roundBounty);
    console.log("Total score for round %s is: %s", _round, totalScore);

    uint workerIndex;
    for (uint i = 0; i < taskMetadata.workersPerRound; i++) {
        if (round.committedWorks[i].committer == msg.sender) {
            workerIndex = i;
            break;
        }
    }
    uint coefficient = (round.scoreboard[workerIndex]) * 100000 / totalScore;
    //log the coegfficient
    console.log("Coefficient for worker %s at round %s is: %s", msg.sender, _round,
    coefficient);
    uint reward = taskMetadata.entranceFee + (roundBounty * coefficient)/100000;

    console.log("Worker %s at round %s got reward %s", msg.sender, _round, reward);

    return reward;
}
```

Listing 6: Compute reward

```
function computeLastRoundReward(uint _taskId, uint _round) internal view returns
(uint) {
    Task storage task  = taskList[_taskId];
    Round storage round = task.metadata.rounds[_round];

    uint fundedAmount = getFundsAmount(_taskId);
    uint roundBounty = fundedAmount / task.metadata.numberOfRounds;

    uint workerScore = task.lastRoundScores.get(msg.sender);
    uint coefficient = workerScore * 100000 / round.totalScore;
    //log the coefficient
    console.log("Coefficient for worker %s at round %s is: %s", msg.sender, _round,
    coefficient);
    uint reward = task.metadata.entranceFee + (roundBounty * coefficient)/100000;

    console.log("Worker %s at round %s got reward %s", msg.sender, _round, reward);

    return reward;
}
```

Listing 7: Compute last round reward

5.2 Frontend

The frontend of our Web3 application is built using React.js, which facilitates the creation of dynamic and responsive user interfaces. To improve the visual appeal and consistency of the UI, we utilized Tailwind in combination with the Shadcn library, which offers a set of reusable, pre-built components to maintain design uniformity. We integrated MetaMask for seamless interaction with the Ethereum blockchain, allowing users to securely access their digital wallets. Additionally, we used Web3.js to bridge the front-end logic with the blockchain environment. Indeed, by means of a provider in this library, the frontend application is able to communicate with Hardhat, which was leveraged as a testing blockchain node to deploy and test the smart contract.

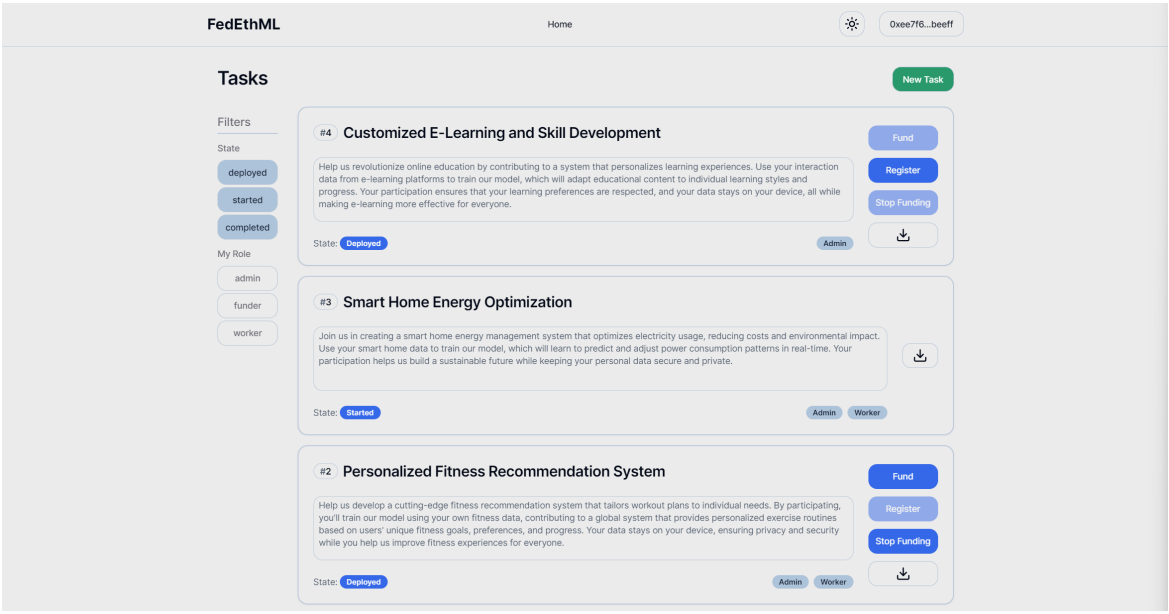


Figure 11: Homepage

On the left-hand side of the homepage, displayed in Figure 11, there are some filters that allow the user to refine the task list accordingly to the state of the task and the roles of the user. Tasks are displayed at the center of the page with their title, a brief description of the task and badges indicating their state and the roles played by the user. Users can interact with the tasks using the buttons on the right-hand side, or create a new task using the green button on the top-right, which will trigger a modal view with a form to be filled by the user.

In this form, shown in Figure 12, the user should carefully insert the details about the new task that he wants to create. Once the task is submitted, the uploaded file, which should include useful resources for the workers, is sent to the IPFS (we are using an IPFS gateway to allow the transfer through http), in response, the CID that is received back is splitted in two bytes32 formatted strings and passed with other parameters to web3.js, ready for the smart contract transaction.
The Figure 13 shows the task page in which users can find much more details and informations about the selected task. This page is used to interact with the task and monitor the evolution of its state. Web3js constantly listens to specific blockchain events of the smart contract to update the UI details accordingly to the upcoming changes.

Figure 12: New Task Creation

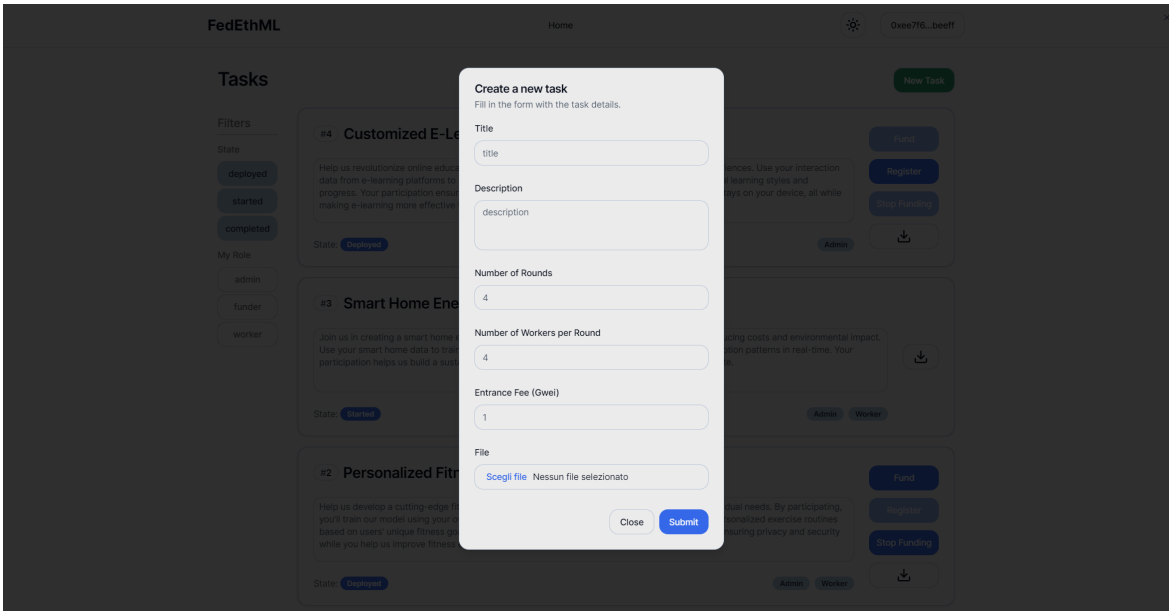
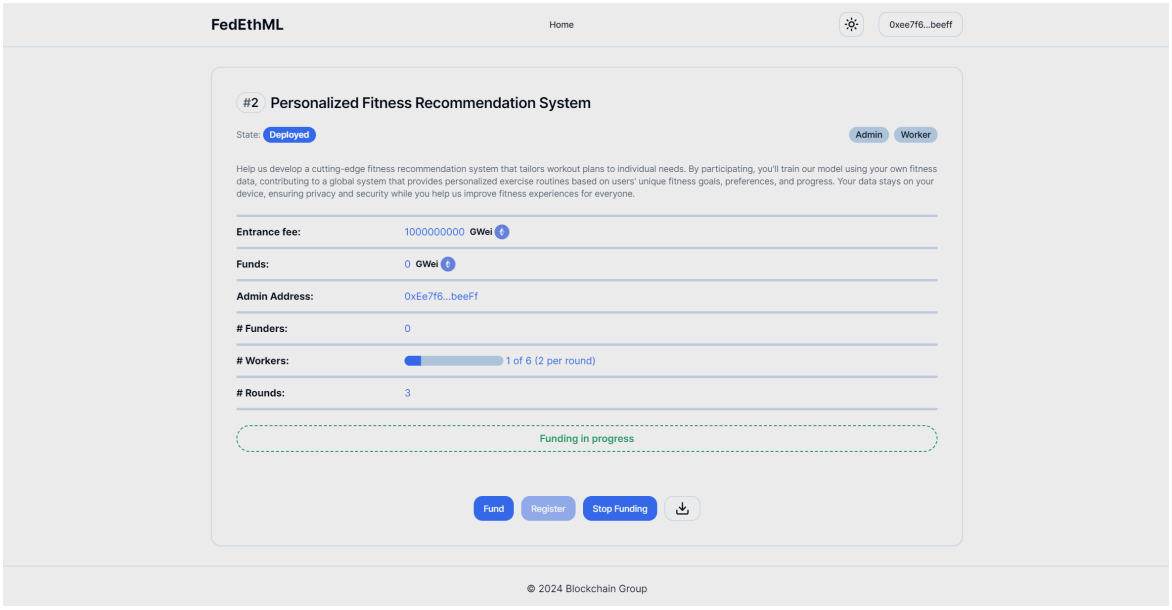


Figure 13: Task Page - Deployed Task



6 Known Issues and Limitations

The present project has only addressed one of the known limitations of Federated Learning (FL), the problem of centralization. Therefore, reverse engineering of model updates and lazy workers are still possible. In addition to these inherent FL problems, our work suffers from several minor implementation issues. First, if the task doesn't reach the required number of workers or the collected funds are insufficient, the task cannot start and remains pending until these conditions are met or the task is aborted. Second, if any worker becomes inactive during the remaining rounds, the task may last indefinitely long and virtually never end. Third, users registering for a task have no economic guarantees, as the transaction fees may exceed the reward. Finally, this project doesn't scale with the size of the models, being limited to tasks involving ML architectures of limited dimensions.

7 Conclusions and Future Remarks

Future work will address the aforementioned issues and limitations. To cope with reverse engineering of model updates and lazy workers, we will adopt the approach enforced in [2], implementing Differential Privacy and Zero-Knowledge Proofs. To solve the issue of pending tasks, we will introduce a deadline by which the conditions to start the task must be satisfied, along with an oracle responsible for aborting the task when the deadline has expired. Additionally, we will introduce a minimum funding threshold, calculated based on the number of workers per round, to guarantee a fair reward to each user participating in the task and to prevent the deployment of tasks that would ultimately be aborted due to a lack of participants. To solve the issue of inactive workers, we will introduce a timeout for the submission of the work. If no commit is performed by the timeout, the inactive worker will be removed, and new registration and funding phases will be started to fill the vacant position. Given the complexity of the project, we have discussed just the most important issues and the corresponding solutions, but the project still has a lot of room for improvement. For example, we could use Chainlink's decentralized oracle infrastructure, with work paid in Link tokens funded by the funders, paying an entrance fee based on the number of rounds and workers. This would add to the bounty and deter malicious behavior. Other useful refinements include the possibility to resume an aborted task by its admin and the introduction of a refund mechanism after the abortion of a task.

References

[1] Pasquini et al. Eluding secure aggregation in federated learning via model inconsistency. *ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[2] Gabriele Quattrocchi. zk-fl: Zero-knowledge proofs for federated learning. `https://quattrocchi.faculty.polimi.it/papers/zk-fl.pdf`, 2023.

[3] Author. Thesis title. `https://www.politesi.polimi.it/bitstream/10589/195173/2/Thesis.pdf`, 2023.

[4] Edoardo Venir. Tesi ruzza venir. `https://github.com/EdoardoV97/Home_Tesi_Ruzza_Venir_2022`, 2022.

[5] Caner Korkmaz, Halil Eralp Kocas, Ahmet Uysal, Ahmed Masry, Oznur Ozkasap, and Baris Akgün. Chain fl: Decentralized federated machine learning via blockchain. In *2020 Second International Conference on Blockchain Computing and Applications (BCCA)*, 2020.

[6] Robert Amler. Blockchain machine learning demo. `https://github.com/robamler/blockchain-machine-learning-demo`, 2023.

[7] Author. Decentralized distributed federated learning. *arXiv preprint*, 2022.

[8] Huggingface. *Model Parallelism with Transformers*, 2023.