// Smart Contract for Decentralized Deepfake Task Management

pragma solidity ^0.8.0;

contract DecentralizedDeepfakeTaskManager {

address public owner;

mapping(address => bool) public authorizedNodes;

event TaskExecuted(address indexed executor, bytes taskData);

modifier onlyOwner() {

require(msg.sender == owner, "Only the owner can execute this");

\_;

}

modifier onlyAuthorizedNode() {

require(authorizedNodes[msg.sender], "Only authorized nodes can execute tasks");

\_;

}

constructor() {

owner = msg.sender;

}

function authorizeNode(address node) external onlyOwner {

authorizedNodes[node] = true;

}

function revokeNodeAuthorization(address node) external onlyOwner {

authorizedNodes[node] = false;

}

function executeTask(bytes calldata taskData) external onlyAuthorizedNode {

// Implement task execution logic using taskData

// This function can interact with edge computing nodes

// and manage the decentralized execution of Deepfake tasks.

// ...

// Emit an event to log the task execution

emit TaskExecuted(msg.sender, taskData);

}

// Smart Contract for Decentralized Deepfake Task Management with Federated Learning

pragma solidity ^0.8.0;

contract DecentralizedFederatedLearning {

address public owner;

mapping(address => bool) public authorizedNodes;

mapping(address => bool) public taskParticipants;

struct Task {

string description;

bool completed;

}

Task public currentTask;

mapping(address => bool) public taskCompletedByNode;

event TaskPublished(string description);

event TaskCompleted(address indexed node);

event ModelFeedback(string feedback);

modifier onlyOwner() {

require(msg.sender == owner, "Only the owner can execute this");

\_;

}

modifier onlyAuthorizedNode() {

require(authorizedNodes[msg.sender], "Only authorized nodes can participate");

\_;

}

modifier onlyTaskParticipant() {

require(taskParticipants[msg.sender], "Only task participants can complete the task");

\_;

}

constructor() {

owner = msg.sender;

}

function publishTask(string memory description) external onlyOwner {

require(bytes(description).length > 0, "Task description should not be empty");

currentTask = Task({

description: description,

completed: false

});

// Emit an event to signal the task publication

emit TaskPublished(description);

}

function authorizeNode(address node) external onlyOwner {

authorizedNodes[node] = true;

}

function participateInTask() external onlyAuthorizedNode {

taskParticipants[msg.sender] = true;

}

function completeTask() external onlyTaskParticipant {

require(!currentTask.completed, "Task already completed");

// Mark the task as completed by the calling node

taskCompletedByNode[msg.sender] = true;

// Check if all authorized nodes have completed the task

bool allNodesCompleted = true;

for (uint256 i = 0; i < authorizedNodes.length; i++) {

if (!taskCompletedByNode[authorizedNodes[i]]) {

allNodesCompleted = false;

break;

}

}

if (allNodesCompleted) {

// All nodes have completed the task, initiate federated learning

trainFederatedModel();

}

// Emit an event to signal the task completion

emit TaskCompleted(msg.sender);

}

function trainFederatedModel() internal {

// Implement federated learning logic

// This function should aggregate model updates from participating nodes

// and update the global federated model.

// ...

// Once the model is trained, provide feedback to the task publisher

emit ModelFeedback("Federated model training completed. Feedback provided.");

}

}