blockchain technology. ElizaOS stands out for its ease of use and accessibility, making it a popular choice for developers looking to create AI agents quickly. The choice between these platforms depends on specific needs, such as infrastructure robustness, flexibility, and ease of use.

19.3.1.4 Internet Computer Protocol

Internet Computer Protocol (ICP) is bringing AI computation directly to the blockchain, creating a new paradigm for verifiable AI execution. https://internetcomputer.org/what-is-the-ic

ICP network is orchestrated by permissionless decentralized governance and is hosted on sovereign hardware devices run by independent parties.

19.3.1.5 Akash Network

Akash Network is revolutionizing how we deploy AI workloads by creating a decentralized marketplace for computing resources. https://docs.akash.network/

Open network that facilitates the secure and efficient buying and selling of computing resources. It's like an "Airbnb/Uber for AI computing power".

Each of these protocols represents a different approach to combining AI with blockchain technology, creating new possibilities for:

- Decentralized AI development and deployment
- Fair and transparent AI service monetization
- Privacy-preserving AI training and inference
- Autonomous AI systems with economic agency
- Verifiable AI computation on the blockchain

19.4 ORA Protocol

As an example of the applications of verifiable AI by leveraging Web3 technologies, let's explore the **ORA Protocol** (https://ora.io/) for some practical examples of combinations of these technologies.

- ORA Protocol is a decentralized protocol providing chain-agnostic infrastructure that bridges the gap between AI and blockchain. https://docs.ora.io/doc/introduction/about-ora
- The base of the ORA Protocol is the use of Onchain AI Oracles to safely execute AI inference off-chain while providing verifiable results on-chain, https://www.ora.io/app/opml/.

19.4.1 Optimistic ML

- The opML paper: https://arxiv.org/abs/2401.17555. OpML (*Optimistic Machine Learning on Blockchain)* combines optimistic rollups with machine learning to enhance blockchain scalability and efficiency. It allows off-chain execution of machine learning models, submitting results to the blockchain for verification, thereby reducing on-chain computational load and increasing throughput.

  Quote NotebookLM on the above paper "*opML (Optimistic Machine Learning on Blockchain) is a new way to perform AI model calculations directly on a blockchain. It works by assuming the AI's answer is correct to save costs and speed things up. If someone thinks the answer is wrong, they can challenge it, and the system uses a special process to check only a tiny part of the work on the blockchain to see who is right. This makes it more practical to use AI within blockchain systems.*"

- Open-source framework for verifying ML inference on-chain.
- Similar to optimistic rollups.
- Example of **opML** powered AI: https://www.ora.io/app/opml/openlm

19.4.2 Resilient Model Services (RMS)

- RMS – Resilient Model Services (https://docs.ora.io/doc/resilient-model-services-rms/overview) works in complement to ORA as an AI service designed to provide computation for all scenarios. It ensures resilient (stable, reliable, fault tolerant, and secure) AI computation by replicating models on several nodes. It is powered by opML.
- Its API service integrates seamlessly with existing AI frameworks.

19.4.3 Initial Model Offerings (IMO)

Model Ownership (ERC-7641 Intrinsic RevShare Token) + Inference Asset (eg. ERC-7007 Verifiable AI-Generated Content Token)

- IMO launches an ERC-20 token (more specifically, ERC-7641 Intrinsic RevShare Token) of any AI model to capture its long-term value
- Anyone who purchases the token becomes one of the owners of this AI model
- Token holders share revenue of the IMO AI model

The IMO launch blog post (https://mirror.xyz/orablog.eth/xYMD27tN23ppbKCluB9faytF_W6M1hKXTuKcfkm3D50) and the first IMO implementation (https://mirror.xyz/orablog.eth/GSjMm-qC4WWsduGqCISSvA1IxicJbyRDES_bl7-Tt2o)

19.4.4 My 2 cents : ORA IMO vs. Hugging Face

19.4.4.1 Primary Function & Purpose

ORA's IMO:

- Focuses on **monetization and ownership** of open-weight AI models via blockchain.
- Allows model creators to launch a **tokenized economy** around their models, where contributors (e.g., trainers, data providers, compute providers) are rewarded with tokens.
- Uses a **decentralized verification system (TrueNet)** to ensure model integrity.
- Aims to create a **self-sustaining ecosystem** where model usage and improvements are incentivized via crypto-economic mechanisms.

Hugging Face:

- Primarily a **centralized hub** for sharing, discovering, and deploying AI models (like GitHub for ML).
- Hosts models (open & proprietary), datasets, and Spaces (demo apps).
- No built-in monetization for most open models (though Hugging Face offers enterprise solutions).
- Focuses on **collaboration and accessibility**, not ownership or tokenized incentives.

19.4.4.2 Monetization & Incentives

ORA IMO:

- Models are tied to a **crypto token**, allowing stakeholders (creators, validators, users) to earn rewards.
- Economic model ensures **long-term sustainability** via token staking, usage fees, and governance.

Hugging Face:

- Free for open models; monetization is mostly through private repositories, inference APIs, and enterprise plans.
- No direct way for model contributors to earn from public usage (unless they license commercially).

19.4.4.3 Decentralization & Trust

ORA IMO:

- **Decentralized** – relies on blockchain for model verification (TrueNet) and economic distribution.
- Immune to single-point censorship (models are stored on IPFS/Arweave).

Hugging Face:

- **Centralized** – Hugging Face controls the platform, can remove models, and hosts weights on its servers.

19.4.4.4 Model Accessibility & Usage

ORA IMO:

- Models are **permissionless and open**, but usage may involve token-based payments or staking.
- Designed for **decentralized AI applications** (e.g., DApps needing verifiable models).

Hugging Face:

- **Free and open access** for most models (unless paywalled by the uploader).
- Used widely for research, prototyping, and production deployments via their inference API.

19.4.4.5 Target Audience

ORA IMO:

- **Crypto-native AI developers** who want decentralized ownership & monetization.
- Projects needing **on-chain verifiable models** (DeFi, decentralized AI agents).

Hugging Face:

- **Traditional ML researchers, engineers, and companies** looking for easy model sharing.
- **Developers** who want plug-and-play AI without dealing with blockchain.

19.4.4.6 Short Summary in conclusion

No real overlap.

- **Hugging Face** = **"GitHub for AI"** (centralized, collaboration-focused).
- **ORA IMO** = **"Tokenized AI model economy"** (decentralized, incentive-driven).

19.4.5 Optimistic Agents

19.4.5.1 Overview

The `opAgent` use case (https://mirror.xyz/orablog.eth/sEFCQVmERNDIsiPDs2LUnU-_SdLmKERpCKcEP7hO08): A blockchain-based AI agent that uses optimistic rollup-like mechanisms for execution.

- The agent performs computations off-chain (e.g., AI inference, decision-making).
- Results are posted on-chain with a dispute window (like Optimistic Rollups).
- If no one challenges the result during the window, it's accepted as valid.

The use cases are: DeFi: Autonomous trading agents, DAOs: Governance proposal analysis, Gaming: NPCs with on-chain accountability.

19.4.5.2 How It Works

1. Execution:
   - The agent runs off-chain (e.g., on RMS nodes).
   - Example: An AI model processes a query like *"What's the best strategy for this trade?"*.
2. Submission:
   - The result is posted on-chain with a **cryptographic proof** (e.g., multi-sig or ZK-SNARK lite).
3. Dispute Period:
   - Anyone can challenge the result by submitting a **fraud proof**.
   - If challenged, the agent's logic is re-run **on-chain** or by a decentralized oracle network.
4. Finalization:
   - If unchallenged, the result becomes **canonical**.

19.4.6 Perpetual Agents

19.4.6.1 Overview

A perpetual agent is always-on, self-sustaining AI agent that operates perpetually on-chain. The agent has a persistent state and continuous funding (e.g., via streaming payments like Superfluid). It can adapt its behavior based on on-chain events (e.g., market conditions). There are token economic incentives for hosting the agent.

Use Cases are: Perpetual DeFi Strategies: Autonomous yield farming, On-Chain Chatbots: Answering queries indefinitely, DAO Delegates: Voting on behalf of token holders.

19.4.6.2 How It Works

Funding:

- Funded by users or DAOs via streaming ETH/USDC.
- Example: A DAO streams $100/month to an agent for analytics.

Autonomous Operation:

- Listens to on-chain triggers (e.g., "*If ETH price drops below $3K, execute X*").
- Uses **opAgent-like execution** for complex logic (off-chain compute + optimistic verification).

Self-Improvement:

- Can upgrade its own logic via **decentralized governance** (e.g., token votes).

19.4.7 Tokenized AI Generated Content (AIGC)

A Tokenized AI-Generated Content (AIGC) is a NFT, ERC-721 or ERC-1155, that represents AI-generated content (images, text, music, etc.). It is different from EIP-7007. The NFT itself doesn't encode AI logic; it's just a record of output and represents ownership of AI outputs.

The EIP-7007 is a proposed standard specifically designed for NFTs that dynamically change based on AI inputs (e.g., an NFT that evolves when you ask it questions): https://eips.ethereum.org/EIPS/eip-7007. The token proposed by EIP-7007 assumes the base NFT follows but adds new metadata and verification methods for AI-generated content. It focuses on on-chain/off-chain interoperability for AI models. For example, it defines how to request and verify AI inferences on-chain.

It is still in draft or discussion phase (not yet finalized).

Example:

```
// Simplified EIP-7007 contract
contract AI_NFT is ERC721 {
    mapping(uint256 => string) public prompts;
    address public aiOracle;

    function updateWithAI(uint256 tokenId, string memory prompt) public {
        require(_isApprovedOrOwner(msg.sender, tokenId), "Not owner");
        prompts[tokenId] = prompt;
        // Oracle signs a response (off-chain AI computes and updates metadata)
    }
}
```

Verifiable "AI Creativity" with the 7007 Protocol (https://www.7007.ai/)

19.4.8 Running AI Text Generation Tasks with Decentralized AI Model Inferences

- The OAO Github repository: https://github.com/ora-io/OAO
- Implementing the `IAIOracle.sol` interface
- Building smart contracts with ORA's AI Oracle: https://docs.ora.io/doc/ai-oracle/ai-oracle/build-with-ai-oracle
- Handling the Callback gas limit estimation for each model ID: https://docs.ora.io/doc/ai-oracle/ai-oracle/callback-gas-limit-estimation.
- Reference list for models and addresses for different networks: https://docs.ora.io/doc/ai-oracle/ai-oracle/references.

19.4.9 Practical Implementation of ORA API Calls

19.4.9.1 General

- To start coding,
  - **Get the ORA API key** through this page: https://rms.ora.io/dashboard/. An API Key is only for authenticating requests. All API keys are managed directly on the smart contract. This includes both the generation and deletion processes of API keys. The API key has the following structure, which enables you to recover it within your local environment: `{`blockchain identifier`}:{base58encoding(message:signature(message, privateKey))}`. Here, the message is a bytes32 data type within the smart contract.
  - **Set Up Environment**: Ensure you have Python installed if you're using the SDK, or a tool like cURL for direct API calls.

- Replace your existing AI API provider with RMS API Key and point it to the RMS endpoint. https://docs.ora.io/doc/resilient-model-services-rms/ora-api - use-ora-api

19.4.9.2 OpenAI SDK-compatible ORA API Call to generate text

Here is an example Python code of calling ORA API using the same SDK interface as OpenAI:

```
import openai

# Define your query
system_content = "You are a helpful assistant."
user_content = "What are some fun things to do in New York?"

# Set your ORA API key
ORA_API_KEY = "YOUR_ORA_API_KEY"

# Initialize the client
client = openai.OpenAI(
    api_key=ORA_API_KEY,
    base_url="https://api.ora.io/v1",
)

# Perform a chat completion
chat_completion = client.chat.completions.create(
    model="deepseek-ai/DeepSeek-V3",
    messages=[
        {"role": "system", "content": system_content},
        {"role": "user", "content": user_content},
    ]
)

# Print the response
response = chat_completion.choices[0].message.content
print("Response:\n", response)
```

Here is an example JavaScript code of calling ORA API using the same SDK interface as OpenAI (NodeJS v18 or higher):

```
const ORA_API_KEY = "YOUR_ORA_API_KEY";
const systemContent = "You are a helpful assistant.";
const userContent = "What are some fun things to do in New York?";

async function getChatCompletion() {
    try {
        const response = await fetch("https://api.ora.io/v1/chat/completions", {
            method: "POST",
            headers: {
                "Content-Type": "application/json",
                "Authorization": `Bearer ${ORA_API_KEY}`
            },
            body: JSON.stringify({
                model: "deepseek-ai/DeepSeek-V3",
                messages: [
                    { role: "system", content: systemContent },
                    { role: "user", content: userContent }
                ]
            })
        });

        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }

        const data = await response.json();
        const chatResponse = data.choices[0].message.content;
        console.log("Response:\n", chatResponse);
    } catch (error) {
        console.error("Error:", error);
    }
}

getChatCompletion();
```

19.5 Reading: Details on opML and RMS

19.5.1 General

OpML (Optimistic Machine Learning on Blockchain) and RMS (Resilient Model Services) are two complementary frameworks designed to enhance the reliability, transparency, and efficiency of decentralized machine learning (ML) systems. Here's how they work together:

19.5.1.1 OpML (Optimistic Machine Learning on Blockchain)

Purpose:

OpML enables trustless, decentralized ML model training and inference by leveraging optimistic rollups (a layer-2 blockchain scaling solution).

Key Features:

- **Optimistic Execution**: Models are trained or inferred off-chain, and results are posted on-chain with a dispute window for fraud proofs.
- **Cost Efficiency**: Reduces on-chain computation costs by batching transactions.
- **Decentralized Verification**: Nodes can challenge incorrect results during the dispute period.

19.5.1.2 RMS (Resilient Model Services)

Purpose:

RMS ensures high availability, fault tolerance, and correctness of ML models in decentralized environments.

Key Features:

- **Model Replication**: Stores multiple copies of models across nodes for redundancy.
- **Consensus-Based Validation**: Uses decentralized consensus to verify model outputs.
- **Self-Healing**: Automatically recovers from node failures or adversarial attacks.

19.5.1.3 How OpML and RMS Work Together

1. Model Deployment:
   - A model is trained or uploaded into the RMS network, which replicates it across nodes.
   - OpML registers the model's metadata on-chain (e.g., model hash, RMS node locations).
2. Inference Request:
   - A user submits an inference request via an OpML smart contract.
   - The request is routed to RMS nodes, which compute the result optimistically (off-chain).
   - The result is posted on-chain with a cryptographic proof. See details further below *19.5.2.1,* Step-by-Step Proof Generation (RMS Nodes).
3. Dispute & Verification:
   - If a challenger disputes the result, RMS nodes re-compute the inference via consensus.
   - Fraudulent results are rolled back, and the correct result is enforced.

19.5.1.4 Key Workflow Summary

1. **Request** → User calls OpML contract.
2. **Compute** → RMS nodes process the request off-chain.
3. **Post Result** → Optimistically posted on-chain.
4. **Dispute (if any)** → Challenged via RMS consensus.
5. **Finalize** → Result accepted after dispute window.

This combination ensures scalability (via OpML's optimistic rollup) and reliability (via RMS's resilient model services).

19.5.2 How RMS nodes compute results

Here's a step-by-step example of how **RMS (Resilient Model Services) nodes** generate and obtain the **proof** required to call `submitInferenceResult` on the OpML smart contract:

19.5.2.1 Step-by-Step Proof Generation (RMS Nodes)

1. Compute the Inference Result Off-Chain

   RMS nodes load the requested ML model (e.g., resnet50) and process the input data (fetched from IPFS using inputDataHash).

   Example JSON Output:

   ```
   { "result": "cat", "confidence": 0.92 }
   ```

2. Generate a Consensus Proof

   RMS nodes use Byzantine Fault Tolerance (BFT) or ZK-SNARKs to agree on the result and generate a proof.
   Option A: Multi-Signature Proof (Simpler): RMS nodes sign the result hash with their private keys.

   ```
   // Pseudocode:
   // Each RMS node signs the result
   const resultHash = ethers.utils.keccak256(ethers.utils.toUtf8Bytes("cat"));
   const signatures = await Promise.all(
     rmsNodes.map(node => node.signMessage(ethers.utils.arrayify(resultHash)))
   );
   // Combine signatures into a single proof
   const proof = ethers.utils.defaultAbiCoder.encode(
     ["bytes32", "bytes[]"],
     [resultHash, signatures]
   );
   ```

   Option B: ZK-SNARK Proof (More Secure): RMS nodes generate a ZK proof that the inference was correct without revealing the raw data.

   ```
   Pseudocode:
   const { proof, publicSignals } = await snarkjs.groth16.fullProve(
     { input: "QmXyZ...", model: "resnet50" },
     "model_compiled.wasm",
     "model_final.zkey"
   );
   const proofCalldata = await snarkjs.groth16.exportSolidityCallData(proof, publicSignals);
   ```

3. Submit to OpML Contract

   The RMS node calls submitInferenceResult with the proof:

   ```
   await opMLContract.submitInferenceResult(
     requestId,     // e.g., 42
     resultHash,    // "0x5d7e9..." (hash of "cat")
     proof          // Combined signatures or ZK-SNARK proof
   );
   ```

Key Components of the Proof

Proof Type	**How It Works**	**Use Case**
Multi-Sig	Majority of RMS nodes sign the result.	Faster, suited for trusted nodes.
ZK-SNARK	Cryptographic proof of correct execution.	Untrusted environments.

19.5.2.2 Complete JavaScript RMS example (Multisig)

- The backend code:

```
const { ethers } = require("ethers");
const rmsNodePrivateKeys = ["0xabc...", "0xdef..."]; // Private keys of RMS nodes

async function generateAndSubmitProof(requestId, inputDataIpfsHash) {
  // 1. Fetch input data from IPFS
  const inputData = await ipfs.get(inputDataIpfsHash);

  // 2. Run inference (e.g., using TensorFlow.js)
  const model = await tf.loadModel("ipfs://QmModel...");
  const result = model.predict(inputData);
  const resultStr = JSON.stringify(result);
  const resultHash = ethers.utils.keccak256(ethers.utils.toUtf8Bytes(resultStr));

  // 3. Generate multi-signature proof
```

```
  const signatures = await Promise.all(
    rmsNodePrivateKeys.map(key => {
      const wallet = new ethers.Wallet(key);
      return wallet.signMessage(ethers.utils.arrayify(resultHash));
    })
  );
  const proof = ethers.utils.defaultAbiCoder.encode(
    ["bytes32", "bytes[]"],
    [resultHash, signatures]
  );

  // 4. Submit to OpML contract
  const opMLContract = new ethers.Contract(OPML_ADDRESS, OPML_ABI, provider);
  await opMLContract.submitInferenceResult(requestId, resultHash, proof);
}
```

- How the Smart Contract Validates the Proof

  The OpML contract checks the proof in `_verifyProof`:

```
// Solidity code
function _verifyProof(
  uint256 requestId,
  bytes32 resultHash,
  bytes memory proof
) internal view returns (bool) {
  // Decode multi-sig proof
  (bytes32 expectedHash, bytes[] memory signatures) =
    abi.decode(proof, (bytes32, bytes[]));

  // Verify hash matches
  require(expectedHash == resultHash, "Hash mismatch");

  // Check ≥2/3 RMS nodes signed
  uint256 validSignatures;
  for (uint i = 0; i < signatures.length; i++) {
    address signer = recoverSigner(resultHash, signatures[i]);
    if (isRMSNode(signer)) validSignatures++;
  }
  return validSignatures >= (TOTAL_RMS_NODES * 2) / 3;
}
```

19.5.3 Example API implementation

19.5.3.1 Flow Diagram

```
User (Frontend) -> requestInference()      -> OpML Contract
OpML Contract   -> Event                   -> RMS Backend
RMS Backend     -> submitInferenceResult() -> OpML Contract
OpML Contract   -> Event                   -> User (Frontend fetches result)
```

19.5.3.2 JavaScript Instantiate opMLContract and submitRequest:

To interact with the OpML smart contract, you need:

- **Contract ABI (OpML_ABI)**: The Application Binary Interface (ABI) defines how to call the smart contract functions.
- **Ethereum Provider (provider)**: Connects to the blockchain (e.g., MetaMask, Infura, Alchemy).

```
// JavaScript
// Import required libraries (using ethers.js)
const { ethers } = require("ethers");

// --- 1. Set up the Ethereum Provider ---
// Option 1: Use MetaMask (browser)
// const provider = new ethers.providers.Web3Provider(window.ethereum);

// Option 2: Use Infura/Alchemy (Node.js)
const provider = new ethers.providers.JsonRpcProvider(
  "https://mainnet.infura.io/v3/YOUR_INFURA_API_KEY"
);

// --- 2. Define the OpML Contract ABI ---
// This is a simplified ABI for the OpML contract.
// In practice, you'd get this from the compiled Solidity artifact.
const OpML_ABI = [
  // Function to request an inference
  {
```

```
      "inputs": [
        { "name": "modelId", "type": "string" },
        { "name": "inputDataHash", "type": "string" }
      ],
      "name": "requestInference",
      "outputs": [],
      "stateMutability": "payable",
      "type": "function"
    },
    // Function to fetch results
    {
      "inputs": [{ "name": "requestId", "type": "uint256" }],
      "name": "getInferenceResult",
      "outputs": [{ "name": "result", "type": "bytes32" }],
      "stateMutability": "view",
      "type": "function"
    }
];

// --- 3. Initialize the Contract ---
const opMLContract = new ethers.Contract(
  "0x123...",      // OpML smart contract address (deployed on-chain)
  OpML_ABI,
  provider
);

// --- 4. Add Signer (for transactions) ---
const privateKey = "YOUR_PRIVATE_KEY"; // Or use MetaMask
const signer = new ethers.Wallet(privateKey, provider);
const opMLContractWithSigner = opMLContract.connect(signer);

// --- 5. Submit an Inference Request ---
async function submitRequest() {
  const tx = await opMLContractWithSigner.requestInference(
    "modelId123",       // Model ID registered in RMS
    "QmXyZ...",         // IPFS hash of input data
    { value: ethers.utils.parseEther("0.01") } // Pay for gas
  );
  console.log("Transaction Hash:", tx.hash);
}
submitRequest();
```

19.5.3.3 Solidity: RMS Node Submitting Results

Here's the full Solidity code for the OpML contract, including the `submitInferenceResult` function and challenge logic:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract OpML {
    // --- Structs ---
    struct InferenceRequest {
        string modelId;
        string inputDataHash;
        address requester;
        bool resolved;
        bytes32 resultHash;
        uint256 challengePeriodEnd;
    }

    // --- State Variables ---
    mapping(uint256 => InferenceRequest) public requests;
    uint256 public nextRequestId;
    uint256 public constant CHALLENGE_PERIOD = 1 hours;

    // --- Events ---
    event InferenceRequested(uint256 requestId, string modelId, string inputDataHash);
    event InferenceResultPosted(uint256 requestId, bytes32 resultHash);
    event InferenceChallenged(uint256 requestId, address challenger);
    event InferenceFinalized(uint256 requestId, bytes32 resultHash);

    // --- Modifiers ---
    modifier onlyRegisteredRMSNode() {
        require(isRMSNode(msg.sender), "Caller is not an RMS node");
        _;
    }
```

```
    // --- Functions ---

    // Submit a new inference request (called by users)
    function requestInference(string memory modelId, string memory inputDataHash) external
payable {
        uint256 requestId = nextRequestId++;
        requests[requestId] = InferenceRequest({
            modelId: modelId,
            inputDataHash: inputDataHash,
            requester: msg.sender,
            resolved: false,
            resultHash: 0,
            challengePeriodEnd: 0
        });
        emit InferenceRequested(requestId, modelId, inputDataHash);
    }

    // RMS nodes submit results optimistically (with proof)
    function submitInferenceResult(
        uint256 requestId,
        bytes32 resultHash,
        bytes calldata proof
    ) external onlyRegisteredRMSNode {
        InferenceRequest storage request = requests[requestId];
        require(!request.resolved, "Request already resolved");
        require(_verifyProof(requestId, resultHash, proof), "Invalid proof");

        request.resultHash = resultHash;
        request.challengePeriodEnd = block.timestamp + CHALLENGE_PERIOD;
        emit InferenceResultPosted(requestId, resultHash);
    }

    // Challenge a result during the dispute window
    function challengeResult(uint256 requestId) external {
        InferenceRequest storage request = requests[requestId];
        require(block.timestamp < request.challengePeriodEnd, "Challenge period expired");
        require(!request.resolved, "Request already resolved");
        emit InferenceChallenged(requestId, msg.sender);
    }

    // Finalize the result after the challenge period
    function finalizeResult(uint256 requestId) external {
        InferenceRequest storage request = requests[requestId];
        require(block.timestamp >= request.challengePeriodEnd, "Challenge period active");
        require(!request.resolved, "Already resolved");
        request.resolved = true;
        emit InferenceFinalized(requestId, request.resultHash);
    }

    // --- Getter Functions ---
    function getInferenceResult(uint256 requestId) external view returns (
        string memory modelId,
        string memory inputDataHash,
        bytes32 resultHash,
        bool resolved,
        uint256 challengePeriodEnd
    ) {
        InferenceRequest storage request = requests[requestId];
        require(request.requester != address(0), "Request does not exist");
        return (
            request.modelId,
            request.inputDataHash,
            request.resultHash,
            request.resolved,
            request.challengePeriodEnd
        );
    }

    // --- Internal Functions ---
    function _verifyProof(uint256 requestId, bytes32 resultHash, bytes memory proof) internal
pure returns (bool) {
        // In production, this would verify a ZK proof or multisig from RMS nodes.
        // Simplified for this example:
        return keccak256(proof) == keccak256(abi.encodePacked(requestId, resultHash));
    }

    function isRMSNode(address node) public pure returns (bool) {
```

```
            // In reality, this would check a registry of RMS nodes.
            return true; // Simplified for demo
        }
    }
```

19.5.3.4 JavaScript: getInferenceResult

```
// Fetch results using the getter
const result = await opMLContract.getInferenceResult(requestId);
console.log("Model ID:", result.modelId);
console.log("Result Hash:", ethers.utils.hexlify(result.resultHash));
console.log("Resolved?", result.resolved);
```

19.5.3.5 Detailed Mechanism Optimistic Rollup Challenge

1. Result Submission:
   - An RMS node submits a result with a cryptographic proof (e.g., a signature from a quorum of RMS nodes).
   - The result is stored on-chain, and a **challenge period** (e.g., 1 hour) begins.
2. Dispute Window:
   - During the challenge period, anyone can call `challengeResult(requestId)`.
   - If challenged, RMS nodes must recompute the inference via consensus.
     - If the original result was wrong, the challenger is rewarded.
     - If correct, the challenger loses a stake (anti-spam measure).
3. Finalization:
   - After the challenge period, `finalizeResult(requestId)` is called.
   - The result is marked as `resolved` and becomes immutable.

Key Takeaways:

- **OpML** handles on-chain coordination (requests, disputes, finalization).
- **RMS** ensures off-chain computation is correct via replication and consensus.
- **Optimistic Rollup** reduces costs by only settling disputes on-chain.

This design ensures **scalability** (most work is off-chain) and **security** (fraud proofs catch errors).

19.5.3.6 JavaScript Frontend functions Arguments

```
await opMLContract.requestInference("resnet50", "QmXyZ...", { value:
ethers.utils.parseEther("0.01") });
```

- **Purpose**: Submit an inference request to the OpML contract.
- **Arguments**:
  - `modelId (string)`: Same as Solidity (e.g., `"resnet50"`).
  - `inputDataHash (string)`: IPFS hash of input data.
  - `options (object)`: Transaction parameters (e.g., `{ value: ethers.utils.parseEther("0.01") }` for gas).

```
// Fetch results using the getter
const result = await opMLContract.getInferenceResult(requestId);
```

- **Purpose**: Fetch results from the smart contract.
- **Arguments**:
  - `requestId (number/BigNumber)`: Request ID (e.g., `42`).

19.5.3.7 Solidity functions Arguments

```
function requestInference(string memory modelId, string memory inputDataHash) external payable;
```

- **Purpose**: Called by users to request an ML inference.
- **Arguments**:
  - `modelId (string)`: Identifier of the ML model (e.g., `"resnet50"`).
  - `inputDataHash (string)`: IPFS hash of the input data (e.g., `"QmXyZ..."`).

```
function submitInferenceResult(uint256 requestId, bytes32 resultHash, bytes calldata proof)
external onlyRegisteredRMSNode;
```

- **Purpose**: Called by RMS nodes to submit computed results optimistically.
- **Arguments**:
  - `requestId (uint256)`: ID of the inference request.
  - `resultHash (bytes32)`: Hash of the inference result (e.g., `keccak256(abi.encodePacked("cat"))`).
  - `proof (bytes)`: Cryptographic proof from RMS consensus (e.g., a multisig or ZK proof).

```
function challengeResult(uint256 requestId) external;
```

- **Purpose**: Called to dispute a potentially fraudulent result.
- **Arguments**:
  - `requestId (uint256)`: ID of the disputed request.

```
const result = await opMLContract.getInferenceResult(requestId);
```

- **Purpose**: Fetch details of a request (called off-chain).
- **Arguments**:
  - `requestId (uint256)`: ID of the request.

19.5.3.8 JavaScript Backend functions Arguments

```
const proof = ethers.utils.hexlify(/* RMS consensus proof */);
await opMLContract.submitInferenceResult(42, "0x123...", proof);
```

- **Purpose**: RMS nodes submit results via a backend script.
- **Arguments**:
  - `requestId (number)`: Request ID.
  - `resultHash (string)`: Hex string of the result hash (e.g., `"0x123..."`).
  - `proof (string/bytes)`: Proof from RMS consensus (e.g., a multisig signature).

19.5.3.9 Critical Considerations

- Data Formats:
  - Solidity uses `bytes32` for hashes, while JS uses hex strings (`"0x..."`).
  - Use `ethers.utils.hexlify()` and `ethers.utils.keccak256()` for conversions.
- Proofs:
  - In Solidity, `proof` is raw `bytes` (e.g., `abi.encodePacked(signatures)`).
  - In JS, it's a hex string (e.g., `"0xabcd..."`).
- Gas Handling:
  - Always include gas estimates in JS (e.g., `{ gasLimit: 500000 }`).

19.6 Decentralized AI Inference

Most AI inference providers we used required credit card payments or similar subscription or usage-based payments.

Some providers are starting to support crypto payments.

- Often these payments are converted to fiat currency by the provider to fund their corporate operations.

The approach of Decentralized AI Inference is to provide a way to execute AI inference off-chain while providing verifiable results on-chain, while handling all the payments through cryptocurrencies and ensuring the proper incentives for the providers

- Tokens as payment medium.
- Tokens as security mechanism (incentives and penalties).
- Tokens as a governance mechanism.

19.6.1 Venice AI Platform

One example of a Decentralized AI Inference platform is the Venice Protocol.

- Privacy-preserving computation.
- Uncensored models.