Building an Oracle Network for Ethereum to Leverage Large Language Models

Madhukara S. Holla (Team 2)

1 Introduction

Smart contracts on blockchain networks are inherently limited to deterministic on-chain data. This limitation poses a challenge when integrating with Large Language Models (LLMs), which provide valuable but non-deterministic outputs. This project implements a decentralized oracle network that enables Ethereum smart contracts to securely interact with LLMs while maintaining consensus across the network.

2 Problem Definition

Smart contracts are deterministic by design to ensure consensus across blockchain nodes. This determinism limits their ability to interact with external, dynamic, and non-deterministic systems like LLMs. The key challenges include:

- **Achieving Consensus:** LLMs generate probabilistic outputs, making it difficult for blockchain nodes to agree on a single result.
- **Ensuring Verifiability:** Blockchain systems rely on transparent and immutable data. Integrating LLMs requires mechanisms to verify and log their outputs in a trustless environment.

3 System Architecture

The system implements a decentralized oracle network that enables smart contracts to access LLM capabilities while maintaining consensus on non-deterministic outputs. A high level system architecture diagram is provided in Appendix A (Figure 1).

3.1 Core Components

- **Decentralized Oracle Network:** A network of 7 nodes operates under a Practical Byzantine Fault Tolerance (PBFT) consensus mechanism to handle non-deterministic LLM outputs.
- **Smart Contract Interface:** Solidity-based contracts deployed on the Polygon ZKEVM testnet facilitate the submission of queries to the oracle network and retrieval of responses.

- **Decentralized Storage:** The InterPlanetary File System (IPFS) stores interaction logs, with corresponding hash references maintained on-chain for verifiability.
- **Response Validation:** Semantic similarity scoring algorithms, implemented using cosine similarity, assess the consistency of LLM responses across oracle nodes.

3.2 System Components and Implementation

The system is implemented in Go and consists of the following key components, with specific implementation details and library usage:

- **Oracle Node Component** (oracle/pkg/):
  - Core PBFT implementation in `consensus/pbft.go`:
    - Complete protocol implementation including normal operation, view change, and recovery
    - Custom message handling using Go's native channels and mutexes
    - SHA-256 for message digests (Go's `crypto/sha256`)
    - ECDSA signatures using Ethereum's `secp256k1` library
    - State management using in-memory maps with periodic checkpointing
  - Network Communication:
    - HTTP/WebSocket-based communication using Go's `net/http` package
    - JSON message serialization for inter-node communication
    - Static node configuration through config files (no dynamic discovery)
    - TLS encryption for secure communication
  - LLM Integration:
    - Direct OpenAI API integration using `go-openai` v1.36.0
    - Response normalization using tokenization and embedding
    - Cosine similarity calculation for response comparison
    - Configurable similarity thresholds for consensus
- **Smart Contract Component** (contracts/):
  - Single Solidity smart contract (v0.8.19):
    - `Oracle.sol`: Implements the complete oracle functionality
      - Request creation and fee handling
      - Response submission with IPFS CID storage
      - Request state tracking and verification
      - Oracle node management and access control
  - Integration with Ethereum-compatible networks
  - Event emission for request and response tracking:
    - RequestCreated: Emitted when new requests are submitted
    - ResponseReceived: Emitted when consensus responses are stored
    - OracleNodeRegistered/Removed: Emitted during oracle management

- **Agent Component** (agent/):
  - LLM Request Processing:
    - Prompt templating and validation
    - Rate limiting and retry mechanisms
    - Error handling and fallback strategies
  - Response Processing:
    - Tokenization using OpenAI's `tiktoken` library for GPT-4
    - Vector embeddings using OpenAI's text-embedding-ada-002 model
    - Semantic similarity computation using cosine distance between embeddings
    - Response format standardization with JSON schema validation
- **Listener Component** (scripts/):
  - Event Monitoring:
    - Direct blockchain interaction using `go-ethereum` v1.13.5
    - Event filtering and parsing using `go-ethereum/accounts/abi`
    - Block polling with configurable intervals (5 seconds)
    - Automatic block range tracking and event filtering
  - Response Management:
    - IPFS integration using Pinata API
    - Secure file pinning with JWT or API key authentication
    - Automatic gas price optimization using `SuggestGasPrice`
    - Concurrent request handling with Go contexts and channels

All components are containerized using Docker with Alpine Linux base images for minimal footprint. Configuration is managed through environment variables and JSON config files, with sensitive data (API keys, private keys) handled through secure environment variables.

3.3 LLM Response Consensus

The system handles non-deterministic LLM outputs through a semantic similarity-based consensus mechanism:

- **Response Collection:**
  - Each node independently queries the LLM with the same prompt
  - Responses are normalized by removing whitespace and formatting
  - Each node broadcasts its response to all other nodes
- **Similarity Calculation:**
  - Each node computes pairwise cosine similarity between all received responses
  - Responses are converted to vector representations using word embeddings

- Similarity score S_{ij} between responses i and j is calculated as:

$$S_{ij} = \frac{\vec{v_i} \cdot \vec{v_j}}{|\vec{v_i}||\vec{v_j}|}$$

where $\vec{v_i}$ and $\vec{v_j}$ are the vector representations

- **Consensus Process:**
  - Each node identifies the response with highest average similarity to all others
  - A response is considered valid if its similarity score exceeds threshold τ (set to 0.8)
  - Nodes only commit if 2f+1 nodes agree on a response meeting the threshold
  - The leader node (current primary) selects the final response from the valid set
- **Response Selection:**
  - Leader chooses the response with highest average similarity score
  - Selected response must have been validated by at least 2f+1 nodes
  - If no response meets criteria, the consensus round fails
  - Failed rounds trigger a new round of LLM queries

This approach ensures that even with non-deterministic LLM outputs, the network can achieve consensus on semantically equivalent responses while filtering out significantly divergent ones.

4 PBFT Implementation

Our implementation extends the traditional PBFT protocol to handle non-deterministic LLM outputs. The complete algorithm is provided in Algorithm 1 in the Appendix. The implementation includes:

- **Message Structure and State Management:**
  - Messages are structured to include type, sender ID, view number, sequence number, digest, and data (see Listing 1)
  - Each PBFT instance maintains state including current view, sequence number, and message buffers
  - Thread-safe operations using mutex locks for concurrent message handling
  - Message types include PrePrepare, Prepare, Commit, and ViewChange
- **Cryptographic Components:**
  - SHA-256 for message digest computation (see Listing 2)
  - Message verification includes sender validation, sequence number checks, and view number validation
  - ECDSA (secp256k1) for digital signatures using Ethereum's crypto libraries

- Digest verification ensures message integrity throughout consensus phases

- **Consensus Protocol Flow:**
  - Pre-prepare phase: Leader computes digest and broadcasts proposal
  - Prepare phase: Nodes validate and broadcast prepare messages after digest verification
  - Commit phase: Nodes wait for 2f+1 prepare messages before committing
  - Each phase includes message validation and state updates
  - Messages are tracked using sequence numbers for ordering
  - Detailed algorithm provided in Appendix (see Algorithm 1)

- **View Change Protocol:**
  - Triggered by view change timeout or leader failure detection
  - Nodes broadcast view change messages with current state (see Listing 3)
  - New view starts when 2f+1 view change messages are received
  - View change includes state transfer and message cleanup
  - Leader selection based on view number modulo number of nodes

- **State Management and Recovery:**
  - Periodic checkpoints every 100 sequences (see Listing 5)
  - Garbage collection of old messages and checkpoints
  - State includes prepare messages, commit messages, and view change messages
  - Recovery possible through checkpoint restoration
  - Message cleanup triggered after consensus or view changes

- **Fault Tolerance:**
  - Tolerates f Byzantine failures where f = (n-1)/3
  - 7-node network configuration handles up to 2 Byzantine nodes
  - Safety guaranteed through 2f+1 matching prepare messages
  - Liveness ensured through view change protocol
  - Message validation prevents equivocation attacks

5 Results

The implementation of the decentralized oracle network achieved the following:

- **Consensus Achievement:** The oracle network successfully achieved consensus with up to $f = \lfloor (n-1)/3 \rfloor$ Byzantine nodes, even with non-deterministic LLM outputs.

- **Efficient Storage:** Using IPFS for off-chain storage minimized on-chain data footprint, reducing gas costs while maintaining verifiability through hash references.

- **Verifiable Logs:** All LLM interactions were logged in an immutable and verifiable manner, ensuring transparency and auditability.
- **Performance Metrics:** The system demonstrated a sub-30-second response time for LLM queries and effectively handled multiple concurrent requests using its Dockerized architecture.
- **Blockchain Integration:** The deployed smart contracts on the Polygon ZKEVM testnet reliably logged IPFS CIDs, enabling end-to-end traceability of oracle responses.

6 Limitations

The current implementation has the following limitations:

- **No Recovery Mechanism for PBFT Nodes:** The system does not implement recovery protocols for failed or disconnected nodes within the PBFT network.
- **No Peer Discovery:** Nodes must be manually configured as the system lacks automated peer discovery mechanisms.
- **No Staking by the Nodes:** Nodes do not stake any assets, resulting in a lack of economic incentives or "skin in the game" for honest participation.

7 Conclusion

This project successfully addresses the challenges of integrating non-deterministic LLM outputs into Ethereum smart contracts. By implementing a decentralized oracle network with robust PBFT consensus, cost-efficient storage using IPFS, and verifiable logging on the Polygon ZKEVM testnet, the system enables blockchain applications to harness the potential of AI-driven insights.

References

[1] Castro, M., & Liskov, B. (1999). *Practical Byzantine fault tolerance*. In OSDI '99: Proceedings of the Third Symposium on Operating Systems Design and Implementation.

[2] Castro, M., & Liskov, B. (2002). *Practical Byzantine fault tolerance and proactive recovery*. ACM Transactions on Computer Systems.

[3] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., & Wong, E. (2007). *Zyzzyva: speculative byzantine fault tolerance*. ACM SIGOPS Operating Systems Review.

[4] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G., & Abraham, I. (2019). *Hotstuff: BFT consensus with linearity and responsiveness*. In Proceedings of the ACM Symposium on Principles of Distributed Computing.

A System Architecture Diagram

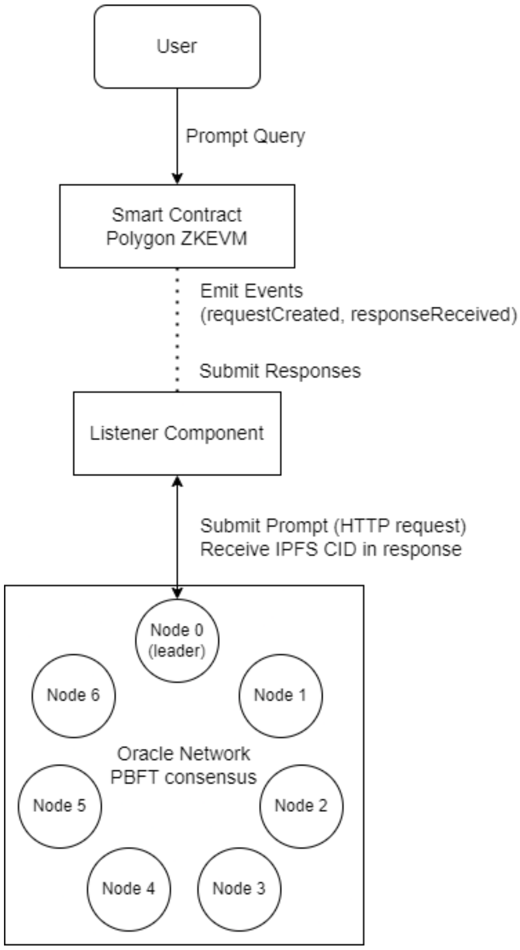

Figure 1: System Architecture Overview showing the interaction between smart contracts, oracle nodes in a PBFT consensus mechanism

B PBFT Algorithm Details

Our PBFT implementation follows Algorithm 1, which extends the traditional PBFT protocol to handle non-deterministic LLM outputs.

```
Initialize: view ← 0, sequence ← 0, f ← ⌊(n − 1)/3⌋;
Function NormalOperation(request):
    if isLeader then
        digest ← SHA256(request);
        Broadcast ⟨PRE-PREPARE, view, sequence, digest, request⟩;
    end
Function HandlePrePrepare(msg):
    if ValidMessage(msg) ∧ ValidDigest(msg) then
        Store msg in prepareMessages[sequence];
        Broadcast ⟨PREPARE, view, sequence, digest⟩;
    end
Function HandlePrepare(msg):
    if ValidMessage(msg) then
        Store msg in prepareMessages[sequence];
        if |prepareMessages[sequence]| ≥ 2f + 1 then
            Broadcast ⟨COMMIT, view, sequence, digest⟩;
        end
    end
Function HandleCommit(msg):
    if ValidMessage(msg) then
        Store msg in commitMessages[sequence];
        if |commitMessages[sequence]| ≥ 2f + 1 then
            response ← GetConsensusResponse();
            ExecuteConsensus(sequence, response);
            sequence ← sequence + 1;
        end
    end
Function ViewChange(newView):
    Broadcast ⟨VIEW-CHANGE, newView, sequence, state⟩;
    if |viewChangeMessages[newView]| ≥ 2f + 1 then
        view ← newView;
        isLeader ← (nodeID mod n = view);
        ResetState();
    end
Function GetConsensusResponse():
    responses ← CollectLLMResponses();
    similarities ← ComputeSemanticSimilarities(responses);
    return SelectMostSimilarResponse(similarities);
```

Algorithm 1: PBFT Consensus for LLM Oracle Network

C Implementation Details

C.1 PBFT Message Structure

The PBFT consensus protocol uses a structured message format for all communication (see Listing 1):

Listing 1: PBFT Message Structure

```
// ConsensusMessage represents a message in the PBFT protocol
type ConsensusMessage struct {
    Type    MessageType // PrePrepare, Prepare, Commit, ViewChange
    NodeID  string    // Sender node identifier
    View    uint64    // Current view number
    Sequence uint64   // Message sequence number
    Digest  []byte    // Message digest (SHA-256)
    Data    []byte    // Actual message content
}

// PBFT represents a PBFT consensus instance
type PBFT struct {
    mu               sync.RWMutex
    nodeID           string
    nodes            []string
    privateKey       *ecdsa.PrivateKey
    networkManager   *NetworkManager
    timeout          time.Duration
    viewChangeTimeout time.Duration
    checkpointInterval uint64

    // State
    view            uint64
    sequence        uint64
    state           PBFTState
    isLeader        bool
    lastCheckpoint  []byte
    lastCheckpointSeq uint64

    // Messages
    prepareMessages map[uint64]map[string]*ConsensusMessage
    commitMessages  map[uint64]map[string]*ConsensusMessage
    viewChangeMsgs  map[uint64]map[string]*ConsensusMessage
    checkpoints     map[uint64][]byte
    consensusReached map[uint64]bool
}
```

C.1.1 Message Digest and Verification

The system uses SHA-256 for message digest computation and verification (see Listing 2):

Listing 2: Message Digest Implementation

```
func (p *PBFT) computeDigest(data []byte) []byte {
    hash := sha256.Sum256(data)
    return hash[:]
}

func (p *PBFT) validateMessage(msg *ConsensusMessage) error {
    if msg == nil {
        return ErrInvalidMessage
    }

    // Validate sender
    senderValid := false
    for _, node := range p.nodes {
        if node == msg.NodeID {
            senderValid = true
            break
        }
    }
    if !senderValid {
        return ErrInvalidSender
    }

    // Validate sequence number
    if msg.Sequence < p.sequence-1 && p.sequence > 1 {
        return ErrInvalidSequence
    }

    // Validate view number
    if msg.View < p.view-1 && p.view > 1 {
        return ErrInvalidView
    }

    return nil
}
```

C.1.2 View Change Protocol

The view change protocol ensures liveness when the leader fails (see Listing 3):

Listing 3: View Change Implementation

```
func (p *PBFT) startViewChange(newView uint64) {
    p.mu.Lock()
    defer p.mu.Unlock()

    // Create view change message
    msg := &ConsensusMessage{
        Type:     ViewChange,
        NodeID:   p.nodeID,
        View:     newView,
        Sequence: p.sequence,
    }

    // Store view change message
    if _, exists := p.viewChangeMsgs[newView]; !exists {
        p.viewChangeMsgs[newView] = make(map[string]*ConsensusMessage)
    }
    p.viewChangeMsgs[newView][p.nodeID] = msg

    // Broadcast view change
    p.broadcast(msg)
}

func (p *PBFT) handleViewChange(msg *ConsensusMessage) {
    p.mu.Lock()
    defer p.mu.Unlock()

    // Store view change message
    if _, exists := p.viewChangeMsgs[msg.View]; !exists {
        p.viewChangeMsgs[msg.View] = make(map[string]*ConsensusMessage)
    }
    p.viewChangeMsgs[msg.View][msg.NodeID] = msg

    // Check if we have enough view change messages
    if len(p.viewChangeMsgs[msg.View]) >= 2*p.f+1 {
        p.changeView(msg.View)
    }
}
```

C.1.3 Consensus Flow

The complete consensus flow implementation (see Listing 4):

Listing 4: Consensus Flow Implementation

```
func (p *PBFT) ProposeValue(value []byte) error {
    if !p.isLeader {
        return fmt.Errorf("node is not the leader")
    }

    p.mu.Lock()
    defer p.mu.Unlock()

    // Compute message digest
    digest := p.computeDigest(value)

    // Create pre-prepare message
    msg := &ConsensusMessage{
        Type:     PrePrepare,
        NodeID:   p.nodeID,
        View:     p.view,
        Sequence: p.sequence,
        Digest:   digest,
        Data:     value,
    }

    // Store message for validation
    if _, exists := p.prepareMessages[p.sequence]; !exists {
        p.prepareMessages[p.sequence] = make(map[string]*ConsensusMessage)
    }
    p.prepareMessages[p.sequence][p.nodeID] = msg

    // Broadcast pre-prepare message
    return p.broadcast(msg)
}

func (p *PBFT) handlePrepare(msg *ConsensusMessage) {
    p.mu.Lock()
    defer p.mu.Unlock()

    // Initialize prepare messages map for this sequence
    if _, exists := p.prepareMessages[msg.Sequence]; !exists {
        p.prepareMessages[msg.Sequence] = make(map[string]*ConsensusMessage)
    }

    // Store prepare message
    p.prepareMessages[msg.Sequence][msg.NodeID] = msg

    // Check if we have enough prepare messages
    if len(p.prepareMessages[msg.Sequence]) >= 2*p.f {
        // Send commit message
        commit := &ConsensusMessage{
```

```
            Type:     Commit,
            NodeID:   p.nodeID,
            View:     msg.View,
            Sequence: msg.Sequence,
            Digest:   msg.Digest,
            Data:     msg.Data,
        }
        p.broadcast(commit)
    }
}
```

C.1.4 Checkpoint and State Management

Implementation of checkpointing and state management (see Listing 5):

Listing 5: Checkpoint Implementation

```go
func (p *PBFT) makeCheckpoint() {
    if p.checkpointInterval == 0 {
        p.checkpointInterval = 100
    }

    if p.sequence%p.checkpointInterval == 0 {
        checkpoint := &struct {
            Sequence uint64
            State    []byte
        }{
            Sequence: p.sequence,
            State:    p.lastCheckpoint,
        }

        checkpointBytes, _ := json.Marshal(checkpoint)
        p.checkpoints[p.sequence] = checkpointBytes
        p.lastCheckpointSeq = p.sequence
    }
}

func (p *PBFT) cleanup(sequence uint64) {
    // Cleanup old prepare messages
    for seq := range p.prepareMessages {
        if seq < sequence {
            delete(p.prepareMessages, seq)
        }
    }

    // Cleanup old commit messages
    for seq := range p.commitMessages {
        if seq < sequence {
            delete(p.commitMessages, seq)
        }
    }

    // Cleanup old consensus reached flags
    for seq := range p.consensusReached {
        if seq < sequence {
            delete(p.consensusReached, seq)
        }
    }

    // Cleanup old checkpoints
    for seq := range p.checkpoints {
        if seq < sequence-p.checkpointInterval {
            delete(p.checkpoints, seq)
        }
```

```
    }
}
```