DAO Voting System Documentation

Table of Contents

System Overview

The DAO Voting System is a comprehensive solution for managing and analyzing decentralized governance proposals. It combines blockchain interaction, proposal analysis, and secure voting mechanisms to provide a robust platform for DAO governance.

Key Features

- Proposal tracking and analysis
- Secure voting mechanism
- Historical data management
- AI-powered governance analytics
- Real-time blockchain integration

Architecture

High-Level Architecture

```
User Interface
      ↓
Application Layer (main.py)
      ↓
Core Services
├── Voting System (voting.py)
├── Proposal Analysis (proposal_analysis.py)
└── Blockchain Interface (blockchain.py)
      ↓
Storage Layer (Redis)
      ↓
External Services
├── Ethereum Network (via Infura)
└── Anthropic API
```

Core Components

ProposalAnalyzer (proposal_analysis.py)

The ProposalAnalyzer class handles proposal evaluation and analysis using AI integration.

Key responsibilities:

- Fetches proposal data from the blockchain
- Evaluates proposal metrics
- Generates AI-powered analysis
- Tracks proposal states and outcomes

VotingSystem (voting.py)

Manages the voting process and user interactions.

Features:

- Session management
- Vote submission and validation
- Historical data tracking
- User statistics

GovernorBravoContract (blockchain.py)

Handles all blockchain interactions.

Capabilities:

- Contract state reading
- Proposal data fetching
- State validation
- Event monitoring

Class Documentation

ProposalAnalyzer

```
class ProposalAnalyzer:
    def __init__(self):
        """
        Initializes the ProposalAnalyzer with required connections and configurations.

        Dependencies:
        - Anthropic API key
        - Web3 provider
        - Contract ABI
        """

    def evaluate_proposal(self, proposal):
        """
        Evaluates a single proposal based on various metrics.

        Parameters:
        - proposal (dict): Proposal data including votes and state

        Returns:
        - dict: Evaluation metrics including feasibility, impact, and risks
        """

    def analyze_proposals(self):
        """
        Analyzes all recent proposals and provides insights.

        Returns:
        - str: Detailed analysis of governance health and trends
        """
```

VotingSystem

```
class VotingSystem:
    def __init__(self, redis_client):
        """
        Initializes the voting system with Redis client for session management.

        Parameters:
        - redis_client: Redis client instance
        """


    def submit_vote(self, session_id: str, proposal_id: int, vote: str) -> str:
        """
        Submits a vote for a specific proposal.

        Parameters:
        - session_id: User session identifier
        - proposal_id: Target proposal ID
        - vote: Vote choice ('for', 'against', 'abstain')

        Returns:
        - str: Status message
        """
```

Data Flow

Voting Process

1. User initializes session with wallet address
2. System validates wallet and loads user history
3. User selects proposal and submits vote
4. Vote is validated and recorded
5. Statistics are updated
6. Transaction is logged

Proposal Analysis

1. System fetches proposal data from blockchain
2. Metrics are calculated
3. AI analysis is generated
4. Results are cached
5. Analysis is presented to users

API Reference

Environment Variables

```
INFURA_URL: Ethereum node provider URL
CONTRACT_ADDRESS: Governor contract address
ANTHROPIC_API_KEY: API key for AI analysis
REDIS_URL: Redis connection string
```

Redis Data Structure

```
wallet:{session_id} -> wallet_address
votes:{wallet_address} -> {
    'for': count,
    'against': count,
    'abstain': count
}
proposal:{proposal_id}:votes -> {
    wallet_address: vote_data
}
```

Deployment Guide

Docker Deployment

1. Build image:

```
docker build -t dao-voting-system .
```

2. Run container:

```
docker run -d \
    -p 8000:8000 \
    --env-file .env \
    dao-voting-system
```

Render Deployment

1. Push Docker image to hub

2. Create new web service on Render
3. Configure environment variables
4. Connect to GitHub repository
5. Deploy

Troubleshooting

Common Issues

1. Connection Errors

```
Error: Web3 connection failed
Solution: Verify INFURA_URL and network status
```

2. Redis Errors

```
Error: Redis connection refused
Solution: Check REDIS_URL and ensure service is running
```

3. Contract Interaction Failures

```
Error: Contract call reverted
Solution: Verify contract ABI and address
```

Logging

The system uses Python's logging module with INFO level by default:

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
```

Health Checks

- Redis connection status
- Web3 provider connection
- Contract accessibility
- API key validation

For additional support or questions, please open an issue on the GitHub repository.