Tymoteusz Piwowarski, Krzysztof Muszyński, Stipe Rimac

1. Overview & Architecture

This project implements an on-chain quadratic voting system for DAO proposals, fulfilling all requirements:

1. **VotingToken.sol**
   - ERC-20 token (OpenZeppelin) extended with owner-only mint and participant-only burn.
   - Manages the supply of voting tokens, each costing a fixed tokenPrice in wei.
2. **IExecutableProposal.sol**
   - Interface for external "action" contracts.
   - Defines executeProposal(uint proposalId, uint numVotes, uint numTokens) to be called when a proposal passes.
3. **QuadraticVoting.sol**
   - Core governance contract (inherits OpenZeppelin's ReentrancyGuard).
   - Manages participants, token sales/refunds, proposal lifecycle, vote staking, auto-execution, and final closing.

Key Data Structures & Flows

- **Participants** register at any time by sending Ether; they receive tokens proportional to msg.value / tokenPrice (with excess refunded).
- **Proposals** are created by participants while voting is open. They can be:
  - **Funding** (budget > 0): auto-approve when votes ≥ threshold and budget is available.
  - **Signaling** (budget = 0): tallied but executed only upon closeVoting().
- **Quadratic staking**: each additional vote on a proposal costs $(totalVotes)^2$ – $(prevVotes)^2$ tokens. Tokens are locked until votes are withdrawn or consumed by execution.
- **Dynamic threshold** for funding proposals:

$$threshold_i = (0{,}2 + \frac{budget_i}{totalbudget}) \cdot numParticipants + numPendingProposals$$

computed on each stake.

---

2. Detailed Design Decisions

2.1. Token Economics & ERC-20 Extension

- **OpenZeppelin's ERC-20** imported for battle-tested correctness.
- **Owner-only mint**, **participant-only burn**, to maintain supply and allow participants to exit.
- **Excess Ether refunds** in addParticipant/buyTokens avoid "dust" accumulation.

2.2. Proposal Representation

- A lightweight struct Proposal holds metadata and vote tallies within QuadraticVoting, optimizing gas over deploying a full contract per proposal.
- **External payloads** (the actual business logic when a proposal passes) live in separate contracts implementing IExecutableProposal, keeping governance data and execution logic decoupled.

2.3. Voting & Budget Accounting

- **lockedTokens mapping** tracks tokens staked in votes, preventing double-spend via sellTokens or removeParticipant.
- On **auto-approval**:
    1. Release locked tokens back to the budget (update lockedTokens and votingBudget).
    2. Transfer p.budget to external executeProposal call with a 100 000 gas cap.
    3. Update remaining votingBudget.

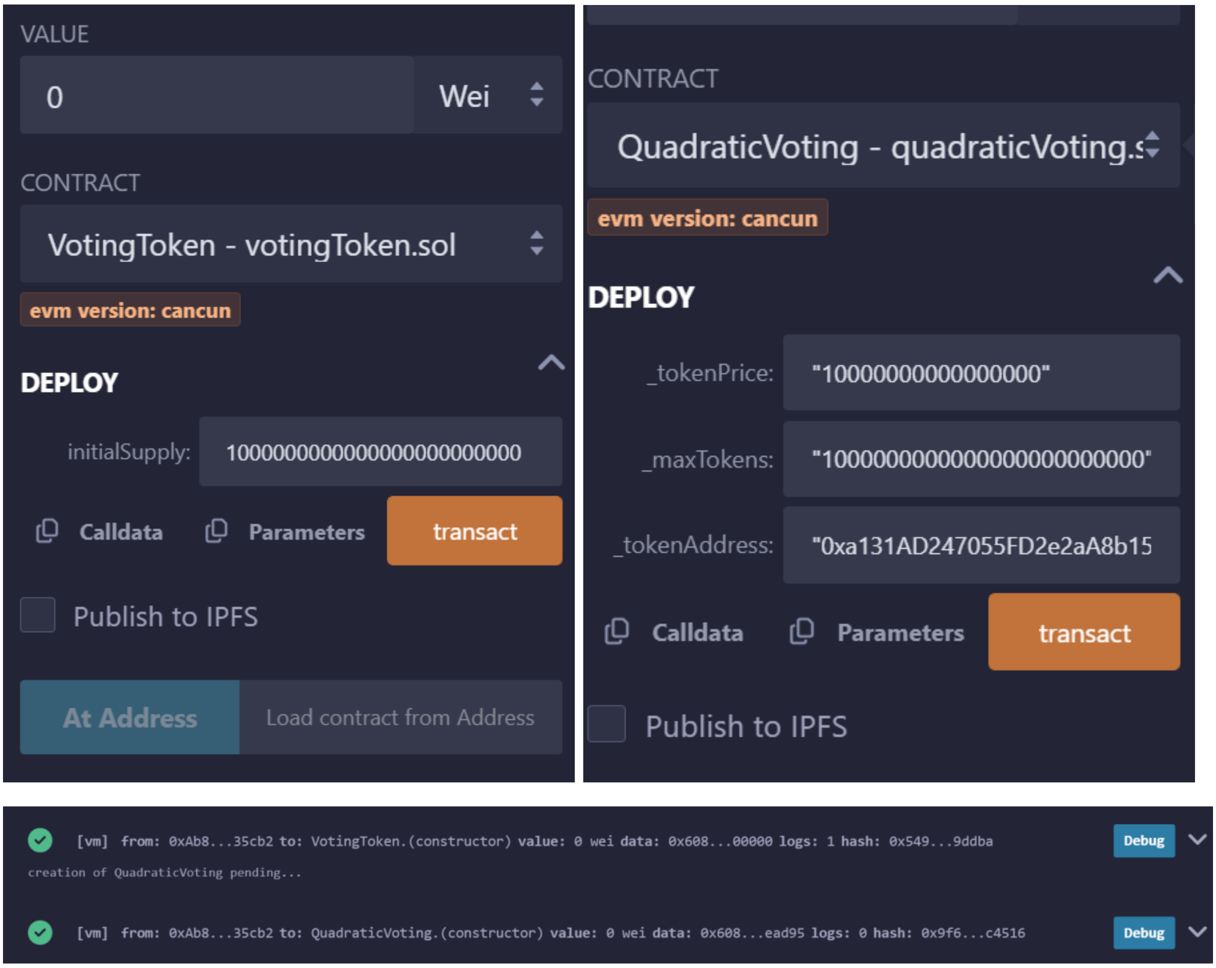
3. Solidity Best Practices & Resources

- **Modifiers** (onlyOwner, onlyParticipant, votingOpen) encapsulate common checks.
- **Inheritance** from OpenZeppelin's ReentrancyGuard for nonReentrant on all Ether-transferring functions.
- **Minimal external state**: only three contracts, each with a single well-defined responsibility.
- **Inline comments** document critical reasoning and guard conditions, aiding maintainability.

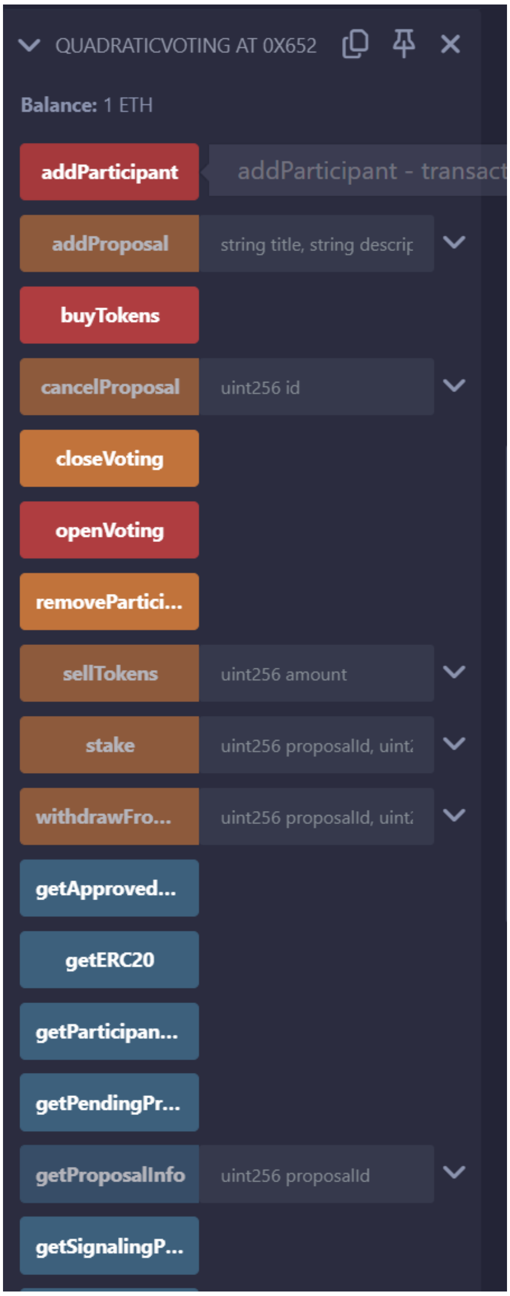
4. Security & Vulnerability Mitigation

Threat	Mitigation
Reentrancy attacks	All functions that transfer Ether or burn tokens (e.g. removeParticipant, sellTokens, _checkAndExecuteProposal) are protected with the nonReentrant modifier from OpenZeppelin's ReentrancyGuard.
Unexpected callbacks	External calls to executeProposal are restricted to 100,000 gas, minimizing the chance for state manipulation via callbacks.

Threat	Mitigation
DAO-style attack	Ether transfers occur after internal state updates. Proposal execution (_checkAndExecuteProposal) ensures proposal status and token burns are finalized *before* any external call.
Integer overflows/underflows	Solidity ^0.8.0 includes built-in overflow/underflow checks. No use of SafeMath is necessary then.
Parity wallet vulnerability	No use of delegatecall or external library contracts that could be uninitialized or destructible.



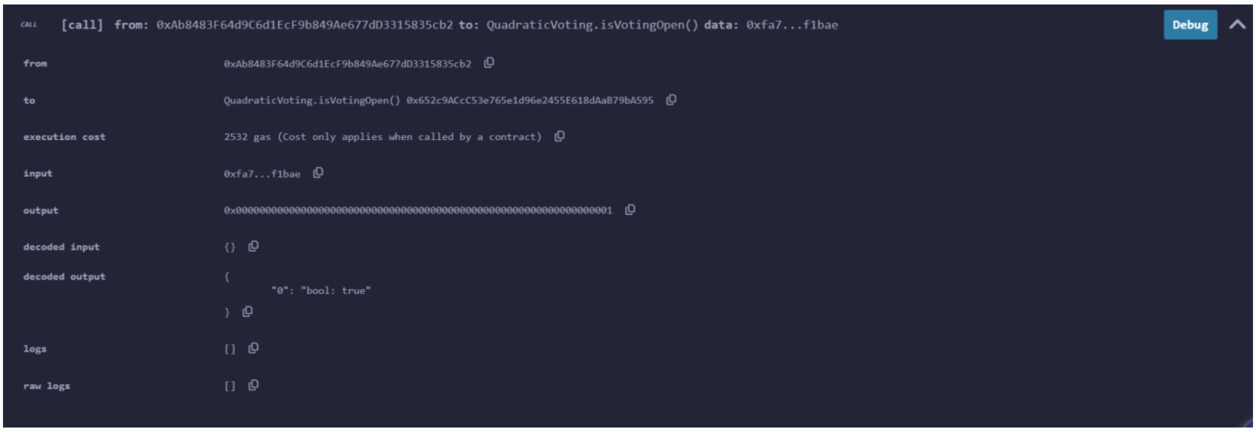
Deployed at first VotingToken contract and then QuadraticVoting contract.



Check if openVoting works

Voting is now opened

Let’s check isVotingOpened()

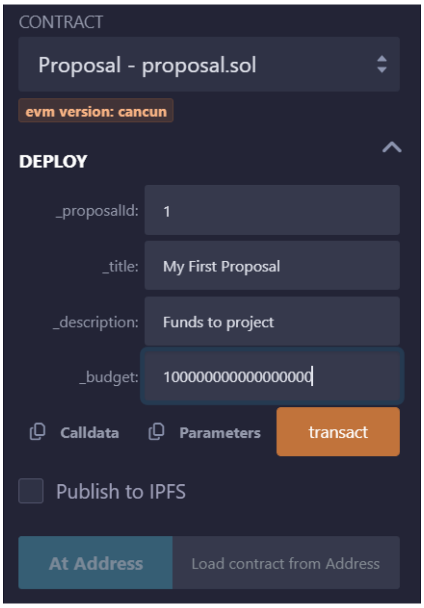


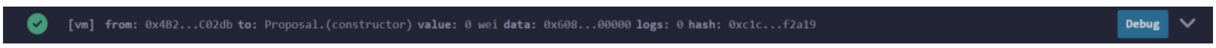
Bool: ‘true’ then voting is opened.

Adding participant

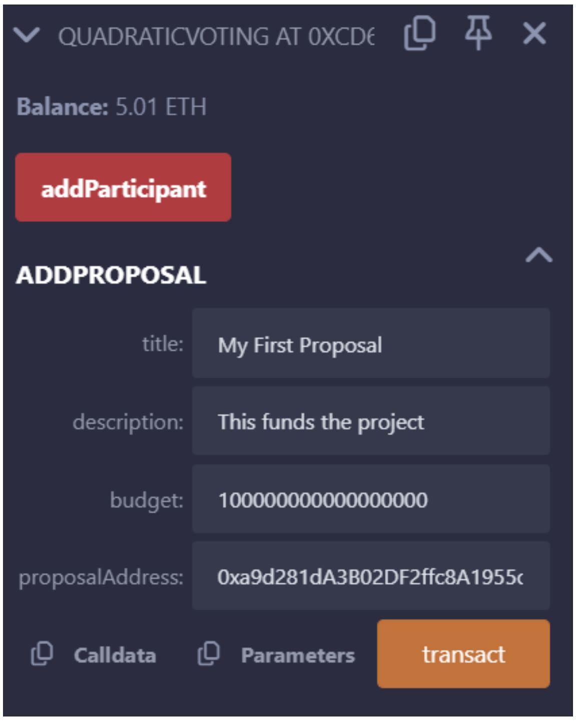
getParticipantCount works

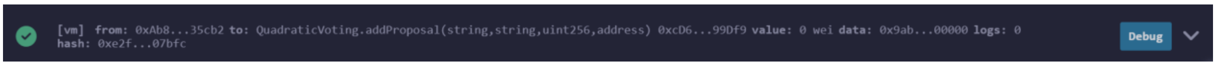
Deploying proposal





Adding deployed proposal to quadratic voting contract

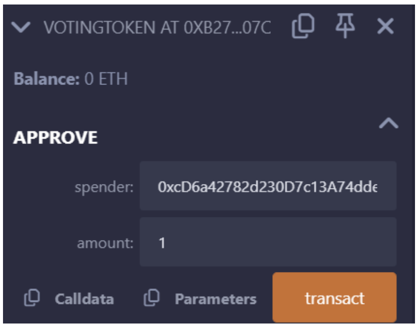




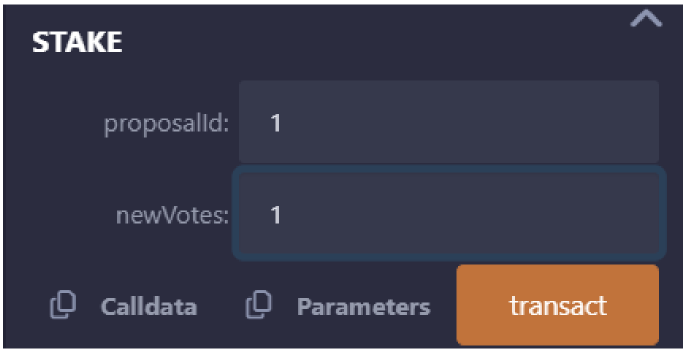
Proposal count

Proposal was added successfully

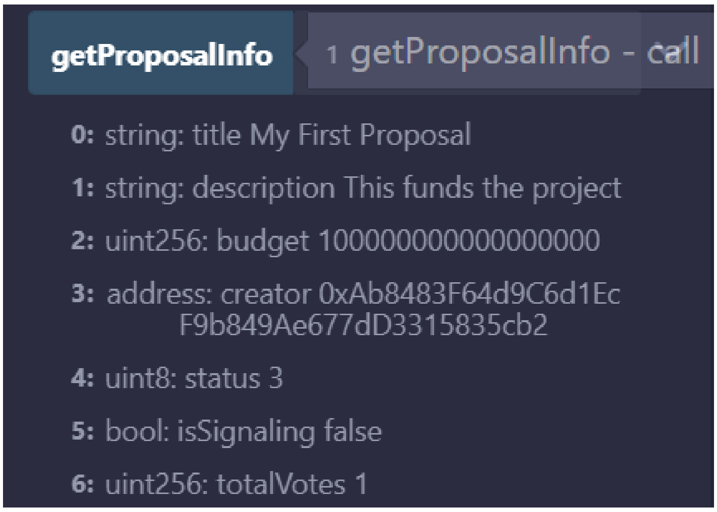
Approving proposal in voting token:



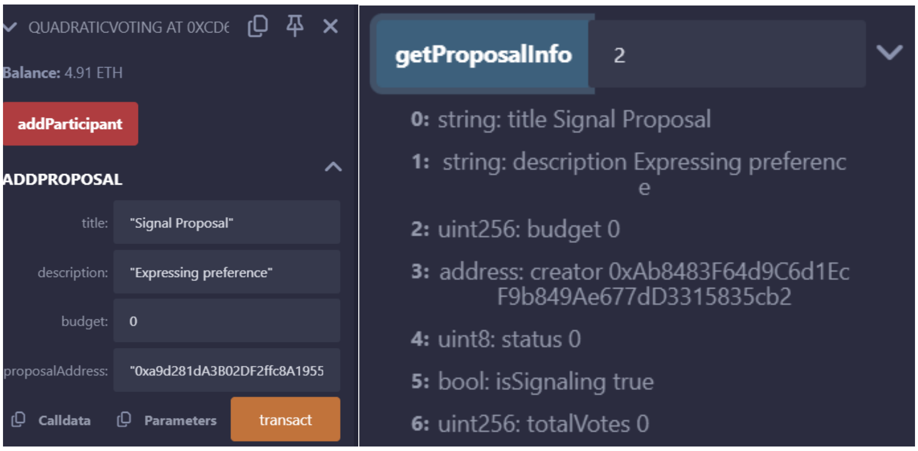
Now call stake function



getProposalInfo with proposal id = 1



Now let's try to check other type of proposal



I cancelled this proposal

This project successfully implements a secure, gas-efficient, and modular on-chain quadratic voting system for DAOs. Through careful use of OpenZeppelin libraries, explicit Ether/token management, and dynamic proposal handling, it ensures both fairness and safety in decentralized governance.