Architecture Documentation

System Overview

The Public Goods Liquidity Engine consists of three main components working together to create a sustainable funding mechanism for public goods.

```
┌─────────────────────────────────────────────────────────────┐
│                    User Deposits Assets                     │
└─────────────────────────────────────────────────────────────┘
                       │
                       │
                       ▼
┌─────────────────────────────────────────────────────────────┐
│              PublicGoodsVault (ERC-4626)                    │
│                                                             │
│  ┌──────────────────────────────────────────────────────┐  │
│  │ • Accepts deposits                                    │  │
│  │ • Issues vault shares to depositors                  │  │
│  │ • Generates yield from strategies                    │  │
│  │ • Harvests yield → mints new shares                  │  │
│  │ • Sends yield shares to allocation address           │  │
│  └──────────────────────────────────────────────────────┘  │
│                                                             │
└─────────────────────────────────────────────────────────────┘
                       │ Yield Shares
                       ▼
┌─────────────────────────────────────────────────────────────┐
│           QuadraticFundingSplitter                          │
│                                                             │
│  ┌──────────────────────────────────────────────────────┐  │
│  │ • Receives vault shares as yield                     │  │
│  │ • Manages funding rounds                             │  │
│  │ • Collects votes from community                      │  │
│  │ • Calculates quadratic funding distribution          │  │
│  │ • Distributes shares to projects                     │  │
│  └──────────────────────────────────────────────────────┘  │
│                                                             │
└─────────────────────────────────────────────────────────────┘
                       │ Distributed Shares
                       ▼
          ┌─────────────────────────────┐
          │   Public Goods Projects     │
          │  • Project A                │
          │  • Project B                │
          │  • Project C                │
          └─────────────────────────────┘
```

Component Details

PublicGoodsVault

Purpose: ERC-4626 compliant vault that generates yield and donates it to public goods

Key Functions:

1. **deposit(uint256 assets, address receiver)** → uint256 shares
   - User deposits underlying assets
   - Receives proportional vault shares
   - Shares represent claim on principal only
2. **withdraw(uint256 assets, address receiver, address owner)** → uint256 shares
   - User withdraws their principal
   - Burns corresponding vault shares
   - Yield shares remain with allocation address
3. **harvest()** → uint256 yieldAmount
   - Called by keeper when yield is available
   - Calculates yield = currentAssets - lastHarvestedAssets
   - Mints new shares equal to yield value
   - Transfers new shares to allocation address
   - Updates lastHarvestedAssets
4. **initializeHarvest()**
   - Sets initial baseline for yield calculation
   - Called once after first deposits

State Variables:

```
address public allocationAddress;     // Receives yield shares
address public keeper;                // Authorized to harvest
address public emergencyAdmin;        // Emergency controls
address public strategy;              // Yield generation strategy
uint256 public lastHarvestedAssets;   // Tracking for yield calc
uint256 public performanceFee;        // Protocol fee (max 10%)
address public feeRecipient;          // Fee destination
bool public paused;                   // Emergency pause
```

Access Control:

- Owner: Can update all configuration parameters
- Keeper: Can harvest yield and initialize
- Emergency Admin: Can pause/unpause and emergency withdraw
- Users: Can deposit/withdraw freely

QuadraticFundingSplitter

Purpose: Allocates yield shares to public goods projects using quadratic funding

Key Functions:

1. **registerProject(address recipient, string name, string description)** → uint256 projectId
    - Anyone can register a project
    - Creates new project entry
    - Returns unique project ID
2. **startRound(uint256 duration)**
    - Owner starts new funding round
    - Sets round duration
    - Activates voting
3. **addToMatchingPool(uint256 amount)**
    - Anyone can contribute to matching pool
    - Transfers vault shares to splitter
    - Increases current round's matching pool
4. **vote(uint256 projectId, uint256 amount)**
    - Community members vote with vault shares
    - Transfers shares from voter to splitter
    - Updates project vote totals and unique voter count
5. **endRound()**
    - Owner ends round after duration expires
    - Calculates quadratic scores for all projects
    - Distributes direct votes + matching funds
    - Formula: `score = sqrt(totalVotes) × uniqueVoters`

Quadratic Funding Math:

```
// For each project i:
avgContribution[i] = totalVotes[i] / uniqueVoters[i]
quadraticScore[i] = sqrt(totalVotes[i]) × uniqueVoters[i]

// Sum all scores
totalScore = Σ quadraticScore[i]

// Distribute matching pool proportionally
matchingAmount[i] = (quadraticScore[i] / totalScore) × matchingPool

// Final distribution to project
finalAmount[i] = totalVotes[i] + matchingAmount[i]
```

Why Quadratic Funding?

Traditional 1-person-1-vote: Easy to game (sybil attacks)
Traditional proportional: Whales dominate

Quadratic: Balances broad support vs. capital

Example:

- Project A: 3 voters × 10 tokens each = 30 tokens
    - QF Score: sqrt(30) × 3 ≈ 5.48 × 3 = 16.44
- Project B: 1 voter × 30 tokens = 30 tokens
    - QF Score: sqrt(30) × 1 ≈ 5.48 × 1 = 5.48

Project A gets ~3x matching multiplier despite same direct votes!

State Variables:

```
struct Project {
    address recipient;
    string name;
    string description;
    uint256 totalVotes;
    uint256 uniqueVoters;
    bool active;
    uint256 totalReceived;
}

mapping(uint256 => Project) public projects;
mapping(address => mapping(uint256 => Vote)) public votes;
uint256 public projectCount;
uint256 public currentRound;
mapping(uint256 => uint256) public matchingPools;
bool public roundActive;
```

Lifecycle Example

Complete Flow: Deposit → Yield → Vote → Distribute

Day 1: Setup

```
1. Deploy vault and splitter
2. Keeper initializes vault
3. Owner starts funding round (30 days)
```

Day 1-30: Deposits & Yield

```
4. Alice deposits 1000 USDC → receives 1000 vault shares
5. Bob deposits 500 USDC → receives 500 vault shares
6. Vault generates 50 USDC yield (5%)
7. Keeper harvests → 50 new shares minted to splitter
```

Day 5-30: Registration & Voting

```
8. Projects register:
   - Project A: Building dev tooling
   - Project B: Education initiative
   - Project C: Research grant

9. Community votes (using their own vault shares):
   - Alice: 10 shares → Project A
   - Bob: 10 shares → Project A
   - Charlie: 20 shares → Project B

10. Ecosystem adds matching pool:
    - Foundation adds 100 shares to matching pool
```

Day 30: Distribution

```
11. Round ends
12. Quadratic scores calculated:
    - Project A: sqrt(20) × 2 = 8.94
    - Project B: sqrt(20) × 1 = 4.47
    - Total score: 13.41

13. Matching distributed:
    - Project A: (8.94/13.41) × 100 = 66.7 shares
    - Project B: (4.47/13.41) × 100 = 33.3 shares

14. Final amounts:
    - Project A: 20 (direct) + 66.7 (matching) = 86.7 shares
    - Project B: 20 (direct) + 33.3 (matching) = 53.3 shares

15. Vault shares redeemed by projects for USDC
```

Security Considerations

Vault Security

- ReentrancyGuard on all state-changing functions
- Pause mechanism for emergencies
- Multi-role access control
- Safe ERC20 operations
- Harvest can only increase totalAssets, never decrease

Splitter Security

- Round-based system prevents manipulation
- Vote tracking ensures fair quadratic calculation

- Project deactivation for malicious actors
- Minimum vote amounts prevent dust attacks
- No withdrawal of votes (commitment mechanism)

Known Limitations

- Sybil resistance relies on share acquisition cost
- No delegation in current version
- Owner has significant control (could be DAO)
- Matching pool size affects game theory

Gas Optimization

Vault

- Minimal storage writes
- Efficient share calculation
- Batch harvesting encouraged

Splitter

- Quadratic calculation optimized
- Two-pass distribution (scoring then distribution)
- View functions for off-chain computation

Upgradeability

Current design is non-upgradeable for security. Future versions could:

- Use proxy patterns (UUPS/Transparent)
- Implement migration functions
- Add governance timelock

Integration Points

For Developers

```
// Integrate vault as yield source
IVault vault = IVault(vaultAddress);
asset.approve(address(vault), amount);
uint256 shares = vault.deposit(amount, address(this));

// Later withdraw
vault.withdraw(amount, address(this), address(this));
```

For Protocols

```
// Route protocol fees to vault
protocolToken.approve(address(vault), fees);
vault.deposit(fees, protocolTreasury);
// Fees generate yield for public goods while treasury retains shares
```

For Communities

```
// Run grants program
splitter.startRound(30 days);
// Community registers projects and votes
// End round distributes funds automatically
```

Future Architecture Enhancements

1. **Strategy Layer**
   - Pluggable yield strategies
   - Risk-adjusted returns
   - Multiple strategy support
2. **Cross-Chain**
   - Bridge vault shares
   - Multi-chain voting
   - Unified distribution
3. **Advanced Allocation**
   - Multiple allocation mechanisms
   - Customizable formulas
   - Streaming distributions
4. **Governance**
   - DAO-controlled parameters
   - Veto system
   - Progressive decentralization