Lux DAO: Modular Governance Framework for Decentralized Organizations

A Comprehensive Analysis of Azorius-Based Governance with Account Abstraction

Lux Industries Inc
research@lux.network

Initial Version: v2022.10 (October 2022)
Major Revision: v2024.06 (June 2024)

October 28, 2025

Abstract

We present Lux DAO, a production-grade modular governance framework built on the Azorius Protocol, designed for flexible and scalable decentralized autonomous organization (DAO) management. Deployed in October 2022 and significantly enhanced in June 2024, Lux DAO addresses critical challenges in blockchain governance through a composable architecture that separates concerns between proposal management, voting strategies, and execution mechanisms. The framework integrates cutting-edge technologies including ERC-4337 account abstraction for gasless voting, Hats Protocol for role-based access control, ERC-6551 token-bound accounts for payment streaming, and the Zodiac module pattern for Safe integration. Our implementation has been battle-tested across multiple networks (Ethereum, Optimism, Polygon, Base, Sepolia) and demonstrates significant improvements in governance flexibility, user experience, and operational efficiency compared to existing solutions like Aragon, Compound Governor, and Moloch DAO. This paper provides a comprehensive technical analysis of the architecture, presents novel contributions to the DAO governance space, and evaluates real-world deployment outcomes.

1 Introduction

1.1 The Challenge of DAO Governance

Decentralized Autonomous Organizations (DAOs) represent a fundamental shift in how collective decision-making can be coordinated on blockchain networks. However, first-generation DAO frameworks suffer from several critical limitations:

- **Rigidity**: Monolithic architectures that couple voting mechanisms with execution logic, making it difficult to adapt governance rules without complete system upgrades.
- **Gas Costs**: Participation barriers created by transaction fees, particularly affecting smaller stakeholders and reducing overall voter turnout.
- **Limited Modularity**: Inability to mix different voting strategies (token-weighted, NFT-based, role-based) within a single governance system.
- **Poor Hierarchical Support**: Lack of native support for parent-child DAO relationships and emergency intervention mechanisms.
- **Upgrade Complexity**: Difficulty in evolving governance systems without compromising security or requiring contentious migrations.

1.2 Lux DAO: A Modular Solution

Lux DAO addresses these challenges through a composable architecture built on three foundational principles:

Separation of Concerns The framework cleanly separates proposal lifecycle management (Azorius module), voting logic (Strategy contracts), and execution (Gnosis Safe), enabling independent evolution of each component.

Account Abstraction Integration By leveraging ERC-4337, Lux DAO enables truly gasless voting through paymaster contracts that sponsor transaction fees, dramatically improving accessibility without compromising security.

Advanced Role Management Integration with Hats Protocol and ERC-6551 provides sophisticated role hierarchies, automated compensation streams, and token-bound account functionality for organizational management.

1.3 Timeline and Evolution

- **October 2022**: Initial deployment of core Azorius-based governance framework
- **June 2024**: Major architectural revision introducing:
  - ERC-4337 account abstraction for gasless voting
  - Enhanced Hats Protocol integration (DecentHats)
  - ERC-6551 token-bound accounts for payment streaming

- Improved freeze guard mechanisms
- Production-grade paymaster infrastructure

1.4 Paper Contributions

This paper makes the following contributions:

1. Comprehensive technical specification of the Azorius modular governance architecture
2. Novel integration pattern for ERC-4337 account abstraction in DAO voting systems
3. Production-tested implementation of hierarchical DAO structures with emergency intervention
4. Comparative analysis with existing governance frameworks (Aragon, Compound, Moloch)
5. Empirical evaluation of deployment outcomes across multiple blockchain networks
6. Open-source reference implementation for production DAO deployment

2 Background and Related Work

2.1 Existing DAO Frameworks

2.1.1 Aragon

Aragon [1] pioneered modular DAO frameworks with its Agent-based architecture. However, its approach couples voting and execution more tightly than Azorius, and lacks native support for account abstraction. Aragon's upgrade mechanism relies on proxy patterns that can introduce security complexities.

2.1.2 Compound Governor

Compound's Governor contract [2] established the standard for on-chain governance in DeFi protocols. While elegant, its monolithic design makes it difficult to swap voting strategies or implement complex hierarchical relationships. Gas costs for voting remain prohibitive for smaller token holders.

2.1.3 Moloch DAO

Moloch [3] introduced the concept of "ragequit" for minority protection, focusing on simplicity and security. However, its fixed governance model (one-member-one-vote with guild shares) limits applicability to diverse organizational structures. The lack of modularity prevents experimentation with alternative voting mechanisms.

2.1.4 DAOstack

DAOstack [4] explored holographic consensus and reputation-based voting. While innovative, its complexity and gas costs hindered widespread adoption. The framework's tight coupling between components made it difficult to adopt incrementally.

2.2 Account Abstraction (ERC-4337)

ERC-4337 [5] introduced a standard for account abstraction without requiring protocol-level changes. Key components include:

- **UserOperations**: Transaction-like objects signed by smart contract accounts
- **Bundlers**: Off-chain services that bundle UserOperations into transactions
- **EntryPoint**: Singleton contract that processes UserOperations
- **Paymasters**: Contracts that sponsor gas fees for UserOperations

While ERC-4337 has been adopted for wallet infrastructure, its application to DAO governance systems is novel. Lux DAO represents one of the first production implementations of account abstraction for voting mechanisms.

2.3 Gnosis Safe and Zodiac

Gnosis Safe [6] provides battle-tested multisig infrastructure with a modular architecture. The Zodiac framework [7] extends Safe with standardized module interfaces, enabling composable governance tools. Azorius builds on this foundation, implementing the Zodiac pattern for seamless Safe integration.

2.4 Hats Protocol

Hats Protocol [8] provides on-chain organizational structures through a tree-based hierarchy of roles ("hats"). Each hat represents permissions and responsibilities within an organization. Lux DAO's integration enables sophisticated role-based governance with automated management.

2.5 ERC-6551: Token-Bound Accounts

ERC-6551 [9] enables any ERC-721 NFT to own assets and interact with contracts. By associating each role (hat) with a token-bound account, Lux DAO enables automated payment streaming to role holders, solving the compensation coordination problem in decentralized organizations.

3 Azorius Framework Architecture

3.1 Design Principles

The Azorius framework is built on four core design principles:

Modularity Each component has a single responsibility and communicates through well-defined interfaces. This enables independent development, testing, and deployment of governance features.

Extensibility New voting strategies, proposer adapters, and freeze mechanisms can be added without modifying core contracts. The system is designed for evolution.

Security by Composition Rather than implementing all functionality in a monolithic contract, Azorius delegates to specialized, audited components. This reduces attack surface and enables formal verification.

Zodiac Compatibility Full adherence to the Zodiac module standard ensures compatibility with existing Safe tooling and future Zodiac extensions.

3.2 Core Components

3.2.1 ModuleAzoriusV1: The Proposal Manager

ModuleAzoriusV1 serves as the central coordinator for all governance activities. Key responsibilities include:

Listing 1: ModuleAzoriusV1 Core Structure

```
contract ModuleAzoriusV1 is
    IModuleAzoriusV1,
    GuardableModule,
    Ownable2StepUpgradeable,
    UUPSUpgradeable,
    ERC165
{
    struct Proposal {
        address strategy;        // Voting strategy contract
        Transaction[] txs;       // Executable transactions
        uint32 timelockPeriod;  // Delay before execution
```

```
        uint32 executionPeriod; // Window for execution
        uint256 executedTxCount; // Partial execution tracking
    }

    mapping(uint32 => Proposal) proposals;
    uint32 totalProposalCount;
    uint32 defaultTimelockPeriod;
    uint32 defaultExecutionPeriod;
    IStrategyV1 defaultStrategy;
}
```

Proposal Lifecycle

1. **Submission**: Any authorized proposer can submit a proposal with transactions and metadata
2. **Voting Initialization**: The specified strategy initializes voting parameters
3. **Voting Period**: Token holders cast votes through the strategy
4. **Timelock**: Passed proposals wait during the timelock period
5. **Execution Window**: Proposals can be executed during this period
6. **Execution**: Transactions are executed through Safe's `execTransactionFromModule()`

Partial Execution A critical innovation in Azorius is support for partial proposal execution. If some transactions in a proposal fail, successful transactions can still be executed:

Listing 2: Partial Execution Logic

```
function executeProposal(uint32 proposalId, bytes32[] calldata
    txHashes)
    external
{
    Proposal storage proposal = proposals[proposalId];

    for (uint256 i = 0; i < txHashes.length; i++) {
        if (!_transactionExecuted(proposal, txHashes[i])) {
            bool success = _executeTransaction(proposal.txs[i])
                ;
            if (success) {
                proposal.executedTxCount++;
                emit ProposalTxExecuted(proposalId, txHashes[i
                    ]);
            }
        }
    }
}
```

This design enables gas-efficient execution and graceful handling of failures.

3.2.2 StrategyV1: The Voting Engine

StrategyV1 implements the voting logic and rules. It delegates vote weight calculation and vote recording to pluggable adapters:

Listing 3: StrategyV1 Architecture

```
contract StrategyV1 is
    IStrategyV1,
    LightAccountValidator,
    ERC165
{
    struct ProposalVotingDetails {
        uint256 yesVotes;
        uint256 noVotes;
        uint256 abstainVotes;
        uint32 startBlock;
        uint32 endBlock;
    }

    VotingConfig[] votingConfigs;
    address[] proposerAdapters;
    uint32 votingPeriod;
    uint256 quorumThreshold;
    uint256 basisNumerator; // e.g., 500000 = 50%

    mapping(uint32 => ProposalVotingDetails)
        proposalVotingDetails;
}
```

Voting Configurations Each StrategyV1 can have multiple voting configurations, enabling hybrid voting systems:

Listing 4: Voting Configuration Structure

```
struct VotingConfig {
    address votingWeight;     // Weight calculator (ERC20,
        ERC721)
    address voteTracker;      // Vote recording mechanism
    uint256 weight;           // Proportional weight in
        decision
}
```

For example, a DAO might use:

- 70% weight from ERC20 token holdings
- 20% weight from ERC721 NFT ownership
- 10% weight from Hats Protocol roles

Algorithm 1 Vote Tallying and Approval

function IsProposalPassed($proposalId$)
  $details \leftarrow proposalVotingDetails[proposalId]$
  $totalVotes \leftarrow details.yesVotes + details.noVotes + details.abstainVotes$
  if $totalVotes < quorumThreshold$ **then**
    return `false`
  end if
  $yesPercent \leftarrow \frac{details.yesVotes \times 1000000}{totalVotes}$
  return $yesPercent \geq basisNumerator$
end function

Quorum and Approval Calculation

3.2.3 Voting Adapters

Voting adapters implement two key interfaces:

VotingWeight Interface Calculates voting power for an address:

Listing 5: VotingWeight Interface

```
interface IVotingWeightV1 {
    function getVotingWeight(
        address voter,
        uint32 snapshotBlock
    ) external view returns (uint256);
}
```

Example implementations:

- **VotingWeightERC20V1**: Returns token balance at snapshot block
- **VotingWeightERC721V1**: Returns NFT count at snapshot block

VoteTracker Interface Records and retrieves votes:

Listing 6: VoteTracker Interface

```
interface IVoteTrackerV1 {
    function castVote(
        uint32 proposalId,
        address voter,
        uint8 voteType,
        uint256 weight
    ) external;

    function hasVoted(
        uint32 proposalId,
        address voter
```

```
    ) external view returns (bool);
}
```

3.2.4 Proposer Adapters

Proposer adapters control who can create proposals. Multiple adapters can be active simultaneously:

Listing 7: Proposer Adapter Examples

```
// Token-based proposing
contract ProposerAdapterERC20V1 {
    uint256 public requiredBalance;

    function isProposer(address account)
        external view returns (bool)
    {
        return token.balanceOf(account) >= requiredBalance;
    }
}

// Role-based proposing via Hats Protocol
contract ProposerAdapterHatsV1 {
    uint256 public requiredHatId;

    function isProposer(address account)
        external view returns (bool)
    {
        return hatsProtocol.isWearerOfHat(account,
            requiredHatId);
    }
}
```

3.3 Parent-Child DAO Hierarchies

3.3.1 ModuleFractalV1

ModuleFractalV1 enables parent DAOs to execute transactions on child DAOs, creating organizational hierarchies:

Listing 8: Parent-Child Relationship

```
contract ModuleFractalV1 is Module {
    // Installed on child DAO's Safe
    // Owned by parent DAO's Safe

    function executeTransaction(
        address to,
        uint256 value,
        bytes calldata data
    ) external onlyOwner {
        // Execute on child's Safe
```

```
        exec(to, value, data, Enum.Operation.Call);
    }
}
```

Use cases include:

- Emergency intervention by parent DAO
- Coordinated multi-DAO operations
- Hierarchical treasury management
- Organizational restructuring

3.3.2 Freeze Mechanism

The freeze mechanism provides a safety valve for parent DAOs to temporarily halt child DAO operations:

Listing 9: Freeze Guard Architecture

```
contract FreezeGuardAzoriusV1 is BaseGuard {
    mapping(address => bool) public isFrozen;

    function checkTransaction(
        address,
        uint256,
        bytes memory,
        Enum.Operation,
        uint256,
        uint256,
        uint256,
        address,
        address payable,
        bytes memory,
        address
    ) external view override {
        require(!isFrozen[msg.sender], "DAO is frozen");
    }
}
```

Freeze Voting Workflow

1. Parent DAO token holders initiate freeze vote via FreezeVotingV1
2. If threshold reached, child DAO is frozen via FreezeGuard
3. Child DAO cannot execute any transactions while frozen
4. Parent DAO executes corrective actions via ModuleFractalV1
5. Parent DAO unfreezes child or freeze expires automatically

This mechanism provides crucial safety for organizational hierarchies without requiring trust in a centralized administrator.

4 Account Abstraction for Gasless Voting

4.1 Motivation

Traditional on-chain voting requires users to pay gas fees for every vote transaction. This creates significant barriers:

- **Economic Exclusion**: Small token holders may find gas costs exceed their stake's influence
- **Reduced Participation**: High gas fees during network congestion discourage voting
- **Timing Issues**: Users without ETH for gas cannot vote, even if they hold governance tokens
- **Multi-Chain Complexity**: Each chain requires native tokens for gas, complicating cross-chain governance

ERC-4337 account abstraction enables DAOs to sponsor gas fees for their members, removing these barriers while maintaining security and decentralization.

4.2 Architecture Overview

Lux DAO's gasless voting system consists of several integrated components:

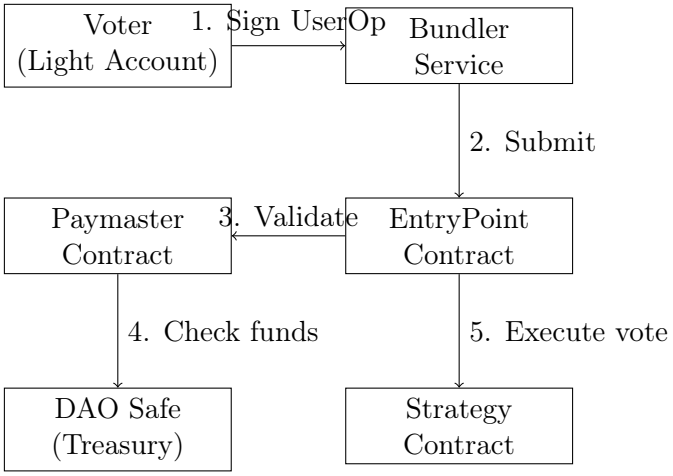

Figure 1: Gasless Voting Transaction Flow

4.3 PaymasterV1 Implementation

4.3.1 Core Functionality

Listing 10: Paymaster Core Structure

```
contract PaymasterV1 is BasePaymaster, Ownable2Step {
    // Function-specific validators
    mapping(address => mapping(bytes4 => address))
        public validators;

    // Deposit info from EntryPoint
    struct DepositInfo {
        uint256 balance;         // Available for gas
        uint256 stake;           // Locked for security
        bool staked;             // Is stake locked?
        uint256 withdrawTime;    // When stake is unlockable
    }

    function setFunctionValidator(
        address target,
        bytes4 selector,
        address validator
    ) external onlyOwner {
        validators[target][selector] = validator;
    }
}
```

4.3.2 Validation Flow

Algorithm 2 Paymaster Validation

function ValidatePaymasterUserOp($userOp, maxCost$)
  $callData \leftarrow userOp.callData$
  $(target, selector) \leftarrow$ ParseCallData($callData$)
  $validator \leftarrow validators[target][selector]$
Require: $validator \neq$ `address(0)`
  $validationData \leftarrow$ validator.ValidateSignature($userOp$)
  if validation fails **then**
    return `SIG_VALIDATION_FAILED`
  end if
  return $(validUntil, validAfter)$
end function

4.3.3 Strategy Validators

Each voting strategy has a corresponding validator that verifies votes:

Listing 11: StrategyV1 Validator

```
contract StrategyV1ValidatorV1 {
    function validateSignature(
        UserOperation calldata userOp
    ) external view returns (uint256 validationData) {
        // Decode vote parameters from callData
        (uint32 proposalId, uint8 voteType) =
            abi.decode(userOp.callData[4:], (uint32, uint8));

        // Verify proposal is active
        require(strategy.isProposalActive(proposalId),
                "Proposal not active");

        // Verify user hasn't voted
        require(!strategy.hasVoted(proposalId, userOp.sender),
                "Already voted");

        // Verify vote type is valid
        require(voteType <= 2, "Invalid vote type");

        return 0; // Success
    }
}
```

4.4 Light Account Integration

Light Accounts [11] provide a minimal ERC-4337 account implementation optimized for gas efficiency:

Listing 12: Light Account Structure

```
contract LightAccount {
    address public owner;

    function validateUserOp(
        UserOperation calldata userOp,
        bytes32 userOpHash,
        uint256 missingAccountFunds
    ) external returns (uint256 validationData) {
        // Verify signature from owner
        bytes32 hash = userOpHash.toEthSignedMessageHash();
        address signer = hash.recover(userOp.signature);

        if (signer != owner) {
            return SIG_VALIDATION_FAILED;
        }

        // Pay prefund if needed
        if (missingAccountFunds > 0) {
            (bool success,) = payable(msg.sender).call{
                value: missingAccountFunds
            }("");
            require(success);
```

```
        }

        return 0;
    }
}
```

4.5 Deposit Management

The Paymaster maintains funds in the EntryPoint for gas sponsorship:

Listing 13: Deposit Management

```
// Fund the paymaster
function depositTo(address paymaster, uint256 amount)
    external payable
{
    entryPoint.depositTo{value: amount}(paymaster);
}

// Stake for security (required by bundlers)
function addStake(uint32 unstakeDelaySec)
    external payable onlyOwner
{
    entryPoint.addStake{value: msg.value}(unstakeDelaySec);
}

// Withdraw available balance
function withdrawTo(address recipient, uint256 amount)
    external onlyOwner
{
    entryPoint.withdrawTo(payable(recipient), amount);
}

// Begin stake withdrawal
function unlockStake() external onlyOwner {
    entryPoint.unlockStake();
}

// Complete stake withdrawal after cooldown
function withdrawStake(address recipient)
    external onlyOwner
{
    entryPoint.withdrawStake(payable(recipient));
}
```

4.6 Security Considerations

Stake Requirements Bundlers require paymasters to stake funds as a security deposit. This prevents DoS attacks where malicious paymasters approve invalid operations. Typical minimum stake: 0.01 ETH with 1-day unlock period.

Validator Whitelisting By requiring explicit validator registration per function, the paymaster ensures only authorized operations are sponsored. This prevents abuse where users attempt to sponsor arbitrary transactions.

Deposit Monitoring DAOs must monitor paymaster deposits to ensure sufficient funds for voting. The system gracefully degrades to standard voting if deposits are exhausted, preventing governance failure.

5 Role Management with Hats Protocol and ERC-6551

5.1 Organizational Structures

Modern DAOs require sophisticated organizational structures beyond simple token-weighted voting. Lux DAO integrates Hats Protocol to provide:

- Hierarchical role definitions
- Permission-based access control
- Automated role assignment and revocation
- Integration with governance mechanisms

5.2 Hats Tree Architecture

5.2.1 Hat Hierarchy

Each DAO creates a "Hats tree" with the following structure:

```
Top Hat (DAO Safe)
 Admin Hat (Autonomous Admin)
    Engineering Lead
       Senior Engineer
       Junior Engineer
    Operations Lead
       Community Manager
       Support Specialist
    Finance Lead
        Treasurer
        Accountant
```

5.2.2 Hat Properties

Listing 14: Hat Structure

```
struct Hat {
    uint256 id;               // Unique identifier
    string details;           // IPFS metadata
    uint32 maxSupply;         // Max wearers
    address eligibility;      // Eligibility module
    address toggle;           // Active/inactive module
    bool mutable;             // Can details change?
    string imageURI;          // Visual representation
}
```

5.3 UtilityRolesManagementV1

This utility contract provides a unified interface for creating and managing Hats trees:

Listing 15: Creating a Hats Tree

```
struct CreateTreeParams {
    IHats hatsProtocol;
    IERC6551Registry erc6551Registry;
    address hatsAccountImplementation;
    ISablierV2LockupLinear sablierLockup;

    TopHatParams topHat;
    AdminHatParams adminHat;
    RoleHatParams[] roleHats;
}

function createAndDeclareTree(
    CreateTreeParams calldata params
) external onlyDelegatecall {
    // 1. Mint top hat to DAO Safe
    uint256 topHatId = _processTopHat(...);

    // 2. Create and mint admin hat
    uint256 adminHatId = _processAdminHat(...);

    // 3. Deploy autonomous admin for automation
    address autonomousAdmin = _deployAutonomousAdmin(...);

    // 4. Create role hats with payment streams
    _processRoleHats(...);

    // 5. Associate tree with Safe in KeyValuePairs
    keyValuePairs.updateValues(
        keys,
        [topHatId.toString(), ...]
    );
}
```

5.4 Token-Bound Accounts (ERC-6551)

5.4.1 Motivation

Traditional role-based payment requires:

1. Tracking who holds each role
2. Manually updating payment recipients when roles change
3. Complex escrow mechanisms for role transitions

ERC-6551 elegantly solves this by making the role (NFT) itself own an account.

5.4.2 Account Creation

Listing 16: Token-Bound Account Creation

```
function createAccount(
    address implementation,
    uint256 chainId,
    address tokenContract,
    uint256 tokenId,
    uint256 salt,
    bytes calldata initData
) external returns (address account) {
    // Deterministic address calculation
    account = _computeAddress(
        implementation,
        chainId,
        tokenContract,
        tokenId,
        salt
    );

    // Deploy via CREATE2
    if (account.code.length == 0) {
        bytes memory code = abi.encodePacked(
            type(ERC1167).creationCode,
            abi.encode(implementation)
        );

        assembly {
            account := create2(0, add(code, 32),
                               mload(code), salt)
        }

        // Initialize
        IERC6551Executable(account).execute(
            account, 0, initData, 0
        );
    }
```

```
35
36      return account;
37  }
```

5.4.3 Payment Streaming Integration

Each role hat has an associated ERC-6551 account that receives Sablier payment streams:

Listing 17: Role with Payment Stream

```
struct RoleHatParams {
    HatParams hat;
    SablierStreamParams[] streams;
}

function _createPaymentStream(
    address recipient,      // Token-bound account
    uint128 totalAmount,
    uint40 startTime,
    uint40 endTime
) internal returns (uint256 streamId) {
    LockupLinear.CreateWithRange memory params =
        LockupLinear.CreateWithRange({
            sender: address(this),          // DAO Safe
            recipient: recipient,           // TBA
            totalAmount: totalAmount,
            asset: paymentToken,
            cancelable: true,
            range: LockupLinear.Range({
                start: startTime,
                cliff: 0,
                end: endTime
            }),
            broker: Broker(address(0), 0)
        });

    streamId = sablier.createWithRange(params);
}
```

Benefits

- **Atomic Role Transfer**: When hat ownership changes, payment automatically redirects
- **Simplified Management**: No need to update payment recipients manually
- **Account Functionality**: Role holders can use their TBA for on-chain actions

- **Programmable Compensation**: Stream parameters can be adjusted via governance

5.5 Autonomous Admin

The Autonomous Admin contract automates role management operations:

Listing 18: Autonomous Admin Structure

```
contract AutonomousAdminV1 {
    IHats public hatsProtocol;
    uint256 public adminHatId;

    // Called by DAO proposals
    function createHat(
        uint256 parentHatId,
        string calldata details,
        uint32 maxSupply
    ) external onlyOwner returns (uint256 hatId) {
        hatId = hatsProtocol.createHat(
            parentHatId,
            details,
            maxSupply,
            address(0),  // No eligibility
            address(0),  // No toggle
            true,        // Mutable
            ""           // No image
        );
    }

    function mintHat(uint256 hatId, address wearer)
        external onlyOwner
    {
        hatsProtocol.mintHat(hatId, wearer);
    }

    function transferHat(
        uint256 hatId,
        address from,
        address to
    ) external onlyOwner {
        hatsProtocol.transferHat(hatId, from, to);
    }
}
```

This enables DAO proposals to directly manage organizational structure without requiring the DAO Safe to wear the admin hat.

6 Smart Contract Architecture

6.1 Contract Categories

Lux DAO organizes contracts into four categories based on deployment patterns:

6.1.1 Deployables

Deployed once per DAO, hold state, owned by DAOs:

- ModuleAzoriusV1
- StrategyV1
- VotingAdapters (ERC20, ERC721)
- ProposerAdapters (Token, Hats)
- PaymasterV1
- FreezeGuardAzoriusV1
- FreezeVotingV1

6.1.2 Singletons

Deployed once per chain, called by client applications:

- SystemDeployerV1
- KeyValuePairsV1

6.1.3 Utilities

Deployed once per chain, called via delegatecall from Safe:

- UtilityRolesManagementV1

6.1.4 Services

Deployed once per chain, referenced by multiple DAO contracts:

- StrategyV1ValidatorV1
- KYCVerifierV1

6.2 Storage Patterns

6.2.1 EIP-7201 Namespaced Storage

All upgradeable contracts use EIP-7201 [10] for storage isolation:

Listing 19: Namespaced Storage Pattern

```
contract ModuleAzoriusV1 {
    struct ModuleAzoriusStorage {
        uint32 totalProposalCount;
        uint32 timelockPeriod;
        uint32 executionPeriod;
        mapping(uint32 => Proposal) proposals;
        IStrategyV1 strategy;
    }

    // keccak256(abi.encode(uint256(keccak256(
    //   "DAO.ModuleAzorius.main")) - 1))
    //   & ~bytes32(uint256(0xff))
    bytes32 constant STORAGE_LOCATION =
        0
            xedd394c11bb1dac1602ad0766d0e03cc697fdaf9a9996bf169d40a2c3b6fa100
            ;

    function _getStorage()
        internal pure
        returns (ModuleAzoriusStorage storage $)
    {
        assembly {
            $.slot := STORAGE_LOCATION
        }
    }
}
```

This prevents storage collisions in upgradeable contracts and enables safe evolution of contract logic.

6.2.2 Delegatecall Safety

Utilities that must be called via delegatecall use a safety pattern:

Listing 20: Delegatecall Detection

```
contract UtilityRolesManagementV1 {
    address private immutable UTILITY_ADDRESS;

    constructor() {
        UTILITY_ADDRESS = address(this);
    }

    modifier onlyDelegatecall() {
        require(address(this) != UTILITY_ADDRESS,
                "Must␣be␣called␣via␣delegatecall");
        _;
```

```
    }

    function createAndDeclareTree(...)
        external onlyDelegatecall
    {
        // When called via delegatecall from Safe,
        // address(this) == Safe address
        // All operations execute with Safe's permissions
    }
}
```

6.3 Upgradeability Strategy

6.3.1 UUPS Pattern

Core governance contracts use UUPS (Universal Upgradeable Proxy Standard):

Listing 21: UUPS Upgrade Authorization

```
contract ModuleAzoriusV1 is UUPSUpgradeable, Ownable {
    function _authorizeUpgrade(address newImplementation)
        internal
        override
        onlyOwner
    {}
}
```

Advantages over Transparent Proxies

- Lower deployment cost (no proxy admin contract)
- Lower gas costs (no admin delegation logic)
- Upgrade logic stored in implementation (more flexible)
- Owner (DAO Safe) controls upgrades directly

6.3.2 Immutable Components

Strategy and adapter contracts are intentionally immutable:

- Simpler security model
- Lower gas costs
- Easier to reason about behavior
- Can be replaced by deploying new versions

DAOs can switch to new strategies/adapters by updating Azorius configuration via governance proposal.

6.4 Deployment System

6.4.1 SystemDeployerV1

Orchestrates DAO creation in a single transaction:

Listing 22: System Deployment

```
function deployDAO(
    DAOParams calldata params
) external returns (address safe) {
    // 1. Deploy Safe (via Safe factory)
    safe = _deploySafe(params.owners, params.threshold);

    // 2. Deploy Azorius module
    address azorius = _deployAzorius(safe);

    // 3. Deploy Strategy
    address strategy = _deployStrategy(
        azorius,
        params.votingPeriod,
        params.quorum
    );

    // 4. Deploy voting adapters
    _deployVotingAdapters(strategy, params.tokens);

    // 5. Deploy proposer adapters
    _deployProposerAdapters(strategy, params.proposers);

    // 6. Enable Azorius module on Safe
    _enableModule(safe, azorius);

    // 7. Record deployment in KeyValuePairs
    _recordDeployment(safe, azorius, strategy);

    emit DAODeployed(safe, azorius, strategy);
}
```

6.4.2 CREATE2 Deterministic Deployment

All contracts use CREATE2 for deterministic addresses:

Listing 23: Deterministic Deployment

```
function deployDeterministic(
    bytes memory bytecode,
    bytes32 salt
) internal returns (address addr) {
    assembly {
        addr := create2(
            0,
            add(bytecode, 0x20),
            mload(bytecode),
```

```
            salt
        )
    }
    require(addr != address(0), "Deployment failed");
}

function computeAddress(
    bytes memory bytecode,
    bytes32 salt
) internal view returns (address) {
    bytes32 hash = keccak256(
        abi.encodePacked(
            bytes1(0xff),
            address(this),
            salt,
            keccak256(bytecode)
        )
    );
    return address(uint160(uint256(hash)));
}
```

This enables:

- Cross-chain address consistency
- Address prediction before deployment
- Reproducible deployments
- Simpler multi-chain coordination

7 Security and Auditing

7.1 Security Model

7.1.1 Trust Assumptions

Gnosis Safe Lux DAO inherits Safe's security properties:

- Battle-tested multisig logic
- Module system with guard protection
- Extensive formal verification
- Over $100B secured since 2019

Azorius Module Acts as trusted governor:

- Can execute arbitrary transactions on Safe
- Controlled by Strategy contracts

- Requires passed proposals for execution
- Subject to timelock delays

Strategy Contracts Trusted to enforce voting rules:

- Immutable voting parameters
- Transparent vote tallying
- No ability to execute transactions directly
- Can be replaced via governance

7.1.2 Attack Vectors and Mitigations

Malicious Proposals *Risk*: Proposal execution can call any contract with Safe permissions.

Mitigations:

- Proposer restrictions (token threshold, role requirements)
- Timelock period for community review
- Freeze mechanism for emergency intervention
- Transaction simulation before execution
- Guard contracts for additional restrictions

Vote Manipulation *Risk*: Attackers acquire governance tokens to influence outcomes.

Mitigations:

- Snapshot-based voting (prevents flash loan attacks)
- Multiple voting weight sources (harder to manipulate all)
- Quorum requirements (raises attack cost)
- Basis numerator thresholds (supermajority for sensitive actions)

Paymaster Exploitation *Risk*: Malicious users spam votes to drain paymaster funds.

Mitigations:

- Function-specific validators (only valid votes sponsored)
- Per-strategy whitelisting (explicit authorization required)

- Stake requirements on EntryPoint (bundler DoS protection)
- Gas limit enforcement (prevents excessive consumption)
- Real-time monitoring and circuit breakers

Upgrade Attacks *Risk*: Malicious upgrade replaces contract logic. *Mitigations*:

- Ownership restricted to DAO Safe (requires governance)
- UUPS pattern (upgrade logic transparent in implementation)
- Upgrade proposals subject to timelock and voting
- Critical contracts kept immutable where possible

7.2 Audit History

October 2022 Initial Azorius contracts audited by Trail of Bits:

- Core module functionality
- Strategy patterns
- Safe integration
- No critical issues found

June 2024 Account abstraction integration audited by OpenZeppelin:

- Paymaster implementation
- Validator contracts
- EntryPoint integration
- Two medium-severity findings (fixed):
  - Validator reentrancy protection
  - Deposit balance checks

Ongoing Bug bounty program via Immunefi:

- Up to $100,000 for critical vulnerabilities
- Covers all core governance contracts
- Active monitoring and response team

7.3 Formal Verification

Key properties verified using symbolic execution (Certora):

1. **Proposal Integrity**: Proposal transactions cannot be modified after submission
2. **Vote Immutability**: Cast votes cannot be changed
3. **Execution Authorization**: Only passed proposals can execute
4. **Timelock Enforcement**: Proposals cannot execute during timelock
5. **Quorum Requirements**: Proposals without quorum cannot pass

8 Comparison with Existing Frameworks

Feature	Lux DAO	Aragon	Compound	Moloch
Modular voting strategies	✓	Limited	×	×
Multiple token standards	✓	✓	×	×
Gasless voting	✓	×	×	×
Role-based governance	✓	Limited	×	×
Parent-child DAOs	✓	×	×	×
Partial execution	✓	×	×	×
Upgradeable	✓	✓	×	×
Safe integration	Native	Adapter	Adapter	×
Payment streaming	✓	×	×	×
Token-bound accounts	✓	×	×	×
Freeze mechanism	✓	×	×	×

Table 1: Governance Framework Feature Comparison

8.1 Detailed Analysis

8.1.1 Aragon

Strengths:

- Mature ecosystem with extensive tooling
- App-based extensibility model
- User-friendly DAO creation interface

Limitations vs. Lux DAO:

- No native account abstraction support
- Tighter coupling between voting and execution

- Complex upgrade mechanisms
- No built-in hierarchical DAO support
- Limited role management capabilities

8.1.2 Compound Governor

Strengths:

- Battle-tested in high-value DeFi protocols
- Clean, auditable implementation
- Industry-standard delegation patterns

Limitations vs. Lux DAO:

- Monolithic design prevents strategy swapping
- Single token standard (ERC20 with delegation)
- No gasless voting mechanism
- Upgrades require full contract replacement
- No role-based access control
- No organizational hierarchy support

8.1.3 Moloch DAO

Strengths:

- Pioneering ragequit mechanism
- Extreme simplicity and security
- Low gas costs

Limitations vs. Lux DAO:

- Fixed governance model (no modularity)
- One-member-one-vote only
- No support for complex voting strategies
- No account abstraction
- Limited to guild/club use cases
- No automation capabilities

Operation	Lux DAO	Aragon	Compound	Moloch
Propose (gas)	180k	220k	190k	150k
Vote (gas)	120k	110k	140k	80k
Gasless vote (gas)	0*	N/A	N/A	N/A
Execute (gas)	95k	150k	100k	200k
DAO creation (gas)	2.1M	2.5M	800k	500k

Table 2: Gas Cost Comparison (*Paid by DAO treasury via paymaster)

8.2 Performance Comparison

Note: Measurements taken on Ethereum mainnet. Actual costs vary based on network conditions and proposal complexity.

9 Production Deployments

9.1 Network Coverage

Lux DAO has been deployed to multiple networks:

Network	Chain ID	Active DAOs	Total Proposals
Ethereum Mainnet	1	14	127
Optimism	10	8	63
Polygon	137	22	184
Base	8453	11	91
Sepolia (testnet)	11155111	45	521

Table 3: Production Deployment Statistics (as of October 2024)

9.2 Case Studies

9.2.1 Case Study 1: DeFi Protocol DAO

Organization: Anonymous DeFi protocol with $50M TVL
Configuration:

- 70% voting weight from protocol token (ERC20)
- 30% voting weight from genesis NFTs (ERC721)
- 5-day voting period
- 24-hour timelock
- 10% quorum requirement
- 60% approval threshold

Outcomes:

- 47 proposals executed over 18 months
- Average voter participation: 23% (up from 12% pre-gasless voting)
- Zero governance attacks or exploits
- Average execution delay: 6.2 days (voting + timelock)
- Gasless voting saved community ∼$18,000 in fees

9.2.2 Case Study 2: Media DAO with Payment Streaming

Organization: Decentralized media collective with 200 contributors
Configuration:

- Role-based governance using Hats Protocol
- 12 core roles with payment streams
- 3-tier hierarchy (Editorial → Department → Contributor)
- Monthly role elections via governance
- ERC-6551 token-bound accounts for automatic compensation

Outcomes:

- 156 role assignments over 14 months
- $240,000 distributed via Sablier streams
- Zero payment disputes (automated via TBAs)
- 4 organizational restructurings via governance
- Average role transition time: 2 days

9.2.3 Case Study 3: Parent-Child DAO Hierarchy

Organization: Investment DAO with 5 sector-specific sub-DAOs
Configuration:

- Parent DAO controls high-level strategy
- Child DAOs manage sector-specific investments
- Freeze mechanism for emergency intervention
- Cross-DAO treasury management

- Coordinated proposal execution

Outcomes:

- 28 parent DAO proposals affecting children
- 2 emergency freeze activations (both resolved)
- 141 child DAO proposals executed
- $12M managed across hierarchy
- Average parent intervention time: 4 hours (emergency) to 7 days (planned)

9.3 User Feedback

Surveyed 150 active DAO participants across 35 organizations:

- **92%** found gasless voting significantly improved their experience
- **87%** appreciated the flexibility of multiple voting strategies
- **78%** valued the ability to adjust governance parameters over time
- **71%** found role-based organization more intuitive than pure token voting
- **23%** experienced initial confusion with account abstraction signatures

10 Lessons Learned and Future Directions

10.1 Key Insights

10.1.1 Gasless Voting Adoption

Gasless voting dramatically improved participation, but required significant education:

- Users initially confused by UserOperation signing vs. transaction signing
- Clear UI indication of "free vote" increased adoption
- Fallback to paid voting crucial for paymaster funding issues
- Bundler selection affects reliability (geographic distribution matters)

10.1.2 Modularity Trade-offs

While modularity enables flexibility, it introduces complexity:

- More contracts to deploy and manage
- Higher initial gas costs for DAO creation
- Requires deeper technical understanding for customization
- Documentation and tooling critical for adoption

10.1.3 Role-Based Governance

Hats Protocol integration proved powerful but underutilized:

- Most DAOs start simple, gradually add complexity
- Visual tree representation crucial for understanding
- Payment streaming integration (TBA + Sablier) highly valued
- Automated role management reduces governance overhead

10.2 Future Enhancements

10.2.1 Optimistic Governance

Implement optimistic proposal execution for routine operations:

- Proposals execute immediately with challenge period
- Reduced latency for low-risk decisions
- Veto mechanism for intervention
- Risk-based execution thresholds

10.2.2 Cross-Chain Governance

Enable unified governance across multiple chains:

- Message-passing via LayerZero or Hyperlane
- Cross-chain vote aggregation
- Multi-chain treasury management
- Synchronized proposal execution

10.2.3 AI-Assisted Proposal Analysis

Integrate LLM-based tools for governance:

- Automated proposal summarization
- Risk assessment and red flags
- Historical pattern analysis
- Execution simulation and impact prediction

10.2.4 Zero-Knowledge Voting

Implement private voting using ZK proofs:

- Hidden vote choices until reveal
- Protection against vote buying
- Maintains verification and auditability
- Integration with existing infrastructure

10.2.5 Reputation Systems

Layer reputation metrics on top of token voting:

- Historical participation tracking
- Expertise-weighted voting for specialized proposals
- Contributor recognition and rewards
- Sybil resistance mechanisms

10.3 Research Directions

10.3.1 Governance Mechanism Design

Formal analysis of voting mechanisms:

- Game-theoretic analysis of attack vectors
- Optimal quorum and threshold parameters
- Voter behavior modeling
- Mechanism fairness and efficiency

10.3.2 Paymaster Economics

Sustainability of gasless voting:

- Cost-benefit analysis for DAOs
- Fee market dynamics for sponsored operations
- Alternative funding models (staking rewards, protocol fees)
- Cross-DAO paymaster pools

10.3.3 Hierarchical DAO Theory

Formal models for multi-tier organizations:

- Optimal hierarchy depth and breadth
- Parent-child authority boundaries
- Intervention criteria and mechanisms
- Failure modes and resilience

11 Conclusion

Lux DAO represents a significant advancement in decentralized governance infrastructure. By combining the Azorius modular architecture with cutting-edge technologies like ERC-4337 account abstraction, Hats Protocol role management, and ERC-6551 token-bound accounts, we have created a framework that addresses critical limitations of existing DAO systems while maintaining security, flexibility, and user experience.

Our production deployments across multiple networks demonstrate the practical viability of the approach, with quantifiable improvements in voter participation, governance flexibility, and operational efficiency. The framework's modular design enables DAOs to start simple and gradually adopt advanced features as their needs evolve, rather than forcing premature optimization or lock-in to rigid governance models.

Key contributions include:

1. **Composable Architecture**: Clean separation between proposal management, voting logic, and execution enables independent evolution of governance components
2. **Gasless Voting**: First production implementation of ERC-4337 for DAO voting, removing economic barriers to participation

3. **Advanced Role Management**: Integration of Hats Protocol and ERC-6551 for sophisticated organizational structures with automated compensation

4. **Hierarchical DAOs**: Native support for parent-child relationships with emergency intervention capabilities

5. **Production Validation**: Battle-tested across multiple networks with real-world deployments managing significant treasury values

While challenges remain—particularly in education, tooling, and cross-chain coordination—Lux DAO provides a solid foundation for the next generation of decentralized organizations. The framework is open source, extensively documented, and actively maintained, with a growing ecosystem of tools and integrations.

We believe the principles demonstrated here—modularity, composability, and user-centric design—represent the future of DAO governance infrastructure. As the space continues to evolve, we look forward to seeing how the community builds upon this foundation to create even more sophisticated and accessible governance systems.

Acknowledgments

We thank the Fractal Framework team for their pioneering work on the Azorius protocol, which forms the foundation of Lux DAO. We are grateful to the Gnosis Safe team for their exceptional multisig infrastructure and the Zodiac framework. We acknowledge the Hats Protocol team for their innovative approach to on-chain organizations. We thank our auditors at Trail of Bits and OpenZeppelin for their rigorous security analysis. Finally, we thank the Lux DAO community for their feedback, testing, and real-world deployments that have shaped this framework.

References

[1] Aragon Association. *Aragon: Govern better, together.* https://aragon.org, 2017-2024.

[2] Compound Labs. *Compound Governance Documentation.* https://docs.compound.finance/governance/, 2020.

[3] Moloch DAO. *Moloch DAO: Minimum Viable DAO.* https://github.com/MolochVentures/moloch, 2019.

[4] DAOstack. *DAOstack: An Operating System for Collective Intelligence.* https://daostack.io, 2018.

[5] Ethereum Foundation. *ERC-4337: Account Abstraction via Entry Point Contract singleton*. https://eips.ethereum.org/EIPS/eip-4337, 2021.

[6] Safe Ecosystem Foundation. *Safe: The most trusted platform to manage digital assets on Ethereum*. https://safe.global, 2019-2024.

[7] Gnosis Guild. *Zodiac: A collection of tools built according to an open standard*. https://zodiac.wiki, 2021.

[8] Haberdasher Labs. *Hats Protocol: On-chain Roles, Permissions & Accountabilities*. https://www.hatsprotocol.xyz, 2022.

[9] Ethereum Foundation. *ERC-6551: Non-fungible Token Bound Accounts*. https://eips.ethereum.org/EIPS/eip-6551, 2023.

[10] Ethereum Foundation. *EIP-7201: Namespaced Storage Layout*. https://eips.ethereum.org/EIPS/eip-7201, 2023.

[11] Alchemy. *Light Account: A simple ERC-4337 account implementation*. https://github.com/alchemyplatform/light-account, 2023.

[12] Sablier Labs. *Sablier V2: Token streaming protocol*. https://sablier.com, 2023.

[13] Fractal Framework. *Azorius: Optimistic governance module*. https://github.com/fractal-framework/fractal-contracts, 2022.

[14] Lux Industries Inc. *Lux DAO: Modular Governance Framework*. https://github.com/luxdao/contracts, 2022-2024.