F.U.M. Smart Contract Architecture

Requirements Document

Project Overview

The F.U.M. (Flexible Unified Management) project will migrate blockchain operations from direct interactions with DeFi protocol contracts to custom smart contracts while maintaining the current adapter pattern for platform agnosticism. The system will support basic CRUD operations initially, with flexibility to add advanced strategies in the future.

Project Objectives

1. Create a modular, extensible smart contract system for liquidity position management
2. Support multiple DeFi platforms starting with Uniswap V3 on Arbitrum
3. Maintain platform and chain agnosticism through the adapter pattern
4. Enable future implementation of advanced strategies
5. Ensure security, gas efficiency, and transparent operation

---

Functional Requirements

1. Core Liquidity Management

1.1 Position Creation

- Create new liquidity positions on supported platforms
- Support custom price range specification
- Allow specification of token amounts and slippage tolerance

1.2 Liquidity Addition

- Add liquidity to existing positions
- Support both single-sided and balanced liquidity addition
- Allow specification of slippage tolerance

1.3 Liquidity Removal

- Remove liquidity partially or completely from positions
- Support percentage-based removal (e.g., remove 50% of liquidity)
- Ensure safe removal with slippage protection

1.4 Fee Collection

- Collect accumulated fees from positions
- Support partial or complete fee collection
- Properly account for collected fees

1.5 Position Closure

- Remove all liquidity from a position
- Collect all fees
- Optionally burn position NFTs for gas refunds

2. Platform Adapter System

2.1 Adapter Registration

- Register new platform adapters with the main contract
- Update existing adapter implementations
- Remove deprecated adapters

2.2 Cross-Platform Management

- Support different liquidity protocols with the same interface
- Handle protocol-specific differences transparently
- Maintain consistent behavior across platforms

3. Strategy System (Foundation for Future Extensions)

3.1 Strategy Registration

- Register strategy contracts with unique identifiers
- Update strategy implementations
- Enable/disable strategies based on security or performance considerations

3.2 Strategy Execution

- Execute strategies on positions
- Pass custom parameters to strategies
- Return execution results to callers

4. Position Information

4.1 Position Querying

- Retrieve detailed information about positions
- Get position status (in-range, out-of-range)
- Calculate uncollected fees

4.2 Position Ownership

- Track position ownership
- Enforce owner-only operations
- Support delegated management (optional)

Technical Requirements

1. Smart Contract Architecture

1.1 Core Contracts

- LiquidityManager.sol: Main entry point for liquidity operations
- StrategyRegistry.sol: Registry for advanced strategies
- FeeCollector.sol: Handling of protocol fees (if implemented)

1.2 Adapter Contracts

- IProtocolAdapter.sol: Interface for all protocol adapters
- UniswapV3Adapter.sol: Implementation for Uniswap V3
- Future adapters for other platforms

1.3 Strategy Contracts

- IStrategy.sol: Interface for strategy contracts
- BasicStrategy.sol: Implementation of basic CRUD operations

- Foundation for future advanced strategies

1.4 Utility Contracts

- Libraries for position management, math operations, etc.
- Security utilities (access control, reentrancy protection)
- Multicall support for batched operations

2. Contract Upgradability

2.1 Upgrade Mechanism

- Implement proxy pattern for core contracts (Optional)
- Allow adapter replacement without affecting stored positions
- Define clear upgrade paths for future enhancements

2.2 Version Control

- Track contract versions
- Ensure compatibility between components
- Implement version checks for critical operations

3. Gas Optimization

3.1 Storage Efficiency

- Optimize storage layout
- Use appropriate data types
- Implement gas-efficient data structures

3.2 Operation Optimization

- Batch related operations when possible
- Optimize critical paths
- Implement gas-saving techniques

4. Event Emission

4.1 Comprehensive Events

- Emit events for all state-changing operations

- Include sufficient information for off-chain tracking
- Maintain consistent event structure

4.2 Indexing Support

- Design events for efficient indexing
- Include indexed parameters for key data
- Support subgraph development

Security Requirements

1. Access Control

1.1 Permission System

- Implement role-based access control
- Restrict sensitive operations to authorized addresses
- Support ownership transfer and management delegation

1.2 Emergency Controls

- Implement pause mechanism for emergencies
- Allow safe value recovery in critical situations
- Define clear emergency procedures

2. Protection Mechanisms

2.1 Reentrancy Protection

- Implement reentrancy guards on all external functions
- Follow checks-effects-interactions pattern
- Prevent cross-function reentrancy attacks

2.2 Integer Overflow/Underflow

- Use SafeMath or Solidity 0.8.x built-in overflow checks
- Validate numerical inputs
- Implement sanity checks on critical calculations

2.3 Input Validation

- Validate all function inputs
- Implement reasonable bounds on parameters
- Reject invalid or suspicious inputs

2.4 Timestamp Manipulation

- Use block.timestamp only for long timeframes
- Implement secure deadline handling
- Avoid reliance on exact timing

3. Protocol Integration Security

3.1 External Call Safety

- Validate return values from external calls
- Handle failures gracefully
- Implement fallback mechanisms

3.2 Protocol-Specific Risks

- Address known vulnerabilities in integrated protocols
- Validate protocol behavior before integration
- Implement safeguards against protocol-specific attacks

Testing Requirements

1. Unit Testing

1.1 Test Coverage

- Achieve >95% test coverage for all contracts
- Test all public and external functions
- Test edge cases and failure modes

1.2 Test Organization

- Organize tests by contract and functionality

- Implement test helpers for common operations
- Document test purpose and expected results

2. Integration Testing

2.1 Cross-Contract Testing

- Test interactions between contracts
- Validate adapter integration
- Test complex flows involving multiple contracts

2.2 Protocol Integration Testing

- Test integration with actual DeFi protocols
- Validate behavior on forked mainnet
- Test with real-world data and scenarios

3. Fuzz Testing

3.1 Property-Based Testing

- Implement invariant tests
- Test with randomized inputs
- Identify edge cases and vulnerabilities

3.2 Stress Testing

- Test under high load conditions
- Identify performance bottlenecks
- Validate gas usage under various conditions

Deployment Requirements

1. Deployment Process

1.1 Deployment Scripts

- Create automated deployment scripts
- Support different network configurations

- Implement deployment verification

1.2 Contract Verification

- Verify all contracts on block explorers
- Publish ABIs and interfaces
- Document deployment addresses

2. Post-Deployment

2.1 Initialization

- Configure initial parameter values
- Register initial adapters
- Set appropriate access controls

2.2 Monitoring

- Implement monitoring for contract activity
- Track key metrics and events
- Set up alerts for suspicious activity

Document Version: 1.0
Last Updated: March 19, 2025
Document Status: Draft