# SimpleAMM v2 Documentation

## Architectural Overview

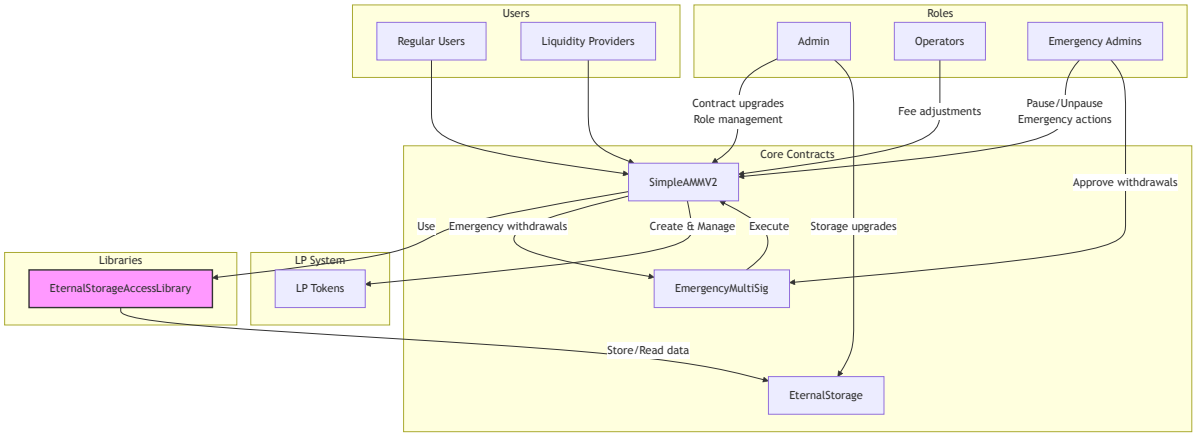
### Core Components

1. **SimpleAMMV2 (Main Contract)**
    - Implements core AMM functionality using constant product formula (x*y=k)
    - Handles liquidity provision, swaps, and emergency functions
    - Uses eternal storage pattern for upgradability
    - Implements role-based access control
2. **EternalStorage**
    - Pure storage contract that persists data across upgrades
    - Implements access control to restrict data access
    - Stores primitive types (uint256, string, address, bool, bytes, int256)
    - SimpleAMMV2 uses `EternalStorageAccessLibrary` to access `EternalStorage`
3. **EmergencyMultiSig**
    - Multi-signature wallet for emergency withdrawals
    - Implements time-bound proposals
    - Requires multiple approvals for execution
4. **LPToken**
    - ERC20-compliant token representing liquidity provider shares
    - Minted when liquidity is added
    - Burned when liquidity is removed
5. **Migration Scripts**
    - Deploy a new contract and upgrade the storage contract
    - See Migration_Guide.pdf

## Core Features

1. **Token Support**
    - Supports ETH and any standard ERC20 tokens
    - Multiple token pairs (ETH-Token pools)
    - Constant product formula (x*y=k) for price discovery
2. **Pool Management**
    - Automated market making
    - Fee earning for liquidity providers
    - Dynamic pricing based on pool reserves
    - Anti-manipulation mechanisms (minimum liquidity)

## System Architecture




### Role Permissions:

1. **Admin (DEFAULT_ADMIN_ROLE)**
    - Upgrade contract implementations
    - Manage roles (grant/revoke)
    - Initialize contracts
2. **Operators (OPERATOR_ROLE)**
    - Adjust trading fees
3. **Emergency Admins (EMERGENCY_ROLE)**
    - Pause/unpause contracts
    - Propose emergency withdrawals
    - Approve emergency actions

**Contract Interactions:**

1. **SimpleAMMV2**
    - Core AMM logic
    - Interacts with `EternalStorage` for data persistence with `EternalStorageAccessLibrary`
    - Creates and manages LP tokens directly
    - Executes emergency actions through `EmergencyMultiSig`
2. **EternalStorage**
    - Stores all contract state
    - Controlled by admin
    - Accessed through `EternalStorageAccessLibrary`
    - only logic contract can set data
3. **EmergencyMultiSig**
    - Requires multiple approvals
    - Time-locked proposals
    - Executes through `SimpleAMMV2`
4. **LPToken System**
    - Factory creates unique tokens per pool
    - ERC20-compliant tokens
    - Managed by `SimpleAMMV2`

# Design Rationale

## 1. Upgradability Pattern

The system uses the eternal storage pattern for upgradability:

- **Why Eternal Storage?**
    - Separates storage from logic
    - Allows contract logic upgrades without data migration
    - Maintains consistent storage layout across versions
    - Simpler than proxy patterns

## 2. Security Features

Multiple security layers are implemented:

- **Role-Based Access Control**
  - `DEFAULT_ADMIN_ROLE`: Contract administration
  - `OPERATOR_ROLE`: Fee management
  - `EMERGENCY_ROLE`: Emergency operations
- **Emergency Controls**
  - Multi-sig requirement for emergency actions
  - Pausable functionality
  - Emergency withdrawal mechanism
- **Safety Checks**
  - Slippage protection
  - Price impact limits
  - Maximum trade size limits
  - Deadline checks
  - Zero address/value checks
  - Maximum trade size limited to `50%` of pool
  - Maximum allowed slippage of `20%`
  - `24-hour` timelock on emergency proposals

## 3. AMM Design

The AMM implements a constant product formula with several optimizations:

- **Pool Management**
  - Dynamic pool creation
  - Minimum liquidity requirement
  - Core invariant: Reserve ratio
- **Trading**
  - Configurable fees
  - Price impact calculation and slippage protection

- Core invariant: Constant product formula

# User Guide

## For Liquidity Providers

1. **Adding Liquidity**

```
function addLiquidity(address tokenAddress, uint25
```

   - Approve token spending first
   - Send ETH along with the transaction
   - Receive LP tokens representing your share

2. **Removing Liquidity**

```
function removeLiquidity(
    address tokenAddress,
    uint256 shares,
    uint256 minEthOut,
    uint256 minTokensOut,
    uint256 deadline
)
```

   - Specify minimum output amounts
   - Set reasonable deadline
   - Burn LP tokens to receive underlying assets

## For Traders

1. **ETH to Token Swaps**

```
function swapETHForTokens(
    address tokenAddress,
    uint256 minTokensOut,
    uint256 maxSlippage,
    uint256 deadline
)
```

- Set max slippage
- Set deadline

2. **Token to ETH Swaps**

```
function swapTokensForETH(
    address tokenAddress,
    uint256 tokenAmount,
    uint256 minEthOut,
    uint256 maxSlippage,
    uint256 deadline
)
```

- Set max slippage
- Set deadline

# Developer Guide

## Contract Deployment

1. **Deploy Storage**

```
forge script script/Deploy.s.sol:DeployScript --rp
```

2. **Upgrade & Migration**

- See Migration_Guide.pdf

## Integration Guide

1. **Pool Integration**

```
// Get pool information
(uint256 tokenReserve, uint256 ethReserve, uint256

// Get spot price
uint256 price = amm.getSpotPrice(tokenAddress);

// Get swap information
(uint256 amountOut, uint256 priceImpact) = amm.get
```

2. **Event Monitoring**

```
event LiquidityAdded(address indexed provider, addı
event LiquidityRemoved(address indexed provider, a
event TokenSwap(address indexed token, uint256 tok
```

### Emergency Procedures

1. **Pausing the System**

```
// Only EMERGENCY_ROLE
amm.pause();
```

2. **Emergency Withdrawal**

```
// Requires multi-sig approval
multiSig.proposeWithdrawal(token, recipient, amoun
multiSig.approveWithdrawal(proposalId);
multiSig.executeWithdrawal(proposalId);
```

3. **Rollback Procedure**

```
forge script script/migration/Rollback.s.sol:Rollb
```

## Out of Scope Features

- Not Support **werid ERC20 tokens**, e.g. different decimals, fee-on-transfer, etc.
- Cancellation of multi-sig proposals
- Adding/revoking signers, operators, emergency admins, etc.

## Contract Security

- Avoid Reentrancy Attacks
  - Follow the CEI pattern(Check-Effect-Interaction)
  - Use `ReentrancyGuardTransient`

- Overflow/Underflow
  - No need to check since soliditVy version >= `0.8.0`
- Denial of Service (DoS)
  - Not Found
- Vault inflation attack
  - Avoid by sending a small amount of reserves to address(1)
- Self-destruct attack
  - Due to time limit, I didn't implement this.
  - Need to add a `skim` function to skim the reserves to the admin
- Static Analysis
  - use slither to analyze the code, config in slither.config.json
  - use aderant to analyze the code, see report.md
- Formal Verification
  - Due to time limit, I didn't implement this.

## Tests

- Unit Tests
  - see `./test` directory
- Fuzz Tests
  - see `./test/fuzz` directory
- Handler-based invariant(property-based) tests
  - `AddLiquidity`/`RemoveLiquidity`: the ratio of `tokenReserve`/`ethReserve` should be the same as the initial value
    - see test/invariant/LiquidityInvariant.t.sol
  - `swapETHForTokens`/`swapTokensForETH`: the reserve product should be a bit greater than the last value considering the fee(`100% < product < 105%`)
    - see test/invariant/SwapInvariant.t.sol
  - Other invariants
    - see test/invariant/SimpleAMMV2.invariant.t.sol

Note: Due to limited time, only important tests are implemented. The coverage is not 100%.

```
forge test
```

▶ Click to see test results

# Gas Optimization

## Key Optimization Techniques

1. **Storage Access Optimization**
    - Cached storage variables to avoid multiple SLOAD operations

```
// Before
if (store.getPoolLPTokenWithAddress(tokenAddress) =
    // do something
}

// After
address lpTokenAddr = store.getPoolLPTokenWithAddr
if (lpTokenAddr == address(0)) {
    // do something
}
```

    - Batched storage updates to minimize SSTORE operations
    - Used memory variables instead of storage where possible

2. **Computation Optimization**
    - Used unchecked blocks for safe arithmetic operations

```
// Before
uint256 tokensAmountBasedOnEth = (ethAmount * toke

// After
unchecked {
    uint256 tokensAmountBasedOnEth = (ethAmount *
}
```

    - Combined calculations to reduce intermediate variables

- Cached frequently used values like msg.value

3. **Gas-Efficient Patterns**
   - Reordered operations to minimize state changes
   - Performed all checks at the start of functions
   - Used custom errors instead of require statements

```
// Before
require(msg.value > 0, "Zero ETH amount");

// After
if (msg.value == 0) revert SimpleAMM__ZeroETHAmoun
```

4. **Memory Management**
   - Minimized memory expansion costs

```
// Before
function getPoolInfo() returns (uint256, uint256, 
    uint256 tokenReserve = store.getPoolTokenReser
    uint256 ethReserve = store.getPoolEthReservesW
    uint256 totalShares = lpToken.totalSupply();
    return (tokenReserve, ethReserve, totalShares)
}

// After
function getPoolInfo() returns (uint256 tokenReser
    (tokenReserve, ethReserve, address lpTokenAddr
    totalShares = LPToken(lpTokenAddr).totalSupply
}
```

   - Avoided unnecessary memory copies
   - Used calldata for read-only function parameters

5. **Event Optimization**
   - Used indexed parameters efficiently for cheaper event filtering

6. **Reentrancy Protection**
   - use `ReentrancyGuardTransient` instead of traditional `ReentrancyGuard**`

```
// Transient storage costs less gas than permanent
contract SimpleAMMV2 is ReentrancyGuardTransient {
    function withdraw() external nonReentrant {
        // Protected against reentrancy
    }
}
```

7. **Constant and Immutable Variables**
    - use `constant` or `immutable` instead of `storage`

## Gas Savings Breakdown

| Operation | Gas Saved |
|---|---|
| SLOAD optimization | ~2100 |
| SSTORE optimization | ~5000 |
| Memory vs Storage | ~2000 |
| Unchecked arithmetic | ~35 |
| Custom errors | ~50 |
| ReentrancyGuardTransient | ~15000 |

## Function-Specific Optimizations

1. **addLiquidity**
    - Cached pool data to single `SLOAD`
    - Combined proportional calculations
    - Used unchecked for safe math
2. **swapETHForTokens**
    - Cached `msg.value`
    - Combined fee calculations
    - Optimized reserve updates
3. **removeLiquidity**
    - Batched reserve updates
    - Cached LP token data

- Optimized share calculations

## Considerations

1. **Trade-offs**
    - Some optimizations may reduce code readability
    - Unchecked math requires careful auditing
    - Complex optimizations increase audit complexity