Smart Contract Fixes - Production Ready

Hardhat Configuration

javascript

```
// hardhat.config.js
require("@nomicfoundation/hardhat-toolbox");
require("@chainlink/hardhat-chainlink");
require("dotenv").config();

module.exports = {
  solidity: {
    version: "0.8.19",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  },
  networks: {
    hardhat: {
      forking: {
        url: process.env.ETHEREUM_RPC_URL,
        blockNumber: 18900000
      }
    },
    ethereum: {
      url: process.env.ETHEREUM_RPC_URL,
      accounts: [process.env.PRIVATE_KEY]
    },
    arbitrum: {
      url: process.env.ARBITRUM_RPC_URL,
      accounts: [process.env.PRIVATE_KEY]
    },
    polygon: {
      url: process.env.POLYGON_RPC_URL,
      accounts: [process.env.PRIVATE_KEY]
```

```
    },
    optimism: {
      url: process.env.OPTIMISM_RPC_URL,
      accounts: [process.env.PRIVATE_KEY]
    }
  },
  etherscan: {
    apiKey: {
      mainnet: process.env.ETHERSCAN_API_KEY,
      arbitrumOne: process.env.ARBISCAN_API_KEY,
      polygon: process.env.POLYGONSCAN_API_KEY,
      optimisticEthereum: process.env.OPTIMISM_API_KEY
    }
  }
};
```

Package.json for Smart Contracts

json

```
{
  "name": "yieldmax-contracts",
  "version": "1.0.0",
  "scripts": {
    "compile": "hardhat compile",
    "test": "hardhat test",
    "deploy": "hardhat run scripts/deploy.js",
    "verify": "hardhat verify"
  },
  "devDependencies": {
    "@nomicfoundation/hardhat-toolbox": "^3.0.0",
    "@chainlink/hardhat-chainlink": "^0.0.1",
    "@chainlink/contracts": "^0.8.0",
    "@openzeppelin/contracts": "^4.9.0",
    "@openzeppelin/contracts-upgradeable": "^4.9.0",
    "hardhat": "^2.19.0",
    "ethers": "^5.7.0",
    "chai": "^4.3.0",
    "dotenv": "^16.0.0"
  }
}
```

Missing Contract Implementations

solidity

```solidity
// contracts/mocks/MockERC20.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockERC20 is ERC20 {
    uint8 private _decimals;

    constructor(
        string memory name,
        string memory symbol,
        uint8 decimals_
    ) ERC20(name, symbol) {
        _decimals = decimals_;
    }

    function decimals() public view virtual override returns (uint8) {
        return _decimals;
    }

    function mint(address to, uint256 amount) public {
        _mint(to, amount);
    }
}
```

```solidity
```

```solidity
// contracts/mocks/MockCCIPRouter.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract MockCCIPRouter {
    mapping(bytes32 => bool) public processedMessages;

    event MessageSent(bytes32 messageId, uint64 destinationChain, bytes data);
    event MessageReceived(bytes32 messageId, uint64 sourceChain, bytes data);

    function getFee(uint64 destinationChain, bytes calldata data)
        external
        pure
        returns (uint256)
    {
        return 0.01 ether + (data.length * 1000);
    }

    function ccipSend(uint64 destinationChain, bytes calldata message)
        external
        payable
        returns (bytes32)
    {
        bytes32 messageId = keccak256(abi.encodePacked(
            block.timestamp,
            msg.sender,
            destinationChain,
            message
        ));

        emit MessageSent(messageId, destinationChain, message);
        return messageId;
    }
```

```
    function deliverMessage(
        address receiver,
        bytes32 messageId,
        uint64 sourceChain,
        address sender,
        bytes calldata data
    ) external {
        require(!processedMessages[messageId], "Already processed");
        processedMessages[messageId] = true;

        (bool success, ) = receiver.call(
            abi.encodeWithSignature(
                "ccipReceive(bytes32,uint64,address,bytes)",
                messageId,
                sourceChain,
                sender,
                data
            )
        );
        require(success, "Message delivery failed");

        emit MessageReceived(messageId, sourceChain, data);
    }

    function simulateDelivery() external {
        // Mock function for testing
    }
}
```

Test Helper Utilities

javascript

```
// test/helpers/index.js
const { ethers } = require("hardhat");
const { time } = require("@nomicfoundation/hardhat-network-helpers");

async function setupPosition(deployment, amount) {
  const { vault, usdc, owner } = deployment;
  await usdc.mint(owner.address, amount);
  await usdc.approve(vault.address, amount);
  await vault.deposit(amount, owner.address);
}

async function executeRebalance(fromDeployment, toDeployment, amount) {
  // Implementation for rebalance execution
  const startTime = Date.now();

  try {
    // Withdraw from source
    await fromDeployment.vault.withdraw(amount);

    // Bridge funds (mock)
    await time.increase(60);

    // Deposit to destination
    await toDeployment.vault.deposit(amount, toDeployment.owner.address);

    return {
      success: true,
      messageDelivered: true,
      duration: Date.now() - startTime
    };
  } catch (error) {
    return {
      success: false,
```

```
      messageDelivered: false,
      error: error.message
    };
  }
}

async function verifyFinalBalances(deployments, expectedTotal) {
  let total = ethers.BigNumber.from(0);

  for (const [chainName, deployment] of deployments) {
    const balance = await deployment.vault.totalAssets();
    total = total.add(balance);
  }

  return total.gte(expectedTotal.mul(995).div(1000)); // 0.5% tolerance
}

async function measureBatchGas(deployment) {
  const { vault, usdc } = deployment;
  const users = await ethers.getSigners();

  const deposits = [];
  for (let i = 1; i <= 10; i++) {
    const user = users[i];
    const amount = ethers.utils.parseUnits("1000", 6);
    await usdc.mint(user.address, amount);
    await usdc.connect(user).approve(vault.address, amount);
    deposits.push({ depositor: user.address, amount });
  }

  const tx = await vault.batchDeposit(deposits);
  const receipt = await tx.wait();
```

```
  return receipt.gasUsed.toNumber();
}

async function sendCrossChainMessage(source, dest, payload) {
  const messageId = ethers.utils.randomBytes(32);
  const tx = await source.router.sendMessage(
    dest.chainId,
    dest.router.address,
    payload
  );

  const receipt = await tx.wait();
  let retries = 0;
  let delivered = false;

  // Simulate retries
  while (!delivered && retries < 3) {
    try {
      await dest.router.processMessage(messageId);
      delivered = true;
    } catch {
      retries++;
      await time.increase(30);
    }
  }

  return { delivered, retries };
}

async function waitForEvent(contract, eventName, timeout) {
  return new Promise((resolve, reject) => {
    const timer = setTimeout(() => {
      contract.removeAllListeners(eventName);
```

```
      reject(new Error(`Timeout waiting for ${eventName}`));
    }, timeout);

    contract.once(eventName, (...args) => {
      clearTimeout(timer);
      resolve({ args });
    });
  });
}

module.exports = {
  setupPosition,
  executeRebalance,
  verifyFinalBalances,
  measureBatchGas,
  sendCrossChainMessage,
  waitForEvent
};
```

ChainlinkClient Mock

javascript

```javascript
// test/utils/chainlink-client.js
class ChainlinkClient {
  constructor(config) {
    this.config = config;
    this.yieldData = {};
  }

  async pushYieldData(data) {
    this.yieldData = { ...this.yieldData, ...data };
    // In real implementation, this would push to Chainlink Data Streams
    return true;
  }

  async getYieldData(protocol, chain) {
    return this.yieldData[protocol]?.[chain] || 0;
  }
}

module.exports = { ChainlinkClient };
```

CCIPSimulator Mock

javascript

```
// test/utils/ccip-simulator.js
class CCIPSimulator {
  constructor(deployments) {
    this.deployments = deployments;
    this.outages = new Map();
  }

  async initialize() {
    // Setup mock CCIP routes between all chains
    for (const [chainA, deploymentA] of this.deployments) {
      for (const [chainB, deploymentB] of this.deployments) {
        if (chainA !== chainB) {
          await deploymentA.router.addRoute(
            deploymentB.chainId,
            deploymentB.router.address
          );
        }
      }
    }
  }

  simulateOutage(chain, duration) {
    this.outages.set(chain, {
      start: Date.now(),
      duration: duration * 1000
    });
  }

  async deliverMessage(router, messageId, sourceChain, sender, payload) {
    const outage = this.outages.get(router.address);
    if (outage && Date.now() - outage.start < outage.duration) {
      throw new Error("CCIP outage");
    }
```

```
    return router.ccipReceive(messageId, sourceChain, sender, payload);
  }
}

module.exports = { CCIPSimulator };
```