NEXUS WEB3 LABS

From: Dr. Eliza Nakamoto, Chief Blockchain Architect & Founder

COMPREHENSIVE DEVELOPMENT BLUEPRINT: MOSAICAL NFT LENDING PLATFORM

Dear Mosaical Team,

Congratulations on your achievement of reaching the Top 10 in the Web3 Ideathon with the Alt30 concept. As requested, I've prepared a highly detailed development blueprint for rebranding to Mosaical and pivoting to the Saga Protocol ecosystem, specifically targeting the GameFi market. Below is a comprehensive breakdown of the POC, Prototype, and MVP phases with specific technical implementations, success metrics, and development considerations.

PROOF OF CONCEPT (POC) DETAILED BLUEPRINT

1. Core Objectives

The POC will demonstrate the fundamental technical feasibility of the GameFi-focused NFT lending model through a minimal implementation that validates:

- NFT collateralization mechanics on Saga Protocol
- GameFi yield collection automation
- Dynamic LTV adjustments based on GameFi NFT utility metrics
- Basic partial liquidation functionality
- Cross-chainlet NFT discoverability and pricing

2. Technical Architecture

2.1 Smart Contract Structure

```
// Core contract architecture
```

```
contract NFTVault {
    // Maps user addresses to their deposited NFTs
    mapping(address => mapping(address => mapping(uint256 => bool)))
public deposits;

    // Chainlet registry for GameFi NFT collections
    mapping(address => bool) public supportedChainlets;
    mapping(address => mapping(address => bool)) public
supportedCollections;

    // Deposit NFT function
    function depositNFT(address collection, uint256 tokenId) external {
        require(isGameFiNFT(collection), "Not a supported GameFi NFT");
        // Transfer NFT to vault
        IERC721(collection).transferFrom(msg.sender, address(this),
tokenId);
        deposits[msg.sender][collection][tokenId] = true;
        emit NFTDeposited(msg.sender, collection, tokenId);
    }

    // Withdraw NFT function (with loan check)
    function withdrawNFT(address collection, uint256 tokenId) external {
        require(deposits[msg.sender][collection][tokenId], "Not your
NFT");
        require(loanManager.getLoanAmount(msg.sender, collection, tokenId)
== 0, "Loan exists");

        deposits[msg.sender][collection][tokenId] = false;
        IERC721(collection).transferFrom(address(this), msg.sender,
tokenId);
        emit NFTWithdrawn(msg.sender, collection, tokenId);
    }

    // Check if NFT is from supported GameFi collection
    function isGameFiNFT(address collection) public view returns (bool) {
        address chainlet = getChainletFromCollection(collection);
        return supportedChainlets[chainlet] &&
supportedCollections[chainlet][collection];
    }

    // Get chainlet from collection address (for POC, simplified
implementation)
    function getChainletFromCollection(address collection) internal view
returns (address) {
        // For POC: Extract chainlet from collection address pattern
        // In practice, will use Saga Protocol chainlet registry
```

```
        return address(uint160(collection) &
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000);
    }

    // Admin function to add supported chainlet (governance controlled in
future)
    function addSupportedChainlet(address chainlet) external onlyAdmin {
        supportedChainlets[chainlet] = true;
    }

    // Admin function to add supported collection within a chainlet
    function addSupportedCollection(address chainlet, address collection)
external onlyAdmin {
        require(supportedChainlets[chainlet], "Chainlet not supported");
        supportedCollections[chainlet][collection] = true;
    }
}

contract LoanManager {
    // Maps NFTs to loan amounts
    mapping(address => mapping(address => mapping(uint256 => uint256)))
public loans;

    // Oracle reference
    IPriceOracle public oracle;

    // GameFi utility multiplier (POC version)
    mapping(address => uint256) public gameFiUtilityMultiplier;

    // Borrow against NFT
    function borrow(address collection, uint256 tokenId, uint256 amount)
external {
        require(nftVault.deposits(msg.sender, collection, tokenId), "Not
your NFT");

        // Get NFT price from oracle
        uint256 nftPrice = oracle.getNFTPrice(collection, tokenId);

        // Apply GameFi utility multiplier to base LTV (default 50%)
        uint256 utilityFactor = gameFiUtilityMultiplier[collection];
        if (utilityFactor == 0) utilityFactor = 100; // Default 100% (no
adjustment)

        // Calculate max LTV with utility adjustment
        uint256 baseMaxLTV = 50; // 50% base LTV
        uint256 adjustedMaxLTV = (baseMaxLTV * utilityFactor) / 100;
```

```
        // Cap at 70% max LTV for safety
        if (adjustedMaxLTV > 70) adjustedMaxLTV = 70;

        uint256 maxLoan = (nftPrice * adjustedMaxLTV) / 100;
        require(amount <= maxLoan, "Exceeds LTV");

        loans[msg.sender][collection][tokenId] = amount;
        IERC20(lendingToken).transfer(msg.sender, amount);
        emit LoanCreated(msg.sender, collection, tokenId, amount);
    }

    // Repay loan
    function repay(address collection, uint256 tokenId, uint256 amount)
external {
        uint256 loanAmount = loans[msg.sender][collection][tokenId];
        require(loanAmount > 0, "No loan exists");

        uint256 repayAmount = amount > loanAmount ? loanAmount : amount;
        loans[msg.sender][collection][tokenId] -= repayAmount;

        IERC20(lendingToken).transferFrom(msg.sender, address(this),
repayAmount);
        emit LoanRepaid(msg.sender, collection, tokenId, repayAmount);
    }

    // Set GameFi utility multiplier (admin function for POC)
    function setGameFiUtilityMultiplier(address collection, uint256
multiplier) external onlyAdmin {
        require(multiplier <= 140, "Multiplier too high"); // Max 140%
(allows up to 70% LTV)
        gameFiUtilityMultiplier[collection] = multiplier;
    }
}

contract SimpleDPOToken is ERC20 {
    // Maps NFT to total supply of DPO tokens
    mapping(address => mapping(uint256 => uint256)) public nftTokenSupply;

    // Mint DPO tokens for NFT
    function mintDPOTokens(address user, address collection, uint256
tokenId) external onlyLoanManager {
        uint256 tokenSupply = 1000 * (10**18); // 1000 tokens per NFT
        _mint(user, tokenSupply);
        nftTokenSupply[collection][tokenId] = tokenSupply;
    }

    // Burn DPO tokens during repayment
```

```
    function burnDPOTokens(address user, address collection, uint256
tokenId, uint256 amount) external onlyLoanManager {
        _burn(user, amount);
    }
}

contract GameFiPriceOracle {
    // Floor prices by collection
    mapping(address => uint256) public collectionFloorPrices;

    // GameFi utility score by collection (e.g., rarity, level, in-game
stats)
    mapping(address => mapping(uint256 => uint256)) public
nftUtilityScores;

    // Set floor price (admin only in POC)
    function setCollectionFloorPrice(address collection, uint256 price)
external onlyAdmin {
        collectionFloorPrices[collection] = price;
    }

    // Set NFT utility score (admin only in POC)
    function setNFTUtilityScore(address collection, uint256 tokenId,
uint256 score) external onlyAdmin {
        nftUtilityScores[collection][tokenId] = score;
    }

    // Get NFT price with utility adjustment
    function getNFTPrice(address collection, uint256 tokenId) external
view returns (uint256) {
        uint256 floorPrice = collectionFloorPrices[collection];
        uint256 utilityScore = nftUtilityScores[collection][tokenId];

        if (utilityScore == 0) return floorPrice; // Default to floor if
no score

        // Apply utility score as multiplier (baseline is 100)
        return (floorPrice * utilityScore) / 100;
    }
}

contract YieldCollector {
    // Simulate yield collection for POC
    function collectYield(address collection, uint256 tokenId) external
returns (uint256) {
        // In POC, return mock yield amount based on GameFi utility
```

```
        uint256 yieldAmount = mockGameFiYieldCalculation(collection,
tokenId);

        // Transfer yield to loan manager for repayment
        IERC20(yieldToken).transfer(loanManager, yieldAmount);
        return yieldAmount;
    }

    // Mock GameFi yield calculation
    function mockGameFiYieldCalculation(address collection, uint256
tokenId) internal view returns (uint256) {
        // For POC: Base yield (1% of NFT value per month) + GameFi
utility bonus
        uint256 nftValue = oracle.getNFTPrice(collection, tokenId);
        uint256 baseYield = (nftValue * 1 * timeElapsed) / (100 * 30
days);

        // GameFi bonus: Extra 0-3% based on utility score
        uint256 utilityScore = oracle.nftUtilityScores(collection,
tokenId);
        if (utilityScore == 0) utilityScore = 100; // Default

        uint256 bonusYield = (baseYield * (utilityScore - 100)) / 100;
        if (bonusYield > (baseYield * 3)) bonusYield = baseYield * 3; //
Cap at 300% bonus

        return baseYield + bonusYield;
    }
}
```

2.2 Oracle Implementation

```
// POC Oracle Implementation for Saga GameFi NFTs
class GameFiNFTPriceOracle {
  constructor() {
    this.prices = {};
    this.utilityScores = {};
    this.lastUpdated = {};
    this.updateFrequency = 1 hour;
    this.sagaChainlets = new Set(); // Set of supported chainlets
    this.gameFiCollections = {}; // Map of chainlet -> collections
  }

  // Register Saga GameFi chainlet
  async registerChainlet(chainletId, name) {
    this.sagaChainlets.add(chainletId);
    this.gameFiCollections[chainletId] = new Set();
```

```
    console.log(`Registered GameFi chainlet: ${name} (${chainletId})`);
  }

  // Register collection within a chainlet
  async registerCollection(chainletId, collectionAddress, name) {
    if (!this.sagaChainlets.has(chainletId)) {
      throw new Error("Chainlet not registered");
    }

    this.gameFiCollections[chainletId].add(collectionAddress);
    console.log(`Registered collection: ${name} on chainlet
${chainletId}`);
  }

  // Update price from external sources
  async updatePrice(chainletId, collection) {
    try {
      if (!this.sagaChainlets.has(chainletId) ||
          !this.gameFiCollections[chainletId].has(collection)) {
        throw new Error("Unsupported collection or chainlet");
      }

      // Primary source (use chainlet's liquidity API)
      const chainletPrice = await this.getChainletPrice(chainletId,
collection);

      // Secondary source (check cross-chain marketplaces)
      const marketplacePrice = await this.getMarketplacePrice(chainletId,
collection);

      // Calculate price with preference for chainlet native price
      let finalPrice;
      if (chainletPrice && marketplacePrice) {
        // Prefer chainlet price but check for major discrepancies
        const priceDiff = Math.abs(chainletPrice - marketplacePrice) /
chainletPrice;

        if (priceDiff <= 0.10) { // Allow up to 10% variance for GameFi
NFTs
          // Weighted average with more weight to chainlet price
          finalPrice = (chainletPrice * 0.7) + (marketplacePrice * 0.3);
        } else {
          // If large discrepancy, use more conservative price
          finalPrice = Math.min(chainletPrice, marketplacePrice);
          this.logPriceDiscrepancy(chainletId, collection, chainletPrice,
marketplacePrice);
        }
```

```
      } else {
        // Fallback to whichever is available
        finalPrice = chainletPrice || marketplacePrice;
      }

      const key = `${chainletId}:${collection}`;
      this.prices[key] = finalPrice;
      this.lastUpdated[key] = Date.now();
      return finalPrice;
    } catch (error) {
      console.error(`Failed to update price for ${collection} on chainlet
${chainletId}:`, error);
      return null;
    }
  }

  // Get utility score for GameFi NFT
  async getNFTUtilityScore(chainletId, collection, tokenId) {
    try {
      // Query chainlet-specific API for NFT utility metrics
      const utilityData = await this.queryChainletUtility(chainletId,
collection, tokenId);

      // Extract key metrics based on game type
      const gameType = await this.getGameType(chainletId, collection);
      let utilityScore = 100; // Default baseline score

      switch (gameType) {
        case 'rpg':
          // For RPG games, consider level, rarity, and power stats
          utilityScore = this.calculateRpgUtilityScore(utilityData);
          break;
        case 'racing':
          // For racing games, consider speed, handling, and unique parts
          utilityScore = this.calculateRacingUtilityScore(utilityData);
          break;
        case 'strategy':
          // For strategy games, consider tactical value and versatility
          utilityScore = this.calculateStrategyUtilityScore(utilityData);
          break;
        default:
          // Generic scoring based on rarity and level
          utilityScore = this.calculateGenericUtilityScore(utilityData);
      }

      return utilityScore;
    } catch (error) {
```

```
      console.error(`Failed to get utility score for NFT ${tokenId}:`,
error);
      return 100; // Default to baseline if error
    }
  }

  // Get current price with freshness check
  async getPrice(chainletId, collection) {
    const key = `${chainletId}:${collection}`;
    const now = Date.now();

    if (!this.lastUpdated[key] ||
        now - this.lastUpdated[key] > this.updateFrequency) {
      await this.updatePrice(chainletId, collection);
    }

    return this.prices[key];
  }
}
```

2.3 GameFi Yield Simulation Framework

```
// GameFi Yield Simulation Framework
class GameFiYieldSimulator {
  constructor(chainlets, collections, timeframe = 90) { // 90 days
simulation
    this.chainlets = chainlets;
    this.collections = collections;
    this.timeframe = timeframe;
    this.dailyYields = {};
    this.volatilityFactors = {};
    this.utilityImpact = {};
    this.gameEngagement = {};
  }

  // Initialize with historical data
  async initialize() {
    for (const chainlet of this.chainlets) {
      for (const collection of this.collections[chainlet] || []) {
        const key = `${chainlet}:${collection}`;

        // Load historical yield data for collection
        const historicalYield = await this.fetchHistoricalYield(chainlet,
collection);

        // Calculate average daily yield
```

```
        this.dailyYields[key] =
this.calculateAverageDailyYield(historicalYield);

        // Calculate volatility factor
        this.volatilityFactors[key] =
this.calculateVolatility(historicalYield);

        // Calculate utility impact on yield
        this.utilityImpact[key] = await
this.analyzeUtilityImpact(chainlet, collection);

        // Calculate game engagement metrics (active users, playtime)
        this.gameEngagement[key] = await
this.fetchGameEngagement(chainlet, collection);
      }
    }
  }

  // Analyze how NFT utility affects yield
  async analyzeUtilityImpact(chainlet, collection) {
    try {
      // Fetch sample of NFTs with different utility scores
      const nftSamples = await this.fetchNFTSamplesWithUtility(chainlet,
collection, 20);

      // Calculate correlation between utility and yield
      let totalCorrelation = 0;
      let sampleCount = 0;

      for (const nft of nftSamples) {
        const yieldHistory = await this.fetchNFTYieldHistory(chainlet,
collection, nft.tokenId);
        if (yieldHistory.length > 0) {
          const avgYield = yieldHistory.reduce((sum, y) => sum + y.yield,
0) / yieldHistory.length;
          const normalizedYield = avgYield /
this.dailyYields[`${chainlet}:${collection}`];
          const utilityFactor = nft.utilityScore / 100;

          // How closely utility predicts yield (1.0 would be perfect
correlation)
          const correlation = normalizedYield / utilityFactor;
          totalCorrelation += correlation;
          sampleCount++;
        }
      }
```

```
      return sampleCount > 0 ? totalCorrelation / sampleCount : 1.0;
    } catch (error) {
      console.error("Error analyzing utility impact:", error);
      return 1.0; // Default: utility has 1:1 impact on yield
    }
  }

  // Run simulation with different market scenarios
  async runSimulation(marketScenario = 'normal', gameActivity = 'normal')
{
    const results = {};

    for (const chainlet of this.chainlets) {
      results[chainlet] = {};

      for (const collection of this.collections[chainlet] || []) {
        const key = `${chainlet}:${collection}`;
        const dailyYield = this.dailyYields[key];
        const volatility = this.volatilityFactors[key];
        const utilityImpact = this.utilityImpact[key];

        // Adjust for market scenario
        let marketMultiplier;
        switch (marketScenario) {
          case 'bull': marketMultiplier = 1.5; break;
          case 'bear': marketMultiplier = 0.6; break;
          default: marketMultiplier = 1.0; // normal
        }

        // Adjust for game activity level
        let activityMultiplier;
        switch (gameActivity) {
          case 'viral': activityMultiplier = 1.8; break;
          case 'growing': activityMultiplier = 1.3; break;
          case 'declining': activityMultiplier = 0.7; break;
          case 'inactive': activityMultiplier = 0.3; break;
          default: activityMultiplier = 1.0; // normal
        }

        // Simulate daily yields
        const simulatedYields = [];
        let cumulativeYield = 0;

        for (let day = 1; day <= this.timeframe; day++) {
          // Add randomness based on volatility
          const randomFactor = 1 + (Math.random() * 2 - 1) * volatility;
```

```
            // Final daily yield with all factors
            const dailySimulatedYield =
              dailyYield *
              marketMultiplier *
              activityMultiplier *
              randomFactor *
              utilityImpact;

            cumulativeYield += dailySimulatedYield;
            simulatedYields.push({
              day,
              dailyYield: dailySimulatedYield,
              cumulativeYield
            });
          }

          results[chainlet][collection] = {
            averageDailyYield: dailyYield * marketMultiplier *
activityMultiplier * utilityImpact,
            totalYield: cumulativeYield,
            yieldTimeSeries: simulatedYields,
            annualizedYield: (cumulativeYield / this.timeframe) * 365,
            gameEngagementLevel: this.gameEngagement[key]
          };
        }
      }

      return results;
    }
}
```

3. Testing Framework

3.1 Smart Contract Testing

```
// Example test cases for NFT Vault with Saga Protocol integration
describe("NFTVault on Saga Protocol", function() {
  let nftVault, mockNFT, owner, user1;
  let mockChainletId, mockCollectionAddress;

  beforeEach(async function() {
    // Deploy mock Saga Chainlet registry
    const MockSagaRegistry = await
ethers.getContractFactory("MockSagaRegistry");
    mockSagaRegistry = await MockSagaRegistry.deploy();
```

```
    // Register mock GameFi chainlet
    mockChainletId = "0x1234567890123456789012345678901234567890";
    await mockSagaRegistry.registerChainlet(mockChainletId,
"MockGameChainlet");

    // Deploy mock NFT contract to represent a GameFi NFT
    const MockNFT = await ethers.getContractFactory("MockERC721");
    mockNFT = await MockNFT.deploy("MockGameNFT", "MGAME");
    mockCollectionAddress = mockNFT.address;

    // Register mock collection in chainlet
    await mockSagaRegistry.registerCollection(mockChainletId,
mockCollectionAddress);

    // Deploy NFT Vault
    const NFTVault = await ethers.getContractFactory("NFTVault");
    nftVault = await NFTVault.deploy(mockSagaRegistry.address);

    [owner, user1] = await ethers.getSigners();

    // Mint NFT to user1
    await mockNFT.connect(owner).mint(user1.address, 1);

    // Add supported chainlet and collection to vault
    await nftVault.connect(owner).addSupportedChainlet(mockChainletId);
    await nftVault.connect(owner).addSupportedCollection(mockChainletId,
mockCollectionAddress);
  });

  it("Should allow users to deposit GameFi NFTs from supported chainlets",
async function() {
    // Approve vault to transfer NFT
    await mockNFT.connect(user1).approve(nftVault.address, 1);

    // Deposit NFT
    await nftVault.connect(user1).depositNFT(mockCollectionAddress, 1);

    // Check deposit status
    expect(await nftVault.deposits(user1.address, mockCollectionAddress,
1)).to.be.true;

    // Check NFT ownership
    expect(await mockNFT.ownerOf(1)).to.equal(nftVault.address);
  });

  it("Should reject NFT deposits from unsupported chainlets", async
function() {
```

```
    // Create new NFT collection not in supported list
    const UnsupportedNFT = await ethers.getContractFactory("MockERC721");
    const unsupportedNFT = await UnsupportedNFT.deploy("UnsupportedNFT",
"UNSUP");

    // Mint to user1
    await unsupportedNFT.connect(owner).mint(user1.address, 1);
    await unsupportedNFT.connect(user1).approve(nftVault.address, 1);

    // Attempt deposit should fail
    await expect(
      nftVault.connect(user1).depositNFT(unsupportedNFT.address, 1)
    ).to.be.revertedWith("Not a supported GameFi NFT");
  });

  it("Should allow users to withdraw NFTs if no loan exists", async
function() {
    // Setup: Deposit NFT
    await mockNFT.connect(user1).approve(nftVault.address, 1);
    await nftVault.connect(user1).depositNFT(mockCollectionAddress, 1);

    // Withdraw NFT
    await nftVault.connect(user1).withdrawNFT(mockCollectionAddress, 1);

    // Check deposit status
    expect(await nftVault.deposits(user1.address, mockCollectionAddress,
1)).to.be.false;

    // Check NFT ownership
    expect(await mockNFT.ownerOf(1)).to.equal(user1.address);
  });
});

// Example test cases for Loan Manager with GameFi utility
describe("LoanManager with GameFi Utility", function() {
  // Similar setup code...

  it("Should adjust LTV based on GameFi utility multiplier", async
function() {
    // Setup: Deposit NFT
    await mockNFT.connect(user1).approve(nftVault.address, 1);
    await nftVault.connect(user1).depositNFT(mockCollectionAddress, 1);

    // Set NFT price in oracle
    await mockOracle.setCollectionFloorPrice(mockCollectionAddress,
ethers.utils.parseEther("100"));
```

```
    // Set GameFi utility multiplier to 120% (allows 60% LTV instead of
base 50%)
    await
loanManager.connect(owner).setGameFiUtilityMultiplier(mockCollectionAddres
s, 120);

    // Borrow 60 ETH against 100 ETH NFT (should succeed with utility
bonus)
    await loanManager.connect(user1).borrow(
      mockCollectionAddress,
      1,
      ethers.utils.parseEther("60")
    );

    // Check loan amount
    expect(await loanManager.loans(user1.address, mockCollectionAddress,
1))
      .to.equal(ethers.utils.parseEther("60"));
  });

  it("Should cap LTV at maximum 70% even with high utility", async
function() {
    // Setup similar to above

    // Set high utility multiplier (150%)
    await
loanManager.connect(owner).setGameFiUtilityMultiplier(mockCollectionAddres
s, 150);

    // Try to borrow 75 ETH against 100 ETH NFT (should fail as max is
70%)
    await expect(
      loanManager.connect(user1).borrow(
        mockCollectionAddress,
        1,
        ethers.utils.parseEther("75")
      )
    ).to.be.revertedWith("Exceeds LTV");

    // Borrow at max allowed LTV (70%)
    await loanManager.connect(user1).borrow(
      mockCollectionAddress,
      1,
      ethers.utils.parseEther("70")
    );

    // Check loan amount
```

```
    expect(await loanManager.loans(user1.address, mockCollectionAddress,
1))
      .to.equal(ethers.utils.parseEther("70"));
  });
});
```

3.2 Oracle Testing

```
describe("GameFi Oracle Testing", function() {
  let oracle, mockChainletAPI, mockMarketplaceAPI;
  const mockChainletId = "0x1234567890123456789012345678901234567890";
  const mockCollectionAddress =
"0xabcdef1234567890abcdef1234567890abcdef12";

  beforeEach(async function() {
    // Setup mock data sources
    mockChainletAPI = {
      getFloorPrice: async () => ({
        price: ethers.utils.parseEther("100")
      })
    };

    mockMarketplaceAPI = {
      getCollectionFloorPrice: async () => ethers.utils.parseEther("102")
    };

    // Initialize oracle with mocks
    oracle = new GameFiNFTPriceOracle();
    oracle.getChainletPrice = async () =>
mockChainletAPI.getFloorPrice().then(r => r.price);
    oracle.getMarketplacePrice = async () =>
mockMarketplaceAPI.getCollectionFloorPrice();

    // Register test chainlet and collection
    await oracle.registerChainlet(mockChainletId, "Test Game Chainlet");
    await oracle.registerCollection(mockChainletId, mockCollectionAddress,
"Test Game Collection");
  });

  it("Should weight chainlet price higher in the weighted average", async
function() {
    const price = await oracle.updatePrice(mockChainletId,
mockCollectionAddress);

    // Should be weighted 70% chainlet, 30% marketplace
    // (100 * 0.7) + (102 * 0.3) = 70 + 30.6 = 100.6
    expect(price).to.equal(ethers.utils.parseEther("100.6"));
```

```
  });

  it("Should use conservative price when difference exceeds threshold",
async function() {
    // Modify Marketplace price to create >10% difference
    mockMarketplaceAPI.getCollectionFloorPrice = async () =>
ethers.utils.parseEther("120");

    const price = await oracle.updatePrice(mockChainletId,
mockCollectionAddress);
    expect(price).to.equal(ethers.utils.parseEther("100")); // Uses lower
price
  });

  it("Should calculate utility scores for different game types", async
function() {
    // Mock NFT utility data
    const rpgUtilityData = {
      level: 75,
      rarity: "epic",
      power: 850,
      attributes: {
        strength: 120,
        agility: 95
      }
    };

    // Setup mocks for utility calculation
    oracle.queryChainletUtility = async () => rpgUtilityData;
    oracle.getGameType = async () => "rpg";
    oracle.calculateRpgUtilityScore = (data) => {
      // Example calculation: base 100 + level bonus + rarity bonus +
power bonus
      const rarityBonus = {common: 0, uncommon: 5, rare: 15, epic: 30,
legendary: 50}[data.rarity] || 0;
      return 100 + (data.level / 2) + rarityBonus + (data.power / 50);
    };

    const utilityScore = await oracle.getNFTUtilityScore(
      mockChainletId,
      mockCollectionAddress,
      123 // Token ID
    );

    // Expected: 100 + (75/2) + 30 + (850/50) = 100 + 37.5 + 30 + 17 =
184.5
    expect(utilityScore).to.be.approximately(184.5, 0.1);
```

```
  });

  it("Should handle chainlet API failures gracefully", async function() {
    // Make chainlet API fail
    oracle.queryChainletUtility = async () => { throw new Error("API
failed"); };

    const utilityScore = await oracle.getNFTUtilityScore(
      mockChainletId,
      mockCollectionAddress,
      123 // Token ID
    );

    // Should default to baseline 100
    expect(utilityScore).to.equal(100);
  });
});
```

3.3 GameFi Yield Simulation Testing

```
describe("GameFi Yield Simulation", function() {
  let simulator;
  const mockChainletId = "0x1234567890123456789012345678901234567890";
  const mockCollectionAddress =
"0xabcdef1234567890abcdef1234567890abcdef12";

  beforeEach(async function() {
    // Initialize simulator with test collections
    simulator = new GameFiYieldSimulator(
      [mockChainletId],
      {[mockChainletId]: [mockCollectionAddress]}
    );

    // Mock historical data fetching
    simulator.fetchHistoricalYield = async (chainlet, collection) => {
      if (collection === mockCollectionAddress) {
        return Array(90).fill().map((_, i) => ({
          day: i + 1,
          yield: 0.002 * (1 + 0.15 * Math.sin(i / 8)) // 0.2% daily with
sine wave variation
        }));
      } else {
        return [];
      }
    };

    // Mock utility impact analysis
```

```
    simulator.fetchNFTSamplesWithUtility = async () => {
      return Array(20).fill().map((_, i) => ({
        tokenId: i + 1,
        utilityScore: 100 + (i * 5) // 100 to 195
      }));
    };

    simulator.fetchNFTYieldHistory = async (_, __, tokenId) => {
      const baseYield = 0.002;
      const utilityFactor = tokenId % 20 ? (tokenId % 20) / 10 : 1; // 0.1
to 1.9
      return Array(30).fill().map((_, i) => ({
        day: i + 1,
        yield: baseYield * utilityFactor * (1 + 0.1 * Math.sin(i / 5))
      }));
    };

    // Mock game engagement data
    simulator.fetchGameEngagement = async () => ({
      dailyActiveUsers: 15000,
      averagePlaytime: 45, // minutes
      retentionRate: 0.72,
      monthlyRevenue: 320000
    });

    await simulator.initialize();
  });

  it("Should calculate accurate average yields", function() {
    const key = `${mockChainletId}:${mockCollectionAddress}`;
    expect(simulator.dailyYields[key]).to.be.approximately(0.002, 0.0003);
  });

  it("Should calculate utility impact on yield", function() {
    const key = `${mockChainletId}:${mockCollectionAddress}`;
    // Should be close to 1.0 if utility correctly predicts yield
    expect(simulator.utilityImpact[key]).to.be.approximately(1.0, 0.2);
  });

  it("Should simulate different game activity scenarios", async function()
{
    // Normal game activity
    const normalResults = await simulator.runSimulation('normal',
'normal');
    const normalYield =
normalResults[mockChainletId][mockCollectionAddress].annualizedYield;
```

```
    // Viral game activity
    const viralResults = await simulator.runSimulation('normal', 'viral');
    const viralYield =
viralResults[mockChainletId][mockCollectionAddress].annualizedYield;
    expect(viralYield).to.be.greaterThan(normalYield * 1.5); // Should be
at least 50% higher

    // Declining game activity
    const decliningResults = await simulator.runSimulation('normal',
'declining');
    const decliningYield =
decliningResults[mockChainletId][mockCollectionAddress].annualizedYield;
    expect(decliningYield).to.be.lessThan(normalYield * 0.8); // Should be
at least 20% lower
  });

  it("Should combine market and game activity impacts", async function() {
    // Bull market + viral game (optimal scenario)
    const bullViralResults = await simulator.runSimulation('bull',
'viral');
    const bullViralYield =
bullViralResults[mockChainletId][mockCollectionAddress].annualizedYield;

    // Bear market + declining game (worst scenario)
    const bearDecliningResults = await simulator.runSimulation('bear',
'declining');
    const bearDecliningYield =
bearDecliningResults[mockChainletId][mockCollectionAddress].annualizedYiel
d;

    // Extreme difference between best and worst scenarios
    expect(bullViralYield / bearDecliningYield).to.be.greaterThan(3.0); //
At least 3x difference
  });
});
```

4. POC Deliverables

4.1 Technical Documentation

- Architecture diagrams showing component interaction between Saga Protocol chainlets and Mosaical platform
- Smart contract specifications with function descriptions tailored for GameFi NFT lending

- Saga Protocol integration documentation and chainlet compatibility checklist
- GameFi yield simulation methodology and results by game category
- Test coverage report with specific GameFi use cases
- Cross-chainlet security considerations

4.2 Demo Environment

- Deployed contracts on Saga Protocol testnet
- Basic CLI interface for interacting with Mosaical contracts
- Simulation dashboard showing GameFi NFT yield projections
- Monitoring tools for cross-chainlet GameFi asset pricing
- Demo integration with at least 2 popular GameFi chainlets

4.3 Validation Report

- Success metrics evaluation against criteria
- Performance analysis under various market and game activity conditions
- Identified limitations and mitigation strategies
- Recommendations for prototype phase
- GameFi-specific risk assessment

5. Success Metrics Validation Plan

5.1 NFT Deposit/Withdrawal Testing

Test Cases:

- Deposit/withdraw NFTs from multiple GameFi chainlets
- Cross-collection NFT management
- Attempt unauthorized withdrawals
- Withdraw with outstanding loans

Success Criteria: 100% successful legitimate transactions, proper rejection of invalid operations, cross-chainlet compatibility

5.2 GameFi Yield Collection Mechanism

Test Cases:

- Simulate yield from multiple GameFi sources
- Test yield distribution to loan repayment
- Measure accuracy against expected yields by game type
- Simulate in-game rewards as yield source

Success Criteria: Yield collection within 5% of expected values for established games, proper distribution to loan accounts, successful handling of multiple reward token types

5.3 Oracle Price Feed Accuracy

Test Cases:

- Compare oracle prices against actual GameFi marketplace data
- Test pricing for NFTs with high vs. low in-game utility
- Introduce price volatility scenarios
- Simulate chainlet source failures

Success Criteria: Price accuracy within 8% of actual market values for GameFi NFTs, proper fallback behavior, appropriate utility-based valuation adjustments

5.4 Market & Game Volatility Handling

Test Cases:

- Simulate 40% price drops over 24 hours (common in GameFi)
- Test rapid price oscillations during game events/updates
- Simulate game shutdown/maintenance scenarios

- Test changing user engagement levels

Success Criteria: System maintains stability during typical GameFi volatility, proper alerts generated, no critical failures, appropriate LTV adjustments based on game activity metrics

PROTOTYPE PHASE DETAILED BLUEPRINT

1. Core Objectives

The Prototype will deliver a minimally interactive system with:

- Full user interface for GameFi NFT lending operations on Saga Protocol
- Enhanced smart contracts with multi-chainlet and multi-collection support
- DPO token implementation with basic trading functionality
- Graduated liquidation thresholds based on GameFi utility metrics
- Basic governance for protocol parameters
- GameFi collection registry with chainlet integration

2. Technical Architecture

2.1 Smart Contract Upgrades

```
// NFTVault V2 with enhanced GameFi support
contract NFTVaultV2 {
    // Extended storage from V1
    mapping(address => mapping(address => mapping(uint256 => bool)))
public deposits;
    mapping(address => bool) public supportedChainlets;
    mapping(address => mapping(address => bool)) public
supportedCollections;

    // New storage for V2
    // GameFi collection category mapping
    mapping(address => uint8) public gameCategory; // 1=RPG, 2=Racing,
3=Strategy, etc.

    // GameFi NFT metadata cache
    mapping(address => mapping(uint256 => bytes)) public nftMetadataCache;
```

```
    // Batch deposit function for improved UX
    function batchDepositNFTs(address collection, uint256[] calldata
tokenIds) external {
        require(isGameFiNFT(collection), "Not a supported GameFi NFT");

        for (uint256 i = 0; i < tokenIds.length; i++) {
            // Transfer NFT to vault
            IERC721(collection).transferFrom(msg.sender, address(this),
tokenIds[i]);
            deposits[msg.sender][collection][tokenIds[i]] = true;
            emit NFTDeposited(msg.sender, collection, tokenIds[i]);

            // Cache NFT metadata if not already cached
            if (nftMetadataCache[collection][tokenIds[i]].length == 0) {
                bytes memory metadata = fetchNFTMetadata(collection,
tokenIds[i]);
                nftMetadataCache[collection][tokenIds[i]] = metadata;
            }
        }
    }

    // Fetch NFT metadata (called internally)
    function fetchNFTMetadata(address collection, uint256 tokenId)
internal view returns (bytes memory) {
        // For V2: Simple implementation to fetch tokenURI and return as
bytes
        // In V3, this will be enhanced to parse and store specific GameFi
attributes
        string memory tokenURI =
IERC721Metadata(collection).tokenURI(tokenId);
        return bytes(tokenURI);
    }

    // Enhanced isGameFiNFT with GameFi category support
    function isGameFiNFT(address collection) public view returns (bool) {
        address chainlet = getChainletFromCollection(collection);
        return supportedChainlets[chainlet] &&
               supportedCollections[chainlet][collection] &&
               gameCategory[collection] > 0;
    }

    // Set GameFi category (admin/governance only)
    function setGameCategory(address collection, uint8 category) external
onlyAdminOrGovernance {
        require(category > 0 && category <= 10, "Invalid game category");
        gameCategory[collection] = category;
```

```
    }

    // Additional functions from V1 with needed enhancements...
}

contract LoanManagerV2 {
    // Extended storage from V1
    mapping(address => mapping(address => mapping(uint256 => uint256)))
public loans;
    IPriceOracle public oracle;
    mapping(address => uint256) public gameFiUtilityMultiplier;

    // New storage for V2
    // Collection-specific interest rates (basis points)
    mapping(address => uint256) public collectionInterestRates;

    // Health factors for loans (higher = better, 1.0 = liquidation
threshold)
    mapping(address => mapping(address => mapping(uint256 => uint256)))
public loanHealthFactors;

    // Default interest rate in basis points (3% = 300)
    uint256 public defaultInterestRate = 300;

    // Graduated liquidation thresholds
    struct LiquidationThreshold {
        uint256 healthFactor; // 1.0 = 10000 (scaled by 10000)
        uint256 liquidationPct; // Percentage to liquidate at this
threshold
    }

    // 3 levels of liquidation
    LiquidationThreshold[3] public liquidationLevels;

    constructor() {
        // Initialize liquidation levels
        liquidationLevels[0] = LiquidationThreshold(9000, 25); // At 0.9
health, liquidate 25%
        liquidationLevels[1] = LiquidationThreshold(8000, 50); // At 0.8
health, liquidate 50%
        liquidationLevels[2] = LiquidationThreshold(7000, 100); // At 0.7
health, liquidate 100%
    }

    // Enhanced borrow with interest calculation
    function borrow(address collection, uint256 tokenId, uint256 amount)
external {
```

```
        require(nftVault.deposits(msg.sender, collection, tokenId), "Not
your NFT");

        // Get NFT price from oracle
        uint256 nftPrice = oracle.getNFTPrice(collection, tokenId);

        // Apply GameFi utility multiplier to base LTV
        uint256 utilityFactor = gameFiUtilityMultiplier[collection];
        if (utilityFactor == 0) utilityFactor = 100; // Default 100% (no
adjustment)

        // Calculate max LTV with utility adjustment
        uint256 baseMaxLTV = 50; // 50% base LTV
        uint256 adjustedMaxLTV = (baseMaxLTV * utilityFactor) / 100;

        // Cap at 70% max LTV for safety
        if (adjustedMaxLTV > 70) adjustedMaxLTV = 70;

        uint256 maxLoan = (nftPrice * adjustedMaxLTV) / 100;
        require(amount <= maxLoan, "Exceeds LTV");

        // Store loan amount
        loans[msg.sender][collection][tokenId] = amount;

        // Initialize health factor (max = 2.0, scales down as price
decreases)
        // Initial health = NFT value / loan amount * 0.5 (scaled by
10000)
        uint256 initialHealth = (nftPrice * 10000) / amount / 2;
        loanHealthFactors[msg.sender][collection][tokenId] =
initialHealth;

        // Issue DPO tokens for the loan
        dpoToken.mintDPOTokens(msg.sender, collection, tokenId, amount);

        // Transfer loan amount to borrower
        IERC20(lendingToken).transfer(msg.sender, amount);
        emit LoanCreated(msg.sender, collection, tokenId, amount,
initialHealth);
    }

    // Calculate interest for a loan
    function calculateInterest(address user, address collection, uint256
tokenId, uint256 timestamp) public view returns (uint256) {
        uint256 loanAmount = loans[user][collection][tokenId];
        if (loanAmount == 0) return 0;
```

```
        // Get loan start time
        uint256 loanStart = loanStartTime[user][collection][tokenId];
        uint256 timeElapsed = timestamp - loanStart;

        // Get interest rate (collection-specific or default)
        uint256 interestRate = collectionInterestRates[collection];
        if (interestRate == 0) interestRate = defaultInterestRate;

        // Calculate interest: principal * rate * time
        // rate is in basis points (1% = 100)
        // time is in seconds converted to years
        return (loanAmount * interestRate * timeElapsed) / (10000 * 365
days);
    }

    // Update loan health factor based on current NFT price
    function updateHealthFactor(address user, address collection, uint256
tokenId) public returns (uint256) {
        uint256 loanAmount = loans[user][collection][tokenId];
        if (loanAmount == 0) return 0;

        // Get current NFT price
        uint256 nftPrice = oracle.getNFTPrice(collection, tokenId);

        // Calculate health factor: NFT value / loan amount * 0.5 (scaled
by 10000)
        uint256 newHealth = (nftPrice * 10000) / loanAmount / 2;
        loanHealthFactors[user][collection][tokenId] = newHealth;

        return newHealth;
    }

    // Partial liquidation function
    function liquidatePartial(address user, address collection, uint256
tokenId) external {
        // Update health factor first
        uint256 health = updateHealthFactor(user, collection, tokenId);

        // Determine liquidation level
        uint256 liquidationPct = 0;
        for (uint256 i = 0; i < liquidationLevels.length; i++) {
            if (health <= liquidationLevels[i].healthFactor) {
                liquidationPct = liquidationLevels[i].liquidationPct;
                break;
            }
        }
```

```
        require(liquidationPct > 0, "No liquidation needed");

        uint256 loanAmount = loans[user][collection][tokenId];
        uint256 liquidationAmount = (loanAmount * liquidationPct) / 100;

        // Execute partial liquidation via DPO token mechanism
        dpoToken.executeLiquidation(user, collection, tokenId,
liquidationAmount, liquidationPct);

        // Reduce loan amount
        loans[user][collection][tokenId] -= liquidationAmount;

        // If full liquidation, return remaining NFT value to user
        if (liquidationPct == 100) {
            nftVault.liquidationReturn(user, collection, tokenId);
        }

        emit LoanLiquidated(user, collection, tokenId, liquidationAmount,
liquidationPct);
    }

    // Additional functions from V1 with needed enhancements...
}

contract DPOTokenV2 is ERC20 {
    // Ownership and admin contracts
    address public admin;
    address public loanManager;

    // DPO token mappings
    mapping(address => mapping(uint256 => uint256)) public nftTokenSupply;
    mapping(address => mapping(uint256 => mapping(address => uint256)))
public tokenHoldings;

    // NFT to DPO token mapping for lookup
    mapping(address => mapping(uint256 => address)) public nftToDPOToken;

    // DPO trading fee in basis points (0.5% = 50)
    uint256 public tradingFee = 50;
    address public feeRecipient;

    // Enhanced mint with amount-based supply
    function mintDPOTokens(
        address user,
        address collection,
        uint256 tokenId,
        uint256 loanAmount
```

```
    ) external onlyLoanManager {
        // Calculate token supply based on loan amount
        // 1 token per 0.001 ETH in loan value (configurable)
        uint256 tokenSupply = (loanAmount * 1000) / (10**15);

        _mint(user, tokenSupply);
        nftTokenSupply[collection][tokenId] = tokenSupply;
        tokenHoldings[collection][tokenId][user] = tokenSupply;

        // Store reverse mapping for lookup
        string memory tokenName = string(abi.encodePacked("DPO-",
IERC721Metadata(collection).symbol(), "-", tokenId.toString()));
        ERC20 newDpoToken = _createChildToken(tokenName, tokenName);
        nftToDPOToken[collection][tokenId] = address(newDpoToken);

        emit DPOTokensMinted(user, collection, tokenId, tokenSupply);
    }

    // Trading functionality for DPO tokens
    function tradeDPOTokens(
        address collection,
        uint256 tokenId,
        address to,
        uint256 amount
    ) external {
        require(tokenHoldings[collection][tokenId][msg.sender] >= amount,
"Insufficient DPO tokens");

        // Calculate fee
        uint256 fee = (amount * tradingFee) / 10000;
        uint256 netAmount = amount - fee;

        // Transfer tokens
        tokenHoldings[collection][tokenId][msg.sender] -= amount;
        tokenHoldings[collection][tokenId][to] += netAmount;

        // Transfer fee to fee recipient
        if (fee > 0) {
            tokenHoldings[collection][tokenId][feeRecipient] += fee;
        }

        // Reflect in ERC20 balances if using child tokens
        address dpoTokenAddr = nftToDPOToken[collection][tokenId];
        if (dpoTokenAddr != address(0)) {
            ERC20 dpoToken = ERC20(dpoTokenAddr);
            bool success = dpoToken.transferFrom(msg.sender, to,
netAmount);
```

```
            require(success, "Transfer failed");

            if (fee > 0) {
                success = dpoToken.transferFrom(msg.sender, feeRecipient,
fee);
                require(success, "Fee transfer failed");
            }
        }

        emit DPOTokensTraded(msg.sender, to, collection, tokenId, amount,
fee);
    }

    // Execute liquidation by burning DPO tokens
    function executeLiquidation(
        address user,
        address collection,
        uint256 tokenId,
        uint256 liquidationAmount,
        uint256 liquidationPct
    ) external onlyLoanManager {
        uint256 totalSupply = nftTokenSupply[collection][tokenId];
        uint256 burnAmount = (totalSupply * liquidationPct) / 100;

        // Burn proportional tokens from all holders
        address[] memory holders = getTokenHolders(collection, tokenId);
        for (uint256 i = 0; i < holders.length; i++) {
            address holder = holders[i];
            uint256 holderBalance =
tokenHoldings[collection][tokenId][holder];

            uint256 holderBurnAmount = (holderBalance * liquidationPct) /
100;
            tokenHoldings[collection][tokenId][holder] -=
holderBurnAmount;

            // Reflect in ERC20 balances if using child tokens
            address dpoTokenAddr = nftToDPOToken[collection][tokenId];
            if (dpoTokenAddr != address(0)) {
                ERC20 dpoToken = ERC20(dpoTokenAddr);
                _burn(holder, holderBurnAmount);
            }

            emit DPOTokensBurned(holder, collection, tokenId,
holderBurnAmount);
        }
```

```
        // Reduce total supply
        nftTokenSupply[collection][tokenId] = totalSupply - burnAmount;
    }

    // Helper to get all token holders for an NFT
    function getTokenHolders(address collection, uint256 tokenId) internal
view returns (address[] memory) {
        // Implementation would track holders in a mapping or array
        // Simplified version for the blueprint
        // In production: Use EnumerableMap or similar for gas-efficient
holder tracking
        return holderRegistry.getHolders(collection, tokenId);
    }

    // Create child DPO token (internal helper)
    function _createChildToken(string memory name, string memory symbol)
internal returns (ERC20) {
        // Implementation would deploy a child ERC20 contract
        // Simplified version for the blueprint
        return new ChildDPOToken(name, symbol, address(this));
    }
}

contract GameFiOracleV2 {
    // Extended storage from V1
    mapping(address => uint256) public collectionFloorPrices;
    mapping(address => mapping(uint256 => uint256)) public
nftUtilityScores;

    // New storage for V2
    // Price history for volatility calculation
    struct PricePoint {
        uint256 timestamp;
        uint256 price;
    }

    // Last 30 price points for each collection
    mapping(address => PricePoint[30]) public priceHistory;
    mapping(address => uint8) public historyIndex; // Current index in
circular buffer

    // Collection volatility (basis points, 1000 = 10% daily volatility)
    mapping(address => uint256) public collectionVolatility;

    // Multi-source aggregation weights (basis points, total should be
10000)
    struct PriceSource {
```

```
        string name;
        uint256 weight;
    }

    // Up to 5 price sources per collection
    mapping(address => PriceSource[5]) public priceSources;

    // Set floor price with historical tracking
    function setCollectionFloorPrice(address collection, uint256 price)
external onlyOracleFeeder {
        // Store previous price
        uint256 prevPrice = collectionFloorPrices[collection];

        // Update current price
        collectionFloorPrices[collection] = price;

        // Update price history in circular buffer
        uint8 index = historyIndex[collection];
        priceHistory[collection][index] = PricePoint({
            timestamp: block.timestamp,
            price: price
        });

        // Update index for next entry
        historyIndex[collection] = (index + 1) % 30;

        // Calculate volatility if we have at least 2 price points
        if (prevPrice > 0) {
            updateVolatility(collection);
        }

        emit FloorPriceUpdated(collection, price, prevPrice);
    }

    // Calculate and update collection volatility
    function updateVolatility(address collection) internal {
        // Find oldest and newest prices in the buffer
        uint256 oldestTimestamp = type(uint256).max;
        uint256 newestTimestamp = 0;
        uint256 oldestPrice = 0;
        uint256 newestPrice = 0;

        for (uint8 i = 0; i < 30; i++) {
            PricePoint memory pp = priceHistory[collection][i];
            if (pp.timestamp == 0) continue;

            if (pp.timestamp < oldestTimestamp) {
```

```
                oldestTimestamp = pp.timestamp;
                oldestPrice = pp.price;
            }

            if (pp.timestamp > newestTimestamp) {
                newestTimestamp = pp.timestamp;
                newestPrice = pp.price;
            }
        }

        // If we don't have at least 2 points, exit
        if (oldestTimestamp == newestTimestamp) return;

        // Calculate time-weighted volatility
        uint256 timespan = newestTimestamp - oldestTimestamp;
        if (timespan < 1 days) return; // Need at least 1 day of data

        // Calculate daily price movement as percentage
        // Simplified: use max deviation from average as volatility
indicator
        uint256 avgPrice = (oldestPrice + newestPrice) / 2;
        uint256 maxDeviation = 0;

        for (uint8 i = 0; i < 30; i++) {
            PricePoint memory pp = priceHistory[collection][i];
            if (pp.timestamp == 0) continue;

            uint256 deviation;
            if (pp.price > avgPrice) {
                deviation = ((pp.price - avgPrice) * 10000) / avgPrice;
            } else {
                deviation = ((avgPrice - pp.price) * 10000) / avgPrice;
            }

            if (deviation > maxDeviation) {
                maxDeviation = deviation;
            }
        }

        // Normalize to daily volatility
        uint256 daysInSample = timespan / 1 days;
        uint256 dailyVolatility = maxDeviation / daysInSample;

        // Update collection volatility
        collectionVolatility[collection] = dailyVolatility;

        emit VolatilityUpdated(collection, dailyVolatility);
```

```
    }

    // Configure price sources for a collection
    function configurePriceSources(
        address collection,
        string[] memory names,
        uint256[] memory weights
    ) external onlyAdmin {
        require(names.length == weights.length, "Array length mismatch");
        require(names.length <= 5, "Too many sources");

        // Validate weights sum to 10000 (100%)
        uint256 totalWeight = 0;
        for (uint256 i = 0; i < weights.length; i++) {
            totalWeight += weights[i];
        }
        require(totalWeight == 10000, "Weights must sum to 10000");

        // Set price sources
        for (uint256 i = 0; i < names.length && i < 5; i++) {
            priceSources[collection][i] = PriceSource({
                name: names[i],
                weight: weights[i]
            });
        }

        // Clear any remaining slots
        for (uint256 i = names.length; i < 5; i++) {
            delete priceSources[collection][i];
        }

        emit PriceSourcesConfigured(collection, names, weights);
    }

    // Get NFT price with enhanced utility adjustment and volatility
factor
    function getNFTPrice(address collection, uint256 tokenId) external
view returns (uint256) {
        uint256 floorPrice = collectionFloorPrices[collection];
        uint256 utilityScore = nftUtilityScores[collection][tokenId];
        uint256 volatility = collectionVolatility[collection];

        if (utilityScore == 0) utilityScore = 100; // Default to baseline

        // Base price adjusted for utility
        uint256 basePrice = (floorPrice * utilityScore) / 100;
```

```
        // Apply volatility discount for high-volatility collections
        // Higher volatility = slightly larger discount
        if (volatility > 0) {
            // Max discount of 10% at 5000 basis points (50%) daily
volatility
            uint256 volatilityDiscount = (volatility * 1000) / 5000;
            if (volatilityDiscount > 1000) volatilityDiscount = 1000; //
Cap at 10%

            basePrice = basePrice - ((basePrice * volatilityDiscount) /
10000);
        }

        return basePrice;
    }
}
```

2.2 Frontend Application Code

```
// NFT Deposit Form Component
const NFTDepositForm = ({ chainlets, collections, onDeposit }) => {
  const [selectedChainlet, setSelectedChainlet] = useState('');
  const [selectedCollection, setSelectedCollection] = useState('');
  const [ownedNFTs, setOwnedNFTs] = useState([]);
  const [selectedNFTs, setSelectedNFTs] = useState([]);
  const [isLoading, setIsLoading] = useState(false);

  // Load user's owned NFTs when collection changes
  useEffect(() => {
    if (selectedCollection) {
      setIsLoading(true);
      fetchUserNFTs(selectedChainlet, selectedCollection)
        .then(nfts => {
          setOwnedNFTs(nfts);
          setIsLoading(false);
        })
        .catch(error => {
          console.error("Error fetching NFTs:", error);
          setIsLoading(false);
        });
    } else {
      setOwnedNFTs([]);
    }
  }, [selectedCollection]);

  // Handle chainlet selection
  const handleChainletChange = (e) => {
```

```
    setSelectedChainlet(e.target.value);
    setSelectedCollection('');
    setOwnedNFTs([]);
    setSelectedNFTs([]);
  };

  // Handle collection selection
  const handleCollectionChange = (e) => {
    setSelectedCollection(e.target.value);
    setSelectedNFTs([]);
  };

  // Toggle NFT selection
const toggleNFTSelection = (tokenId) => {
  if (selectedNFTs.includes(tokenId)) {
    setSelectedNFTs(selectedNFTs.filter(id => id !== tokenId));
  } else {
    setSelectedNFTs([...selectedNFTs, tokenId]);
  }
};

// Handle deposit submission
const handleSubmit = async (e) => {
  e.preventDefault();
  if (selectedNFTs.length === 0) return;

  setIsLoading(true);
  try {
    // Call blockchain method to deposit NFTs
    await onDeposit(selectedChainlet, selectedCollection, selectedNFTs);
    // Reset selection after successful deposit
    setSelectedNFTs([]);
    // Refresh NFT list
    const updatedNFTs = await fetchUserNFTs(selectedChainlet,
selectedCollection);
    setOwnedNFTs(updatedNFTs);
  } catch (error) {
    console.error("Error depositing NFTs:", error);
    alert(`Failed to deposit NFTs: ${error.message}`);
  }
  setIsLoading(false);
};

return (
  <div className="nft-deposit-container">
    <h2>Deposit GameFi NFTs as Collateral</h2>
```

```
    <form onSubmit={handleSubmit}>
      <div className="form-group">
        <label>Select GameFi Chainlet:</label>
        <select
          value={selectedChainlet}
          onChange={handleChainletChange}
          className="chainlet-selector"
        >
          <option value="">Select a Chainlet</option>
          {chainlets.map(chainlet => (
            <option key={chainlet.id} value={chainlet.id}>
              {chainlet.name}
            </option>
          ))}
        </select>
      </div>

      {selectedChainlet && (
        <div className="form-group">
          <label>Select NFT Collection:</label>
          <select
            value={selectedCollection}
            onChange={handleCollectionChange}
            className="collection-selector"
          >
            <option value="">Select a Collection</option>
            {collections
              .filter(coll => coll.chainletId === selectedChainlet)
              .map(collection => (
                <option key={collection.address}
value={collection.address}>
                  {collection.name}
                </option>
              ))}
          </select>
        </div>
      )}

      {isLoading && <div className="loading-spinner">Loading your
NFTs...</div>}

      {selectedCollection && ownedNFTs.length > 0 && (
        <div className="nft-grid">
          {ownedNFTs.map(nft => (
            <div
              key={nft.tokenId}
```

```
              className={`nft-item ${selectedNFTs.includes(nft.tokenId) ?
'selected' : ''}`}
              onClick={() => toggleNFTSelection(nft.tokenId)}
            >
              <img src={nft.imageUrl} alt={`NFT #${nft.tokenId}`} />
              <div className="nft-info">
                <span className="nft-id">#{nft.tokenId}</span>
                {nft.gameStats && (
                  <div className="game-stats">
                    {Object.entries(nft.gameStats).map(([key, value]) => (
                      <span key={key} className="stat">
                        {key}: {value}
                      </span>
                    ))}
                  </div>
                )}
              </div>
              <div className="nft-utility">
                Utility Score: {nft.utilityScore || 'N/A'}
              </div>
            </div>
          ))}
        </div>
      )}

      {selectedCollection && ownedNFTs.length === 0 && !isLoading && (
        <div className="no-nfts">
          You don't own any NFTs in this collection
        </div>
      )}

      <button
        type="submit"
        className="deposit-button"
        disabled={selectedNFTs.length === 0 || isLoading}
      >
        {isLoading ? 'Processing...' : `Deposit Selected NFTs
(${selectedNFTs.length})`}
      </button>
    </form>
  </div>
);
};

// Borrow Dashboard Component
const BorrowDashboard = ({ userAddress }) => {
  const [depositedNFTs, setDepositedNFTs] = useState([]);
```

```
  const [activeLoans, setActiveLoans] = useState([]);
  const [isLoading, setIsLoading] = useState(true);

  // Fetch user's deposited NFTs and active loans
  useEffect(() => {
    const fetchUserAssets = async () => {
      try {
        const [nfts, loans] = await Promise.all([
          fetchUserDepositedNFTs(userAddress),
          fetchUserLoans(userAddress)
        ]);
        setDepositedNFTs(nfts);
        setActiveLoans(loans);
      } catch (error) {
        console.error("Error fetching user assets:", error);
      } finally {
        setIsLoading(false);
      }
    };

    fetchUserAssets();
  }, [userAddress]);

  // Handle taking a loan
  const handleBorrow = async (nft, amount) => {
    setIsLoading(true);
    try {
      await borrowAgainstNFT(nft.chainletId, nft.collection, nft.tokenId,
amount);
      // Refresh data
      const [updatedNFTs, updatedLoans] = await Promise.all([
        fetchUserDepositedNFTs(userAddress),
        fetchUserLoans(userAddress)
      ]);
      setDepositedNFTs(updatedNFTs);
      setActiveLoans(updatedLoans);
    } catch (error) {
      console.error("Error borrowing against NFT:", error);
      alert(`Failed to borrow: ${error.message}`);
    } finally {
      setIsLoading(false);
    }
  };

  // Handle loan repayment
  const handleRepay = async (loan, amount) => {
    setIsLoading(true);
```

```
    try {
      await repayLoan(loan.chainletId, loan.collection, loan.tokenId,
amount);
      // Refresh data
      const [updatedNFTs, updatedLoans] = await Promise.all([
        fetchUserDepositedNFTs(userAddress),
        fetchUserLoans(userAddress)
      ]);
      setDepositedNFTs(updatedNFTs);
      setActiveLoans(updatedLoans);
    } catch (error) {
      console.error("Error repaying loan:", error);
      alert(`Failed to repay: ${error.message}`);
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <div className="borrow-dashboard">
      <h2>Your GameFi Assets & Loans</h2>

      {isLoading ? (
        <div className="loading-spinner">Loading your assets...</div>
      ) : (
        <>
          <div className="dashboard-section">
            <h3>Deposited NFTs</h3>
            {depositedNFTs.length > 0 ? (
              <div className="nft-grid">
                {depositedNFTs.map(nft => (
                  <div key={`${nft.collection}-${nft.tokenId}`}
className="nft-card">
                    <img src={nft.imageUrl} alt={`NFT #${nft.tokenId}`} />
                    <div className="nft-details">
                      <h4>{nft.name || `#${nft.tokenId}`}</h4>
                      <p>Collection: {nft.collectionName}</p>
                      <p>Game: {nft.gameName}</p>
                      <p>Utility Score: {nft.utilityScore}</p>
                      <p className="nft-value">Estimated Value:
{formatCurrency(nft.estimatedValue)}</p>

                      {!nft.hasLoan && (
                        <BorrowForm
                          nft={nft}
                          maxLTV={nft.maxLTV}
                          onBorrow={handleBorrow}
```

```
                      />
                    )}
                  </div>
                </div>
              ))}
            </div>
          ) : (
            <p className="no-assets">You don't have any deposited
NFTs</p>
          )}
        </div>

        <div className="dashboard-section">
          <h3>Active Loans</h3>
          {activeLoans.length > 0 ? (
            <div className="loans-table">
              <table>
                <thead>
                  <tr>
                    <th>NFT</th>
                    <th>Loan Amount</th>
                    <th>Interest Rate</th>
                    <th>Health Factor</th>
                    <th>Accrued Interest</th>
                    <th>Actions</th>
                  </tr>
                </thead>
                <tbody>
                  {activeLoans.map(loan => (
                    <tr key={`${loan.collection}-${loan.tokenId}`}>
                      <td>
                        <div className="loan-nft">
                          <img
                            src={loan.nft.imageUrl}
                            alt={`NFT #${loan.tokenId}`}
                            className="mini-nft"
                          />
                          <span>{loan.nft.collectionName}
#{loan.tokenId}</span>
                        </div>
                      </td>
                      <td>{formatCurrency(loan.amount)}</td>
                      <td>{loan.interestRate}%</td>
                      <td>
                        <div className={`health-factor
${getHealthClass(loan.healthFactor)}`}>
                          {formatHealthFactor(loan.healthFactor)}
```

```
                        </div>
                      </td>
                      <td>{formatCurrency(loan.accruedInterest)}</td>
                      <td>
                        <RepayForm
                          loan={loan}
                          onRepay={handleRepay}
                        />
                      </td>
                    </tr>
                  ))}
                </tbody>
              </table>
            </div>
          ) : (
            <p className="no-assets">You don't have any active loans</p>
          )}
        </div>
      </>
    )}
  </div>
);
};

// DPO Token Trading Component
const DPOTokenTrading = () => {
  const [availableTokens, setAvailableTokens] = useState([]);
  const [userTokens, setUserTokens] = useState([]);
  const [searchQuery, setSearchQuery] = useState('');
  const [isLoading, setIsLoading] = useState(true);
  const [selectedToken, setSelectedToken] = useState(null);
  const [tradeAmount, setTradeAmount] = useState('');
  const [orderType, setOrderType] = useState('buy'); // 'buy' or 'sell'

  // Fetch DPO token data
  useEffect(() => {
    const fetchTokenData = async () => {
      try {
        const [market, owned] = await Promise.all([
          fetchMarketDPOTokens(),
          fetchUserDPOTokens()
        ]);
        setAvailableTokens(market);
        setUserTokens(owned);
      } catch (error) {
        console.error("Error fetching token data:", error);
      } finally {
```

```
        setIsLoading(false);
      }
    };

    fetchTokenData();
  }, []);

  // Handle token selection
  const handleSelectToken = (token) => {
    setSelectedToken(token);
    setTradeAmount('');
  };

  // Filter tokens based on search
  const filteredTokens = useMemo(() => {
    if (!searchQuery) return availableTokens;

    const query = searchQuery.toLowerCase();
    return availableTokens.filter(token =>
      token.name.toLowerCase().includes(query) ||
      token.collectionName.toLowerCase().includes(query) ||
      token.gameName.toLowerCase().includes(query)
    );
  }, [availableTokens, searchQuery]);

  // Handle trade submission
  const handleTrade = async (e) => {
    e.preventDefault();
    if (!selectedToken || !tradeAmount || parseFloat(tradeAmount) <= 0)
return;

    setIsLoading(true);
    try {
      if (orderType === 'buy') {
        await buyDPOTokens(
          selectedToken.chainletId,
          selectedToken.collection,
          selectedToken.tokenId,
          parseFloat(tradeAmount)
        );
      } else {
        await sellDPOTokens(
          selectedToken.chainletId,
          selectedToken.collection,
          selectedToken.tokenId,
          parseFloat(tradeAmount)
        );
```

```
      }

      // Refresh data
      const [market, owned] = await Promise.all([
        fetchMarketDPOTokens(),
        fetchUserDPOTokens()
      ]);
      setAvailableTokens(market);
      setUserTokens(owned);

      // Reset form
      setSelectedToken(null);
      setTradeAmount('');
    } catch (error) {
      console.error(`Error ${orderType === 'buy' ? 'buying' : 'selling'}
tokens:`, error);
      alert(`Failed to ${orderType} tokens: ${error.message}`);
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <div className="dpo-trading">
      <h2>DPO Token Trading</h2>

      <div className="trading-container">
        <div className="market-panel">
          <div className="search-bar">
            <input
              type="text"
              placeholder="Search by name, collection or game..."
              value={searchQuery}
              onChange={(e) => setSearchQuery(e.target.value)}
            />
          </div>

          {isLoading ? (
            <div className="loading-spinner">Loading tokens...</div>
          ) : (
            <div className="token-list">
              <h3>Available DPO Tokens</h3>
              {filteredTokens.length > 0 ? (
                <table>
                  <thead>
                    <tr>
                      <th>Token</th>
```

```
                    <th>Game</th>
                    <th>NFT Value</th>
                    <th>Price</th>
                    <th>Supply</th>
                    <th>Actions</th>
                  </tr>
                </thead>
                <tbody>
                  {filteredTokens.map(token => (
                    <tr
                      key={`${token.collection}-${token.tokenId}`}
                      className={selectedToken?.id === token.id ?
'selected' : ''}
                    >
                      <td>
                        <div className="token-info">
                          <img src={token.imageUrl} alt={token.name}
className="token-icon" />
                          <span>{token.name}</span>
                        </div>
                      </td>
                      <td>{token.gameName}</td>
                      <td>{formatCurrency(token.nftValue)}</td>
                      <td>{formatCurrency(token.price)}</td>
                      <td>{formatNumber(token.supply)}</td>
                      <td>
                        <button
                          className="select-token-btn"
                          onClick={() => handleSelectToken(token)}
                        >
                          Select
                        </button>
                      </td>
                    </tr>
                  ))}
                </tbody>
              </table>
            ) : (
              <p className="no-tokens">No tokens found matching your
search</p>
            )}
          </div>
        )}
      </div>

      <div className="trade-panel">
        {selectedToken ? (
```

```
            <div className="trade-form">
              <h3>Trade {selectedToken.name}</h3>

              <div className="token-details">
                <img src={selectedToken.imageUrl} alt={selectedToken.name}
className="selected-token-img" />
                <div>
                  <p><strong>Collection:</strong>
{selectedToken.collectionName}</p>
                  <p><strong>Game:</strong> {selectedToken.gameName}</p>
                  <p><strong>Current Price:</strong>
{formatCurrency(selectedToken.price)}</p>
                  <p><strong>Your Balance:</strong> {formatNumber(
                    userTokens.find(t =>
                      t.collection === selectedToken.collection &&
                      t.tokenId === selectedToken.tokenId
                    )?.balance || 0
                  )}</p>
                </div>
              </div>

              <form onSubmit={handleTrade}>
                <div className="form-group">
                  <label>Order Type:</label>
                  <div className="order-type-toggle">
                    <button
                      type="button"
                      className={`toggle-btn ${orderType === 'buy' ?
'active' : ''}`}
                      onClick={() => setOrderType('buy')}
                    >
                      Buy
                    </button>
                    <button
                      type="button"
                      className={`toggle-btn ${orderType === 'sell' ?
'active' : ''}`}
                      onClick={() => setOrderType('sell')}
                    >
                      Sell
                    </button>
                  </div>
                </div>

                <div className="form-group">
                  <label>Amount:</label>
                  <input
```

```
                    type="number"
                    value={tradeAmount}
                    onChange={(e) => setTradeAmount(e.target.value)}
                    min="0"
                    step="0.01"
                    placeholder="Enter amount to trade"
                    required
                  />
                </div>

                {orderType === 'buy' ? (
                  <div className="trade-summary">
                    <p>Total Cost: {formatCurrency(
                      parseFloat(tradeAmount || 0) * selectedToken.price
                    )}</p>
                    <p className="fee-info">
                      +0.5% Trading Fee: {formatCurrency(
                        parseFloat(tradeAmount || 0) * selectedToken.price
* 0.005
                      )}
                    </p>
                  </div>
                ) : (
                  <div className="trade-summary">
                    <p>Total Proceeds: {formatCurrency(
                      parseFloat(tradeAmount || 0) * selectedToken.price
                    )}</p>
                    <p className="fee-info">
                      -0.5% Trading Fee: {formatCurrency(
                        parseFloat(tradeAmount || 0) * selectedToken.price
* 0.005
                      )}
                    </p>
                  </div>
                )}

                <button
                  type="submit"
                  className="trade-btn"
                  disabled={isLoading || !tradeAmount ||
parseFloat(tradeAmount) <= 0}
                >
                  {isLoading ? 'Processing...' : `${orderType === 'buy' ?
'Buy' : 'Sell'} Tokens`}
                </button>
              </form>
            </div>
```

```
          ) : (
            <div className="no-selection">
              <p>Select a DPO token from the list to start trading</p>
            </div>
          )}
        </div>
      </div>
    </div>
  );
};
```

3. End-to-End Testing Framework

```
// E2E testing for complete loan lifecycle
describe("E2E: Full Loan Lifecycle on Saga Protocol", function() {
  let borrower, lender, oracle, admin;
  let nftVault, loanManager, dpoToken;
  let gameNFT;
  let chainletId, collectionAddress;
  const tokenId = 1;

  before(async function() {
    // Set up test environment with longer timeout for blockchain
interactions
    this.timeout(60000);

    // Get accounts for testing
    [admin, borrower, lender] = await ethers.getSigners();

    // Deploy and configure contracts
    chainletId = "0x1234567890123456789012345678901234567890";

    // Deploy mock GameFi NFT
    const GameNFT = await ethers.getContractFactory("MockGameNFT");
    gameNFT = await GameNFT.deploy("TestGame NFT", "TGNFT");
    collectionAddress = gameNFT.address;

    // Mint NFT to borrower
    await gameNFT.mint(borrower.address, tokenId);

    // Deploy NFT Vault
    const NFTVault = await ethers.getContractFactory("NFTVaultV2");
    nftVault = await NFTVault.deploy();

    // Deploy Oracle
```

```
    const GameFiOracle = await
ethers.getContractFactory("GameFiOracleV2");
    oracle = await GameFiOracle.deploy();

    // Deploy DPO Token
    const DPOToken = await ethers.getContractFactory("DPOTokenV2");
    dpoToken = await DPOToken.deploy();

    // Deploy Loan Manager
    const LoanManager = await ethers.getContractFactory("LoanManagerV2");
    loanManager = await LoanManager.deploy(nftVault.address,
oracle.address, dpoToken.address);

    // Set up contract connections
    await nftVault.setLoanManager(loanManager.address);
    await dpoToken.setLoanManager(loanManager.address);

    // Register chainlet and collection
    await nftVault.addSupportedChainlet(chainletId);
    await nftVault.addSupportedCollection(chainletId, collectionAddress);
    await nftVault.setGameCategory(collectionAddress, 1); // RPG game

    // Set oracle price and utility score
    await oracle.setCollectionFloorPrice(collectionAddress,
ethers.utils.parseEther("10"));
    await oracle.setNFTUtilityScore(collectionAddress, tokenId, 150); //
50% bonus for high utility

    // Set utility multiplier
    await loanManager.setGameFiUtilityMultiplier(collectionAddress, 120);
// 20% LTV bonus

    // Fund loan manager with tokens for lending
    await admin.sendTransaction({
      to: loanManager.address,
      value: ethers.utils.parseEther("100")
    });
  });

  it("Should complete an end-to-end loan cycle with NFT collateral", async
function() {
    // Step 1: Borrower deposits NFT
    await gameNFT.connect(borrower).approve(nftVault.address, tokenId);
    await nftVault.connect(borrower).depositNFT(collectionAddress,
tokenId);

    // Verify deposit
```

```
    expect(await nftVault.deposits(borrower.address, collectionAddress,
tokenId)).to.be.true;
    expect(await gameNFT.ownerOf(tokenId)).to.equal(nftVault.address);

    // Step 2: Borrower takes a loan
    const loanAmount = ethers.utils.parseEther("6"); // 60% of 10 ETH NFT
value
    await loanManager.connect(borrower).borrow(collectionAddress, tokenId,
loanAmount);

    // Verify loan
    expect(await loanManager.loans(borrower.address, collectionAddress,
tokenId))
      .to.equal(loanAmount);

    // Verify DPO tokens minted
    const expectedTokens = 6000; // 1000 tokens per ETH borrowed
    expect(await
dpoToken.balanceOf(borrower.address)).to.equal(expectedTokens);

    // Step 3: Borrower sells some DPO tokens to lender
    const saleAmount = 2000; // Sell 2000 tokens
    await dpoToken.connect(borrower).approve(dpoToken.address,
saleAmount);
    await dpoToken.connect(borrower).tradeDPOTokens(
      collectionAddress,
      tokenId,
      lender.address,
      saleAmount
    );

    // Verify token transfer
    const lenderBalance = await dpoToken.tokenHoldings(collectionAddress,
tokenId, lender.address);
    // Should be slightly less than 2000 due to trading fee
    expect(lenderBalance).to.be.gt(1980);
    expect(lenderBalance).to.be.lt(2000);

    // Step 4: Test health factor updates
    // Simulate price drop of 20%
    await oracle.setCollectionFloorPrice(collectionAddress,
ethers.utils.parseEther("8"));

    // Update health factor
    const newHealth = await loanManager.updateHealthFactor(
      borrower.address,
      collectionAddress,
```

```
      tokenId
    );

    // Calculate expected health factor: (8 ETH * 1.5 utility) / (6 ETH
loan) * 0.5 = 1.0
    // Should be around 10000 (scaled by 10000)
    expect(newHealth).to.be.gt(9500);
    expect(newHealth).to.be.lt(10500);

    // Step 5: Simulate partial liquidation due to low health factor
    // Further reduce price to trigger liquidation
    await oracle.setCollectionFloorPrice(collectionAddress,
ethers.utils.parseEther("6.5"));

    // Update health and check liquidation level
    await loanManager.updateHealthFactor(
      borrower.address,
      collectionAddress,
      tokenId
    );

    // Execute partial liquidation
    await loanManager.liquidatePartial(
      borrower.address,
      collectionAddress,
      tokenId
    );

    // Verify loan amount reduced (should be 25% liquidated at first
threshold)
    const remainingLoan = await loanManager.loans(borrower.address,
collectionAddress, tokenId);
    const expectedRemaining = loanAmount.mul(75).div(100);
    expect(remainingLoan).to.equal(expectedRemaining);

    // Verify DPO tokens burned proportionally
    const borrowerRemainingBalance = await dpoToken.tokenHoldings(
      collectionAddress,
      tokenId,
      borrower.address
    );
    const lenderRemainingBalance = await dpoToken.tokenHoldings(
      collectionAddress,
      tokenId,
      lender.address
    );
```

```
    // Both should be reduced by ~25%
    expect(borrowerRemainingBalance).to.be.lt(expectedTokens -
saleAmount);
    expect(lenderRemainingBalance).to.be.lt(lenderBalance);

    // Step 6: Repay remaining loan
    await loanManager.connect(borrower).repay(
      collectionAddress,
      tokenId,
      remainingLoan,
      { value: remainingLoan }
    );

    // Verify loan fully repaid
    expect(await loanManager.loans(borrower.address, collectionAddress,
tokenId))
      .to.equal(0);

    // Step 7: Withdraw NFT
    await nftVault.connect(borrower).withdrawNFT(collectionAddress,
tokenId);

    // Verify withdrawal
    expect(await nftVault.deposits(borrower.address, collectionAddress,
tokenId)).to.be.false;
    expect(await gameNFT.ownerOf(tokenId)).to.equal(borrower.address);
  });
});
```

4. Load Testing Framework

```
// Load testing script for simulating multiple users and transactions
describe("Load Testing: Mosaical Protocol on Saga", function() {
  // Configure for longer-running tests
  this.timeout(300000); // 5 minutes

  const NUM_USERS = 100;
  const TRANSACTIONS_PER_USER = 5;
  const NFT_PRICE = ethers.utils.parseEther("10");

  let admin;
  let users = [];
  let gameChainlets = [];
  let gameCollections = [];
  let nftVault, loanManager, dpoToken, oracle;
```

```
  before(async function() {
    [admin, ...users] = await ethers.getSigners();

    // Ensure we have enough users for testing
    const additionalUsers = NUM_USERS - users.length;
    if (additionalUsers > 0) {
      for (let i = 0; i < additionalUsers; i++) {
        const wallet =
ethers.Wallet.createRandom().connect(ethers.provider);
        users.push(wallet);
        // Fund wallet for gas
        await admin.sendTransaction({
          to: wallet.address,
          value: ethers.utils.parseEther("1")
        });
      }
    }

    // Cut down to exact number needed
    users = users.slice(0, NUM_USERS);

    // Set up test chainlets and collections
    gameChainlets = [
      { id: "0x1111111111111111111111111111111111111111", name: "RPG
World" },
      { id: "0x2222222222222222222222222222222222222222", name: "Racing
League" },
      { id: "0x3333333333333333333333333333333333333333", name: "Strategy
Masters" }
    ];

    gameCollections = [
      { chainlet: gameChainlets[0].id, address: "0xC1", name: "Heroes",
category: 1 },
      { chainlet: gameChainlets[0].id, address: "0xC2", name: "Monsters",
category: 1 },
      { chainlet: gameChainlets[1].id, address: "0xC3", name: "Racecars",
category: 2 },
      { chainlet: gameChainlets[1].id, address: "0xC4", name: "Tracks",
category: 2 },
      { chainlet: gameChainlets[2].id, address: "0xC5", name:
"Commanders", category: 3 }
    ];

    // Deploy contracts
    const NFTVault = await ethers.getContractFactory("NFTVaultV2");
    nftVault = await NFTVault.deploy();
```

```
    const GameFiOracle = await
ethers.getContractFactory("GameFiOracleV2");
    oracle = await GameFiOracle.deploy();

    const DPOToken = await ethers.getContractFactory("DPOTokenV2");
    dpoToken = await DPOToken.deploy();

    const LoanManager = await ethers.getContractFactory("LoanManagerV2");
    loanManager = await LoanManager.deploy(nftVault.address,
oracle.address, dpoToken.address);

    // Set up contract connections
    await nftVault.setLoanManager(loanManager.address);
    await dpoToken.setLoanManager(loanManager.address);

    // Register chainlets and collections
    for (const chainlet of gameChainlets) {
      await nftVault.addSupportedChainlet(chainlet.id);
    }

    for (const collection of gameCollections) {
      await nftVault.addSupportedCollection(collection.chainlet,
collection.address);
      await nftVault.setGameCategory(collection.address,
collection.category);

      // Set oracle price and utility multiplier
      await oracle.setCollectionFloorPrice(collection.address, NFT_PRICE);
      await loanManager.setGameFiUtilityMultiplier(collection.address,
120); // 20% LTV bonus
    }

    // Deploy mock NFTs for each collection
    for (const collection of gameCollections) {
      const MockNFT = await ethers.getContractFactory("MockGameNFT");
      const mockNFT = await MockNFT.deploy(collection.name,
collection.name.substring(0, 4).toUpperCase());
      collection.contractInstance = mockNFT;

      // Override address with actual deployment
      collection.address = mockNFT.address;

      // Update vault and oracle with new address
      await nftVault.addSupportedCollection(collection.chainlet,
mockNFT.address);
```

```
      await nftVault.setGameCategory(mockNFT.address,
collection.category);
      await oracle.setCollectionFloorPrice(mockNFT.address, NFT_PRICE);
      await loanManager.setGameFiUtilityMultiplier(mockNFT.address, 120);
    }

    // Fund loan manager with tokens for lending
    await admin.sendTransaction({
      to: loanManager.address,
      value: ethers.utils.parseEther("1000")
    });

    // Distribute NFTs to users
    console.log("Distributing NFTs to test users...");
    for (let i = 0; i < NUM_USERS; i++) {
      const user = users[i];

      // Each user gets one NFT from a random collection
      const collectionIndex = i % gameCollections.length;
      const collection = gameCollections[collectionIndex];
      const tokenId = i + 1;

      // Mint NFT to user
      await collection.contractInstance.connect(admin).mint(user.address,
tokenId);

      // Set utility score (varies by user)
      const utilityScore = 100 + (i % 5) * 20; // 100, 120, 140, 160, 180
      await oracle.setNFTUtilityScore(collection.address, tokenId,
utilityScore);
    }

    console.log("Setup complete. Beginning load test...");
  });

  it("Should handle multiple concurrent users and transactions", async
function() {
    // Phase 1: NFT Deposits
    console.log("Phase 1: Testing concurrent NFT deposits...");
    const depositPromises = [];

    for (let i = 0; i < NUM_USERS; i++) {
      const user = users[i];
      const collectionIndex = i % gameCollections.length;
      const collection = gameCollections[collectionIndex];
      const tokenId = i + 1;
```

```
      const depositPromise = (async () => {
        // Approve and deposit NFT
        await
collection.contractInstance.connect(user).approve(nftVault.address,
tokenId);
        await nftVault.connect(user).depositNFT(collection.address,
tokenId);

        // Verify deposit
        const isDeposited = await nftVault.deposits(user.address,
collection.address, tokenId);
        expect(isDeposited).to.be.true;
      })();

      depositPromises.push(depositPromise);
    }

    await Promise.all(depositPromises);
    console.log(`✓ ${NUM_USERS} concurrent deposits completed
successfully`);

    // Phase 2: Borrowing
    console.log("Phase 2: Testing concurrent borrowing...");
    const borrowPromises = [];

    for (let i = 0; i < NUM_USERS; i++) {
      const user = users[i];
      const collectionIndex = i % gameCollections.length;
      const collection = gameCollections[collectionIndex];
      const tokenId = i + 1;

      // Vary loan amount by user (40-60% of NFT value)
      const loanPct = 40 + (i % 5) * 5; // 40%, 45%, 50%, 55%, 60%
      const loanAmount = NFT_PRICE.mul(loanPct).div(100);

      const borrowPromise = (async () => {
        await loanManager.connect(user).borrow(collection.address,
tokenId, loanAmount);

        // Verify loan
        const loan = await loanManager.loans(user.address,
collection.address, tokenId);
        expect(loan).to.equal(loanAmount);
      })();

      borrowPromises.push(borrowPromise);
    }
```

```
    await Promise.all(borrowPromises);
    console.log(`✓ ${NUM_USERS} concurrent borrows completed
successfully`);

    // Phase 3: DPO Token Trading
    console.log("Phase 3: Testing DPO token trading...");
    const tradePromises = [];

    // Create user pairs for trading
    for (let i = 0; i < NUM_USERS; i += 2) {
      if (i + 1 >= NUM_USERS) break; // Skip if no partner

      const seller = users[i];
      const buyer = users[i + 1];

      const sellerCollectionIndex = i % gameCollections.length;
      const sellerCollection = gameCollections[sellerCollectionIndex];
      const sellerTokenId = i + 1;

      // Trade ~30% of tokens
      const tradePromise = (async () => {
        // Get seller's balance
        const sellerBalance = await dpoToken.tokenHoldings(
          sellerCollection.address,
          sellerTokenId,
          seller.address
        );

        const tradeAmount = sellerBalance.mul(30).div(100);

        // Approve and trade
        await dpoToken.connect(seller).approve(dpoToken.address,
tradeAmount);
        await dpoToken.connect(seller).tradeDPOTokens(
          sellerCollection.address,
          sellerTokenId,
          buyer.address,
          tradeAmount
        );

        // Verify trade
        const buyerBalance = await dpoToken.tokenHoldings(
          sellerCollection.address,
          sellerTokenId,
          buyer.address
        );
```

```
        expect(buyerBalance).to.be.gt(0);
      })();

      tradePromises.push(tradePromise);
    }

    await Promise.all(tradePromises);
    console.log(`✓ ${Math.floor(NUM_USERS / 2)} concurrent trades
completed successfully`);

    // Phase 4: Loan Health Updates and Simulated Oracle Price Changes
    console.log("Phase 4: Testing health factor updates with price
changes...");

    // Simulate market volatility - update prices for all collections
    for (const collection of gameCollections) {
      // Reduce price by 10-30% randomly for each collection
      const reduction = 10 + Math.floor(Math.random() * 20);
      const newPrice = NFT_PRICE.mul(100 - reduction).div(100);
      await oracle.setCollectionFloorPrice(collection.address, newPrice);
    }

    // Update health factors for all loans
    const healthUpdatePromises = [];

    for (let i = 0; i < NUM_USERS; i++) {
      const user = users[i];
      const collectionIndex = i % gameCollections.length;
      const collection = gameCollections[collectionIndex];
      const tokenId = i + 1;

      const healthUpdatePromise = (async () => {
        await loanManager.updateHealthFactor(user.address,
collection.address, tokenId);

        // Verify health factor updated
        const health = await loanManager.loanHealthFactors(user.address,
collection.address, tokenId);
        expect(health).to.be.gt(0);
      })();

      healthUpdatePromises.push(healthUpdatePromise);
    }

    await Promise.all(healthUpdatePromises);
    console.log(`✓ ${NUM_USERS} health factor updates completed
successfully`);
```

```
    // Phase 5: Partial Liquidations and Repayments
    console.log("Phase 5: Testing partial liquidations and
repayments...");

    // Simulate severe price drop for half of collections to trigger
liquidations
    const collectionsToLiquidate = gameCollections.slice(0,
Math.ceil(gameCollections.length / 2));
    for (const collection of collectionsToLiquidate) {
      // Drop price by 40%
      const newPrice = NFT_PRICE.mul(60).div(100);
      await oracle.setCollectionFloorPrice(collection.address, newPrice);
    }

    // Process a mix of liquidations and repayments
    const finalPromises = [];

    for (let i = 0; i < NUM_USERS; i++) {
      const user = users[i];
      const collectionIndex = i % gameCollections.length;
      const collection = gameCollections[collectionIndex];
      const tokenId = i + 1;

      // Users with NFTs in first half of collections get liquidated
      const shouldLiquidate = collectionIndex <
Math.ceil(gameCollections.length / 2);

      const actionPromise = (async () => {
        if (shouldLiquidate) {
          // Update health factor and liquidate if needed
          const health = await
loanManager.updateHealthFactor(user.address, collection.address, tokenId);

          // If health is low enough, process liquidation
          if (health < 9000) { // Below first threshold
            await loanManager.liquidatePartial(user.address,
collection.address, tokenId);

            // Verify loan reduced
            const loan = await loanManager.loans(user.address,
collection.address, tokenId);
            const initialLoanPct = 40 + (i % 5) * 5; // 40-60%
            const initialLoanAmount =
NFT_PRICE.mul(initialLoanPct).div(100);
            expect(loan).to.be.lt(initialLoanAmount);
          }
```

```
        } else {
          // Repay 50% of loan
          const loan = await loanManager.loans(user.address,
collection.address, tokenId);
          const repayAmount = loan.div(2);

          await loanManager.connect(user).repay(
            collection.address,
            tokenId,
            repayAmount,
            { value: repayAmount }
          );

          // Verify loan reduced
          const remainingLoan = await loanManager.loans(user.address,
collection.address, tokenId);
          expect(remainingLoan).to.be.closeTo(loan.sub(repayAmount), 100);
// Allow for rounding
        }
      })();

      finalPromises.push(actionPromise);
    }

    await Promise.all(finalPromises);
    console.log(`✓ Final phase of liquidations and repayments completed
successfully`);

    // Summary statistics
    const activeLoans = await countActiveLoans();
    const liquidatedLoans = await countLiquidatedLoans();
    const partiallyRepaidLoans = await countPartiallyRepaidLoans();

    console.log("Load test complete. Summary:");
    console.log(`- Active loans: ${activeLoans}`);
    console.log(`- Partially liquidated: ${liquidatedLoans}`);
    console.log(`- Partially repaid: ${partiallyRepaidLoans}`);
    console.log(`- Total transactions processed: ~${NUM_USERS *
TRANSACTIONS_PER_USER}`);
  });

  // Helper functions
  async function countActiveLoans() {
    let count = 0;
    for (let i = 0; i < NUM_USERS; i++) {
      const user = users[i];
      const collectionIndex = i % gameCollections.length;
```

```
      const collection = gameCollections[collectionIndex];
      const tokenId = i + 1;

      const loan = await loanManager.loans(user.address,
collection.address, tokenId);
      if (loan.gt(0)) count++;
    }
    return count;
  }

  async function countLiquidatedLoans() {
    let count = 0;
    for (let i = 0; i < NUM_USERS; i++) {
      const user = users[i];
      const collectionIndex = i % gameCollections.length;
      const collection = gameCollections[collectionIndex];
      const tokenId = i + 1;

      const events = await loanManager.queryFilter(
        loanManager.filters.LoanLiquidated(user.address,
collection.address, tokenId)
      );
      if (events.length > 0) count++;
    }
    return count;
  }

  async function countPartiallyRepaidLoans() {
    let count = 0;
    for (let i = 0; i < NUM_USERS; i++) {
      const user = users[i];
      const collectionIndex = i % gameCollections.length;
      const collection = gameCollections[collectionIndex];
      const tokenId = i + 1;

      const events = await loanManager.queryFilter(
        loanManager.filters.LoanRepaid(user.address, collection.address,
tokenId)
      );
      if (events.length > 0) count++;
    }
    return count;
  }
});
```

5. Prototype Deliverables

5.1 Technical Deliverables

- Deployed contracts on Saga Protocol devnet
- Interactive web UI with NFT deposit, loan management, and DPO token trading
- Integration with at least 5 popular GameFi projects on Saga Protocol
- Comprehensive testing suite (unit, integration, E2E)
- Performance benchmarking reports
- Security analysis and threat model

5.2 Business Documentation

- Updated whitepaper focused on GameFi lending and Saga integration
- Technical documentation (API, SDK, integration guides)
- User guides and tutorials for GameFi players and investors
- Economic model simulations for various in-game economies
- Market analysis of GameFi assets and yield opportunities

5.3 Community Resources

- Developer documentation for GameFi integrations
- Community forum for feedback and feature requests
- Mosaical SDK for game developers
- Initial business development relationships with Saga Protocol chainlets

MVP PHASE DETAILED BLUEPRINT

1. Core Objectives

The MVP will deliver a fully functional platform with:

- Production-ready smart contracts with comprehensive security measures
- Full-featured web application with intuitive UX for GameFi users
- AI-driven NFT price prediction system optimized for GameFi assets

- Advanced risk management system with GameFi-specific parameters
- Cross-chainlet interoperability for Saga Protocol GameFi assets
- Comprehensive governance system for protocol parameters

2. Technical Architecture

2.1 Smart Contract Suite

```
// Core contract suite for Mosaical MVP

// Governance contract for protocol management
contract MosaicalGovernance {
    // Governance token
    IERC20 public governanceToken;

    // Proposal counter
    uint256 public proposalCount;

    // Proposal details
    struct Proposal {
        uint256 id;
        address proposer;
        string description;
        bytes[] calls;      // Contract calls to execute
        address[] targets;  // Target contracts to call
        uint256[] values;   // ETH values for calls
        bool executed;
        uint256 startBlock;
        uint256 endBlock;
        uint256 forVotes;
        uint256 againstVotes;
    }

    // Proposal mapping
    mapping(uint256 => Proposal) public proposals;

    // Voting receipts
    mapping(uint256 => mapping(address => Receipt)) public receipts;
    struct Receipt {
        bool hasVoted;
        bool support;
        uint256 votes;
    }

    // Vote delegation
```

```
    mapping(address => address) public delegates;
    mapping(address => uint256) public checkpoints;

    // Governance parameters
    uint256 public votingDelay = 13140; // ~2 days in blocks
    uint256 public votingPeriod = 40320; // ~1 week in blocks
    uint256 public proposalThreshold = 1000000 * 10**18; // 1M tokens to
propose
    uint256 public quorumVotes = 5000000 * 10**18; // 5M tokens for quorum

    // Events
    event ProposalCreated(uint256 id, address proposer, string
description);
    event VoteCast(address voter, uint256 proposalId, bool support,
uint256 votes);
    event ProposalExecuted(uint256 id);

    constructor(address _governanceToken) {
        governanceToken = IERC20(_governanceToken);
    }

    // Create a new proposal
    function propose(
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        string memory description
    ) public returns (uint256) {
        require(
            governanceToken.balanceOf(msg.sender) >= proposalThreshold,
            "MosaicalGovernance::propose: below threshold"
        );

        require(targets.length == values.length && targets.length ==
calldatas.length,
            "MosaicalGovernance::propose: invalid proposal"
        );

        proposalCount++;
        Proposal memory newProposal = Proposal({
            id: proposalCount,
            proposer: msg.sender,
            description: description,
            calls: calldatas,
            targets: targets,
            values: values,
            executed: false,
```

```
            startBlock: block.number + votingDelay,
            endBlock: block.number + votingDelay + votingPeriod,
            forVotes: 0,
            againstVotes: 0
        });

        proposals[proposalCount] = newProposal;
        emit ProposalCreated(proposalCount, msg.sender, description);

        return proposalCount;
    }

    // Cast vote on proposal
    function castVote(uint256 proposalId, bool support) public {
        Proposal storage proposal = proposals[proposalId];
        require(block.number >= proposal.startBlock, "Voting not
started");
        require(block.number < proposal.endBlock, "Voting closed");

        Receipt storage receipt = receipts[proposalId][msg.sender];
        require(!receipt.hasVoted, "Already voted");

        // Get voting power (direct + delegated)
        uint256 votes = getVotingPower(msg.sender);

        if (support) {
            proposal.forVotes += votes;
        } else {
            proposal.againstVotes += votes;
        }

        receipt.hasVoted = true;
        receipt.support = support;
        receipt.votes = votes;

        emit VoteCast(msg.sender, proposalId, support, votes);
    }

    // Execute a successful proposal
    function execute(uint256 proposalId) public {
        Proposal storage proposal = proposals[proposalId];
        require(block.number > proposal.endBlock, "Voting still active");
        require(!proposal.executed, "Already executed");
        require(
            proposal.forVotes > proposal.againstVotes &&
            proposal.forVotes >= quorumVotes,
            "Proposal not passed"
```

```
        );

        proposal.executed = true;

        for (uint256 i = 0; i < proposal.targets.length; i++) {
            (bool success, ) = proposal.targets[i].call{value:
proposal.values[i]}(proposal.calls[i]);
            require(success, "Proposal execution failed");
        }

        emit ProposalExecuted(proposalId);
    }

    // Get current voting power of an account
    function getVotingPower(address account) public view returns (uint256)
{
        // Direct voting power
        uint256 votes = governanceToken.balanceOf(account);

        // Delegated power calculation would be implemented here

        return votes;
    }

    // Delegate voting power
    function delegate(address delegatee) public {
        address currentDelegate = delegates[msg.sender];
        delegates[msg.sender] = delegatee;

        // Update voting power checkpoints
        _moveDelegates(currentDelegate, delegatee,
governanceToken.balanceOf(msg.sender));
    }

    // Internal function to update delegate checkpoints
    function _moveDelegates(address srcRep, address dstRep, uint256
amount) internal {
        // Implementation for checkpoint accounting
        // Omitted for blueprint brevity
    }

    // Update governance parameters (only through governance)
    function updateVotingDelay(uint256 newDelay) external onlyGov {
        votingDelay = newDelay;
    }

    function updateVotingPeriod(uint256 newPeriod) external onlyGov {
```

```
        votingPeriod = newPeriod;
    }

    function updateProposalThreshold(uint256 newThreshold) external
onlyGov {
        proposalThreshold = newThreshold;
    }

    function updateQuorumVotes(uint256 newQuorum) external onlyGov {
        quorumVotes = newQuorum;
    }

    // Access control
    modifier onlyGov() {
        require(msg.sender == address(this), "Only governance");
        _;
    }
}

// NFT Vault V3 with risk tiers and enhanced GameFi support
contract NFTVaultV3 {
    // Extended storage from V2
    mapping(address => mapping(address => mapping(uint256 => bool)))
public deposits;
    mapping(address => bool) public supportedChainlets;
    mapping(address => mapping(address => bool)) public
supportedCollections;
    mapping(address => uint8) public gameCategory;
    mapping(address => mapping(uint256 => bytes)) public nftMetadataCache;

    // New storage for V3
    // Risk tier for each collection (1-5, where 1 is lowest risk, 5 is
highest)
    mapping(address => uint8) public collectionRiskTier;

    // Risk models by tier (parameters, thresholds)
    struct RiskModel {
        uint256 baseLTV; // Base LTV in percentage (e.g., 50 = 50%)
        uint256 liquidationThreshold; // LT in percentage (e.g., 65 = 65%)
        uint256 maxUtilityBonus; // Maximum boost from utility in
percentage (e.g., 20 = 20%)
        uint256 minCollateralAmount; // Minimum collateral value required
    }

    // Risk model params by tier
    mapping(uint8 => RiskModel) public riskModels;
```

```
// Enhanced NFT data structure
struct NFTData {
    address collection;
    uint256 tokenId;
    uint8 gameTier; // In-game tier/rarity
    uint256 utilityScore;
    bool isStaked; // If NFT is actively used in-game
    uint256 lastActivityTimestamp;
}

// NFT data mapping
mapping(address => mapping(uint256 => NFTData)) public nftData;

// Initialize risk models
constructor() {
    // Set default risk model parameters
    riskModels[1] = RiskModel({
        baseLTV: 70,
        liquidationThreshold: 80,
        maxUtilityBonus: 20,
        minCollateralAmount: 0.1 ether
    });

    riskModels[2] = RiskModel({
        baseLTV: 65,
        liquidationThreshold: 75,
        maxUtilityBonus: 15,
        minCollateralAmount: 0.25 ether
    });

    riskModels[3] = RiskModel({
        baseLTV: 60,
        liquidationThreshold: 70,
        maxUtilityBonus: 10,
        minCollateralAmount: 0.5 ether
    });

    riskModels[4] = RiskModel({
        baseLTV: 50,
        liquidationThreshold: 65,
        maxUtilityBonus: 10,
        minCollateralAmount: 1 ether
    });

    riskModels[5] = RiskModel({
        baseLTV: 40,
        liquidationThreshold: 55,
```

```
            maxUtilityBonus: 5,
            minCollateralAmount: 2 ether
        });
    }

    // Enhanced deposit with GameFi metadata parsing
    function depositNFT(address collection, uint256 tokenId) external {
        require(isGameFiNFT(collection), "Not a supported GameFi NFT");

        // Transfer NFT to vault
        IERC721(collection).transferFrom(msg.sender, address(this),
tokenId);
        deposits[msg.sender][collection][tokenId] = true;

        // Fetch and process NFT metadata
        NFTData memory data = parseNFTGameData(collection, tokenId);
        nftData[collection][tokenId] = data;

        emit NFTDeposited(msg.sender, collection, tokenId,
data.utilityScore, data.gameTier);
    }

    // Parse game-specific data from NFT metadata
    function parseNFTGameData(address collection, uint256 tokenId)
internal returns (NFTData memory) {
        NFTData memory data;
        data.collection = collection;
        data.tokenId = tokenId;

        // Fetch token URI
        string memory tokenURI =
IERC721Metadata(collection).tokenURI(tokenId);

        // Cache metadata
        nftMetadataCache[collection][tokenId] = bytes(tokenURI);

        // In production: Parse JSON from tokenURI to extract
game-specific attributes
        // For MVP blueprint: Use oracle or API to get game-specific data

        // Example of fetching from oracle or game API
        (uint8 gameTier, uint256 utilityScore, bool isStaked) =
gameDataOracle.getNFTGameData(collection, tokenId);

        data.gameTier = gameTier;
        data.utilityScore = utilityScore;
        data.isStaked = isStaked;
```

```
        data.lastActivityTimestamp = block.timestamp;

        return data;
    }

    // Get max LTV for an NFT based on risk tier and utility
    function getMaxLTV(address collection, uint256 tokenId) public view
returns (uint256) {
        uint8 tier = collectionRiskTier[collection];
        if (tier == 0) tier = 3; // Default to middle tier

        RiskModel memory model = riskModels[tier];

        // Get NFT utility score
        uint256 utilityScore = nftData[collection][tokenId].utilityScore;
        if (utilityScore == 0) utilityScore = 100; // Default base score

        // Calculate utility bonus (capped by maxUtilityBonus)
        uint256 utilityBonus = 0;
        if (utilityScore > 100) {
            // Convert to percentage points (e.g., 150 score = 50% above
baseline)
            utilityBonus = ((utilityScore - 100) * model.maxUtilityBonus)
/ 100;
            if (utilityBonus > model.maxUtilityBonus) {
                utilityBonus = model.maxUtilityBonus;
            }
        }

        // Final LTV with utility bonus
        return model.baseLTV + utilityBonus;
    }

    // Set collection risk tier (admin/governance only)
    function setCollectionRiskTier(address collection, uint8 tier)
external onlyAdminOrGovernance {
        require(tier >= 1 && tier <= 5, "Invalid risk tier");
        collectionRiskTier[collection] = tier;
        emit RiskTierUpdated(collection, tier);
    }

    // Update risk model parameters (governance only)
    function updateRiskModel(
        uint8 tier,
        uint256 baseLTV,
        uint256 liquidationThreshold,
        uint256 maxUtilityBonus,
```

```
        uint256 minCollateralAmount
    ) external onlyGovernance {
        require(tier >= 1 && tier <= 5, "Invalid risk tier");
        require(baseLTV < liquidationThreshold, "LTV must be < liquidation
threshold");
        require(liquidationThreshold <= 90, "Liquidation threshold too
high");

        riskModels[tier] = RiskModel({
            baseLTV: baseLTV,
            liquidationThreshold: liquidationThreshold,
            maxUtilityBonus: maxUtilityBonus,
            minCollateralAmount: minCollateralAmount
        });

        emit RiskModelUpdated(tier, baseLTV, liquidationThreshold,
maxUtilityBonus);
    }

    // Additional functions from V2 with needed enhancements...
}

// Loan Manager V3 with dynamic interest and health factors
contract LoanManagerV3 {
    // Extended storage from V2
    mapping(address => mapping(address => mapping(uint256 => uint256)))
public loans;
    IPriceOracle public oracle;
    mapping(address => uint256) public collectionInterestRates;
    mapping(address => mapping(address => mapping(uint256 => uint256)))
public loanHealthFactors;

    // New storage for V3
    // Dynamic interest rate model parameters
    struct InterestRateModel {
        uint256 baseRate; // Base interest rate in basis points (e.g., 200
= 2%)
        uint256 slopeUtilization1; // Slope below optimal utilization
        uint256 slopeUtilization2; // Slope above optimal utilization
        uint256 optimalUtilization; // Optimal utilization point (e.g.,
8000 = 80%)
    }

    // Interest rate model by collection
    mapping(address => InterestRateModel) public interestRateModels;

    // Default interest rate model
```

```
    InterestRateModel public defaultInterestModel;

    // Utilization by collection
    mapping(address => uint256) public collectionUtilization; // Scaled by
10000

    // Total borrowed and supplied by collection
    mapping(address => uint256) public totalBorrowed;
    mapping(address => uint256) public totalSupplied;

    // Enhanced loan data
    struct LoanData {
        uint256 principal;
        uint256 originationFee;
        uint256 interestRate;
        uint256 lastInterestUpdate;
        uint256 accruedInterest;
    }

    // Loan data mapping
    mapping(address => mapping(address => mapping(uint256 => LoanData)))
public loanData;

    constructor() {
        // Initialize default interest rate model
        defaultInterestModel = InterestRateModel({
            baseRate: 200, // 2%
            slopeUtilization1: 1000, // 10% slope below optimal
            slopeUtilization2: 5000, // 50% slope above optimal
            optimalUtilization: 8000 // 80% optimal utilization
        });
    }

    // Calculate current interest rate based on utilization
    function calculateInterestRate(address collection) public view returns
(uint256) {
        InterestRateModel memory model = interestRateModels[collection];
        if (model.baseRate == 0) {
            model = defaultInterestModel; // Use default if not set
        }

        uint256 utilization = collectionUtilization[collection];
        uint256 interestRate;

        if (utilization <= model.optimalUtilization) {
            // Below optimal: baseRate + slope1 * (util / optimal)
            interestRate = model.baseRate +
```

```
                (model.slopeUtilization1 * utilization) /
model.optimalUtilization;
        } else {
            // Above optimal: baseRate + slope1 + slope2 * (util -
optimal) / (10000 - optimal)
            interestRate = model.baseRate + model.slopeUtilization1 +
                (model.slopeUtilization2 * (utilization -
model.optimalUtilization)) /
                (10000 - model.optimalUtilization);
        }

        return interestRate;
    }

    // Enhanced borrow function with interest model
    function borrow(address collection, uint256 tokenId, uint256 amount)
external {
        require(nftVault.deposits(msg.sender, collection, tokenId), "Not
your NFT");

        // Get NFT price from oracle
        uint256 nftPrice = oracle.getNFTPrice(collection, tokenId);

        // Get max LTV from NFT vault (accounts for risk tier and utility)
        uint256 maxLTV = nftVault.getMaxLTV(collection, tokenId);
        uint256 maxLoan = (nftPrice * maxLTV) / 100;

        require(amount <= maxLoan, "Exceeds max LTV");
        require(nftPrice >=
nftVault.riskModels(nftVault.collectionRiskTier(collection)).minCollateral
Amount,
                "Collateral below minimum");

        // Calculate origination fee (0.5% = 50 basis points)
        uint256 originationFee = (amount * 50) / 10000;
        uint256 netBorrowAmount = amount - originationFee;

        // Store loan amount
        loans[msg.sender][collection][tokenId] = amount;

        // Update total borrowed for collection
        totalBorrowed[collection] += amount;

        // Update utilization rate
        _updateUtilization(collection);

        // Get current interest rate
```

```
        uint256 interestRate = calculateInterestRate(collection);

        // Store loan data
        loanData[msg.sender][collection][tokenId] = LoanData({
            principal: amount,
            originationFee: originationFee,
            interestRate: interestRate,
            lastInterestUpdate: block.timestamp,
            accruedInterest: 0
        });

        // Initialize health factor
        uint256 initialHealth = (nftPrice * 10000) / amount;
        loanHealthFactors[msg.sender][collection][tokenId] =
initialHealth;

        // Issue DPO tokens for the loan
        dpoToken.mintDPOTokens(msg.sender, collection, tokenId, amount);

        // Transfer net loan amount to borrower
        IERC20(lendingToken).transfer(msg.sender, netBorrowAmount);

        // Add origination fee to treasury
        IERC20(lendingToken).transfer(treasury, originationFee);

        emit LoanCreated(
            msg.sender,
            collection,
            tokenId,
            amount,
            initialHealth,
            interestRate,
            originationFee
        );
    }

    // Update loan interest accrued
    function updateLoanInterest(address user, address collection, uint256
tokenId) public returns (uint256) {
        LoanData storage loan = loanData[user][collection][tokenId];
        if (loan.principal == 0) return 0;

        uint256 timeElapsed = block.timestamp - loan.lastInterestUpdate;
        if (timeElapsed == 0) return loan.accruedInterest;

        // Calculate accrued interest: principal * rate * time
        // rate is in basis points (1% = 100)
```

```
        // time is in seconds converted to years
        uint256 newInterest = (loan.principal * loan.interestRate *
timeElapsed) / (10000 * 365 days);

        // Update loan data
        loan.accruedInterest += newInterest;
        loan.lastInterestUpdate = block.timestamp;

        return loan.accruedInterest;
    }

    // Enhanced repay function with interest payment
    function repay(address collection, uint256 tokenId, uint256 amount)
external {
        uint256 loanAmount = loans[msg.sender][collection][tokenId];
        require(loanAmount > 0, "No loan exists");

        // Update accrued interest
        uint256 interest = updateLoanInterest(msg.sender, collection,
tokenId);

        // Calculate total owed (principal + interest)
        uint256 totalOwed = loanAmount + interest;

        // Determine repayment allocation (first to interest, then
principal)
        uint256 repayAmount = amount > totalOwed ? totalOwed : amount;
        uint256 interestPayment = repayAmount > interest ? interest :
repayAmount;
        uint256 principalPayment = repayAmount - interestPayment;

        // Update loan state
        if (principalPayment > 0) {
            loans[msg.sender][collection][tokenId] -= principalPayment;
            loanData[msg.sender][collection][tokenId].principal -=
principalPayment;

            // Update total borrowed for collection
            totalBorrowed[collection] -= principalPayment;

            // Update utilization rate
            _updateUtilization(collection);
        }

        // Update accrued interest
        if (interestPayment > 0) {
```

```
            loanData[msg.sender][collection][tokenId].accruedInterest -=
interestPayment;
        }

        // Transfer repayment
        IERC20(lendingToken).transferFrom(msg.sender, address(this),
repayAmount);

        // Distribute interest to lenders/treasury
        if (interestPayment > 0) {
            // 80% to lenders (DPO token holders), 20% to treasury
            uint256 treasuryFee = (interestPayment * 20) / 100;
            IERC20(lendingToken).transfer(treasury, treasuryFee);

            // Distribute remaining interest to DPO token holders
            dpoToken.distributeInterest(collection, tokenId,
interestPayment - treasuryFee);
        }

        // If fully repaid, burn DPO tokens proportionally
        if (loans[msg.sender][collection][tokenId] == 0) {
            dpoToken.repaymentBurn(msg.sender, collection, tokenId);
        }

        emit LoanRepaid(
            msg.sender,
            collection,
            tokenId,
            principalPayment,
            interestPayment
        );
    }

    // Update utilization for a collection
    function _updateUtilization(address collection) internal {
        uint256 borrowed = totalBorrowed[collection];
        uint256 supplied = totalSupplied[collection];

        if (supplied == 0) {
            collectionUtilization[collection] = 0;
        } else {
            collectionUtilization[collection] = (borrowed * 10000) /
supplied;
        }
    }

    // Enhanced health factor calculation
```

```
    function calculateHealthFactor(
        address user,
        address collection,
        uint256 tokenId
    ) public view returns (uint256) {
        uint256 loanAmount = loans[user][collection][tokenId];
        if (loanAmount == 0) return type(uint256).max; // No loan =
infinite health

        // Get accrued interest
        LoanData memory loan = loanData[user][collection][tokenId];
        uint256 timeElapsed = block.timestamp - loan.lastInterestUpdate;
        uint256 additionalInterest = (loan.principal * loan.interestRate *
timeElapsed) / (10000 * 365 days);
        uint256 totalDebt = loanAmount + loan.accruedInterest +
additionalInterest;

        // Get current NFT price
        uint256 nftPrice = oracle.getNFTPrice(collection, tokenId);

        // Calculate health factor: collateral value / total debt
        return (nftPrice * 10000) / totalDebt;
    }

    // Update health factor for a loan
    function updateHealthFactor(address user, address collection, uint256
tokenId) public returns (uint256) {
        // Update accrued interest first
        updateLoanInterest(user, collection, tokenId);

        // Calculate new health factor
        uint256 newHealth = calculateHealthFactor(user, collection,
tokenId);

        // Update health factor
        loanHealthFactors[user][collection][tokenId] = newHealth;

        return newHealth;
    }

    // Set interest rate model for collection
    function setInterestRateModel(
        address collection,
        uint256 baseRate,
        uint256 slopeUtilization1,
        uint256 slopeUtilization2,
        uint256 optimalUtilization
```

```
    ) external onlyAdminOrGovernance {
        require(optimalUtilization < 10000, "Invalid optimal
utilization");

        interestRateModels[collection] = InterestRateModel({
            baseRate: baseRate,
            slopeUtilization1: slopeUtilization1,
            slopeUtilization2: slopeUtilization2,
            optimalUtilization: optimalUtilization
        });

        emit InterestRateModelUpdated(
            collection,
            baseRate,
            slopeUtilization1,
            slopeUtilization2,
            optimalUtilization
        );
    }

    // Additional functions from V2 with needed enhancements...
}

// DPO Token V3 with enhanced trading and tracking
contract DPOTokenV3 is ERC20 {
    // Extended storage from V2
    address public admin;
    address public loanManager;
    mapping(address => mapping(uint256 => uint256)) public nftTokenSupply;
    mapping(address => mapping(uint256 => mapping(address => uint256)))
public tokenHoldings;
    mapping(address => mapping(uint256 => address)) public nftToDPOToken;
    uint256 public tradingFee;
    address public feeRecipient;

    // New storage for V3
    // DPO token trading markets
    struct MarketOrder {
        address maker;
        uint256 price; // Per token price in lending token, scaled by
10^18
        uint256 amount;
        uint256 timestamp;
        bool isBuy; // true = buy order, false = sell order
    }

    // Order book for each NFT's DPO tokens
```

```
    mapping(address => mapping(uint256 => MarketOrder[])) public
buyOrders;
    mapping(address => mapping(uint256 => MarketOrder[])) public
sellOrders;

    // Order IDs for tracking
    mapping(address => mapping(uint256 => mapping(address => uint256[])))
public userOrders;

    // Interest distribution tracking
    struct InterestAccrual {
        uint256 totalDistributed;
        uint256 lastDistributionTimestamp;
        mapping(address => uint256) lastClaimedTimestamp;
        mapping(address => uint256) unclaimed;
    }

    // Interest accrual by NFT
    mapping(address => mapping(uint256 => InterestAccrual)) public
interestAccrual;

    // Enhanced mint function
    function mintDPOTokens(
        address user,
        address collection,
        uint256 tokenId,
        uint256 loanAmount
    ) external onlyLoanManager {
        // Same as V2 implementation
        // Calculate token supply based on loan amount
        // 1 token per 0.001 ETH in loan value (configurable)
        uint256 tokenSupply = (loanAmount * 1000) / (10**15);

        _mint(user, tokenSupply);
        nftTokenSupply[collection][tokenId] = tokenSupply;
        tokenHoldings[collection][tokenId][user] = tokenSupply;

        // Store reverse mapping for lookup
        string memory tokenName = string(abi.encodePacked("DPO-",
IERC721Metadata(collection).symbol(), "-", tokenId.toString()));
        ERC20 newDpoToken = _createChildToken(tokenName, tokenName);
        nftToDPOToken[collection][tokenId] = address(newDpoToken);

        emit DPOTokensMinted(user, collection, tokenId, tokenSupply);
    }

    // Place buy order for DPO tokens
```

```
    function placeBuyOrder(
        address collection,
        uint256 tokenId,
        uint256 amount,
        uint256 pricePerToken
    ) external {
        require(amount > 0, "Invalid amount");
        require(pricePerToken > 0, "Invalid price");

        // Calculate total cost
        uint256 totalCost = amount * pricePerToken;

        // Transfer funds to contract as escrow
        IERC20(lendingToken).transferFrom(msg.sender, address(this),
totalCost);

        // Create order
        MarketOrder memory order = MarketOrder({
            maker: msg.sender,
            price: pricePerToken,
            amount: amount,
            timestamp: block.timestamp,
            isBuy: true
        });

        // Add to order book
        uint256 orderId = buyOrders[collection][tokenId].length;
        buyOrders[collection][tokenId].push(order);

        // Track user orders
        userOrders[collection][tokenId][msg.sender].push(orderId);

        emit BuyOrderPlaced(
            msg.sender,
            collection,
            tokenId,
            orderId,
            amount,
            pricePerToken
        );

        // Check if order can be matched immediately
        _tryMatchOrders(collection, tokenId);
    }

    // Place sell order for DPO tokens
    function placeSellOrder(
```

```
        address collection,
        uint256 tokenId,
        uint256 amount,
        uint256 pricePerToken
    ) external {
        require(amount > 0, "Invalid amount");
        require(pricePerToken > 0, "Invalid price");
        require(tokenHoldings[collection][tokenId][msg.sender] >= amount,
"Insufficient tokens");

        // Transfer tokens to contract as escrow
        tokenHoldings[collection][tokenId][msg.sender] -= amount;

        // Create order
        MarketOrder memory order = MarketOrder({
            maker: msg.sender,
            price: pricePerToken,
            amount: amount,
            timestamp: block.timestamp,
            isBuy: false
        });

        // Add to order book
        uint256 orderId = sellOrders[collection][tokenId].length;
        sellOrders[collection][tokenId].push(order);

        // Track user orders
        userOrders[collection][tokenId][msg.sender].push(orderId);

        emit SellOrderPlaced(
            msg.sender,
            collection,
            tokenId,
            orderId,
            amount,
            pricePerToken
        );

        // Check if order can be matched immediately
        _tryMatchOrders(collection, tokenId);
    }

    // Cancel order
    function cancelOrder(
        address collection,
        uint256 tokenId,
        bool isBuy,
```

```
        uint256 orderId
    ) external {
        MarketOrder[] storage orders = isBuy ?
            buyOrders[collection][tokenId] :
            sellOrders[collection][tokenId];

        require(orderId < orders.length, "Invalid order ID");
        require(orders[orderId].maker == msg.sender, "Not your order");
        require(orders[orderId].amount > 0, "Order already filled");

        uint256 amount = orders[orderId].amount;

        if (isBuy) {
            // Refund escrowed funds
            uint256 refundAmount = amount * orders[orderId].price;
            IERC20(lendingToken).transfer(msg.sender, refundAmount);
        } else {
            // Return escrowed tokens
            tokenHoldings[collection][tokenId][msg.sender] += amount;
        }

        // Mark order as filled (amount = 0)
        orders[orderId].amount = 0;

        emit OrderCancelled(
            msg.sender,
            collection,
            tokenId,
            orderId,
            isBuy
        );
    }

    // Try to match buy and sell orders
    function _tryMatchOrders(address collection, uint256 tokenId) internal
{
        MarketOrder[] storage buys = buyOrders[collection][tokenId];
        MarketOrder[] storage sells = sellOrders[collection][tokenId];

        // Find highest buy and lowest sell
        uint256 bestBuyId = _findBestBuyOrder(collection, tokenId);
        uint256 bestSellId = _findBestSellOrder(collection, tokenId);

        if (bestBuyId == type(uint256).max || bestSellId ==
type(uint256).max) {
            return; // No valid orders to match
        }
```

```
        MarketOrder storage bestBuy = buys[bestBuyId];
        MarketOrder storage bestSell = sells[bestSellId];

        // Check if orders can be matched
        if (bestBuy.price >= bestSell.price) {
            // Orders match - execute trade
            uint256 execPrice = bestSell.price; // Seller's price (lower
gets priority)
            uint256 execAmount = bestBuy.amount < bestSell.amount ?
bestBuy.amount : bestSell.amount;

            // Calculate value
            uint256 tradeValue = execAmount * execPrice;

            // Calculate and deduct fee
            uint256 fee = (tradeValue * tradingFee) / 10000;
            uint256 sellerProceeds = tradeValue - fee;

            // Transfer tokens from seller escrow to buyer
            tokenHoldings[collection][tokenId][bestBuy.maker] +=
execAmount;

            // Transfer payment from buyer escrow to seller
            IERC20(lendingToken).transfer(bestSell.maker, sellerProceeds);

            // Transfer fee to fee recipient
            IERC20(lendingToken).transfer(feeRecipient, fee);

            // Update order amounts
            bestBuy.amount -= execAmount;
            bestSell.amount -= execAmount;

            // If buyer paid more than execution price, refund difference
            if (bestBuy.price > execPrice) {
                uint256 refund = execAmount * (bestBuy.price - execPrice);
                IERC20(lendingToken).transfer(bestBuy.maker, refund);
            }

            emit OrdersMatched(
                collection,
                tokenId,
                bestBuyId,
                bestSellId,
                execAmount,
                execPrice,
                fee
```

```
            );

            // Recursively try to match more orders
            if (bestBuy.amount > 0 && bestSell.amount > 0) {
                _tryMatchOrders(collection, tokenId);
            }
        }
    }

    // Find best buy order (highest price)
    function _findBestBuyOrder(address collection, uint256 tokenId)
internal view returns (uint256) {
        MarketOrder[] storage buys = buyOrders[collection][tokenId];
        uint256 bestPrice = 0;
        uint256 bestId = type(uint256).max;

        for (uint256 i = 0; i < buys.length; i++) {
            if (buys[i].amount > 0 && buys[i].price > bestPrice) {
                bestPrice = buys[i].price;
                bestId = i;
            }
        }

        return bestId;
    }

    // Find best sell order (lowest price)
    function _findBestSellOrder(address collection, uint256 tokenId)
internal view returns (uint256) {
        MarketOrder[] storage sells = sellOrders[collection][tokenId];
        uint256 bestPrice = type(uint256).max;
        uint256 bestId = type(uint256).max;

        for (uint256 i = 0; i < sells.length; i++) {
            if (sells[i].amount > 0 && sells[i].price < bestPrice) {
                bestPrice = sells[i].price;
                bestId = i;
            }
        }

        return bestId;
    }

    // Distribute interest to DPO token holders
    function distributeInterest(
        address collection,
        uint256 tokenId,
```

```
        uint256 interestAmount
    ) external onlyLoanManager {
        InterestAccrual storage accrual =
interestAccrual[collection][tokenId];
        accrual.totalDistributed += interestAmount;
        accrual.lastDistributionTimestamp = block.timestamp;

        emit InterestDistributed(collection, tokenId, interestAmount);
    }

    // Claim pending interest
    function claimInterest(address collection, uint256 tokenId) external {
        InterestAccrual storage accrual =
interestAccrual[collection][tokenId];
        uint256 unclaimed = accrual.unclaimed[msg.sender];

        require(unclaimed > 0, "No interest to claim");

        // Update tracking
        accrual.unclaimed[msg.sender] = 0;
        accrual.lastClaimedTimestamp[msg.sender] = block.timestamp;

        // Transfer interest
        IERC20(lendingToken).transfer(msg.sender, unclaimed);

        emit InterestClaimed(msg.sender, collection, tokenId, unclaimed);
    }

    // Calculate pending interest for a user
    function calculatePendingInterest(
        address user,
        address collection,
        uint256 tokenId
    ) public view returns (uint256) {
        uint256 userBalance = tokenHoldings[collection][tokenId][user];
        if (userBalance == 0) return 0;

        InterestAccrual storage accrual =
interestAccrual[collection][tokenId];
        uint256 totalSupply = nftTokenSupply[collection][tokenId];

        // First add any stored unclaimed amount
        uint256 pending = accrual.unclaimed[user];

        // Calculate portion of any new distributions
        uint256 lastClaimed = accrual.lastClaimedTimestamp[user];
        if (lastClaimed < accrual.lastDistributionTimestamp) {
```

```
            // User's share of interest = their balance / total supply
            uint256 share = (userBalance * 10000) / totalSupply;
            uint256 newInterest = (accrual.totalDistributed * share) /
10000;

            pending += newInterest;
        }

        return pending;
    }

    // Burn DPO tokens on full loan repayment
    function repaymentBurn(
        address borrower,
        address collection,
        uint256 tokenId
    ) external onlyLoanManager {
        // Get total supply
        uint256 totalSupply = nftTokenSupply[collection][tokenId];
        if (totalSupply == 0) return;

        // Get all holders
        address[] memory holders = getTokenHolders(collection, tokenId);

        // Distribute any remaining interest
        InterestAccrual storage accrual =
interestAccrual[collection][tokenId];

        for (uint256 i = 0; i < holders.length; i++) {
            address holder = holders[i];
            uint256 balance = tokenHoldings[collection][tokenId][holder];

            if (balance == 0) continue;

            // Calculate final interest distribution
            uint256 pendingInterest = calculatePendingInterest(holder,
collection, tokenId);
            if (pendingInterest > 0) {
                accrual.unclaimed[holder] += pendingInterest;
            }

            // Burn tokens
            tokenHoldings[collection][tokenId][holder] = 0;
            _burn(holder, balance);

            emit DPOTokensBurned(holder, collection, tokenId, balance);
        }
```

```
        // Reset token supply
        nftTokenSupply[collection][tokenId] = 0;
    }

    // Additional functions from V2 with needed enhancements...
}

// Oracle System V3 with AI Integration
contract GameFiOracleV3 {
    // Extended storage from V2
    mapping(address => uint256) public collectionFloorPrices;
    mapping(address => mapping(uint256 => uint256)) public
nftUtilityScores;
    mapping(address => PricePoint[30]) public priceHistory;
    mapping(address => uint8) public historyIndex;
    mapping(address => uint256) public collectionVolatility;

    // New storage for V3
    // AI price prediction integration
    address public aiPricePredictionSystem;

    // Confidence scores for price predictions (0-100)
    mapping(address => uint256) public predictionConfidence;

    // Price source weights (dynamic adjustment)
    struct PriceSource {
        string name;
        uint256 weight; // Basis points, total should be 10000
        uint256 reliability; // 0-100 score for reliability
        uint256 lastUpdateTimestamp;
    }

    mapping(address => PriceSource[]) public priceSources;

    // Game activity metrics
    struct GameMetrics {
        uint256 activeUsers; // Daily active users
        uint256 avgPlaytime; // Average playtime in minutes
        uint256 revenue; // Daily revenue in USD
        uint256 retention; // 30-day retention rate (basis points)
    }

    mapping(address => GameMetrics) public gameMetrics;

    // Oracle feeder permissions
    mapping(address => bool) public authorizedFeeders;
```

```
    // Set AI price prediction system
    function setAIPredictionSystem(address _aiSystem) external onlyAdmin {
        aiPricePredictionSystem = _aiSystem;
    }

    // Get NFT price with AI integration
    function getNFTPrice(address collection, uint256 tokenId) external
view returns (uint256) {
        // Get base floor price
        uint256 floorPrice = collectionFloorPrices[collection];

        // Get utility score
        uint256 utilityScore = nftUtilityScores[collection][tokenId];
        if (utilityScore == 0) utilityScore = 100; // Default to baseline

        // Try to get AI-enhanced price prediction if available
        uint256 aiPricePrediction = 0;
        if (aiPricePredictionSystem != address(0)) {
            try
IAIPricePredictor(aiPricePredictionSystem).predictNFTPrice(collection,
tokenId) returns (uint256 price, uint256 confidence) {
                if (confidence >= 70) { // Only use AI if confidence is
high enough
                    aiPricePrediction = price;

                    // Blend prices based on confidence
                    // Higher confidence in AI = more weight to AI price
                    uint256 aiWeight = confidence * 100; // 70-100
confidence = 7000-10000 weight
                    uint256 blendedPrice = ((floorPrice * (10000 -
aiWeight)) + (aiPricePrediction * aiWeight)) / 10000;

                    // Apply utility adjustments to the blended price
                    return (blendedPrice * utilityScore) / 100;
                }
            } catch {
                // Fallback to standard pricing if AI call fails
            }
        }

        // If no AI price or low confidence, use standard pricing with
utility adjustment
        uint256 basePrice = (floorPrice * utilityScore) / 100;

        // Apply volatility discount for high-volatility collections
        uint256 volatility = collectionVolatility[collection];
```

```
        if (volatility > 0) {
            // Max discount of 10% at 5000 basis points (50%) daily
volatility
            uint256 volatilityDiscount = (volatility * 1000) / 5000;
            if (volatilityDiscount > 1000) volatilityDiscount = 1000; //
Cap at 10%

            basePrice = basePrice - ((basePrice * volatilityDiscount) /
10000);
        }

        return basePrice;
    }

    // Update game metrics
    function updateGameMetrics(
        address collection,
        uint256 activeUsers,
        uint256 avgPlaytime,
        uint256 revenue,
        uint256 retention
    ) external onlyAuthorizedFeeder {
        gameMetrics[collection] = GameMetrics({
            activeUsers: activeUsers,
            avgPlaytime: avgPlaytime,
            revenue: revenue,
            retention: retention
        });

        emit GameMetricsUpdated(
            collection,
            activeUsers,
            avgPlaytime,
            revenue,
            retention
        );
    }

    // Get game engagement factor (normalized 0-100)
    function getGameEngagementFactor(address collection) public view
returns (uint256) {
        GameMetrics memory metrics = gameMetrics[collection];

        if (metrics.activeUsers == 0) return 50; // Default to neutral

        // Combine metrics into a single engagement score
```

```
        // This is a simplified example - production would use more
sophisticated weighting

        // User base score (0-30)
        uint256 userScore = 0;
        if (metrics.activeUsers >= 100000) userScore = 30;
        else if (metrics.activeUsers >= 50000) userScore = 25;
        else if (metrics.activeUsers >= 10000) userScore = 20;
        else if (metrics.activeUsers >= 5000) userScore = 15;
        else if (metrics.activeUsers >= 1000) userScore = 10;
        else userScore = 5;

        // Playtime score (0-20)
        uint256 playtimeScore = 0;
        if (metrics.avgPlaytime >= 120) playtimeScore = 20;
        else if (metrics.avgPlaytime >= 60) playtimeScore = 15;
        else if (metrics.avgPlaytime >= 30) playtimeScore = 10;
        else playtimeScore = 5;

        // Revenue score (0-30)
        uint256 revenueScore = 0;
        if (metrics.revenue >= 100000) revenueScore = 30;
        else if (metrics.revenue >= 50000) revenueScore = 25;
        else if (metrics.revenue >= 10000) revenueScore = 20;
        else if (metrics.revenue >= 5000) revenueScore = 15;
        else if (metrics.revenue >= 1000) revenueScore = 10;
        else revenueScore = 5;

        // Retention score (0-20)
        uint256 retentionScore = 0;
        if (metrics.retention >= 8000) retentionScore = 20; // 80%+
retention
        else if (metrics.retention >= 6000) retentionScore = 15; // 60%+
retention
        else if (metrics.retention >= 4000) retentionScore = 10; // 40%+
retention
        else retentionScore = 5;

        // Sum all scores
        return userScore + playtimeScore + revenueScore + retentionScore;
    }

    // Add or update a price source
    function updatePriceSource(
        address collection,
        string memory name,
        uint256 weight,
```

```
        uint256 reliability
    ) external onlyAdmin {
        // Find existing source or add new one
        bool found = false;
        for (uint256 i = 0; i < priceSources[collection].length; i++) {
            if (keccak256(bytes(priceSources[collection][i].name)) ==
keccak256(bytes(name))) {
                priceSources[collection][i].weight = weight;
                priceSources[collection][i].reliability = reliability;
                priceSources[collection][i].lastUpdateTimestamp =
block.timestamp;
                found = true;
                break;
            }
        }

        if (!found) {
            priceSources[collection].push(PriceSource({
                name: name,
                weight: weight,
                reliability: reliability,
                lastUpdateTimestamp: block.timestamp
            }));
        }

        // Validate that weights sum to 10000
        uint256 totalWeight = 0;
        for (uint256 i = 0; i < priceSources[collection].length; i++) {
            totalWeight += priceSources[collection][i].weight;
        }
        require(totalWeight == 10000, "Weights must sum to 10000");

        emit PriceSourceUpdated(collection, name, weight, reliability);
    }

    // Import AI model prediction confidence
    function updatePredictionConfidence(
        address collection,
        uint256 confidence
    ) external onlyAISystem {
        require(confidence <= 100, "Confidence must be 0-100");
        predictionConfidence[collection] = confidence;

        emit PredictionConfidenceUpdated(collection, confidence);
    }

    // Authorize oracle feeders
```

```
    function setOracleFeeder(address feeder, bool authorized) external
onlyAdmin {
        authorizedFeeders[feeder] = authorized;
    }

    // Access control modifiers
    modifier onlyAuthorizedFeeder() {
        require(authorizedFeeders[msg.sender], "Not authorized feeder");
        _;
    }

    modifier onlyAISystem() {
        require(msg.sender == aiPricePredictionSystem, "Not AI system");
        _;
    }

    // Additional functions from V2 with needed enhancements...
}

// Cross-Chain Bridge for Saga Protocol Integration
contract MosaicalSagaBridge {
    // LayerZero endpoint for cross-chain messaging
    ILayerZeroEndpoint public endpoint;

    // Supported chainlets
    mapping(uint16 => bool) public supportedChainlets;

    // Collection mapping (local address -> remote address)
    mapping(address => mapping(uint16 => address)) public remoteMappings;

    // Trusted remote addresses
    mapping(uint16 => bytes) public trustedRemoteLookup;

    // Pending bridged NFTs
    struct PendingNFT {
        address collection;
        uint256 tokenId;
        address owner;
        bool isBridged;
    }

    mapping(bytes32 => PendingNFT) public pendingNFTs;

    // Events
    event NFTBridgeInitiated(address indexed collection, uint256 indexed
tokenId, address indexed owner, uint16 dstChainId);
```

```
    event NFTBridgeCompleted(address indexed collection, uint256 indexed
tokenId, address indexed owner, uint16 srcChainId);

    constructor(address _endpoint) {
        endpoint = ILayerZeroEndpoint(_endpoint);
    }

    // Add supported chainlet
    function addSupportedChainlet(uint16 chainletId) external onlyAdmin {
        supportedChainlets[chainletId] = true;
    }

    // Set trusted remote address
    function setTrustedRemote(uint16 _remoteChainId, bytes calldata _path)
external onlyAdmin {
        trustedRemoteLookup[_remoteChainId] = _path;
    }

    // Map collection between chainlets
    function mapCollection(address localCollection, uint16 remoteChainId,
address remoteCollection) external onlyAdmin {
        remoteMappings[localCollection][remoteChainId] = remoteCollection;
    }

    // Bridge NFT to another chainlet
    function bridgeNFT(
        address collection,
        uint256 tokenId,
        uint16 dstChainId
    ) external payable {
        require(supportedChainlets[dstChainId], "Destination not
supported");
        require(remoteMappings[collection][dstChainId] != address(0),
"Collection not mapped");

        // Transfer NFT to bridge
        IERC721(collection).transferFrom(msg.sender, address(this),
tokenId);

        // Create payload for cross-chain message
        bytes memory payload = abi.encode(
            collection,
            tokenId,
            msg.sender,
            remoteMappings[collection][dstChainId]
        );
```

```
        // Estimate fee for LayerZero message
        (uint256 fee, ) = endpoint.estimateFees(
            dstChainId,
            address(this),
            payload,
            false,
            bytes("")
        );

        require(msg.value >= fee, "Insufficient fee");

        // Send cross-chain message
        endpoint.send{value: fee}(
            dstChainId,                         // destination chain ID
            trustedRemoteLookup[dstChainId],// destination address (remote
bridge)
            payload,                            // payload
            payable(msg.sender),                // refund address
            address(0),                         // future parameter, unused
now
            bytes("")                           // adapter parameters, empty
for now
        );

        // Store pending NFT
        bytes32 bridgeId = keccak256(abi.encodePacked(collection, tokenId,
dstChainId));
        pendingNFTs[bridgeId] = PendingNFT({
            collection: collection,
            tokenId: tokenId,
            owner: msg.sender,
            isBridged: true
        });

        emit NFTBridgeInitiated(collection, tokenId, msg.sender,
dstChainId);
    }

    // Receive bridged NFT from remote chainlet
    function lzReceive(
        uint16 _srcChainId,
        bytes memory _srcAddress,
        uint64 _nonce,
        bytes memory _payload
    ) external {
        // Verify sender is trusted remote
        require(msg.sender == address(endpoint), "Invalid endpoint");
```

```
        require(_srcAddress.length ==
trustedRemoteLookup[_srcChainId].length, "Invalid source");

        // Decode payload
        (
            address originalCollection,
            uint256 tokenId,
            address owner,
            address remoteCollection
        ) = abi.decode(_payload, (address, uint256, address, address));

        // Find local collection mapping
        address localCollection = findLocalCollection(remoteCollection,
_srcChainId);
        require(localCollection != address(0), "Collection not mapped");

        // Mint or transfer NFT to owner
        if (IERC721(localCollection).exists(tokenId)) {
            // If already exists, transfer from bridge to owner
            IERC721(localCollection).transferFrom(address(this), owner,
tokenId);
        } else {
            // If doesn't exist, mint to owner
            // This requires the bridge to have minter role on the local
collection
            INFTMinter(localCollection).mint(owner, tokenId);
        }

        // Record completion
        bytes32 bridgeId = keccak256(abi.encodePacked(remoteCollection,
tokenId, _srcChainId));
        pendingNFTs[bridgeId] = PendingNFT({
            collection: localCollection,
            tokenId: tokenId,
            owner: owner,
            isBridged: false
        });

        emit NFTBridgeCompleted(localCollection, tokenId, owner,
_srcChainId);
    }

    // Helper to find local collection from remote mapping
    function findLocalCollection(address remoteCollection, uint16
remoteChainId) internal view returns (address) {
        // Inefficient but functional for MVP - production would use a
reverse lookup
```

```
        for (uint256 i = 0; i < registeredCollections.length; i++) {
            address localCollection = registeredCollections[i];
            if (remoteMappings[localCollection][remoteChainId] ==
remoteCollection) {
                return localCollection;
            }
        }
        return address(0);
    }

    // Admin recovery function for stuck NFTs
    function adminRecoverNFT(
        address collection,
        uint256 tokenId,
        address recipient
    ) external onlyAdmin {
        IERC721(collection).transferFrom(address(this), recipient,
tokenId);
    }
}
```

2.2 AI Price Prediction System

```python
# AI Price Prediction System for GameFi NFTs
import tensorflow as tf
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import LSTM, Dense, Input, Dropout,
Concatenate, MultiHeadAttention
from tensorflow.keras.optimizers import Adam
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from typing import Dict, List, Tuple, Optional, Union
import json
import requests
import time
from web3 import Web3

class NFTDataCollector:
    """Collects data for training and prediction"""

    def __init__(self, saga_api_key: str, chainlet_endpoints: Dict[str,
str], web3_provider: str):
        self.saga_api_key = saga_api_key
        self.chainlet_endpoints = chainlet_endpoints  # Map of chainlet_id
-> endpoint
        self.headers = {
```

```
            "Authorization": f"Bearer {saga_api_key}",
            "Content-Type": "application/json"
        }
        self.web3 = Web3(Web3.HTTPProvider(web3_provider))

    def get_collection_history(self, chainlet_id: str, collection_address:
str, days: int = 90) -> pd.DataFrame:
        """Get historical price and volume data for NFT collection"""
        endpoint = self.chainlet_endpoints[chainlet_id]
        url = f"{endpoint}/collections/{collection_address}/stats"
        params = {"days": days}

        response = requests.get(url, headers=self.headers, params=params)
        if response.status_code != 200:
            raise Exception(f"Failed to fetch collection history:
{response.text}")

        data = response.json()

        # Convert to dataframe
        df = pd.DataFrame(data["dailyStats"])

        # Add chainlet-specific columns
        df["chainlet_id"] = chainlet_id

        # Add game activity metrics if available
        try:
            activity_data = self.get_game_activity_metrics(chainlet_id,
collection_address, days)
            df = pd.merge(df, activity_data, on="date", how="left")
        except Exception as e:
            print(f"Warning: Failed to get game activity data - {str(e)}")

        return df

    def get_game_activity_metrics(self, chainlet_id: str,
collection_address: str, days: int = 90) -> pd.DataFrame:
        """Get game-specific activity metrics"""
        endpoint = self.chainlet_endpoints[chainlet_id]
        url = f"{endpoint}/games/activity/{collection_address}"
        params = {"days": days}

        response = requests.get(url, headers=self.headers, params=params)
        if response.status_code != 200:
            raise Exception(f"Failed to fetch game activity:
{response.text}")
```

```
        data = response.json()

        # Convert to dataframe
        df = pd.DataFrame(data["activity"])

        # Ensure these columns exist
        required_cols = ["date", "dau", "avg_playtime_minutes",
"revenue_usd", "retention_30d"]
        for col in required_cols:
            if col not in df.columns:
                df[col] = None

        return df

    def get_nft_metadata(self, chainlet_id: str, collection_address: str,
token_id: int) -> dict:
        """Get metadata for a specific NFT"""
        endpoint = self.chainlet_endpoints[chainlet_id]
        url = f"{endpoint}/tokens/{collection_address}/{token_id}"

        response = requests.get(url, headers=self.headers)
        if response.status_code != 200:
            raise Exception(f"Failed to fetch NFT metadata:
{response.text}")

        return response.json()

    def get_nft_transaction_history(self, chainlet_id: str,
collection_address: str, token_id: int) -> pd.DataFrame:
        """Get transaction history for a specific NFT"""
        endpoint = self.chainlet_endpoints[chainlet_id]
        url = f"{endpoint}/tokens/{collection_address}/{token_id}/history"

        response = requests.get(url, headers=self.headers)
        if response.status_code != 200:
            raise Exception(f"Failed to fetch NFT history:
{response.text}")

        data = response.json()

        # Convert to dataframe
        df = pd.DataFrame(data["transactions"])

        return df

    def get_nft_game_stats(self, chainlet_id: str, collection_address:
str, token_id: int) -> dict:
```

```
        """Get game-specific stats for an NFT (level, power, etc.)"""
        endpoint = self.chainlet_endpoints[chainlet_id]
        url =
f"{endpoint}/games/nft-stats/{collection_address}/{token_id}"

        response = requests.get(url, headers=self.headers)
        if response.status_code != 200:
            raise Exception(f"Failed to fetch NFT game stats:
{response.text}")

        return response.json()

    def fetch_training_data(self, chainlet_id: str, collection_address:
str, days: int = 180) -> Tuple[pd.DataFrame, pd.DataFrame]:
        """Fetch comprehensive data for model training"""
        # Get collection price history
        collection_df = self.get_collection_history(chainlet_id,
collection_address, days)

        # Get sample of NFTs from collection
        nfts_sample = self.get_collection_nfts_sample(chainlet_id,
collection_address, 50)

        # Build NFT-specific dataset
        nft_data = []
        for nft in nfts_sample:
            token_id = nft["token_id"]

            # Get NFT transaction history
            try:
                tx_history = self.get_nft_transaction_history(chainlet_id,
collection_address, token_id)

                # Get NFT game stats
                game_stats = self.get_nft_game_stats(chainlet_id,
collection_address, token_id)

                # Get NFT metadata
                metadata = self.get_nft_metadata(chainlet_id,
collection_address, token_id)

                # Combine data
                nft_info = {
                    "token_id": token_id,
                    "tx_history": tx_history.to_dict("records") if not
tx_history.empty else [],
                    "game_stats": game_stats,
```

```
                    "metadata": metadata,
                    "last_price": nft.get("last_price", 0),
                    "rarity_score": nft.get("rarity_score", 0)
                }

                nft_data.append(nft_info)
            except Exception as e:
                print(f"Warning: Failed to process NFT {token_id} -
{str(e)}")
                continue

        # Convert to dataframe
        nft_df = pd.DataFrame(nft_data)

        return collection_df, nft_df

    def get_collection_nfts_sample(self, chainlet_id: str,
collection_address: str, sample_size: int = 50) -> List[dict]:
        """Get a sample of NFTs from a collection"""
        endpoint = self.chainlet_endpoints[chainlet_id]
        url = f"{endpoint}/collections/{collection_address}/tokens"
        params = {"limit": sample_size, "sample": "true"}

        response = requests.get(url, headers=self.headers, params=params)
        if response.status_code != 200:
            raise Exception(f"Failed to fetch collection NFTs:
{response.text}")

        data = response.json()
        return data["tokens"]


class NFTFeatureEngineer:
    """Processes raw data into features for the ML models"""

    def __init__(self, sequence_length: int = 30):
        self.sequence_length = sequence_length
        self.price_scaler = MinMaxScaler(feature_range=(0, 1))
        self.volume_scaler = MinMaxScaler(feature_range=(0, 1))
        self.game_metrics_scaler = MinMaxScaler(feature_range=(0, 1))

    def prepare_collection_features(self, collection_df: pd.DataFrame) ->
Dict[str, np.ndarray]:
        """Prepare collection-level time series features"""
        # Sort by date
        collection_df = collection_df.sort_values("date")
```

```
        # Fill missing values
        collection_df =
collection_df.fillna(method="ffill").fillna(method="bfill").fillna(0)

        # Extract price features
        price_features = collection_df[["floor_price", "avg_price",
"max_price"]].values
        scaled_price = self.price_scaler.fit_transform(price_features)

        # Extract volume features
        volume_features = collection_df[["volume", "sales_count"]].values
        scaled_volume = self.volume_scaler.fit_transform(volume_features)

        # Extract game activity features if available
        if "dau" in collection_df.columns:
            game_features = collection_df[["dau", "avg_playtime_minutes",
"revenue_usd", "retention_30d"]].values
            scaled_game =
self.game_metrics_scaler.fit_transform(game_features)
        else:
            scaled_game = np.zeros((len(collection_df), 4))

        # Combine features
        combined_features = np.hstack([scaled_price, scaled_volume,
scaled_game])

        # Create sequences
        X = []
        y = []

        for i in range(len(combined_features) - self.sequence_length):
            X.append(combined_features[i:i + self.sequence_length])
            y.append(price_features[i + self.sequence_length, 0])  # floor
price as target

        X = np.array(X)
        y = np.array(y)

        return {
            "X": X,
            "y": y,
            "dates": collection_df["date"].values[self.sequence_length:],
            "price_scaler": self.price_scaler,
            "volume_scaler": self.volume_scaler,
            "game_metrics_scaler": self.game_metrics_scaler
        }
```

```
    def prepare_nft_features(self, nft_df: pd.DataFrame) -> Dict[str,
np.ndarray]:
        """Prepare NFT-specific features for rarity and trait-based
models"""
        # Extract game stats features
        game_stats_features = []

        for _, row in nft_df.iterrows():
            stats = row["game_stats"]

            # Extract common game stats
            features = {
                "level": stats.get("level", 0),
                "power": stats.get("power", 0),
                "rarity_tier":
self.encode_rarity_tier(stats.get("rarity_tier", "common")),
                "utility_score": stats.get("utility_score", 100),
            }

            # Add game-specific stats if available
            if "attributes" in stats:
                for attr in stats["attributes"]:
                    if "trait_type" in attr and "value" in attr:
                        # Convert trait values to numeric if possible
                        try:
                            features[f"trait_{attr['trait_type']}"] =
float(attr["value"])
                        except (ValueError, TypeError):
                            # For non-numeric traits, use one-hot encoding
or embedding lookup
                            features[f"trait_{attr['trait_type']}"] =
self.encode_categorical_trait(
                                attr["trait_type"], attr["value"]
                            )

            game_stats_features.append(features)

        # Convert to dataframe
        game_stats_df = pd.DataFrame(game_stats_features)

        # Fill missing values
        game_stats_df = game_stats_df.fillna(0)

        # Extract transaction history features
        tx_features = []

        for _, row in nft_df.iterrows():
```

```
            tx_history = row["tx_history"]

            if not tx_history:
                # No transaction history
                tx_features.append({
                    "tx_count": 0,
                    "avg_price": 0,
                    "price_volatility": 0,
                    "days_since_last_tx": 999,
                    "price_trend": 0
                })
                continue

            # Convert to dataframe if not already
            if isinstance(tx_history, list):
                tx_df = pd.DataFrame(tx_history)
            else:
                tx_df = tx_history

            # Calculate features
            prices = tx_df.get("price", [0])
            if isinstance(prices, pd.Series):
                prices = prices.values

            features = {
                "tx_count": len(tx_df),
                "avg_price": np.mean(prices) if len(prices) > 0 else 0,
                "price_volatility": np.std(prices) if len(prices) > 1 else
0,
                "days_since_last_tx":
self.calculate_days_since_last_tx(tx_df),
                "price_trend": self.calculate_price_trend(tx_df)
            }

            tx_features.append(features)

        # Convert to dataframe
        tx_features_df = pd.DataFrame(tx_features)

        # Combine all features
        X = pd.concat([game_stats_df, tx_features_df], axis=1)
        y = nft_df["last_price"].values

        return {
            "X": X.values,
            "y": y,
            "feature_names": X.columns.tolist(),
```

```
            "token_ids": nft_df["token_id"].values
        }

    def encode_rarity_tier(self, tier: str) -> int:
        """Convert rarity tier to numeric value"""
        tier_map = {
            "common": 1,
            "uncommon": 2,
            "rare": 3,
            "epic": 4,
            "legendary": 5,
            "mythic": 6
        }
        return tier_map.get(tier.lower() if isinstance(tier, str) else
"common", 1)

    def encode_categorical_trait(self, trait_type: str, trait_value: str)
-> int:
        """Encode categorical traits using simple mapping"""
        # In production, would use more sophisticated encoding or
embedding lookup
        # For now, use hash value modulo 100 as a simple encoding
        hash_val = hash(f"{trait_type}:{trait_value}")
        return abs(hash_val) % 100

    def calculate_days_since_last_tx(self, tx_df: pd.DataFrame) -> int:
        """Calculate days since the last transaction"""
        if "timestamp" not in tx_df.columns or tx_df.empty:
            return 999

        latest_ts = pd.to_datetime(tx_df["timestamp"].max())
        now = pd.Timestamp.now()
        days_diff = (now - latest_ts).days

        return max(0, days_diff)

    def calculate_price_trend(self, tx_df: pd.DataFrame) -> float:
        """Calculate price trend direction and magnitude"""
        if "price" not in tx_df.columns or len(tx_df) < 2:
            return 0

        # Sort by timestamp
        if "timestamp" in tx_df.columns:
            tx_df = tx_df.sort_values("timestamp")

        prices = tx_df["price"].values
```

```
        if len(prices) < 2:
            return 0

        # Calculate slope of linear regression
        x = np.arange(len(prices))
        slope, _ = np.polyfit(x, prices, 1)

        # Normalize by average price
        avg_price = np.mean(prices)
        if avg_price > 0:
            normalized_slope = slope / avg_price
        else:
            normalized_slope = 0

        return normalized_slope


class NFTPricePredictionModel:
    """Combined model for NFT price prediction"""

    def __init__(self, checkpoint_path: Optional[str] = None):
        self.lstm_model = None
        self.transformer_model = None
        self.nft_model = None
        self.feature_engineer = NFTFeatureEngineer()
        self.checkpoint_path = checkpoint_path

    def build_lstm_model(self, input_shape: Tuple[int, int]) -> Model:
        """Build LSTM model for time series data"""
        inputs = Input(shape=input_shape)
        x = LSTM(128, return_sequences=True)(inputs)
        x = Dropout(0.2)(x)
        x = LSTM(64)(x)
        x = Dropout(0.2)(x)
        x = Dense(32, activation="relu")(x)
        outputs = Dense(1)(x)
        model = Model(inputs=inputs, outputs=outputs)
        model.compile(optimizer=Adam(0.001), loss="mse")
        return model

    def build_transformer_model(self, input_shape: Tuple[int, int]) ->
Model:
        """Build Transformer model for time series data"""
        inputs = Input(shape=input_shape)
        x = MultiHeadAttention(
            num_heads=4, key_dim=64
        )(inputs, inputs)
```

```
        x = Dropout(0.1)(x)
        x = Dense(64, activation="relu")(x)
        x = Dropout(0.1)(x)
        x = Dense(32, activation="relu")(x)
        x = tf.keras.layers.GlobalAveragePooling1D()(x)
        outputs = Dense(1)(x)
        model = Model(inputs=inputs, outputs=outputs)
        model.compile(optimizer=Adam(0.001), loss="mse")
        return model

    def build_nft_model(self, input_dim: int) -> Model:
        """Build model for NFT-specific features"""
        inputs = Input(shape=(input_dim,))
        x = Dense(64, activation="relu")(inputs)
        x = Dropout(0.2)(x)
        x = Dense(32, activation="relu")(x)
        x = Dropout(0.1)(x)
        outputs = Dense(1)(x)
        model = Model(inputs=inputs, outputs=outputs)
        model.compile(optimizer=Adam(0.001), loss="mse")
        return model

    def train(self, collection_df: pd.DataFrame, nft_df: pd.DataFrame,
epochs: int = 50) -> Dict:
        """Train the ensemble of models"""
        # Prepare collection features
        collection_features =
self.feature_engineer.prepare_collection_features(collection_df)
        X_ts, y_ts = collection_features["X"], collection_features["y"]

        # Prepare NFT features
        nft_features = self.feature_engineer.prepare_nft_features(nft_df)
        X_nft, y_nft = nft_features["X"], nft_features["y"]

        # Build models if not already built
        if self.lstm_model is None:
            self.lstm_model = self.build_lstm_model(X_ts.shape[1:])

        if self.transformer_model is None:
            self.transformer_model =
self.build_transformer_model(X_ts.shape[1:])

        if self.nft_model is None:
            self.nft_model = self.build_nft_model(X_nft.shape[1])

        # Train time series models
```

```
        lstm_history = self.lstm_model.fit(X_ts, y_ts, epochs=epochs,
batch_size=32, validation_split=0.2, verbose=1)
        transformer_history = self.transformer_model.fit(X_ts, y_ts,
epochs=epochs, batch_size=32, validation_split=0.2, verbose=1)

        # Train NFT model
        nft_history = self.nft_model.fit(X_nft, y_nft, epochs=epochs,
batch_size=32, validation_split=0.2, verbose=1)

        # Save checkpoints if path provided
        if self.checkpoint_path:
            self.save_models()

        return {
            "lstm_history": lstm_history.history,
            "transformer_history": transformer_history.history,
            "nft_history": nft_history.history
        }

    def predict_nft_price(
        self,
        collection_df: pd.DataFrame,
        nft_data: Dict,
        chainlet_id: str,
        collection_address: str,
        token_id: int
    ) -> Tuple[float, float]:
        """Predict price for a specific NFT"""
        # Prepare collection features
        collection_features =
self.feature_engineer.prepare_collection_features(collection_df)

        # Use last available sequence for prediction
        X_ts = collection_features["X"][-1:]

        # Create DataFrame with single NFT data
        nft_df = pd.DataFrame([nft_data])
        nft_features = self.feature_engineer.prepare_nft_features(nft_df)
        X_nft = nft_features["X"]

        # Get predictions from different models
        lstm_pred = self.lstm_model.predict(X_ts, verbose=0)[0][0]
        transformer_pred = self.transformer_model.predict(X_ts,
verbose=0)[0][0]
        nft_pred = self.nft_model.predict(X_nft, verbose=0)[0][0]

        # Inverse transform predictions to original scale
```

```
        price_scaler = collection_features["price_scaler"]
        lstm_price = self._inverse_transform_price(lstm_pred,
price_scaler)
        transformer_price =
self._inverse_transform_price(transformer_pred, price_scaler)

        # Ensemble prediction (with weightings)
        # Time series models: 60%, NFT model: 40%
        # Within time series: LSTM 50%, Transformer 50%
        ts_weight = 0.6
        nft_weight = 0.4
        lstm_weight = 0.5
        transformer_weight = 0.5

        ensemble_ts_pred = (lstm_price * lstm_weight) + (transformer_price
* transformer_weight)
        final_pred = (ensemble_ts_pred * ts_weight) + (nft_pred *
nft_weight)

        # Calculate confidence score (0-100)
        confidence = self._calculate_prediction_confidence(
            lstm_price, transformer_price, nft_pred, collection_df,
nft_data
        )

        # Return prediction and confidence
        return final_pred, confidence

    def _inverse_transform_price(self, pred: float, scaler) -> float:
        """Convert scaled prediction back to original price scale"""
        # Create dummy array with zeros for other features
        dummy = np.zeros((1, scaler.scale_.shape[0]))
        dummy[0, 0] = pred  # Set first feature (floor_price) to
prediction

        # Inverse transform
        unscaled = scaler.inverse_transform(dummy)
        return unscaled[0, 0]

    def _calculate_prediction_confidence(
        self,
        lstm_price: float,
        transformer_price: float,
        nft_price: float,
        collection_df: pd.DataFrame,
        nft_data: Dict
    ) -> int:
```

```
        """Calculate confidence score (0-100) for the prediction"""
        # Starting confidence score
        confidence = 70

        # 1. Model agreement factor (up to +/- 15 points)
        # Calculate relative differences between models
        if max(lstm_price, transformer_price, nft_price) > 0:
            max_diff_pct = (max(lstm_price, transformer_price, nft_price)
-
                            min(lstm_price, transformer_price, nft_price))
/ max(lstm_price, transformer_price, nft_price)

            # Strong agreement increases confidence
            if max_diff_pct < 0.05:  # Less than 5% difference
                confidence += 15
            elif max_diff_pct < 0.10:  # Less than 10% difference
                confidence += 10
            elif max_diff_pct < 0.20:  # Less than 20% difference
                confidence += 5
            elif max_diff_pct > 0.50:  # More than 50% difference
                confidence -= 15
            elif max_diff_pct > 0.35:  # More than 35% difference
                confidence -= 10
            elif max_diff_pct > 0.20:  # More than 20% difference
                confidence -= 5

        # 2. Data quality factor (up to +/- 10 points)
        # Check collection data recency and completeness
        if collection_df.empty or len(collection_df) < 30:
            confidence -= 10  # Insufficient historical data

        if "tx_history" in nft_data and nft_data["tx_history"]:
            tx_history = nft_data["tx_history"]
            # Convert to dataframe if necessary
            if isinstance(tx_history, list):
                tx_df = pd.DataFrame(tx_history)
            else:
                tx_df = tx_history

            # Check for recent transactions
            if "timestamp" in tx_df.columns and not tx_df.empty:
                latest_ts = pd.to_datetime(tx_df["timestamp"].max())
                days_diff = (pd.Timestamp.now() - latest_ts).days

                if days_diff < 7:  # Less than a week old
                    confidence += 10
                elif days_diff < 30:  # Less than a month old
```

```
                    confidence += 5
                elif days_diff > 180:  # More than 6 months old
                    confidence -= 10
                elif days_diff > 90:  # More than 3 months old
                    confidence -= 5
            else:
                confidence -= 5  # No transaction history

            # 3. Market volatility factor (up to +/- 10 points)
            # Higher volatility = lower confidence
            if len(collection_df) >= 7:  # Need at least a week of data
                try:
                    recent_prices = collection_df.sort_values("date",
ascending=False).head(7)["floor_price"].values
                    volatility = np.std(recent_prices) /
np.mean(recent_prices) if np.mean(recent_prices) > 0 else 0

                    if volatility < 0.05:  # Very stable
                        confidence += 10
                    elif volatility < 0.10:  # Moderately stable
                        confidence += 5
                    elif volatility > 0.30:  # Highly volatile
                        confidence -= 10
                    elif volatility > 0.20:  # Moderately volatile
                        confidence -= 5
                except Exception:
                    pass  # Skip volatility adjustment if calculation fails

            # 4. GameFi-specific factor (up to +/- 10 points)
            if "game_stats" in nft_data:
                game_stats = nft_data["game_stats"]

                # Check for utility score
                if "utility_score" in game_stats:
                    utility_score = game_stats["utility_score"]

                    # Higher utility generally means more predictable value
                    if utility_score > 180:  # Extremely high utility
                        confidence += 10
                    elif utility_score > 150:  # Very high utility
                        confidence += 5
                    elif utility_score < 70:  # Low utility
                        confidence -= 5

                # Check for game metrics
                if "dau" in collection_df.columns:
```

```
                recent_dau = collection_df.sort_values("date",
ascending=False).head(7)["dau"].mean()

                # Higher user activity generally means more predictable
market
                if recent_dau > 10000:  # Very active game
                    confidence += 5
                elif recent_dau < 1000:  # Low activity game
                    confidence -= 5

        # Cap confidence between 0-100
        confidence = max(0, min(100, confidence))

        return int(confidence)

    def save_models(self):
        """Save model checkpoints"""
        if not self.checkpoint_path:
            return

        try:
            # Create directory if it doesn't exist
            import os
            os.makedirs(self.checkpoint_path, exist_ok=True)

            # Save models
            self.lstm_model.save(f"{self.checkpoint_path}/lstm_model")

self.transformer_model.save(f"{self.checkpoint_path}/transformer_model")
            self.nft_model.save(f"{self.checkpoint_path}/nft_model")

            # Save scalers
            import pickle
            with open(f"{self.checkpoint_path}/scalers.pkl", "wb") as f:
                pickle.dump({
                    "price_scaler": self.feature_engineer.price_scaler,
                    "volume_scaler": self.feature_engineer.volume_scaler,
                    "game_metrics_scaler":
self.feature_engineer.game_metrics_scaler
                }, f)

            print(f"Models saved to {self.checkpoint_path}")
        except Exception as e:
            print(f"Error saving models: {str(e)}")

    def load_models(self):
        """Load model checkpoints"""
```

```
        if not self.checkpoint_path:
            return False

        try:
            # Load models
            self.lstm_model =
tf.keras.models.load_model(f"{self.checkpoint_path}/lstm_model")
            self.transformer_model =
tf.keras.models.load_model(f"{self.checkpoint_path}/transformer_model")
            self.nft_model =
tf.keras.models.load_model(f"{self.checkpoint_path}/nft_model")

            # Load scalers
            import pickle
            with open(f"{self.checkpoint_path}/scalers.pkl", "rb") as f:
                scalers = pickle.load(f)
                self.feature_engineer.price_scaler =
scalers["price_scaler"]
                self.feature_engineer.volume_scaler =
scalers["volume_scaler"]
                self.feature_engineer.game_metrics_scaler =
scalers["game_metrics_scaler"]

            print(f"Models loaded from {self.checkpoint_path}")
            return True
        except Exception as e:
            print(f"Error loading models: {str(e)}")
            return False


class MosaicalPriceOracle:
    """On-chain oracle service for Mosaical platform"""

    def __init__(
        self,
        model: NFTPricePredictionModel,
        data_collector: NFTDataCollector,
        web3_provider: str,
        oracle_contract_address: str,
        private_key: str
    ):
        self.model = model
        self.data_collector = data_collector
        self.web3 = Web3(Web3.HTTPProvider(web3_provider))
        self.oracle_contract_address = oracle_contract_address
        self.private_key = private_key
```

```
        # Load oracle ABI
        self.oracle_abi = self._load_oracle_abi()
        self.oracle_contract = self.web3.eth.contract(
            address=self.web3.toChecksumAddress(oracle_contract_address),
            abi=self.oracle_abi
        )

        # Set up account from private key
        self.account =
self.web3.eth.account.privateKeyToAccount(private_key)

    def _load_oracle_abi(self) -> list:
        """Load ABI for oracle contract"""
        # In production, would load from file or config
        return [
            {
                "inputs": [
                    {"internalType": "address", "name": "collection",
"type": "address"},
                    {"internalType": "uint256", "name": "tokenId", "type":
"uint256"}
                ],
                "name": "getNFTPrice",
                "outputs": [{"internalType": "uint256", "name": "",
"type": "uint256"}],
                "stateMutability": "view",
                "type": "function"
            },
            {
                "inputs": [
                    {"internalType": "address", "name": "collection",
"type": "address"},
                    {"internalType": "uint256", "name": "tokenId", "type":
"uint256"},
                    {"internalType": "uint256", "name": "price", "type":
"uint256"},
                    {"internalType": "uint256", "name": "confidence",
"type": "uint256"}
                ],
                "name": "updateNFTPrice",
                "outputs": [],
                "stateMutability": "nonpayable",
                "type": "function"
            },
            {
                "inputs": [
```

```
                    {"internalType": "address", "name": "collection",
"type": "address"},
                    {"internalType": "uint256", "name": "confidence",
"type": "uint256"}
                ],
                "name": "updatePredictionConfidence",
                "outputs": [],
                "stateMutability": "nonpayable",
                "type": "function"
            }
        ]

    def update_nft_price(
        self,
        chainlet_id: str,
        collection_address: str,
        token_id: int
    ) -> dict:
        """Predict NFT price and update on-chain oracle"""
        try:
            # Fetch data for prediction
            collection_df =
self.data_collector.get_collection_history(chainlet_id,
collection_address)
            nft_data = {
                "token_id": token_id,
                "tx_history":
self.data_collector.get_nft_transaction_history(
                    chainlet_id, collection_address, token_id
                ).to_dict("records"),
                "game_stats": self.data_collector.get_nft_game_stats(
                    chainlet_id, collection_address, token_id
                ),
                "metadata": self.data_collector.get_nft_metadata(
                    chainlet_id, collection_address, token_id
                )
            }

            # Get prediction
            price_prediction, confidence = self.model.predict_nft_price(
                collection_df,
                nft_data,
                chainlet_id,
                collection_address,
                token_id
            )
```

```
            # Convert prediction to wei (assuming price_prediction is in
ETH)
            price_wei = self.web3.toWei(price_prediction, "ether")

            # Update oracle
            nonce =
self.web3.eth.getTransactionCount(self.account.address)
            tx = self.oracle_contract.functions.updateNFTPrice(
                self.web3.toChecksumAddress(collection_address),
                token_id,
                price_wei,
                confidence
            ).buildTransaction({
                "chainId": self.web3.eth.chainId,
                "gas": 2000000,
                "gasPrice": self.web3.eth.gasPrice,
                "nonce": nonce
            })

            # Sign and send transaction
            signed_tx = self.web3.eth.account.signTransaction(tx,
self.private_key)
            tx_hash =
self.web3.eth.sendRawTransaction(signed_tx.rawTransaction)

            # Wait for transaction receipt
            receipt = self.web3.eth.waitForTransactionReceipt(tx_hash)

            return {
                "success": True,
                "price": price_prediction,
                "confidence": confidence,
                "tx_hash": tx_hash.hex(),
                "block_number": receipt.blockNumber
            }
        except Exception as e:
            print(f"Error updating NFT price: {str(e)}")
            return {
                "success": False,
                "error": str(e)
            }

    def update_collection_confidence(
        self,
        chainlet_id: str,
        collection_address: str
    ) -> dict:
```

```
        """Update overall prediction confidence for a collection"""
        try:
            # Get sample of NFTs
            nfts_sample = self.data_collector.get_collection_nfts_sample(
                chainlet_id, collection_address, 10
            )

            # Get collection history
            collection_df = self.data_collector.get_collection_history(
                chainlet_id, collection_address
            )

            # Calculate average confidence across sample
            confidences = []
            for nft in nfts_sample:
                token_id = nft["token_id"]

                try:
                    # Get NFT data
                    nft_data = {
                        "token_id": token_id,
                        "tx_history":
self.data_collector.get_nft_transaction_history(
                            chainlet_id, collection_address, token_id
                        ).to_dict("records"),
                        "game_stats":
self.data_collector.get_nft_game_stats(
                            chainlet_id, collection_address, token_id
                        ),
                        "metadata": self.data_collector.get_nft_metadata(
                            chainlet_id, collection_address, token_id
                        )
                    }

                    # Get prediction confidence
                    _, confidence = self.model.predict_nft_price(
                        collection_df,
                        nft_data,
                        chainlet_id,
                        collection_address,
                        token_id
                    )

                    confidences.append(confidence)
                except Exception as e:
                    print(f"Error predicting for token {token_id}:
{str(e)}")
```

```
                    continue

            if not confidences:
                return {
                    "success": False,
                    "error": "Failed to calculate confidence for any NFTs
in sample"
                }

            # Calculate average confidence
            avg_confidence = int(np.mean(confidences))

            # Update oracle
            nonce =
self.web3.eth.getTransactionCount(self.account.address)
            tx =
self.oracle_contract.functions.updatePredictionConfidence(
                self.web3.toChecksumAddress(collection_address),
                avg_confidence
            ).buildTransaction({
                "chainId": self.web3.eth.chainId,
                "gas": 1000000,
                "gasPrice": self.web3.eth.gasPrice,
                "nonce": nonce
            })

            # Sign and send transaction
            signed_tx = self.web3.eth.account.signTransaction(tx,
self.private_key)
            tx_hash =
self.web3.eth.sendRawTransaction(signed_tx.rawTransaction)

            # Wait for transaction receipt
            receipt = self.web3.eth.waitForTransactionReceipt(tx_hash)

            return {
                "success": True,
                "confidence": avg_confidence,
                "tx_hash": tx_hash.hex(),
                "block_number": receipt.blockNumber
            }
        except Exception as e:
            print(f"Error updating collection confidence: {str(e)}")
            return {
                "success": False,
                "error": str(e)
            }
```

```python
# Example usage of the system
def main():
    # Configuration variables
    saga_api_key = "SAGA_API_KEY"
    web3_provider = "https://rpc.saga.xyz/mainnet"
    oracle_contract_address = "0x1234567890123456789012345678901234567890"
    private_key = "0xPRIVATE_KEY"
    model_checkpoint_path = "models/nft_price_prediction"
    chainlet_endpoints = {
        "chainlet1": "https://api.chainlet1.saga.xyz",
        "chainlet2": "https://api.chainlet2.saga.xyz",
    }

    # Initialize components
    data_collector = NFTDataCollector(saga_api_key, chainlet_endpoints,
web3_provider)
    price_model =
NFTPricePredictionModel(checkpoint_path=model_checkpoint_path)

    # Try to load existing models, or train new ones if needed
    if not price_model.load_models():
        print("Training new models...")

        # Get training data for a collection
        collection_df, nft_df =
data_collector.fetch_training_data("chainlet1", "0xCOLLECTION_ADDRESS")

        # Train models
        price_model.train(collection_df, nft_df, epochs=50)

        # Save models
        price_model.save_models()

    # Initialize oracle service
    oracle_service = MosaicalPriceOracle(
        model=price_model,
        data_collector=data_collector,
        web3_provider=web3_provider,
        oracle_contract_address=oracle_contract_address,
        private_key=private_key
    )

    # Update NFT price example
    result = oracle_service.update_nft_price(
        "chainlet1",
```

```
        "0xCOLLECTION_ADDRESS",
        123  # token ID
    )
    print(f"Update NFT price result: {result}")

    # Update collection confidence example
    result = oracle_service.update_collection_confidence(
        "chainlet1",
        "0xCOLLECTION_ADDRESS"
    )
    print(f"Update collection confidence result: {result}")


if __name__ == "__main__":
    main()
```

2.3 Risk Management System

```
# Risk Management System for GameFi NFTs
import pandas as pd
import numpy as np
from typing import Dict, List, Tuple, Optional, Union
import json
import requests
import time
from web3 import Web3
import logging
import os
from datetime import datetime, timedelta


# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("risk_management.log"),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger("MosaicalRiskManager")


class GameFiRiskManager:
    """Risk management system for Mosaical platform"""

    def __init__(
```

```
        self,
        web3_provider: str,
        contracts: Dict[str, str],
        private_key: str,
        saga_api_key: str,
        chainlet_endpoints: Dict[str, str]
    ):
        self.web3 = Web3(Web3.HTTPProvider(web3_provider))
        self.contracts = {k: self.web3.eth.contract(
            address=self.web3.toChecksumAddress(v),
            abi=self._load_abi(k)
        ) for k, v in contracts.items()}

        self.private_key = private_key
        self.account =
self.web3.eth.account.privateKeyToAccount(private_key)
        self.saga_api_key = saga_api_key
        self.chainlet_endpoints = chainlet_endpoints
        self.headers = {
            "Authorization": f"Bearer {saga_api_key}",
            "Content-Type": "application/json"
        }

        # Risk parameters
        self.risk_parameters = self._load_risk_parameters()

    def _load_abi(self, contract_name: str) -> list:
        """Load ABI for contract"""
        # In production, would load from file or config
        abis = {
            "nftVault": [],  # NFTVaultV3 ABI
            "loanManager": [],  # LoanManagerV3 ABI
            "oracle": [],  # GameFiOracleV3 ABI
            "governance": []  # MosaicalGovernance ABI
        }

        return abis.get(contract_name, [])

    def _load_risk_parameters(self) -> dict:
        """Load risk parameters"""
        # In production, would load from DB or config
        return {
            "global": {
                "max_total_exposure": 1000,  # ETH
                "max_collection_exposure": 200,  # ETH
                "emergency_ltv_reduction": 10,  # %
                "min_confidence_score": 70
```

```
        },
        "tiers": {
            1: {  # Lowest risk
                "base_ltv": 70,
                "liquidation_threshold": 80,
                "max_utility_bonus": 20,
                "min_collateral_amount": 0.1,
                "price_feed_frequency": 12  # Hours
            },
            2: {
                "base_ltv": 65,
                "liquidation_threshold": 75,
                "max_utility_bonus": 15,
                "min_collateral_amount": 0.25,
                "price_feed_frequency": 8
            },
            3: {  # Medium risk
                "base_ltv": 60,
                "liquidation_threshold": 70,
                "max_utility_bonus": 10,
                "min_collateral_amount": 0.5,
                "price_feed_frequency": 6
            },
            4: {
                "base_ltv": 50,
                "liquidation_threshold": 65,
                "max_utility_bonus": 10,
                "min_collateral_amount": 1,
                "price_feed_frequency": 4
            },
            5: {  # Highest risk
                "base_ltv": 40,
                "liquidation_threshold": 55,
                "max_utility_bonus": 5,
                "min_collateral_amount": 2,
                "price_feed_frequency": 2
            }
        }
    }

def analyze_collection_risk(
    self,
    chainlet_id: str,
    collection_address: str,
    days_history: int = 30
) -> dict:
    """Analyze risk profile of a GameFi NFT collection"""
```

```
        try:
            # Get collection stats history
            collection_stats = self._get_collection_stats(chainlet_id,
collection_address, days_history)

            # Get game activity metrics
            game_metrics = self._get_game_metrics(chainlet_id,
collection_address, days_history)

            # Calculate risk metrics
            price_volatility =
self._calculate_price_volatility(collection_stats)
            trading_volume =
self._calculate_trading_volume(collection_stats)
            price_trend = self._calculate_price_trend(collection_stats)
            liquidity = self._calculate_liquidity(collection_stats)
            user_activity = self._calculate_user_activity(game_metrics)

            # Compute overall risk score (1-100, higher = riskier)
            risk_score = self._compute_risk_score(
                price_volatility,
                trading_volume,
                price_trend,
                liquidity,
                user_activity
            )

            # Determine risk tier (1-5)
            risk_tier = self._determine_risk_tier(risk_score)

            return {
                "collection_address": collection_address,
                "chainlet_id": chainlet_id,
                "risk_score": risk_score,
                "risk_tier": risk_tier,
                "metrics": {
                    "price_volatility": price_volatility,
                    "trading_volume": trading_volume,
                    "price_trend": price_trend,
                    "liquidity": liquidity,
                    "user_activity": user_activity
                },
                "recommended_params":
self.risk_parameters["tiers"][risk_tier],
                "timestamp": datetime.now().isoformat()
            }
```

```
        except Exception as e:
            logger.error(f"Error analyzing collection risk: {str(e)}")
            return {
                "success": False,
                "error": str(e)
            }

    def _get_collection_stats(self, chainlet_id: str, collection_address:
str, days: int) -> pd.DataFrame:
        """Get collection stats history"""
        endpoint = self.chainlet_endpoints[chainlet_id]
        url = f"{endpoint}/collections/{collection_address}/stats"
        params = {"days": days}

        response = requests.get(url, headers=self.headers, params=params)
        if response.status_code != 200:
            raise Exception(f"Failed to fetch collection stats:
{response.text}")

        data = response.json()
        df = pd.DataFrame(data["dailyStats"])

        return df

    def _get_game_metrics(self, chainlet_id: str, collection_address: str,
days: int) -> pd.DataFrame:
        """Get game activity metrics"""
        endpoint = self.chainlet_endpoints[chainlet_id]
        url = f"{endpoint}/games/activity/{collection_address}"
        params = {"days": days}

        response = requests.get(url, headers=self.headers, params=params)
        if response.status_code != 200:
            raise Exception(f"Failed to fetch game metrics:
{response.text}")

        data = response.json()
        df = pd.DataFrame(data["activity"])

        return df

    def _calculate_price_volatility(self, df: pd.DataFrame) -> float:
        """Calculate price volatility score (0-100)"""
        if df.empty or "floor_price" not in df.columns:
            return 50  # Default median risk if data unavailable

        # Calculate daily price changes
```

```
        df = df.sort_values("date")
        df["price_change_pct"] = df["floor_price"].pct_change() * 100

        # Get absolute daily change
        df["abs_change"] = df["price_change_pct"].abs()

        # Calculate average daily absolute change
        avg_daily_change = df["abs_change"].mean()

        # Convert to 0-100 score (higher = more volatile = riskier)
        # 0% change = 0 score, 20% daily change = 100 score
        volatility_score = min(100, avg_daily_change * 5)

        return volatility_score

    def _calculate_trading_volume(self, df: pd.DataFrame) -> float:
        """Calculate trading volume score (0-100)"""
        if df.empty or "volume" not in df.columns or "floor_price" not in
df.columns:
            return 50  # Default median risk

        # Calculate average daily volume relative to floor price
        df["volume_to_floor"] = df["volume"] / df["floor_price"]
        avg_volume_ratio = df["volume_to_floor"].mean()

        # Convert to 0-100 score (higher = more volume = less risky)
        # Invert for risk score (higher = riskier)
        volume_score_raw = min(100, avg_volume_ratio / 2)
        volume_score = 100 - volume_score_raw

        return volume_score

    def _calculate_price_trend(self, df: pd.DataFrame) -> float:
        """Calculate price trend score (0-100)"""
        if df.empty or "floor_price" not in df.columns or len(df) < 7:
            return 50  # Default median risk

        # Sort by date
        df = df.sort_values("date")

        # Calculate 7-day price change
        last_7d = df.tail(7)
        if len(last_7d) < 7:
            return 50

        start_price = last_7d.iloc[0]["floor_price"]
        end_price = last_7d.iloc[-1]["floor_price"]
```

```
        if start_price == 0:
            return 50

        price_change_pct = ((end_price - start_price) / start_price) * 100

        # Convert to 0-100 score (negative trend = higher risk)
        if price_change_pct >= 0:
            # Positive trend (less risky)
            # 0% change = 50, 20% gain or more = 0 (lowest risk)
            trend_score = max(0, 50 - price_change_pct * 2.5)
        else:
            # Negative trend (more risky)
            # 0% change = 50, 20% loss or more = 100 (highest risk)
            trend_score = min(100, 50 + abs(price_change_pct) * 2.5)

        return trend_score

    def _calculate_liquidity(self, df: pd.DataFrame) -> float:
        """Calculate liquidity score (0-100)"""
        if df.empty or "sales_count" not in df.columns:
            return 50  # Default median risk

        # Calculate average daily sales count
        avg_daily_sales = df["sales_count"].mean()

        # Convert to 0-100 score (higher sales = lower risk)
        # Invert for risk score (higher = riskier)
        liquidity_score_raw = min(100, avg_daily_sales / 2)
        liquidity_score = 100 - liquidity_score_raw

        return liquidity_score

    def _calculate_user_activity(self, df: pd.DataFrame) -> float:
        """Calculate user activity score (0-100)"""
        if df.empty or "dau" not in df.columns:
            return 50  # Default median risk

        # Calculate average DAU
        avg_dau = df["dau"].mean()

        # Check for trend (rising or falling)
        if len(df) >= 7:
            df = df.sort_values("date")
            last_7d = df.tail(7)
            start_dau = last_7d.iloc[0]["dau"]
            end_dau = last_7d.iloc[-1]["dau"]
```

```
            if start_dau > 0:
                dau_change_pct = ((end_dau - start_dau) / start_dau) * 100
            else:
                dau_change_pct = 0
        else:
            dau_change_pct = 0

        # Calculate activity score based on absolute DAU
        # Higher DAU = lower risk
        if avg_dau >= 10000:  # Very popular game
            activity_score_base = 10
        elif avg_dau >= 5000:
            activity_score_base = 25
        elif avg_dau >= 1000:
            activity_score_base = 40
        elif avg_dau >= 500:
            activity_score_base = 60
        elif avg_dau >= 100:
            activity_score_base = 75
        else:
            activity_score_base = 90  # Very small user base = high risk

        # Adjust for trend
        # Growing = less risky, shrinking = more risky
        if dau_change_pct >= 20:  # Rapid growth
            trend_adjustment = -20
        elif dau_change_pct >= 10:
            trend_adjustment = -10
        elif dau_change_pct <= -20:  # Rapid decline
            trend_adjustment = 20
        elif dau_change_pct <= -10:
            trend_adjustment = 10
        else:
            trend_adjustment = 0

        # Calculate final score (bounded 0-100)
        activity_score = min(100, max(0, activity_score_base +
trend_adjustment))

        return activity_score

    def _compute_risk_score(
        self,
        price_volatility: float,
        trading_volume: float,
        price_trend: float,
```

```
        liquidity: float,
        user_activity: float
    ) -> float:
        """Compute overall risk score (0-100)"""
        # Weighted average of risk factors
        weights = {
            "price_volatility": 0.3,
            "trading_volume": 0.15,
            "price_trend": 0.25,
            "liquidity": 0.15,
            "user_activity": 0.15
        }

        risk_score = (
            price_volatility * weights["price_volatility"] +
            trading_volume * weights["trading_volume"] +
            price_trend * weights["price_trend"] +
            liquidity * weights["liquidity"] +
            user_activity * weights["user_activity"]
        )

        return round(risk_score, 1)

    def _determine_risk_tier(self, risk_score: float) -> int:
        """Determine risk tier (1-5) from risk score (0-100)"""
        if risk_score < 20:
            return 1  # Lowest risk
        elif risk_score < 40:
            return 2
        elif risk_score < 60:
            return 3  # Medium risk
        elif risk_score < 80:
            return 4
        else:
            return 5  # Highest risk

    def update_collection_risk_tier(
        self,
        chainlet_id: str,
        collection_address: str
    ) -> dict:
        """Analyze and update risk tier for a collection on-chain"""
        try:
            # Analyze risk
            risk_analysis = self.analyze_collection_risk(chainlet_id,
collection_address)
```

```
            if "success" in risk_analysis and risk_analysis["success"] ==
False:
                return risk_analysis

            risk_tier = risk_analysis["risk_tier"]

            # Update on-chain risk tier
            nonce =
self.web3.eth.getTransactionCount(self.account.address)
            tx =
self.contracts["nftVault"].functions.setCollectionRiskTier(
                self.web3.toChecksumAddress(collection_address),
                risk_tier
            ).buildTransaction({
                "chainId": self.web3.eth.chainId,
                "gas": 2000000,
                "gasPrice": self.web3.eth.gasPrice,
                "nonce": nonce
            })

            # Sign and send transaction
            signed_tx = self.web3.eth.account.signTransaction(tx,
self.private_key)
            tx_hash =
self.web3.eth.sendRawTransaction(signed_tx.rawTransaction)

            # Wait for transaction receipt
            receipt = self.web3.eth.waitForTransactionReceipt(tx_hash)

            logger.info(f"Updated risk tier for {collection_address} to
{risk_tier}")

            return {
                "success": True,
                "collection_address": collection_address,
                "risk_tier": risk_tier,
                "risk_score": risk_analysis["risk_score"],
                "tx_hash": tx_hash.hex(),
                "block_number": receipt.blockNumber
            }

        except Exception as e:
            logger.error(f"Error updating collection risk tier: {str(e)}")
            return {
                "success": False,
                "error": str(e)
            }
```

```
    def scan_unhealthy_loans(self) -> List[dict]:
        """Scan for loans approaching liquidation thresholds"""
        try:
            # Get all active loans
            active_loans = self._get_all_active_loans()

            # Check health factor for each loan
            unhealthy_loans = []

            for loan in active_loans:
                try:
                    # Get current health factor
                    health_factor =
self.contracts["loanManager"].functions.calculateHealthFactor(
                        loan["borrower"],
                        loan["collection"],
                        loan["tokenId"]
                    ).call()

                    # Get liquidation threshold
                    risk_tier =
self.contracts["nftVault"].functions.collectionRiskTier(
                        loan["collection"]
                    ).call()

                    if risk_tier == 0:
                        risk_tier = 3  # Default to medium risk

                    liquidation_threshold =
self.risk_parameters["tiers"][risk_tier]["liquidation_threshold"] * 100  #
Scale to match contract

                    # Check if health is near threshold
                    if health_factor < (liquidation_threshold * 1.1):  #
Within 10% of liquidation
                        loan["health_factor"] = health_factor / 100  #
Scale down for readability
                        loan["liquidation_threshold"] =
liquidation_threshold / 100
                        loan["risk_tier"] = risk_tier

                        # Calculate distance to liquidation
                        loan["liquidation_distance"] = (health_factor -
liquidation_threshold) / 100

                        unhealthy_loans.append(loan)
```

```
                except Exception as e:
                    logger.error(f"Error checking loan health for
{loan['collection']}-{loan['tokenId']}: {str(e)}")
                    continue

            return unhealthy_loans

        except Exception as e:
            logger.error(f"Error scanning unhealthy loans: {str(e)}")
            return []

    def _get_all_active_loans(self) -> List[dict]:
        """Get all active loans (simplified for blueprint)"""
        # In production: would query events or indexed database
        # For blueprint: return mock data
        return [
            {
                "borrower": "0xBorrowerAddress1",
                "collection": "0xCollectionAddress1",
                "tokenId": 123,
                "amount": 5 * 10**18,  # 5 ETH
                "startTime": int(time.time()) - 86400  # 1 day ago
            },
            {
                "borrower": "0xBorrowerAddress2",
                "collection": "0xCollectionAddress2",
                "tokenId": 456,
                "amount": 10 * 10**18,  # 10 ETH
                "startTime": int(time.time()) - 604800  # 1 week ago
            }
        ]

    def assess_protocol_risk(self) -> dict:
        """Assess overall protocol risk metrics"""
        try:
            # Get global risk metrics
            total_loans = self._get_total_loans_value()
            collection_exposures = self._get_collection_exposures()

            # Get top exposed collections
            top_collections = sorted(
                collection_exposures.items(),
                key=lambda x: x[1],
                reverse=True
            )[:5]

            # Check if any exposure limits are exceeded
```

```
            global_limit =
self.risk_parameters["global"]["max_total_exposure"] * 10**18
            collection_limit =
self.risk_parameters["global"]["max_collection_exposure"] * 10**18

            global_limit_exceeded = total_loans > global_limit
            collection_limits_exceeded = [
                {"collection": coll, "exposure": exp}
                for coll, exp in collection_exposures.items()
                if exp > collection_limit
            ]

            # Calculate protocol health score (0-100, higher = healthier)
            protocol_health = self._calculate_protocol_health(
                total_loans,
                global_limit,
                collection_exposures,
                collection_limit
            )

            return {
                "timestamp": datetime.now().isoformat(),
                "total_loans_value": total_loans / 10**18,  # Convert to
ETH
                "protocol_health_score": protocol_health,
                "global_limit_exceeded": global_limit_exceeded,
                "collection_limits_exceeded": collection_limits_exceeded,
                "top_exposed_collections": [
                    {"collection": coll, "exposure": exp / 10**18}  #
Convert to ETH
                    for coll, exp in top_collections
                ],
                "recommended_actions": self._get_risk_recommendations(
                    protocol_health,
                    global_limit_exceeded,
                    collection_limits_exceeded
                )
            }

        except Exception as e:
            logger.error(f"Error assessing protocol risk: {str(e)}")
            return {
                "success": False,
                "error": str(e)
            }

    def _get_total_loans_value(self) -> int:
```

```
        """Get total value of all outstanding loans"""
        # In production: would query contract state or indexed database
        # For blueprint: return mock value
        return 800 * 10**18  # 800 ETH

    def _get_collection_exposures(self) -> Dict[str, int]:
        """Get exposure by collection"""
        # In production: would query contract state or indexed database
        # For blueprint: return mock values
        return {
            "0xCollection1": 150 * 10**18,  # 150 ETH
            "0xCollection2": 200 * 10**18,  # 200 ETH
            "0xCollection3": 120 * 10**18,  # 120 ETH
            "0xCollection4": 180 * 10**18,  # 180 ETH
            "0xCollection5": 90 * 10**18,   # 90 ETH
            "0xCollection6": 60 * 10**18    # 60 ETH
        }

    def _calculate_protocol_health(
        self,
        total_loans: int,
        global_limit: int,
        collection_exposures: Dict[str, int],
        collection_limit: int
    ) -> float:
        """Calculate protocol health score (0-100)"""
        # Global utilization score (lower utilization = higher health)
        global_utilization = total_loans / global_limit if global_limit >
0 else 1
        global_health = 100 * (1 - min(1, global_utilization))

        # Collection concentration score
        # Higher concentration in few collections = lower health
        if total_loans > 0:
            collection_concentrations = [
                min(1, exp / collection_limit) for exp in
collection_exposures.values()
            ]
            avg_concentration = sum(collection_concentrations) /
len(collection_concentrations) if collection_concentrations else 0
            concentration_health = 100 * (1 - avg_concentration)
        else:
            concentration_health = 100

        # Combined health score
        health_score = (global_health * 0.6) + (concentration_health *
0.4)
```

```
        return round(health_score, 1)

    def _get_risk_recommendations(
        self,
        health_score: float,
        global_limit_exceeded: bool,
        collection_limits_exceeded: List[dict]
    ) -> List[str]:
        """Get risk management recommendations"""
        recommendations = []

        if global_limit_exceeded:
            recommendations.append(
                "URGENT: Global exposure limit exceeded. Pause new loans
immediately."
            )

        if collection_limits_exceeded:
            for item in collection_limits_exceeded:
                recommendations.append(
                    f"URGENT: Collection {item['collection']} exposure
limit exceeded. "
                    f"Pause new loans for this collection."
                )

        if health_score < 30:
            recommendations.append(
                "Critical protocol health. Initiate emergency risk
reduction measures."
            )
        elif health_score < 50:
            recommendations.append(
                "Poor protocol health. Consider reducing LTV parameters
across all risk tiers."
            )
        elif health_score < 70:
            recommendations.append(
                "Moderate protocol health. Monitor closely for changes."
            )

        if not recommendations:
            recommendations.append("Protocol health is good. No immediate
actions needed.")

        return recommendations
```

```
    def emergency_risk_reduction(self) -> dict:
        """Execute emergency risk reduction measures"""
        try:
            # Get current protocol health
            protocol_risk = self.assess_protocol_risk()

            if protocol_risk.get("success") == False:
                return protocol_risk

            health_score = protocol_risk.get("protocol_health_score", 0)

            if health_score >= 70:
                return {
                    "success": True,
                    "message": "No emergency measures needed. Protocol
health is good.",
                    "actions_taken": []
                }

            # Define actions based on health score
            actions_taken = []

            # 1. Reduce LTV across all risk tiers
            ltv_reduction =
self.risk_parameters["global"]["emergency_ltv_reduction"]

            for tier in range(1, 6):
                try:
                    # Get current LTV for tier
                    current_ltv =
self.contracts["nftVault"].functions.riskModels(tier).call()[0]

                    # Calculate new LTV with reduction
                    new_ltv = max(20, current_ltv - ltv_reduction)  #
Don't go below 20% LTV

                    # Get other risk model parameters
                    risk_model =
self.contracts["nftVault"].functions.riskModels(tier).call()
                    liquidation_threshold = risk_model[1]
                    max_utility_bonus = risk_model[2]
                    min_collateral_amount = risk_model[3]

                    # Update risk model
                    nonce =
self.web3.eth.getTransactionCount(self.account.address)
```

```
                    tx =
self.contracts["nftVault"].functions.updateRiskModel(
                        tier,
                        new_ltv,
                        liquidation_threshold,
                        max_utility_bonus,
                        min_collateral_amount
                    ).buildTransaction({
                        "chainId": self.web3.eth.chainId,
                        "gas": 2000000,
                        "gasPrice": self.web3.eth.gasPrice,
                        "nonce": nonce
                    })

                    signed_tx = self.web3.eth.account.signTransaction(tx,
self.private_key)
                    tx_hash =
self.web3.eth.sendRawTransaction(signed_tx.rawTransaction)

                    receipt =
self.web3.eth.waitForTransactionReceipt(tx_hash)

                    actions_taken.append({
                        "action": f"Reduced LTV for tier {tier} from
{current_ltv} to {new_ltv}",
                        "tx_hash": tx_hash.hex()
                    })

                except Exception as e:
                    logger.error(f"Error updating risk model for tier
{tier}: {str(e)}")
                    continue

            # 2. If health is critical, disable borrowing temporarily
            if health_score < 30:
                try:
                    nonce =
self.web3.eth.getTransactionCount(self.account.address)
                    tx =
self.contracts["loanManager"].functions.setPaused(True).buildTransaction({
                        "chainId": self.web3.eth.chainId,
                        "gas": 1000000,
                        "gasPrice": self.web3.eth.gasPrice,
                        "nonce": nonce
                    })
```

```
                    signed_tx = self.web3.eth.account.signTransaction(tx,
self.private_key)
                    tx_hash =
self.web3.eth.sendRawTransaction(signed_tx.rawTransaction)

                    receipt =
self.web3.eth.waitForTransactionReceipt(tx_hash)

                    actions_taken.append({
                        "action": "Paused borrowing due to critical
protocol health",
                        "tx_hash": tx_hash.hex()
                    })

                except Exception as e:
                    logger.error(f"Error pausing borrowing: {str(e)}")

            logger.info(f"Emergency risk reduction completed. Actions
taken: {len(actions_taken)}")

            return {
                "success": True,
                "health_score_before": health_score,
                "actions_taken": actions_taken
            }

        except Exception as e:
            logger.error(f"Error executing emergency risk reduction:
{str(e)}")
            return {
                "success": False,
                "error": str(e)
            }

    def schedule_risk_assessment_tasks(self) -> None:
        """Schedule regular risk assessment tasks"""
        # In production, this would set up automated tasks
        # For blueprint, we'll just define them

        # Task 1: Daily protocol risk assessment
        logger.info("Scheduling daily protocol risk assessment")
        # schedule.every().day.at("00:00").do(self.assess_protocol_risk)

        # Task 2: Hourly loan health scan
        logger.info("Scheduling hourly loan health scan")
        # schedule.every().hour.do(self.scan_unhealthy_loans)
```

```
        # Task 3: Weekly collection risk tier updates
        logger.info("Scheduling weekly collection risk tier updates")
        # def update_all_collections():
        #     collections = self._get_all_supported_collections()
        #     for chainlet_id, collection_addr in collections:
        #         self.update_collection_risk_tier(chainlet_id,
collection_addr)
        # schedule.every().monday.at("12:00").do(update_all_collections)

        logger.info("Risk assessment tasks scheduled")


# Example usage of the system
def risk_management_example():
    # Configuration
    web3_provider = "https://rpc.saga.xyz/mainnet"
    contracts = {
        "nftVault": "0xNFTVaultAddress",
        "loanManager": "0xLoanManagerAddress",
        "oracle": "0xOracleAddress",
        "governance": "0xGovernanceAddress"
    }
    private_key = "0xPRIVATE_KEY"
    saga_api_key = "SAGA_API_KEY"
    chainlet_endpoints = {
        "chainlet1": "https://api.chainlet1.saga.xyz",
        "chainlet2": "https://api.chainlet2.saga.xyz",
    }

    # Initialize risk manager
    risk_manager = GameFiRiskManager(
        web3_provider=web3_provider,
        contracts=contracts,
        private_key=private_key,
        saga_api_key=saga_api_key,
        chainlet_endpoints=chainlet_endpoints
    )

    # Example 1: Analyze collection risk
    collection_risk = risk_manager.analyze_collection_risk(
        "chainlet1",
        "0xCollectionAddress"
    )
    print(f"Collection risk analysis: {collection_risk}")

    # Example 2: Update collection risk tier
    update_result = risk_manager.update_collection_risk_tier(
```

```
        "chainlet1",
        "0xCollectionAddress"
    )
    print(f"Update collection risk tier: {update_result}")

    # Example 3: Scan unhealthy loans
    unhealthy_loans = risk_manager.scan_unhealthy_loans()
    print(f"Found {len(unhealthy_loans)} unhealthy loans")

    # Example 4: Assess protocol risk
    protocol_risk = risk_manager.assess_protocol_risk()
    print(f"Protocol risk assessment: {protocol_risk}")

    # Example 5: Schedule risk assessment tasks
    risk_manager.schedule_risk_assessment_tasks()


if __name__ == "__main__":
    risk_management_example()
```

3. Testing and Security Framework

3.1 Unit & Integration Testing

```
// Test suite for Mosaical Smart Contract Suite
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("Mosaical MVP Test Suite", function () {
  // Common test variables
  let admin, borrower, lender, treasury;
  let governance, nftVault, loanManager, dpoToken, oracle, bridge;
  let gameNFT;
  let chainletId, collectionAddress;

  before(async function () {
    // Set up longer timeout for comprehensive tests
    this.timeout(300000);

    // Get signers
    [admin, borrower, lender, treasury, ...others] = await
ethers.getSigners();

    // Deploy all contracts
    const MosaicalGovernance = await
ethers.getContractFactory("MosaicalGovernance");
```

```
    const NFTVaultV3 = await ethers.getContractFactory("NFTVaultV3");
    const LoanManagerV3 = await
ethers.getContractFactory("LoanManagerV3");
    const DPOTokenV3 = await ethers.getContractFactory("DPOTokenV3");
    const GameFiOracleV3 = await
ethers.getContractFactory("GameFiOracleV3");
    const MosaicalSagaBridge = await
ethers.getContractFactory("MosaicalSagaBridge");

    // Deploy governance token
    const GovernanceToken = await
ethers.getContractFactory("GovernanceToken");
    const govToken = await GovernanceToken.deploy("Mosaical Governance",
"MSCLGOV");

    governance = await MosaicalGovernance.deploy(govToken.address);
    nftVault = await NFTVaultV3.deploy();
    oracle = await GameFiOracleV3.deploy();
    dpoToken = await DPOTokenV3.deploy();
    loanManager = await LoanManagerV3.deploy();
    bridge = await MosaicalSagaBridge.deploy("0xLayerZeroEndpoint");

    // Set up contract connections
    await nftVault.initialize(oracle.address, loanManager.address);
    await loanManager.initialize(nftVault.address, oracle.address,
dpoToken.address, treasury.address);
    await dpoToken.initialize(loanManager.address, treasury.address);
    await oracle.initialize();

    // Deploy mock GameFi NFT
    const MockGameNFT = await ethers.getContractFactory("MockGameNFT");
    gameNFT = await MockGameNFT.deploy("Test GameFi NFT", "TGNFT");

    // Set up test data
    chainletId = "0x1111111111111111111111111111111111111111";
    collectionAddress = gameNFT.address;

    // Register collection
    await nftVault.addSupportedChainlet(chainletId);
    await nftVault.addSupportedCollection(chainletId, collectionAddress);
    await nftVault.setGameCategory(collectionAddress, 1); // RPG game
    await nftVault.setCollectionRiskTier(collectionAddress, 2);

    // Set oracle price
    await oracle.setCollectionFloorPrice(collectionAddress,
ethers.utils.parseEther("10"));
```

```
    // Mint NFT to borrower
    await gameNFT.mint(borrower.address, 1);
    await gameNFT.mint(borrower.address, 2);

    // Set up bridge
    await bridge.addSupportedChainlet(1); // Chainlet ID

    // Fund treasury for lending
    await admin.sendTransaction({
      to: treasury.address,
      value: ethers.utils.parseEther("1000")
    });
  });

  // Governance Tests
  describe("Governance System", function () {
    it("Should allow creating proposals with sufficient tokens", async
function () {
      // Mint governance tokens to admin
      const govToken = await ethers.getContractAt(
        "GovernanceToken",
        await governance.governanceToken()
      );
      await govToken.mint(admin.address,
ethers.utils.parseEther("2000000"));

      // Create proposal
      const targets = [nftVault.address];
      const values = [0];
      const calldatas = [
        nftVault.interface.encodeFunctionData("setCollectionRiskTier", [
          collectionAddress, 1
        ])
      ];
      const description = "Update collection risk tier";

      const tx = await governance.connect(admin).propose(
        targets, values, calldatas, description
      );
      const receipt = await tx.wait();

      // Get proposal ID from events
      const event = receipt.events.find(e => e.event ===
"ProposalCreated");
      expect(event).to.not.be.undefined;

      const proposalId = event.args.id;
```

```
      expect(proposalId).to.equal(1);
    });

    it("Should allow voting on proposals", async function () {
      // Skip blocks to reach voting period
      for (let i = 0; i < 13150; i++) {
        await ethers.provider.send("evm_mine", []);
      }

      // Vote on proposal
      await governance.connect(admin).castVote(1, true);

      // Verify vote recorded
      const receipt = await governance.receipts(1, admin.address);
      expect(receipt.hasVoted).to.be.true;
      expect(receipt.support).to.be.true;
    });

    it("Should execute passed proposals", async function () {
      // Skip blocks to end voting period
      for (let i = 0; i < 40330; i++) {
        await ethers.provider.send("evm_mine", []);
      }

      // Execute proposal
      await governance.connect(admin).execute(1);

      // Verify proposal executed
      const proposal = await governance.proposals(1);
      expect(proposal.executed).to.be.true;

      // Verify effect of execution
      const riskTier = await
nftVault.collectionRiskTier(collectionAddress);
      expect(riskTier).to.equal(1);
    });
  });

  // NFT Vault Tests
  describe("NFT Vault V3", function () {
    it("Should correctly process GameFi NFT metadata on deposit", async
function () {
      // Set NFT game data in oracle
      await oracle.setNFTUtilityScore(collectionAddress, 1, 150);

      // Approve and deposit NFT
      await gameNFT.connect(borrower).approve(nftVault.address, 1);
```

```
      await nftVault.connect(borrower).depositNFT(collectionAddress, 1);

      // Verify NFT data stored correctly
      const nftData = await nftVault.nftData(collectionAddress, 1);
      expect(nftData.utilityScore).to.equal(150);
    });

    it("Should calculate max LTV based on risk tier and utility", async
function () {
      const maxLTV = await nftVault.getMaxLTV(collectionAddress, 1);

      // Risk tier 1 has base LTV 70% + utility bonus
      // Utility 150 = 50% above baseline = 10% bonus (capped at
maxUtilityBonus 20%)
      expect(maxLTV).to.equal(80); // 70% base + 10% bonus
    });

    it("Should adjust risk model parameters through governance", async
function () {
      // Set up governance proposal to update risk model
      const govToken = await ethers.getContractAt(
        "GovernanceToken",
        await governance.governanceToken()
      );

      const targets = [nftVault.address];
      const values = [0];
      const calldatas = [
        nftVault.interface.encodeFunctionData("updateRiskModel", [
          1, // tier
          75, // baseLTV
          85, // liquidationThreshold
          15, // maxUtilityBonus
          ethers.utils.parseEther("0.2") // minCollateralAmount
        ])
      ];
      const description = "Update risk model for tier 1";

      // Create, vote on, and execute proposal
      await governance.connect(admin).propose(
        targets, values, calldatas, description
      );

      // Skip blocks to voting period
      for (let i = 0; i < 13150; i++) {
        await ethers.provider.send("evm_mine", []);
      }
```

```
      await governance.connect(admin).castVote(2, true);

      // Skip blocks to end voting
      for (let i = 0; i < 40330; i++) {
        await ethers.provider.send("evm_mine", []);
      }

      await governance.connect(admin).execute(2);

      // Verify risk model updated
      const riskModel = await nftVault.riskModels(1);
      expect(riskModel.baseLTV).to.equal(75);
      expect(riskModel.liquidationThreshold).to.equal(85);
      expect(riskModel.maxUtilityBonus).to.equal(15);

expect(riskModel.minCollateralAmount).to.equal(ethers.utils.parseEther("0.
2"));
    });
  });

  // Loan Manager Tests
  describe("Loan Manager V3", function () {
    it("Should dynamically calculate interest rates based on utilization",
async function () {
      // Set up interest rate model
      await loanManager.setInterestRateModel(
        collectionAddress,
        200,  // 2% base rate
        1000, // 10% slope1
        5000, // 50% slope2
        8000  // 80% optimal utilization
      );

      // Mock total borrowed and supplied
      await loanManager.setTotalBorrowed(collectionAddress,
ethers.utils.parseEther("80"));
      await loanManager.setTotalSupplied(collectionAddress,
ethers.utils.parseEther("100"));

      // Check interest rate at 80% utilization
      const rate = await
loanManager.calculateInterestRate(collectionAddress);
      // Expected: baseRate + slope1 = 2% + 10% = 12%
      expect(rate).to.equal(1200);

      // Update to 90% utilization (above optimal)
```

```
      await loanManager.setTotalBorrowed(collectionAddress,
ethers.utils.parseEther("90"));

      // Check new interest rate
      const newRate = await
loanManager.calculateInterestRate(collectionAddress);
      // Expected: baseRate + slope1 + slope2 * (90-80)/(100-80) = 2% +
10% + 50%*0.5 = 37%
      expect(newRate).to.equal(3700);
    });
    it("Should correctly handle loan creation with dynamic interest",
async function () {
      // Fund loan manager with lending tokens
      await treasury.sendTransaction({
        to: loanManager.address,
        value: ethers.utils.parseEther("100")
      });

      // Approve and borrow
      // Note: NFT #1 already deposited in vault
      const borrowAmount = ethers.utils.parseEther("5"); // 5 ETH
      await loanManager.connect(borrower).borrow(
        collectionAddress,
        1,
        borrowAmount
      );

      // Verify loan created
      const loan = await loanManager.loans(borrower.address,
collectionAddress, 1);
      expect(loan).to.equal(borrowAmount);

      // Verify loan data
      const loanData = await loanManager.loanData(borrower.address,
collectionAddress, 1);
      expect(loanData.principal).to.equal(borrowAmount);

      // Verify origination fee
      const originationFee = borrowAmount.mul(50).div(10000); // 0.5%
      expect(loanData.originationFee).to.equal(originationFee);

      // Verify interest rate set
      expect(loanData.interestRate).to.be.gt(0);

      // Verify DPO tokens minted
      const dpoBalance = await dpoToken.tokenHoldings(
        collectionAddress,
```

```
        1,
        borrower.address
      );
      expect(dpoBalance).to.be.gt(0);
    });

    it("Should correctly calculate and accrue interest over time", async
function () {
      // Initial interest should be 0
      const initialLoanData = await loanManager.loanData(borrower.address,
collectionAddress, 1);
      expect(initialLoanData.accruedInterest).to.equal(0);

      // Advance time by 30 days
      await ethers.provider.send("evm_increaseTime", [30 * 86400]);
      await ethers.provider.send("evm_mine", []);

      // Update interest
      await loanManager.updateLoanInterest(borrower.address,
collectionAddress, 1);

      // Verify interest accrued
      const updatedLoanData = await loanManager.loanData(borrower.address,
collectionAddress, 1);
      expect(updatedLoanData.accruedInterest).to.be.gt(0);

      // Verify interest is approximately correct
      // With ~12% APR, 30 days interest on 5 ETH should be around 0.05
ETH
      const expectedInterest = ethers.utils.parseEther("0.05");
      const actualInterest = updatedLoanData.accruedInterest;

      // Allow 10% margin of error due to block timestamp variations
      const lowerBound = expectedInterest.mul(90).div(100);
      const upperBound = expectedInterest.mul(110).div(100);

      expect(actualInterest).to.be.gte(lowerBound);
      expect(actualInterest).to.be.lte(upperBound);
    });

    it("Should correctly repay loan with interest allocation", async
function () {
      // Advance time to accrue more interest
      await ethers.provider.send("evm_increaseTime", [10 * 86400]);
      await ethers.provider.send("evm_mine", []);

      // Get current loan state
```

```
      const loanBefore = await loanManager.loans(borrower.address,
collectionAddress, 1);
      const loanDataBefore = await loanManager.loanData(borrower.address,
collectionAddress, 1);
      const totalOwed = loanBefore.add(loanDataBefore.accruedInterest);

      // Repay full loan
      await loanManager.connect(borrower).repay(
        collectionAddress,
        1,
        totalOwed,
        { value: totalOwed }
      );

      // Verify loan state after repayment
      const loanAfter = await loanManager.loans(borrower.address,
collectionAddress, 1);
      expect(loanAfter).to.equal(0);

      const loanDataAfter = await loanManager.loanData(borrower.address,
collectionAddress, 1);
      expect(loanDataAfter.accruedInterest).to.equal(0);
      expect(loanDataAfter.principal).to.equal(0);
    });

    it("Should correctly handle health factor updates with price changes",
async function () {
      // Create a new loan for testing
      await gameNFT.connect(borrower).approve(nftVault.address, 2);
      await nftVault.connect(borrower).depositNFT(collectionAddress, 2);

      // Set NFT utility score
      await oracle.setNFTUtilityScore(collectionAddress, 2, 120);

      // Borrow against NFT
      const borrowAmount = ethers.utils.parseEther("6"); // 6 ETH
      await loanManager.connect(borrower).borrow(
        collectionAddress,
        2,
        borrowAmount
      );

      // Initial NFT price is 10 ETH, borrowing 6 ETH
      // Initial health factor should be around 1.67
      const initialHealth = await
loanManager.loanHealthFactors(borrower.address, collectionAddress, 2);
      expect(initialHealth).to.be.gt(16500); // 1.65 scaled by 10000
```

```
      expect(initialHealth).to.be.lt(17000); // 1.70 scaled by 10000

      // Simulate price drop to 8 ETH
      await oracle.setCollectionFloorPrice(collectionAddress,
ethers.utils.parseEther("8"));

      // Update health factor
      await loanManager.updateHealthFactor(borrower.address,
collectionAddress, 2);

      // New health factor should be around 1.33
      const newHealth = await
loanManager.loanHealthFactors(borrower.address, collectionAddress, 2);
      expect(newHealth).to.be.gt(13000); // 1.30 scaled by 10000
      expect(newHealth).to.be.lt(13500); // 1.35 scaled by 10000

      // Simulate further price drop to trigger liquidation threshold
      await oracle.setCollectionFloorPrice(collectionAddress,
ethers.utils.parseEther("7"));

      // Update health factor
      await loanManager.updateHealthFactor(borrower.address,
collectionAddress, 2);

      // Health factor now below first liquidation threshold
      const finalHealth = await
loanManager.loanHealthFactors(borrower.address, collectionAddress, 2);
      expect(finalHealth).to.be.lt(12000); // 1.20 scaled by 10000
    });
  });

  // DPO Token Tests
  describe("DPO Token V3", function () {
    it("Should set up DPO token market orders correctly", async function
() {
      // Borrow against a new NFT to get DPO tokens
      await gameNFT.mint(borrower.address, 3);
      await gameNFT.connect(borrower).approve(nftVault.address, 3);
      await nftVault.connect(borrower).depositNFT(collectionAddress, 3);
      await oracle.setNFTUtilityScore(collectionAddress, 3, 140);

      const borrowAmount = ethers.utils.parseEther("5");
      await loanManager.connect(borrower).borrow(
        collectionAddress,
        3,
        borrowAmount
      );
```

```
      // Check DPO token balance
      const dpoBalance = await dpoToken.tokenHoldings(
        collectionAddress,
        3,
        borrower.address
      );
      expect(dpoBalance).to.be.gt(0);

      // Place sell order for 30% of tokens
      const sellAmount = dpoBalance.mul(30).div(100);
      const sellPrice = ethers.utils.parseEther("0.001"); // Price per
token

      await dpoToken.connect(borrower).placeSellOrder(
        collectionAddress,
        3,
        sellAmount,
        sellPrice
      );

      // Verify order placed
      const orderCount = await
dpoToken.getSellOrdersCount(collectionAddress, 3);
      expect(orderCount).to.equal(1);

      const order = await dpoToken.sellOrders(collectionAddress, 3, 0);
      expect(order.maker).to.equal(borrower.address);
      expect(order.amount).to.equal(sellAmount);
      expect(order.price).to.equal(sellPrice);
    });

    it("Should match buy and sell orders automatically", async function ()
{
      // Get sell order details
      const sellOrder = await dpoToken.sellOrders(collectionAddress, 3,
0);
      const sellAmount = sellOrder.amount;
      const sellPrice = sellOrder.price;

      // Calculate total cost
      const totalCost = sellAmount.mul(sellPrice);

      // Place buy order with sufficient funds
      await dpoToken.connect(lender).placeBuyOrder(
        collectionAddress,
        3,
```

```
        sellAmount,
        sellPrice,
        { value: totalCost }
      );

      // Orders should be automatically matched
      const sellOrderAfter = await dpoToken.sellOrders(collectionAddress,
3, 0);
      expect(sellOrderAfter.amount).to.equal(0); // Order should be filled

      // Verify tokens transferred to buyer
      const buyerBalance = await dpoToken.tokenHoldings(
        collectionAddress,
        3,
        lender.address
      );

      // Should have received tokens minus fee
      const fee = totalCost.mul(50).div(10000); // 0.5% fee
      const expectedTokens =
sellAmount.mul(totalCost.sub(fee)).div(totalCost);

      expect(buyerBalance).to.be.gt(0);
      expect(buyerBalance).to.be.closeTo(expectedTokens,
expectedTokens.div(100)); // Allow 1% rounding error
    });

    it("Should distribute interest to DPO token holders", async function
() {
      // Simulate interest payment
      const interestAmount = ethers.utils.parseEther("0.1");

      await loanManager.connect(admin).simulateInterestPayment(
        collectionAddress,
        3,
        interestAmount
      );

      // Distribute interest
      await dpoToken.connect(admin).distributeInterest(
        collectionAddress,
        3,
        interestAmount
      );

      // Verify interest distribution recorded
```

```
      const interestAccrual = await
dpoToken.getTotalDistributed(collectionAddress, 3);
      expect(interestAccrual).to.equal(interestAmount);

      // Calculate borrower's pending interest
      const borrowerPending = await dpoToken.calculatePendingInterest(
        borrower.address,
        collectionAddress,
        3
      );

      // Calculate lender's pending interest
      const lenderPending = await dpoToken.calculatePendingInterest(
        lender.address,
        collectionAddress,
        3
      );

      // Combined should equal total interest (minus rounding errors)
      const combinedInterest = borrowerPending.add(lenderPending);
      expect(combinedInterest).to.be.closeTo(interestAmount,
ethers.utils.parseEther("0.001"));

      // Claim interest
      await dpoToken.connect(borrower).claimInterest(collectionAddress,
3);

      // Verify claim recorded
      const borrowerPendingAfter = await
dpoToken.calculatePendingInterest(
        borrower.address,
        collectionAddress,
        3
      );
      expect(borrowerPendingAfter).to.equal(0);
    });

    it("Should burn DPO tokens on full loan repayment", async function ()
{
      // Get holder balances before
      const borrowerBalance = await dpoToken.tokenHoldings(
        collectionAddress,
        3,
        borrower.address
      );

      const lenderBalance = await dpoToken.tokenHoldings(
```

```
        collectionAddress,
        3,
        lender.address
      );

      expect(borrowerBalance).to.be.gt(0);
      expect(lenderBalance).to.be.gt(0);

      // Repay the loan fully
      const loanAmount = await loanManager.loans(borrower.address,
collectionAddress, 3);
      const loanData = await loanManager.loanData(borrower.address,
collectionAddress, 3);
      const totalOwed = loanAmount.add(loanData.accruedInterest);

      await loanManager.connect(borrower).repay(
        collectionAddress,
        3,
        totalOwed,
        { value: totalOwed }
      );

      // Verify DPO tokens burned
      const borrowerBalanceAfter = await dpoToken.tokenHoldings(
        collectionAddress,
        3,
        borrower.address
      );

      const lenderBalanceAfter = await dpoToken.tokenHoldings(
        collectionAddress,
        3,
        lender.address
      );

      expect(borrowerBalanceAfter).to.equal(0);
      expect(lenderBalanceAfter).to.equal(0);
    });
  });

  // Oracle Tests
  describe("GameFi Oracle V3", function () {
    it("Should integrate with AI prediction system", async function () {
      // Deploy mock AI prediction system
      const MockAIPredictor = await
ethers.getContractFactory("MockAIPredictor");
      const aiPredictor = await MockAIPredictor.deploy();
```

```
      // Set AI prediction system in oracle
      await oracle.setAIPredictionSystem(aiPredictor.address);

      // Set up mock AI prediction
      const aiPrice = ethers.utils.parseEther("12"); // AI predicts 12 ETH
      const confidence = 80; // 80% confidence

      await aiPredictor.setMockPrediction(
        collectionAddress,
        1,
        aiPrice,
        confidence
      );

      // Get NFT price with AI integration
      const price = await oracle.getNFTPrice(collectionAddress, 1);

      // Price should be weighted blend of AI and oracle price
      // Oracle price is 7 ETH (from previous test)
      // AI price is 12 ETH with 80% confidence
      // Expected: (7 * 0.2) + (12 * 0.8) = 1.4 + 9.6 = 11 ETH
      // Then adjusted for utility (150%) = 11 * 1.5 = 16.5 ETH

      // Allow 5% margin for rounding differences
      const expectedPrice = ethers.utils.parseEther("16.5");
      const lowerBound = expectedPrice.mul(95).div(100);
      const upperBound = expectedPrice.mul(105).div(100);

      expect(price).to.be.gte(lowerBound);
      expect(price).to.be.lte(upperBound);
    });

    it("Should track game activity metrics for risk assessment", async
function () {
      // Update game metrics
      await oracle.connect(admin).updateGameMetrics(
        collectionAddress,
        10000, // 10K active users
        45,    // 45 min avg playtime
        50000, // $50K daily revenue
        7000   // 70% retention
      );

      // Verify metrics recorded
      const metrics = await oracle.gameMetrics(collectionAddress);
      expect(metrics.activeUsers).to.equal(10000);
```

```
      expect(metrics.avgPlaytime).to.equal(45);
      expect(metrics.revenue).to.equal(50000);
      expect(metrics.retention).to.equal(7000);

      // Get engagement factor
      const engagement = await
oracle.getGameEngagementFactor(collectionAddress);

      // Should be high with good metrics
      expect(engagement).to.be.gte(80);
    });
  });

  // Bridge Tests
  describe("Saga Protocol Bridge", function () {
    it("Should set up cross-chainlet mappings correctly", async function
() {
      const remoteChainletId = 123;
      const remoteCollectionAddress = "0xRemoteCollectionAddress";

      // Set trusted remote path
      const remotePath = ethers.utils.defaultAbiCoder.encode(
        ["address"],
        ["0xRemoteBridgeAddress"]
      );

      await bridge.setTrustedRemote(remoteChainletId, remotePath);

      // Map collection between chainlets
      await bridge.mapCollection(
        collectionAddress,
        remoteChainletId,
        remoteCollectionAddress
      );

      // Verify mapping
      const mapping = await bridge.remoteMappings(
        collectionAddress,
        remoteChainletId
      );

      expect(mapping).to.equal(remoteCollectionAddress);
    });

    it("Should simulate NFT bridging process", async function () {
      // This is a simplified test since we can't fully test cross-chain
in Hardhat
```

```
      // Mint new NFT for bridging
      await gameNFT.mint(borrower.address, 10);

      // Approve bridge to transfer NFT
      await gameNFT.connect(borrower).approve(bridge.address, 10);

      // Simulate bridging
      // In a real scenario, this would emit an event that the mock would
handle
      await expect(
        bridge.connect(borrower).bridgeNFT(
          collectionAddress,
          10,
          123, // remoteChainletId
          { value: ethers.utils.parseEther("0.1") } // Pay bridge fee
        )
      ).to.emit(bridge, "NFTBridgeInitiated")
        .withArgs(collectionAddress, 10, borrower.address, 123);

      // Verify NFT transferred to bridge
      expect(await gameNFT.ownerOf(10)).to.equal(bridge.address);

      // Verify pending NFT recorded
      const pendingNFTId = ethers.utils.keccak256(
        ethers.utils.defaultAbiCoder.encode(
          ["address", "uint256", "uint16"],
          [collectionAddress, 10, 123]
        )
      );

      const pendingNFT = await bridge.pendingNFTs(pendingNFTId);
      expect(pendingNFT.collection).to.equal(collectionAddress);
      expect(pendingNFT.tokenId).to.equal(10);
      expect(pendingNFT.owner).to.equal(borrower.address);
      expect(pendingNFT.isBridged).to.be.true;
    });
  });

  // Integration Tests
  describe("Complete GameFi NFT Lending Flow", function () {
    it("Should perform complete lending cycle with all components", async
function () {
      // Mint a new NFT for this test
      await gameNFT.mint(borrower.address, 100);

      // 1. Set up collection in vault
```

```
      await nftVault.setGameCategory(collectionAddress, 1); // RPG
      await nftVault.setCollectionRiskTier(collectionAddress, 3); //
Medium risk

      // 2. Set up oracle price and utility score
      await oracle.setCollectionFloorPrice(collectionAddress,
ethers.utils.parseEther("20"));
      await oracle.setNFTUtilityScore(collectionAddress, 100, 160);

      // 3. Update game metrics in oracle
      await oracle.updateGameMetrics(
        collectionAddress,
        8000,  // 8K active users
        60,    // 60 min avg playtime
        30000, // $30K daily revenue
        6000   // 60% retention
      );

      // 4. Set up interest rate model
      await loanManager.setInterestRateModel(
        collectionAddress,
        300,  // 3% base rate
        1000, // 10% slope1
        6000, // 60% slope2
        7000  // 70% optimal utilization
      );

      // 5. Deposit NFT
      await gameNFT.connect(borrower).approve(nftVault.address, 100);
      await nftVault.connect(borrower).depositNFT(collectionAddress, 100);

      // 6. Borrow against NFT
      const maxLTV = await nftVault.getMaxLTV(collectionAddress, 100);
      const nftPrice = await oracle.getNFTPrice(collectionAddress, 100);
      const borrowAmount = nftPrice.mul(maxLTV).div(100);

      await loanManager.connect(borrower).borrow(
        collectionAddress,
        100,
        borrowAmount
      );

      // 7. Verify DPO tokens minted
      const dpoBalance = await dpoToken.tokenHoldings(
        collectionAddress,
        100,
        borrower.address
```

```
      );
      expect(dpoBalance).to.be.gt(0);

      // 8. Place sell order for 40% of DPO tokens
      const sellAmount = dpoBalance.mul(40).div(100);
      const sellPrice = ethers.utils.parseEther("0.001");

      await dpoToken.connect(borrower).placeSellOrder(
        collectionAddress,
        100,
        sellAmount,
        sellPrice
      );

      // 9. Buy tokens from another account
      const totalCost = sellAmount.mul(sellPrice);
      await dpoToken.connect(lender).placeBuyOrder(
        collectionAddress,
        100,
        sellAmount,
        sellPrice,
        { value: totalCost }
      );

      // 10. Verify trade completed
      const lenderBalance = await dpoToken.tokenHoldings(
        collectionAddress,
        100,
        lender.address
      );
      expect(lenderBalance).to.be.gt(0);

      // 11. Simulate time passing for interest accrual
      await ethers.provider.send("evm_increaseTime", [30 * 86400]); // 30
days
      await ethers.provider.send("evm_mine", []);

      // 12. Update interest
      await loanManager.updateLoanInterest(borrower.address,
collectionAddress, 100);

      // 13. Distribute some interest
      const loanData = await loanManager.loanData(borrower.address,
collectionAddress, 100);
      const interest = loanData.accruedInterest;
      expect(interest).to.be.gt(0);
```

```
      // 14. Distribute interest
      await dpoToken.distributeInterest(
        collectionAddress,
        100,
        interest
      );

      // 15. Claim interest
      await dpoToken.connect(lender).claimInterest(collectionAddress,
100);
      await dpoToken.connect(borrower).claimInterest(collectionAddress,
100);

      // 16. Repay loan
      const loan = await loanManager.loans(borrower.address,
collectionAddress, 100);
      const updatedLoanData = await loanManager.loanData(borrower.address,
collectionAddress, 100);
      const totalOwed = loan.add(updatedLoanData.accruedInterest);

      await loanManager.connect(borrower).repay(
        collectionAddress,
        100,
        totalOwed,
        { value: totalOwed }
      );

      // 17. Verify loan closed
      const loanAfter = await loanManager.loans(borrower.address,
collectionAddress, 100);
      expect(loanAfter).to.equal(0);

      // 18. Withdraw NFT
      await nftVault.connect(borrower).withdrawNFT(collectionAddress,
100);

      // 19. Verify NFT returned
      expect(await gameNFT.ownerOf(100)).to.equal(borrower.address);
    });
  });
});
```

3.2 Security Testing

```
// Security tests for Mosaical Protocol
const { expect } = require("chai");
const { ethers } = require("hardhat");
```

```
const { time } = require("@nomicfoundation/hardhat-network-helpers");

describe("Mosaical Protocol Security Tests", function () {
  let admin, attacker, user, treasury;
  let nftVault, loanManager, dpoToken, oracle;
  let gameNFT;

  before(async function () {
    [admin, attacker, user, treasury] = await ethers.getSigners();

    // Deploy contracts
    const NFTVaultV3 = await ethers.getContractFactory("NFTVaultV3");
    const LoanManagerV3 = await
ethers.getContractFactory("LoanManagerV3");
    const DPOTokenV3 = await ethers.getContractFactory("DPOTokenV3");
    const GameFiOracleV3 = await
ethers.getContractFactory("GameFiOracleV3");

    nftVault = await NFTVaultV3.deploy();
    oracle = await GameFiOracleV3.deploy();
    dpoToken = await DPOTokenV3.deploy();
    loanManager = await LoanManagerV3.deploy();

    // Set up contracts
    await nftVault.initialize(oracle.address, loanManager.address);
    await loanManager.initialize(nftVault.address, oracle.address,
dpoToken.address, treasury.address);
    await dpoToken.initialize(loanManager.address, treasury.address);
    await oracle.initialize();

    // Deploy mock GameFi NFT
    const MockGameNFT = await ethers.getContractFactory("MockGameNFT");
    gameNFT = await MockGameNFT.deploy("Security Test NFT", "STNFT");

    // Register collection
    await nftVault.addSupportedChainlet("0x1111");
    await nftVault.addSupportedCollection("0x1111", gameNFT.address);
    await nftVault.setGameCategory(gameNFT.address, 1);
    await nftVault.setCollectionRiskTier(gameNFT.address, 2);

    // Set oracle price
    await oracle.setCollectionFloorPrice(gameNFT.address,
ethers.utils.parseEther("10"));

    // Fund treasury for lending
    await admin.sendTransaction({
      to: treasury.address,
```

```
      value: ethers.utils.parseEther("1000")
    });

    // Give attacker some funds
    await admin.sendTransaction({
      to: attacker.address,
      value: ethers.utils.parseEther("10")
    });

    // Mint NFTs
    await gameNFT.mint(user.address, 1);
    await gameNFT.mint(user.address, 2);
    await gameNFT.mint(attacker.address, 3);
  });

  describe("Access Control", function () {
    it("Should prevent unauthorized access to admin functions", async
function () {
      // Attacker tries to add a new collection
      await expect(
        nftVault.connect(attacker).addSupportedCollection("0x1111",
attacker.address)
      ).to.be.reverted;

      // Attacker tries to set collection risk tier
      await expect(
        nftVault.connect(attacker).setCollectionRiskTier(gameNFT.address,
1)
      ).to.be.reverted;

      // Attacker tries to set interest rate model
      await expect(
        loanManager.connect(attacker).setInterestRateModel(
          gameNFT.address,
          200,
          1000,
          5000,
          8000
        )
      ).to.be.reverted;

      // Attacker tries to set oracle price
      await expect(
        oracle.connect(attacker).setCollectionFloorPrice(gameNFT.address,
ethers.utils.parseEther("100"))
      ).to.be.reverted;
    });
```

```
    it("Should prevent unauthorized calls between contracts", async
function () {
      // Attacker tries to mint DPO tokens directly
      await expect(
        dpoToken.connect(attacker).mintDPOTokens(
          attacker.address,
          gameNFT.address,
          1,
          ethers.utils.parseEther("10")
        )
      ).to.be.reverted;

      // Attacker tries to distribute interest directly
      await expect(
        dpoToken.connect(attacker).distributeInterest(
          gameNFT.address,
          1,
          ethers.utils.parseEther("1")
        )
      ).to.be.reverted;
    });
  });

  describe("Oracle Security", function () {
    it("Should prevent price manipulation attacks", async function () {
      // Set up proper loan for user
      await gameNFT.connect(user).approve(nftVault.address, 1);
      await nftVault.connect(user).depositNFT(gameNFT.address, 1);

      const borrowAmount = ethers.utils.parseEther("5");
      await loanManager.connect(user).borrow(
        gameNFT.address,
        1,
        borrowAmount
      );

      // Try to manipulate price to liquidate user
      // Since attacker can't directly set price, this should fail
      await expect(
        oracle.connect(attacker).setCollectionFloorPrice(gameNFT.address,
ethers.utils.parseEther("1"))
      ).to.be.reverted;

      // Even admin should use proper governance for price changes
      // Verify multi-source oracle properly validates price sources
```

```
      // This would be an integration test with the AI system in
production
    });

    it("Should have circuit breakers for extreme price movements", async
function () {
      // Set up circuit breaker in oracle
      await oracle.connect(admin).setCircuitBreaker(
        gameNFT.address,
        30, // 30% max daily change
        true // enabled
      );

      // Try to update price beyond circuit breaker threshold
      await expect(
        oracle.connect(admin).setCollectionFloorPrice(gameNFT.address,
ethers.utils.parseEther("5"))
      ).to.be.reverted; // 50% drop from 10 ETH

      // Allowed change within threshold
      await oracle.connect(admin).setCollectionFloorPrice(gameNFT.address,
ethers.utils.parseEther("7"));
      expect(await
oracle.collectionFloorPrices(gameNFT.address)).to.equal(ethers.utils.parse
Ether("7"));
    });
  });

  describe("Reentrancy Protection", function () {
    it("Should prevent reentrancy attacks on lending functions", async
function () {
      // Deploy malicious NFT that attempts reentrancy
      const MaliciousNFT = await
ethers.getContractFactory("MaliciousNFT");
      const maliciousNFT = await MaliciousNFT.connect(attacker).deploy(
        loanManager.address,
        nftVault.address
      );

      // Try adding malicious NFT to protocol
      await nftVault.connect(admin).addSupportedCollection("0x1111",
maliciousNFT.address);
      await nftVault.connect(admin).setGameCategory(maliciousNFT.address,
1);
      await
nftVault.connect(admin).setCollectionRiskTier(maliciousNFT.address, 3);
```

```
      await
oracle.connect(admin).setCollectionFloorPrice(maliciousNFT.address,
ethers.utils.parseEther("10"));

      // Mint malicious NFT to attacker
      await maliciousNFT.connect(attacker).mint(attacker.address, 1);

      // Try reentrancy attack through deposit
      await maliciousNFT.connect(attacker).approve(nftVault.address, 1);
      await expect(
        maliciousNFT.connect(attacker).executeAttack(1)
      ).to.be.reverted;

      // Manually deposit NFT for next test
      await nftVault.connect(attacker).depositNFT(maliciousNFT.address,
1);

      // Try reentrancy attack through borrow
      await expect(
        maliciousNFT.connect(attacker).executeBorrowAttack(1,
ethers.utils.parseEther("5"))
      ).to.be.reverted;

      // Try reentrancy attack through repayment
      await loanManager.connect(attacker).borrow(
        maliciousNFT.address,
        1,
        ethers.utils.parseEther("5")
      );

      await expect(
        maliciousNFT.connect(attacker).executeRepayAttack(1,
ethers.utils.parseEther("5"), { value: ethers.utils.parseEther("5") })
      ).to.be.reverted;
    });
  });

  describe("Flash Loan Attack Protection", function () {
    it("Should prevent flash loan price manipulation attacks", async
function () {
      // Set up attack scenario
      // User has NFT worth 10 ETH and borrows 5 ETH
      await gameNFT.connect(user).approve(nftVault.address, 2);
      await nftVault.connect(user).depositNFT(gameNFT.address, 2);
      await oracle.connect(admin).setCollectionFloorPrice(gameNFT.address,
ethers.utils.parseEther("10"));
```

```
      const borrowAmount = ethers.utils.parseEther("5");
      await loanManager.connect(user).borrow(
        gameNFT.address,
        2,
        borrowAmount
      );

      // In a real attack, attacker would:
      // 1. Take flash loan
      // 2. Manipulate price feed (not possible with proper oracle)
      // 3. Liquidate position
      // 4. Repay flash loan

      // Our oracle should use TWAP or similar mechanism to prevent this
      // Verify that oracle uses price volatility checks
      expect(await
oracle.collectionVolatility(gameNFT.address)).to.not.equal(0);

      // Oracle should check multiple price sources
      const priceSourceCount = await
oracle.getPriceSourceCount(gameNFT.address);
      expect(priceSourceCount).to.be.gt(0);

      // Oracle should have a delay between price updates
      const lastUpdateTime = await
oracle.lastPriceUpdateTime(gameNFT.address);
      expect(lastUpdateTime).to.not.equal(0);

      // Try to update price again immediately (should be rate limited)
      await expect(
        oracle.connect(admin).setCollectionFloorPrice(gameNFT.address,
ethers.utils.parseEther("9"))
      ).to.be.revertedWith("Rate limited");

      // Wait for rate limit to expire
      await ethers.provider.send("evm_increaseTime", [3600]); // 1 hour
      await ethers.provider.send("evm_mine", []);

      // Now should be able to update
      await oracle.connect(admin).setCollectionFloorPrice(gameNFT.address,
ethers.utils.parseEther("9"));
    });
  });

  describe("Liquidation System Security", function () {
    it("Should prevent unfair liquidations", async function () {
      // Set up liquidation scenario
```

```
      await gameNFT.mint(user.address, 10);
      await gameNFT.connect(user).approve(nftVault.address, 10);
      await nftVault.connect(user).depositNFT(gameNFT.address, 10);

      // Set NFT utility score
      await oracle.connect(admin).setNFTUtilityScore(gameNFT.address, 10,
100);

      // Borrow near max LTV
      await oracle.connect(admin).setCollectionFloorPrice(gameNFT.address,
ethers.utils.parseEther("10"));

      // Risk tier 2 has ~65% max LTV
      const borrowAmount = ethers.utils.parseEther("6.5");
      await loanManager.connect(user).borrow(
        gameNFT.address,
        10,
        borrowAmount
      );

      // Drop price just below liquidation threshold
      await ethers.provider.send("evm_increaseTime", [3600]); // Skip time
limit
      await ethers.provider.send("evm_mine", []);
      await oracle.connect(admin).setCollectionFloorPrice(gameNFT.address,
ethers.utils.parseEther("8.5"));

      // Update health factor
      await loanManager.updateHealthFactor(user.address, gameNFT.address,
10);

      // Attacker tries to liquidate but shouldn't be able to yet
      await expect(
        loanManager.connect(attacker).liquidatePartial(user.address,
gameNFT.address, 10)
      ).to.be.revertedWith("No liquidation needed");

      // Drop price further to trigger liquidation
      await ethers.provider.send("evm_increaseTime", [3600]); // Skip time
limit
      await ethers.provider.send("evm_mine", []);
      await oracle.connect(admin).setCollectionFloorPrice(gameNFT.address,
ethers.utils.parseEther("7.5"));

      // Update health factor
      await loanManager.updateHealthFactor(user.address, gameNFT.address,
10);
```

```
      // Now liquidation should be possible but only at the correct tier
level
      // First level should be 25% liquidation
      await loanManager.connect(attacker).liquidatePartial(user.address,
gameNFT.address, 10);

      // Verify only 25% was liquidated
      const remainingLoan = await loanManager.loans(user.address,
gameNFT.address, 10);
      expect(remainingLoan).to.be.closeTo(
        borrowAmount.mul(75).div(100),
        ethers.utils.parseEther("0.01") // Allow small rounding error
      );
    });
  });

  describe("Denial of Service Protection", function () {
    it("Should prevent DoS attacks on critical functions", async function
() {
      // Test gas limits on loops

      // Try to create many risk models at once
      // Should use bounded loops
      for (let i = 1; i <= 10; i++) {
        await nftVault.connect(admin).updateRiskModel(
          i,
          65,
          75,
          15,
          ethers.utils.parseEther("0.1")
        );
      }

      // Verify gas usage remains reasonable
      const gasEstimate = await nftVault.estimateGas.getAllRiskModels();
      expect(gasEstimate).to.be.lt(5000000);

      // Should properly handle DPO token holder enumeration
      // Create multiple DPO holders
      await gameNFT.mint(user.address, 20);
      await gameNFT.connect(user).approve(nftVault.address, 20);
      await nftVault.connect(user).depositNFT(gameNFT.address, 20);

      const borrowAmount = ethers.utils.parseEther("5");
      await loanManager.connect(user).borrow(
        gameNFT.address,
```

```
        20,
        borrowAmount
      );

      const dpoBalance = await dpoToken.tokenHoldings(
        gameNFT.address,
        20,
        user.address
      );

      // Split tokens between multiple addresses
      const splitAddresses = 10;
      const transferAmount = dpoBalance.div(splitAddresses * 2);

      for (let i = 0; i < splitAddresses; i++) {
        const wallet =
ethers.Wallet.createRandom().connect(ethers.provider);

        // Fund wallet for gas
        await admin.sendTransaction({
          to: wallet.address,
          value: ethers.utils.parseEther("0.1")
        });

        // Transfer DPO tokens by creating sell order and buy order
        await dpoToken.connect(user).placeSellOrder(
          gameNFT.address,
          20,
          transferAmount,
          ethers.utils.parseEther("0.001")
        );

        await dpoToken.connect(wallet).placeBuyOrder(
          gameNFT.address,
          20,
          transferAmount,
          ethers.utils.parseEther("0.001"),
          { value: transferAmount.mul(ethers.utils.parseEther("0.001")) }
        );
      }

      // Verify repayment with many DPO holders doesn't exceed gas limits
      await ethers.provider.send("evm_increaseTime", [30 * 86400]); // 30
days
      await ethers.provider.send("evm_mine", []);

      // Update interest
```

```
      await loanManager.updateLoanInterest(user.address, gameNFT.address,
20);

      // Get repayment amount
      const loan = await loanManager.loans(user.address, gameNFT.address,
20);
      const loanData = await loanManager.loanData(user.address,
gameNFT.address, 20);
      const totalOwed = loan.add(loanData.accruedInterest);

      // Repay loan
      await loanManager.connect(user).repay(
        gameNFT.address,
        20,
        totalOwed,
        { value: totalOwed }
      );

      // Verify loan closed
      const loanAfter = await loanManager.loans(user.address,
gameNFT.address, 20);
      expect(loanAfter).to.equal(0);
    });
  });

  describe("Overflow/Underflow Protection", function () {
    it("Should safely handle large numbers and prevent overflows", async
function () {
      // Test extreme values

      // Very large NFT price
      const largePrice = ethers.constants.MaxUint256.div(2);
      await oracle.connect(admin).setCollectionFloorPrice(gameNFT.address,
largePrice);

      // Mint and deposit NFT
      await gameNFT.mint(user.address, 50);
      await gameNFT.connect(user).approve(nftVault.address, 50);
      await nftVault.connect(user).depositNFT(gameNFT.address, 50);

      // Borrowing should still work without overflow
      // But should be limited by protocol max limits
      await expect(
        loanManager.connect(user).borrow(
          gameNFT.address,
          50,
          largePrice
```

```
        )
      ).to.be.revertedWith("Exceeds protocol max loan");

      // Valid large loan
      const maxProtocolLoan = await loanManager.maxLoanAmount();
      await loanManager.connect(user).borrow(
        gameNFT.address,
        50,
        maxProtocolLoan.sub(ethers.utils.parseEther("1")) // Just under
max
      );

      // Test with tiny amounts
      await gameNFT.mint(user.address, 51);
      await gameNFT.connect(user).approve(nftVault.address, 51);
      await nftVault.connect(user).depositNFT(gameNFT.address, 51);

      await oracle.connect(admin).setCollectionFloorPrice(gameNFT.address,
1); // 1 wei

      // Should fail due to minimum loan amount
      await expect(
        loanManager.connect(user).borrow(
          gameNFT.address,
          51,
          1
        )
      ).to.be.revertedWith("Below minimum loan amount");
    });
  });

  describe("Cross-Chain Security", function () {
    it("Should have proper safeguards for cross-chainlet assets", async
function () {
      // Deploy mock bridge
      const MockBridge = await
ethers.getContractFactory("MockMosaicalSagaBridge");
      const bridge = await MockBridge.deploy();

      // Add as trusted source in vault
      await nftVault.connect(admin).setTrustedBridge(bridge.address);

      // Register remote chainlet
      const remoteChainletId = 456;
      await bridge.addSupportedChainlet(remoteChainletId);

      // Set up mapping
```

```
      const remoteCollection = "0xRemoteGameCollection";
      await bridge.mapCollection(gameNFT.address, remoteChainletId,
remoteCollection);

      // Register cross-chainlet collection in vault
      await nftVault.connect(admin).addCrossChainletCollection(
        remoteChainletId,
        remoteCollection,
        gameNFT.address
      );

      // Attacker tries to fake a bridge message
      const fakeNFTId = 999;

      await expect(
        nftVault.connect(attacker).receiveCrossChainNFT(
          remoteChainletId,
          remoteCollection,
          fakeNFTId,
          attacker.address
        )
      ).to.be.reverted;

      // Proper bridging from trusted bridge
      await bridge.connect(admin).simulateIncomingNFT(
        nftVault.address,
        remoteChainletId,
        remoteCollection,
        100,
        user.address
      );

      // Verify cross-chain NFT registered
      expect(await nftVault.isCrossChainNFT(gameNFT.address,
100)).to.be.true;
    });
  });
});
```

4. MVP Deliverables

4.1 Smart Contract Deployment

- Mosaical protocol contracts deployed to Saga Protocol mainnet
- Integration with at least 10 GameFi projects across multiple chainlets

- Formal security audit completed with all critical findings addressed
- Cross-chain bridges set up with LayerZero integration
- AI price oracle integration with Chainlink data feeds

4.2 Backend & Frontend System

- React-based web application with GameFi-focused UI/UX
- Mobile-responsive design with wallet connection support
- NFT discovery and filtering system for available GameFi assets
- Real-time loan health monitoring dashboard
- DPO token trading interface with order book visualization
- Bridge interface for cross-chainlet asset management
- User profile section with loan history and yield statistics

4.3 AI/Service Infrastructure

- Deployed AI model for GameFi NFT price prediction
- Backend service for data collection and feature engineering
- Data pipeline for collection, indexing, and analysis of GameFi metrics
- Automated risk assessment service with alert system
- Real-time monitoring system for protocol health metrics
- Asset valuation API for external integrations

4.4 Documentation & Community Resources

- Technical documentation for smart contracts and architecture
- API documentation for developers
- Integration guides for GameFi projects
- User guides and tutorials
- Risk model documentation and parameters
- Community forum and social media presence

- Developer documentation for Mosaical SDK

5. SWOT Analysis & MoSCoW Prioritization

5.1 SWOT Analysis

Strengths:

- GameFi-specific NFT lending solution designed for Saga Protocol ecosystem
- Innovative DPO token mechanism for partial liquidation and yield distribution
- AI-driven price prediction for better risk assessment
- Advanced risk management system with tiered parameters
- Cross-chainlet compatibility providing unified lending across multiple games

Weaknesses:

- Reliance on accurate GameFi utilization data from chainlets
- Complexity of partial liquidation mechanism for new users
- Dependencies on third-party oracles and cross-chain messaging
- Initial liquidity requirements for DPO token markets
- Technical complexity of managing multiple GameFi asset types

Opportunities:

- GameFi sector rapid growth on Saga Protocol
- Limited competition in specialized GameFi NFT lending
- Potential for exclusive partnerships with major GameFi projects
- Expansion to additional Saga Protocol chainlets
- Integration with in-game marketplaces for seamless lending experience

Threats:

- Potential GameFi market volatility and project failures

- Security risks from cross-chain bridging
- Regulatory uncertainty in NFT lending
- Competing protocols entering the GameFi lending space
- Technical challenges with Saga Protocol's evolving architecture

5.2 MoSCoW Prioritization

Must Have:

- NFT vault with GameFi asset support for Saga Protocol chainlets
- Dynamic LTV based on risk tiers and NFT utility
- Basic DPO token mechanism for partial liquidation
- Oracle system with multi-source validation
- Cross-chainlet bridge for Saga Protocol assets
- Core frontend functionality for deposits, loans, and repayments
- Basic risk management system

Should Have:

- AI-powered NFT price prediction with confidence scores
- DPO token trading interface with order books
- Comprehensive risk dashboard for users
- GameFi utility score visualization and explanation
- Enhanced cross-chainlet discoverability
- Interest distribution system for DPO tokens
- Mobile-responsive UI

Could Have:

- Advanced analytics for GameFi asset performance
- Integration with multiple wallet providers

- Governance system for protocol parameters
- Yield farming programs for liquidity incentives
- SDK for GameFi developer integrations
- Social features for community engagement
- Notification system for loan health alerts

Won't Have (in MVP):

- Full decentralized governance system (will be phased in later)
- Custom wallet implementation
- Institutional lending features
- Margin trading with GameFi assets
- Self-custody solution for bridged assets
- Fiat on/off ramp integration
- Secondary market for loan positions

NEXUS WEB3 LABS

Dr. Eliza Nakamoto, Chief Blockchain Architect & Founder