SC6107: Blockchain Development Fundamentals (Part 2)

Development Project Documentation

MSc (Blockchain) Programme
Academic Year: 2025-26 | Trimester: 1

General Information

Project Overview

In the Development Project, you are expected to design, architect, and develop an advanced decentralized application (dApp) or blockchain infrastructure component that demonstrates mastery of blockchain development principles, smart contract composition patterns, and DeFi protocols. This project accounts for **40% of your total course grade**.

Team Composition

- Work in groups of **2 to 4 students**
- Single-member groups require explicit instructor approval
- Each member must contribute meaningfully to both implementation and presentation
- Individual contributions will be tracked via GitHub commits

Project Timeline

Weeks 1-6: Full development lifecycle

Week 1-2: Project selection, research, architecture design

Week 3-4: Core implementation and contract development

Week 5: Testing, security analysis, optimization

Week 6: Final integration, documentation, and presentation preparation

---

Development Resources

References

- **Ethereum Development**:
  - Hardhat/Foundry documentation
  - OpenZeppelin Contracts library
  - Ethereum Improvement Proposals (EIPs)
- **DeFi Patterns**:

  - Uniswap V2/V3 documentation
  - Compound Finance documentation
  - Flashbots documentation (MEV)
- **Security Resources**:
  - SWC Registry (Smart Contract Weaknesses)
  - Consensys Smart Contract Best Practices
  - Trail of Bits security guidelines

Support Channels

- **Discussion Board**: NTULearn course forum
- **Instructors**: Available after lectures

Project Options

Option 1: Flash Loan Arbitrage & MEV Platform

Project in a Nutshell

Build an advanced flash loan platform that enables users to execute complex arbitrage strategies across multiple DEXs, with MEV-aware transaction ordering and optimization. The platform should include a sophisticated routing engine, gas optimization, and profit simulation capabilities.

Background & Problem Statement

Flash loans represent one of the most innovative primitives in DeFi, enabling uncollateralized loans within a single transaction. However, executing profitable arbitrage requires: - Real-time price monitoring across multiple liquidity sources - Complex multi-hop routing through DEX aggregators - Gas optimization to ensure profitability after transaction costs - MEV protection to prevent front-running and sandwich attacks

Traditional arbitrage bots are vulnerable to MEV extractors who can front-run profitable transactions. A well-designed platform must incorporate MEV-awareness and implement strategies such as private transaction pools (e.g., Flashbots) or commit-reveal schemes.

Feature Requirements

Core Features:

1. **Flash Loan Integration** - Integrate with at least 2 flash loan providers (e.g., Aave, dYdX) - Support multi-asset flash loans - Implement atomic transaction execution with rollback on failure
2. **Multi-DEX Arbitrage Engine**
   - Support at least 3 different DEX protocols (e.g., Uniswap V2/V3, Sushiswap, Curve)
   - Implement optimal routing algorithm for multi-hop trades

- Calculate and display expected profit before execution

3. **MEV Protection**
   - Implement at least one MEV protection mechanism:
     - Private transaction submission (Flashbots)
     - Commit-reveal scheme
     - Time-locked transactions
4. **Gas Optimization**
   - Optimize contract bytecode for minimal gas consumption
   - Implement gas price prediction for transaction timing
   - Display gas cost vs. profit analysis

Advanced Features (Bonus):

- Cross-chain arbitrage using bridges

- Liquidation bot functionality for lending protocols

- Machine learning-based profit prediction

- Real-time monitoring dashboard with WebSocket updates

Technical Considerations

- How to handle slippage across multiple hops?
- How to ensure atomic execution of the entire arbitrage chain?
- How to optimize for MEV-Geth or other MEV-aware infrastructure?
- How to handle failed transactions and gas refunds?

Leading Projects for Reference

- Flashbots (https://docs.flashbots.net/)
- Aave Flash Loans (https://docs.aave.com/developers/guides/flash-loans)
- MEV-Inspect (https://github.com/flashbots/mev-inspect-py)

Option 2: Cross-Chain Bridge with Oracle-Based Security

Project in a Nutshell

Develop a trustless cross-chain bridge that enables asset transfers between Ethereum and another EVM-compatible chain (e.g., Polygon, Arbitrum, or BSC), using oracle networks for transaction verification and fraud proofs for security.

Background & Problem Statement

Blockchain bridges are critical infrastructure for a multi-chain ecosystem, but they represent one of the highest-risk components in DeFi, with billions lost to bridge hacks. Traditional bridge designs suffer from: - Centralized validator sets vulnerable to collusion -

Lack of fraud-proof mechanisms - Slow finality for secure transfers - Oracle manipulation risks

A secure bridge must implement robust security mechanisms including: - Decentralized oracle networks for cross-chain verification - Fraud-proof systems with challenge periods - Multi-signature validation with threshold requirements - Circuit breakers for abnormal activity detection

Feature Requirements

Core Features:

1. **Bridge Protocol** - Lock-and-mint mechanism for asset transfers - Support for ERC-20 token bridging - Burn-and-unlock for reverse transfers - Transaction proof verification
2. **Oracle Integration**
   - Integrate with Chainlink or similar oracle network
   - Multi-oracle validation (minimum 3 oracles)
   - Consensus mechanism for cross-chain transaction confirmation
   - Oracle reputation and slashing mechanism
3. **Security Features**
   - Challenge period for fraud proofs (configurable duration)
   - Multi-signature validation for large transfers
   - Rate limiting and daily transfer caps
   - Emergency pause functionality with timelock
4. **Monitoring & Analytics**
   - Transaction status tracking across chains
   - Bridge liquidity monitoring
   - Failed transaction handling and recovery

Advanced Features (Bonus): - Optimistic rollup-style fraud proofs - Zero-knowledge proofs for privacy-preserving transfers - Support for NFT bridging - Automated liquidity rebalancing

Technical Considerations

- How to handle chain reorganizations and finality?
- How to design the fraud-proof challenge mechanism?
- How to ensure oracle decentralization and prevent collusion?
- How to manage liquidity across both chains efficiently?

Leading Projects for Reference

- Hop Protocol (https://docs.hop.exchange/)
- Connext (https://docs.connext.network/)
- Chainlink CCIP (https://docs.chain.link/ccip)

Option 3: Upgradeable DeFi Protocol Suite

Project in a Nutshell

Build a composable DeFi protocol suite (lending pool + liquidity mining + governance) using upgradeable proxy patterns, allowing protocol evolution while maintaining state and user funds. Implement a complete governance system for upgrade proposals and execution.

Background & Problem Statement

Smart contracts are immutable by default, but DeFi protocols need to evolve to fix bugs, add features, and respond to market conditions. The challenge is implementing upgradeability without: - Creating centralization risks (admin keys) - Introducing new attack vectors - Losing existing state or user funds - Breaking composability with other protocols

Modern upgradeable patterns (EIP-1967, UUPS, Diamond) provide frameworks, but proper implementation requires: - Careful storage layout management - Governance-controlled upgrades with timelocks - Emergency pause mechanisms - Thorough testing of upgrade paths

Feature Requirements

Core Features:

1. **Upgradeable Architecture** - Implement UUPS (Universal Upgradeable Proxy Standard) or Diamond pattern - Maintain storage layout compatibility across upgrades - Support multiple independent modules - Version control and rollback capability
2. **Lending Pool Protocol**
   - Deposit and withdraw functionality for multiple assets
   - Interest rate model (utilization-based)
   - Collateralized borrowing with liquidation mechanism
   - Health factor calculation and monitoring
3. **Liquidity Mining**
   - Staking mechanism for LP tokens
   - Reward distribution based on time-weighted deposits
   - Multiple reward tokens support
   - Fair distribution algorithm
4. **Governance System**
   - Proposal creation and voting mechanism
   - Timelock for upgrade execution (minimum 24-hour delay)
   - Quorum requirements and voting power calculation
   - Emergency actions with multi-sig override

Advanced Features (Bonus):

- Gas-efficient snapshot-based voting

- Delegated voting system

- On-chain governance analytics

- Automated proposal execution

Technical Considerations

- How to prevent storage collisions during upgrades?
- How to handle initialization of new contract versions?
- How to design governance to prevent malicious upgrades?
- How to test upgrade scenarios comprehensively?

Leading Projects for Reference

- OpenZeppelin Upgradeable Contracts (https://docs.openzeppelin.com/upgrades-plugins/)
- Compound Governance (https://compound.finance/governance)
- Aave Protocol (https://docs.aave.com/developers/)

Option 4: On-Chain Verifiable Random Game Platform

Project in a Nutshell

Create a provably fair on-chain gaming platform that uses Chainlink VRF (Verifiable Random Function) or similar oracle solutions for generating random outcomes. Implement at least 2 different game types with betting mechanisms, reward pools, and anti-cheating measures.

Background & Problem Statement

On-chain gaming faces a fundamental challenge: generating randomness in a deterministic environment. Traditional approaches using block hashes are exploitable by miners. True randomness requires: - Unpredictable and unmanipulable random number generation - Verifiable proof that results weren't tampered with - Fair distribution of rewards - Protection against MEV exploitation

Chainlink VRF and similar solutions provide cryptographically secure randomness, but implementing them correctly requires understanding: - Callback patterns and gas limitations - Economic security models - Request-response timing - Failure handling and retry mechanisms

Feature Requirements

Core Features:

1. **Verifiable Randomness** - Integrate Chainlink VRF or similar oracle - Implement callback pattern for random number consumption - Handle VRF request failures and retries - Display proof of randomness to users
2. **Game Implementations** (choose at least 2):

- Lottery/Raffle system with time-based draws
- Dice game with multiplier betting
- Card game (e.g., simplified poker or blackjack)
- Prediction market with binary outcomes

3. **Betting & Treasury Management**
   - Support ETH and ERC-20 token betting
   - Pooled prize mechanism with house edge
   - Automatic payout system
   - Minimum/maximum bet limits
4. **Anti-Cheating & Fairness**
   - Commitment schemes for user actions
   - Time-locked reveals
   - Slashing for malicious behavior
   - Transparent outcome verification

Advanced Features (Bonus):

- Multi-player games with turn-based mechanics

- NFT-based game items or achievements

- Referral and reward system

- Integration with ENS for player identities

Technical Considerations

- How to handle VRF callback gas limitations?
- How to prevent MEV exploitation in betting games?
- How to design fair house edge and economic sustainability?
- How to manage edge cases (e.g., no winner in lottery)?

Leading Projects for Reference

- Chainlink VRF (https://docs.chain.link/vrf/v2/introduction)
- PoolTogether (https://docs.pooltogether.com/)
- Wolf Game (https://wolf.game/)

Option 5: Automated Market Maker with Novel Features

Project in a Nutshell

Design and implement an AMM (Automated Market Maker) with innovative features beyond standard constant product formula (x*y=k). Implement advanced features such as concentrated liquidity, dynamic fees, multiple asset pools, or novel curve designs.

Background & Problem Statement

While Uniswap's constant product formula revolutionized DEXs, it has limitations: - Capital inefficiency (liquidity spread across all price ranges) - Fixed fee structures regardless of market volatility - Impermanent loss for liquidity providers - Limited support for stablecoin or correlated asset pairs

Modern AMMs address these through:

- Concentrated liquidity (Uniswap V3)

- Dynamic fee adjustment based on volatility

- Curve-based designs for stablecoin swaps

- Virtual reserves and oracle integration

Feature Requirements

Core Features:

1. **AMM Implementation** - Implement one of the following curve designs:
   - Concentrated liquidity (Uni V3-style)
   - StableSwap curve (Curve.fi-style)
   - Hybrid curve with multiple regions
   - Novel curve design (must justify mathematically)
2. **Liquidity Management**
   - Add/remove liquidity functionality
   - LP token minting and burning
   - Position management (if using concentrated liquidity)
   - Fee accumulation and distribution
3. **Trading Features**
   - Swap functionality with price impact calculation
   - Slippage protection
   - Multi-hop routing for best price
   - Price oracle integration (TWAP)
4. **Analytics & Monitoring**
   - Real-time price chart
   - Liquidity depth visualization
   - Historical volume and fee data
   - Impermanent loss calculator for LPs

Advanced Features (Bonus):

- Dynamic fee adjustment based on volatility

- Just-in-time (JIT) liquidity protection

- Range orders (limit orders on AMM)

- Protocol-owned liquidity (POL) mechanisms

Technical Considerations

- How to optimize gas costs for complex mathematical operations?
- How to prevent sandwich attacks and MEV exploitation?
- How to design fee structures that balance LP and trader interests?
- How to handle edge cases (e.g., pool depletion)?

Leading Projects for Reference

- Uniswap V3 (https://docs.uniswap.org/contracts/v3/overview)
- Curve Finance (https://curve.readthedocs.io/)
- Balancer V2 (https://docs.balancer.fi/)

Option 6: Stablecoin Protocol with Algorithmic Stability

Project in a Nutshell

Develop a stablecoin protocol with a robust stability mechanism, collateral management system, and liquidation engine. The stablecoin should maintain its peg through algorithmic mechanisms, over-collateralization, or a hybrid approach.

Background & Problem Statement

Stablecoin designs have evolved from simple fiat-backed tokens to complex algorithmic systems:

- Fiat-backed (USDC, USDT): Centralized, censorship risk

- Crypto-collateralized (DAI): Decentralized but capital inefficient

- Algorithmic (failed examples like UST): Prone to death spirals

A robust stablecoin requires:

- Sufficient collateralization to withstand market volatility

- Efficient liquidation mechanism to maintain peg

- Incentive systems to encourage stability

- Oracle integration for accurate pricing

Feature Requirements

Core Features:

1. **Collateral Management**
- Support multiple collateral types (ETH, wrapped BTC, other tokens)

- Collateral ratio calculation and monitoring
- Vault/CDP (Collateralized Debt Position) creation
- Partial withdrawals and deposits

2. **Stability Mechanism**
   - Choose one approach:
     - Over-collateralization (MakerDAO-style)
     - Algorithmic supply adjustment
     - Hybrid model with collateral + algorithm
   - Peg maintenance mechanism (stability fees, savings rate)
3. **Liquidation System**
   - Automated liquidation when collateral ratio falls below threshold
   - Auction mechanism for liquidated collateral
   - Liquidation penalty to incentivize healthy positions
   - Debt socialization for underwater positions
4. **Oracle Integration**
   - Price feed integration (Chainlink or similar)
   - Multi-oracle validation
   - Circuit breaker for extreme price movements
   - TWAP for manipulation resistance

Advanced Features (Bonus):

- Flash loan protection for liquidations

- Governance system for parameter adjustment

- Savings rate for stablecoin holders

- Integration with lending protocols

Technical Considerations

- How to prevent death spiral scenarios?
- How to handle oracle failures or manipulation?
- How to design efficient liquidation auctions?
- How to ensure long-term protocol sustainability?

Leading Projects for Reference

- MakerDAO (https://docs.makerdao.com/)
- Liquity (https://docs.liquity.org/)
- Frax Finance (https://docs.frax.finance/)

Option 7: EVM Transaction Debugger & Analyzer

Project in a Nutshell

Build a sophisticated debugging and analysis tool for Ethereum transactions, providing detailed trace analysis, gas profiling, state diff visualization, and vulnerability detection. The tool should help developers understand complex contract interactions and optimize their code.

Background & Problem Statement

Debugging smart contracts is significantly more challenging than traditional software:

- Limited visibility into execution flow

- No traditional debugging tools (breakpoints, step-through)

- Gas cost optimization requires deep understanding of EVM opcodes

- Complex contract interactions are hard to trace

Developers need tools that can:

- Provide detailed execution traces

- Identify gas optimization opportunities

- Detect common vulnerabilities

- Visualize state changes across transactions

Feature Requirements

Core Features:

1. **Transaction Trace Analysis**
   - Parse and display detailed execution traces
   - Show all internal transactions and calls
   - Display opcode-level execution (optional)
   - Event log decoding and display
2. **Gas Profiling**
   - Gas consumption breakdown by function
   - Identify gas-intensive operations
   - Compare gas usage across similar functions
   - Suggest optimization opportunities
3. **State Diff Visualization**
   - Display all storage changes in transaction
   - Show before/after state for affected contracts
   - Visualize balance changes across addresses
   - Track token transfers (ERC-20/721/1155)

4. **Vulnerability Detection**
    - Static analysis for common vulnerabilities:
        - Reentrancy
        - Integer overflow/underflow
        - Unchecked external calls
        - Access control issues
    - Integration with existing tools (Slither, Mythril)

Advanced Features (Bonus):

- Real-time transaction monitoring and alerting

- Contract interaction graph visualization

- Historical trend analysis for gas optimization

- Integration with popular development frameworks

Technical Considerations

- How to efficiently parse and store large transaction traces?
- How to present complex data in user-friendly format?
- How to balance depth of analysis with performance?
- How to keep vulnerability detection up-to-date?

Leading Projects for Reference

- Tenderly (https://tenderly.co/)
- Etherscan Transaction Analyzer
- Foundry/Hardhat debugging tools

Option 8: Bring Your Own Project (BYOP)

Requirements for Custom Projects

⚠️ **MANDATORY APPROVAL REQUIRED**

You **MUST** obtain approval from the course instructor before proceeding with a custom project. Talk to your instructor to get approval by the end **of Week 2**.

Proposal Requirements

Please make sure you are clear on the followings before talking to the instructor:

1. **Project Title and Team Members**
2. **Background & Problem Statement**
    - What problem are you solving?
    - Why is this problem important?

   - What are current limitations of existing solutions?
3. **Technical Architecture**
   - Blockchain platform (Ethereum preferred)
   - Smart contract components
   - Off-chain components (if any)
   - Integration points with existing protocols
4. **Core Feature Requirements** (minimum viable product)
   - List 4-6 essential features
   - Indicate which course topics each feature relates to
   - Specify acceptance criteria for each feature
5. **Advanced Features** (optional bonus)
   - List 2-4 stretch goals
   - Indicate added complexity and value
6. **Evaluation Alignment**
   - Explain how your project demonstrates:
     - Technical depth
     - Originality
     - Practicality
     - Scope appropriate for 6 weeks

Project Idea Sources

1. DeFi Innovation

- Novel lending/borrowing mechanisms

- Advanced yield optimization strategies

- New collateral types or risk models

- DeFi composability patterns

2. Infrastructure & Tooling

- Development tools

- EVM analysis or optimization tools

- Multi-call batching solutions

3. Emerging Trends

- Appchain-specific applications

- MEV-aware applications

- On-chain AI/ML integration

- Account abstraction implementations

4. Research-Oriented Projects

- Novel cryptographic primitives

- Layer 2 scalability solutions

- Privacy-preserving protocols

- Cross-chain communication protocols

Recommended Resources

- DeFi Security Summit talks
- Ethereum Research forum
- Paradigm Research blog
- a16z Crypto research
- Recent EIPs (Ethereum Improvement Proposals)

Evaluation Note

Custom projects will be evaluated using the same rubric as provided options, with emphasis on:

- Innovation and originality

- Technical complexity comparable to provided options

- Clear demonstration of course concepts

- Practical utility and potential real-world application

Technical Requirements (All Projects)

Blockchain Platform

- **Primary**: Ethereum (Mainnet fork, Sepolia, or Goerli testnet)
- **Alternative**: Other EVM-compatible chains with instructor approval
- **Alternative**: Solana or other non-EVM platforms with instructor approval and justification

Smart Contract Development

- **Language**: Solidity 0.8.x (latest stable)
- **Development Framework**: Foundry or Hardhat
- **Testing Framework**: Foundry/Forge or Hardhat/Chai
- **Libraries**: OpenZeppelin Contracts 5.x

Testing Requirements

- **Minimum test coverage**: 80% line coverage
- **Required test types**:
  - Unit tests for all public/external functions
  - Integration tests for contract interactions
  - Invariant/fuzz testing for critical functions
  - Gas optimization tests

Security Requirements

- **Static Analysis**: Run Slither or similar tool, address critical findings
- **Common Vulnerabilities**: Demonstrate protection against:
  - Reentrancy attacks
  - Integer overflow/underflow
  - Front-running (where applicable)
  - Access control bypass
- **External Calls**: Proper checks-effects-interactions pattern
- **Emergency Controls**: Pause mechanism for critical functions (if applicable)

Gas Optimization

- Demonstrate gas optimization efforts:
  - Use of appropriate data types
  - Storage vs. memory optimization
  - Loop optimization
  - Batch operations where possible
- Document gas costs for main operations

Frontend Requirements

- **Framework**: React, Next.js, or similar modern framework
- **Web3 Library**: ethers.js v6 or viem
- **Wallet Connection**: Support MetaMask (minimum), WalletConnect (bonus)
- **UI/UX**: Clean, functional interface (styling framework of your choice)
  - Not required to be beautiful, but must be usable
  - Mobile-responsive (bonus)

Documentation Requirements

- **README.md**: Comprehensive project overview
- **Architecture documentation**: System design and component interaction
- **Contract documentation**: NatSpec comments for all public functions
- **Deployment guide**: Step-by-step instructions
- **User guide**: How to interact with the application

Submission Requirements

GitHub Repository

Repository Structure:

```
project-name/
├── README.md (comprehensive overview)
├── contracts/ (Solidity smart contracts)
│   ├── src/
│   └── test/
├── frontend/ (React/Next.js application)
├── docs/ (additional documentation)
│   ├── architecture.md
│   ├── security-analysis.md
│   └── gas-optimization.md
├── scripts/ (deployment scripts)
└── package.json
```

Critical Requirements:

1. **Individual Commits**: Each team member MUST commit using their own GitHub username
   - We track individual contributions through commit histor
   - Commits should be meaningful (not all code in one commit)
   - Use descriptive commit messages
2. **Commit Frequency**: Regular commits throughout the 6-week period
   - Distributed across multiple weeks
   - Demonstrates consistent contribution
3. **Branch Strategy** (recommended):
   - `main`: Production-ready code
   - `develop`: Integration branch
   - Feature branches: Individual feature development
4. **Documentation**:
   - Clear README with setup instructions
   - Architecture diagrams (use tools like Mermaid, Draw.io)
   - API documentation for smart contracts
   - Known limitations and future improvements

Presentation Requirements

Format: 5-minute presentation

Content Structure:

1. **Introduction (1 min)**
   - Problem statement
   - Solution overview

    - Team member roles
2. **Technical Architecture (1 min)**
    - System design diagram
    - Smart contract architecture
    - Key design decisions and trade-offs
3. **Live Demonstration (1 min)**
    - Core functionality walkthrough
    - User flow demonstration
    - Transaction on testnet (if feasible)
4. **Technical Deep Dive (1 min)**
    - Interesting technical challenge solved
    - Code snippet showing implementation
    - Gas optimization or security consideration
5. **Results & Reflection (1 min)**
    - What worked well
    - What you would do differently
    - Key learnings

Presentation Deliverables:

- Slide deck (PDF format, submit to NTULearn)

- Include: project demo link, GitHub repo link, deployed contract addresses

- Optional: Pre-recorded demo video (backup if live demo fails)

Evaluation Criteria for Presentation:

- Clarity of explanation (20%)

- Technical depth demonstrated (30%)

- Quality of demo (20%)

- Handling of Q&A (20%)

- Professional delivery (10%)

Evaluation Framework

Overall Grade Breakdown

- **Technical Implementation (50%)**
    - Smart contract quality and correctness
    - Security and gas optimization
    - Testing coverage and quality

  - Code organization and documentation
- **Technical Depth (15%)**
  - Use of advanced patterns and concepts
  - Integration of course topics
  - Innovation and complexity
- **Originality (10%)**
  - Novel features or approaches
  - Creative problem-solving
  - Differentiation from existing solutions
- **Practicality (10%)**
  - Real-world applicability
  - User experience considerations
  - Scalability and maintainability
- **UI/UX (10%)**
  - Frontend functionality
  - User flow design
  - Visual presentation
- **WOW Factor (5%)**
  - Exceeds expectations
  - Impressive features or execution
  - Presentation quality

Detailed Rubric

Technical Implementation (50%)

Criteria	Fail (0-40%)	Pass (41-74%)	High (75-100%)
Smart Contracts	Contracts have critical bugs, fail to compile, or don't implement core features. Little evidence of understanding Solidity.	Contracts work correctly for core features. Minor bugs present. Reasonable understanding of Solidity demonstrated. Could improve organization.	Contracts are robust, well-organized, and implement all core features correctly. Excellent Solidity practices. Professional-grade code quality.
Security	Critical vulnerabilities present (reentrancy, access control	Common vulnerabilities addressed. Basic security analysis performed. Some	Comprehensive security analysis. All major vulnerabilities addressed. Evidence

Criteria	Fail (0-40%)	Pass (41-74%)	High (75-100%)
	issues). No security analysis performed.	edge cases not handled.	of security-first thinking throughout.
Testing	<40% coverage, or tests don't run. Few meaningful tests.	60-80% coverage. Tests cover main functionality. Some edge cases missing.	>80% coverage. Comprehensive unit, integration, and fuzz tests. Edge cases well-tested.
Gas Optimization	No evidence of gas optimization. Inefficient patterns used throughout.	Some gas optimization attempts. Reasonable gas costs for main operations.	Systematic gas optimization. Efficient use of storage, memory. Gas costs documented and justified.

Technical Depth (15%)

Criteria	Fail (0-40%)	Pass (41-74%)	High (75-100%)
Complexity	Basic implementation only. Does not demonstrate mastery of course concepts.	Moderate complexity. Demonstrates understanding of some advanced concepts from course.	High complexity. Excellent demonstration of multiple advanced concepts (flash loans, oracles, MEV, etc.).
Integration	Isolated contracts with minimal integration.	Reasonable integration between components. Demonstrates contract composition.	Excellent integration showing deep understanding of DeFi composability and contract interactions.

Originality (10%)

Criteria	Fail (0-40%)	Pass (41-74%)	High (75-100%)
Innovation	Direct copy of existing project with minimal modifications.	Implements known patterns with some novel features or improvements.	Significant original contributions. Novel approach to solving problems.

Practicality (10%)

Criteria	Fail (0-40%)	Pass (41-74%)	High (75-100%)
Real-world Utility	Solution is not practical or doesn't solve stated problem.	Solution addresses the problem reasonably well. Some practical limitations.	Solution is highly practical, well-thought-out, and addresses real-world needs effectively.

UI/UX (10%)

Criteria	Fail (0-40%)	Pass (41-74%)	High (75-100%)
Functionality	UI is broken or unusable. Many features don't work.	UI is functional with minor issues. Main user flows work correctly.	UI is polished and fully functional. All features work smoothly. Professional quality.
Design	Confusing layout, poor information architecture.	Reasonable layout. Could improve user experience.	Excellent UX. Intuitive design. Good information architecture.

WOW Factor (5%)

Criteria	Description
Exceptional Achievement	Reserved for projects that significantly exceed expectations through innovative features, exceptional execution, impressive technical achievements, or outstanding presentation quality.

Peer Evaluation (Moderating Factor)

Purpose

Since the instructor cannot directly observe each member's contribution, peer evaluation ensures fair assessment of individual efforts within the team.

Process

- **Confidential**: No team member will know how others rated them
- **Mandatory**: You MUST submit peer evaluation to receive project marks
- **Fair Assessment**: Rate yourself and teammates objectively and fairly

Rating Scale

Score	Description
10-9	**Outstanding**: Demonstrated exceptional contributions and leadership. Went above and beyond. Coordinated major portions of work. Always available and reliable.
8-7	**Strong**: Exhibited equal and appropriate contributions. Took fair share of work. Coordinated own parts effectively. Reliable team member.
6-4	**Adequate**: Made sufficient contributions but could have done more. Some inconsistency in engagement. Little coordination effort.
3-1	**Insufficient**: Did not contribute much effort or work. Often unavailable. Minimal engagement with team.
0	**No Contribution**: Made no effort to contribute. Did not participate in teamwork.

Grade Calculation

Your individual grade is calculated from the team's base grade using the following formula:

If Mean Rating ≥ 8: - Individual Grade = 100% of Team Grade

If 2 ≤ Mean Rating < 8: - Individual Grade = (20% + Mean Rating × 10%) of Team Grade

If Mean Rating < 2: - Case investigated by instructor - May result in 0% of Team Grade

Example Scenarios

Assume team receives **75/100** base grade:

Mean Rating	Calculation	Individual Grade
9.2	100% × 75	75/100
7.5	(20% + 75%) × 75	71.25/100
5.0	(20% + 50%) × 75	52.5/100
1.5	Investigation	TBD (possibly 0)

Peer Evaluation Submission

- **Platform**: NTULearn assignment submission
- **Deadline**: End of Week 6 (with project submission)
- **Format**: Rate each team member (including yourself) using 0-10 scale
- **Required**: Brief justification for each rating (2-3 sentences)

Important Notes

- Instructor reserves the right to adjust ratings in cases of bias, discrimination, or malicious intent
- GitHub commit history will be reviewed alongside peer evaluations
- Significant discrepancies between commit history and peer ratings may trigger investigation
- Be honest but fair - consider the impact on your teammates' grades

---

Important Dates & Milestones

Week	Milestone	Deliverable
Week 1	Project Selection & Team Formation	Team registration, BYOP proposal (if applicable)
Week 2	Architecture Design	Architecture document submission (recommended)
Week 3-4	Core Development	Mid-project check-in (optional)
Week 5	Testing & Optimization	Test coverage report (recommended)
Week 6	Final Integration & Submission	GitHub repository, presentation slides, peer evaluation

Submission Deadlines

- **Code & Documentation**: End of Week 6 (Friday 11:59 PM SGT)
- **Presentation Slides**: End of Week 6 (Friday 11:59 PM SGT)
- **Peer Evaluation**: End of Week 6 (Friday 11:59 PM SGT)
- **Presentations**: End of Week 6 (Feb. 12, 2026)

Tips for Success

Getting Started

1. **Week 1**: Carefully review all project options. Discuss with your team. Consider your team's strengths.
2. **Research**: Look at the leading projects mentioned. Understand their architecture.
3. **Start Simple**: Build core features first. Add complexity iteratively.
4. **Regular Commits**: Commit early, commit often. Demonstrates consistent work.

Development Best Practices

1. **Test-Driven Development**: Write tests before implementation (or alongside)
2. **Code Reviews**: Review each other's pull requests
3. **Documentation**: Document as you go, not at the end

4. **Gas Profiling**: Profile gas costs early and often
5. **Security First**: Think about security implications of every function

Common Pitfalls to Avoid

- ❌ Leaving everything to the last week
- ❌ Not testing edge cases
- ❌ Ignoring gas optimization until the end
- ❌ Poor communication within team
- ❌ Not using version control properly
- ❌ Underestimating time for debugging and testing
- ❌ Not seeking help when stuck

Getting Help

- **Discussion Board**: Post questions (others likely have same questions)
- **Office Hours**: Book time with instructors/TAs
- **Team Communication**: Use Discord/Telegram for team coordination
- **Documentation**: Read the docs for libraries/frameworks you're using
- **Community**: Ethereum StackExchange, GitHub issues of projects you're referencing

---

Academic Integrity

Acceptable Practices

✅ Using open-source libraries (OpenZeppelin, etc.)
✅ Referencing documentation and tutorials
✅ Discussing concepts and approaches with other teams
✅ Using code snippets from official documentation
✅ Using AI tools (ChatGPT, GitHub Copilot) for code suggestions

Unacceptable Practices

❌ Copying code from other teams
❌ Submitting code you didn't write or understand
❌ Having someone outside your team write significant portions
❌ Using complete contract implementations without attribution
❌ Plagiarizing documentation or presentations

Proper Attribution

- **Code**: Comment source of any substantial borrowed code
- **Concepts**: Cite papers, blog posts, or projects that inspired your design
- **Libraries**: Document all dependencies in README

AI Tool Usage

AI tools are permitted but:

- You must understand all code you submit

- You are responsible for security and correctness

- Document when AI tools were used significantly

- AI-generated code still requires proper testing

Violation of academic integrity will result in serious consequences, including possible failure of the course.

Frequently Asked Questions

Q: Can we switch project options after starting?
A: Yes, but only in Week 1-2. After Week 2, switching requires instructor approval and may affect your grade.

Q: What if our team cannot agree on a project?
A: Escalate to instructor immediately. Do not delay beyond Week 2.

Q: Can we use languages other than Solidity (e.g., Vyper)?
A: Requires explicit instructor approval. Must justify why alternative is better for your project.

Q: What if a team member is not contributing?
A: Document the situation. Use peer evaluation to reflect this. Inform instructor if situation is severe.

Q: Can we deploy to Mainnet?
A: Not required and not recommended. Testnet deployment is sufficient. Mainnet deployment is your own risk.

Q: How much help can we get from instructors?
A: We can guide you, answer questions, and help debug. We will not write significant code for you.

Q: What if we can't complete all features?
A: Focus on core features first. A working MVP with good testing and documentation is better than incomplete advanced features.

Q: Can we continue working on this project after the course?
A: Yes! Many successful projects started as course projects. We encourage you to continue development.

Conclusion

This development project is your opportunity to demonstrate mastery of blockchain development principles and to build something you can be proud of. Approach it with:

- **Ambition**: Push yourself to learn new concepts

- **Rigor**: Test thoroughly, optimize diligently

- **Creativity**: Add your own unique touches

- **Professionalism**: Treat this as a real-world project

The skills you develop here will serve you well in your blockchain career. Take this seriously, work collaboratively, and build something impressive.

Good luck!

Questions? Contact instructors via:

- NTULearn Discussion Board

- After lectures