Stony Brook University

Department of Computer Science

CSE 416: Software Engineering

Course Information & Project Guide

Instructor: Mohammad Javad Amiri

Fall 2026

Class: Tuesday & Thursday, 3:30–4:50 PM, Harriman Hall 116
Office Hours: Tuesday, 12:00–2:00 PM, NCS 368

Contents

1 Course Overview

This course introduces the basic concepts and modern tools and techniques of software engineering, and prepares you for a software engineer job at modern tech companies, from FAANG to early-stage startups, in the age of AI. It emphasizes the development of reliable and maintainable software via system requirements and specifications, software methodologies including analysis, design, implementation, integration, and testing, software project management, life-cycle documentation, software maintenance, and consideration of human factor issues. We build these skills mainly through practice: working in a team on a semester-long project of your own choosing in the blockchain domain, carried through its complete lifecycle; analysis, design, implementation, testing, improving, and release, checked at six milestones along the way. Alongside the project, lectures and student-led presentations cover the practices that matter most today, including using AI coding tools responsibly, so you leave this course able to build real software and explain your decisions to a team.

1.1 Getting Started

Two early deadlines are not tied to a milestone grade, but everything else in the semester depends on them:

- **Form your team by 08/30.** Groups must be finalized by the end of the first week of class. It is your responsibility to either form a group or join an existing one.
- **Lock in your project topic and presentation slot by 09/06.** Your group's project topic (Section 8) and which of the ten presentation slots (Section 7) your group will present in should both be finalized by the end of the second week of class. The project and presentation topics will be booked on a first-come, first-served basis. We will provide instructions on creating and sharing the project GitHub repository with us later.

1.2 Learning Outcomes

By the end of this course, you should be able to:

- Elicit, write, and evaluate software requirements and user stories, and translate them into a system design using appropriate modeling notation (e.g., UML).
- Choose and apply a software development methodology (e.g., Agile, Scrum, Kanban) appropriate to a team and project's constraints.
- Design a software architecture, justify its trade-offs, and apply relevant design patterns.
- Implement, test, debug, and iteratively improve a non-trivial software system as part of a team, using modern engineering practices such as version control, code review, and CI/CD.
- Use AI coding assistants (autocomplete and agentic tools) productively and responsibly, understanding both their capabilities and their limitations.
- Communicate technical decisions clearly in both written documentation and oral presentations to technical and non-technical audiences.
- Recognize the security, maintenance, project-management, and ethical considerations that shape real-world software engineering, including in decentralized/blockchain systems.

2 Time and Location

- **Class:** Tuesday and Thursday, 3:30–4:50 PM, Harriman Hall 116.
- **Office Hours:** Tuesday, 12:00–2:00 PM, NCS 368.

3 Requirements

3.1 Prerequisites

For CSE majors, completion of CSE 316 is required to enroll in this course, and you are expected to have senior standing (U4 in Stony Brook's academic guidelines); informally, to be almost ready to become a software engineer. CSE 305, CSE 333, CSE 336, CSE 337, and CSE 356 are not required but provide useful background; depending on how many of these you've completed, you may need to spend extra time on the course project picking up the related database technologies, user interface principles, and programming languages.

3.2 Expected Background

You should already be comfortable with different types of programming languages, for example, a scripting language such as JavaScript, and a strongly typed language with garbage collection (Java) or without (C/C++). This course expects you to be able to pick up new languages of these types and complete simple tasks using online resources without extra tutoring, which is a common expectation for working software engineers. No specific systems background is required; the project may look like a "systems" project to some, but we don't assume you already have that knowledge, nor that you'll learn it entirely on your own; we'll cover the related concepts in lecture.

4 Evaluation

Your final grade is based on two components: the course project, delivered across six milestones, and a single group class presentation. We will have no exams. At every project milestone, each of you will also assess your teammates.

Component	Description	Deadline	Weight
Milestone 1	Proposal and Requirements	09/19	10%
Milestone 2	Design and Setup	10/03	10%
Milestone 3	MVP	10/24	10%
Milestone 4	Feature-Complete	11/07	10%
Milestone 5	Testing and Integration	11/21	10%
Milestone 6	Final Product	12/05	30%
Course Project Total			**80%**
Class Presentation	Delivered by the group (see §7)	—	20%
Course Total			**100%**

4.1 The Six Project Milestones

The course project is a semester-long, team-built software system in the blockchain domain (see Section 8 for project ideas), carried through its complete lifecycle and checked at six milestones roughly every two weeks.

Every milestone has two parts:

1. **A 10-minute in-class presentation**, given during the milestone review session(s) shown in the schedule (Section 5). Every group must cover:
   - What has been completed during this phase.
   - What design choices were made, and the reasoning behind each choice.
   - How AI tools were used during this phase of the work.
2. **Artifacts submitted by the end of that week**, due by the deadline listed in the Evaluation table above. The specific artifacts expected for each milestone are listed below. For each milestone, a report template has been provided.

Milestone 1: Proposal and Requirements
Presentation: (a) your problem statement and target users,(b) why the problem is a good fit for blockchain, (c) why this problem necessitates a semester-long project and cannot be effectively tackled using AI tools, (d) what existing systems tackle the same or similar issues, and what are their limitations, (e) the initial scope of the system, and (f) how your team divided roles and responsibilities.
Artifacts: a written project proposal (problem statement, goals, target users); a requirements list covering both functional and non-functional requirements; an initial set of user stories; a statement of team roles; and a rough architecture sketch.

Milestone 2: Design and Setup
Presentation: your system architecture and the reasoning behind it, the tech stack you chose and why, and a demo of whatever is running so far.
Artifacts: an architecture/design document (including relevant UML diagrams); a written justification of your tech stack decisions; a project repository with a CI skeleton in place; and an initial, minimal working prototype.

Milestone 3: MVP
Presentation: a live demo of your minimum viable product, the design choices made while implementing it, and your testing approach so far.
Artifacts: a working MVP implementing your core feature end to end; updated design documentation reflecting any changes since Milestone 2; and an initial test suite or documented test plan.

Milestone 4: Feature-Complete
Presentation: a demo covering your full feature set, the trade-offs made while completing the remaining features, and your plan for testing and integration.
Artifacts: a complete implementation of all features planned for the project; a full, documented test plan; and updated documentation reflecting the final design.

Milestone 5: Testing and Integration
Presentation: your integration testing results, the bugs you found and how you fixed them, and your deployment plan.
Artifacts: integration test results; a deployed version of your system (e.g., on a public testnet, where applicable); and a draft of your final documentation (user guide and developer guide).

Milestone 6: Final Product
Presentation: your final demo, plus a project retrospective covering what worked, what didn't, and how AI tools were used across the whole project.
Artifacts: the final, working product; a finalized repository and documentation; and a final written report including your project retrospective.

5 Course Schedule

The schedule below lists every class session for the semester: instructor-led lectures, the two-session review windows for each project milestone, and the ten student-group presentations. Two sessions of the lecture (09/01 and 09/03) are held online via Zoom; a Zoom link will be shared in advance. The schedule is subject to change; announcements in class and on the course website take precedence over this document.

Date	Session	Topic
08/25	Lecture	Course Introduction
08/27	Lecture	Blockchain Fundamentals
09/01	Lecture (online)	Software Engineering in the Age of AI
09/03	Lecture (online)	Software Development Methodologies
09/08	Lecture	Software Modeling and UML Diagrams
09/10	Lecture	Requirements Engineering and User Stories
09/15	Milestone 1 Review	Project Proposal and Requirements
09/17	Milestone 1 Review	Project Proposal and Requirements
09/22	Presentation 1	Project Management Tools and Practices
09/24	Presentation 2	Software Architecture and Design Patterns
09/29	Milestone 2 Review	Design and Setup
10/01	Milestone 2 Review	Design and Setup
10/06	Presentation 3	Prompt Engineering and AI-Assisted Development
10/08	Presentation 4	API Design and Documentation
10/13	—	*No Class (Fall Break)*
10/15	Presentation 5	CI/CD and DevOps
10/20	Milestone 3 Review	MVP
10/22	Milestone 3 Review	MVP
10/27	Presentation 6	Software Testing
10/29	Presentation 7	Debugging and Observability
11/03	Milestone 4 Review	Feature-Complete
11/05	Milestone 4 Review	Feature-Complete
11/10	Presentation 8	Software Security and Secure Coding Practices
11/12	Presentation 9	Software Maintenance, Technical Debt, and Refactoring
11/17	Milestone 5 Review	Testing and Integration
11/19	Milestone 5 Review	Testing and Integration
11/24	Presentation 10	Responsible and Ethical Tech

Date	Session	Topic
11/26	—	*No Class (Thanksgiving)*
12/01	Milestone 6	Final Demo
12/03	Milestone 6	Final Demo

5.1 Lecture Topics at a Glance

- **Course Introduction:** syllabus walkthrough, team formation, and an overview of the semester project.
- **Blockchain Fundamentals:** the core concepts (distributed ledgers, consensus, smart contracts) needed to scope a blockchain project.
- **Software Engineering in the Age of AI:** how AI coding assistants are changing the software development lifecycle, grounded in recent research on how professional developers actually use AI at work.
- **Software Development Methodologies:** Waterfall, Agile/Scrum, Kanban, and how to choose a process that fits a small team on a fixed timeline.
- **Software Modeling and UML Diagrams:** use case, class, and sequence diagrams as tools for designing a system before you build it.
- **Requirements Engineering and User Stories:** eliciting requirements, writing testable user stories, and defining acceptance criteria.

6 Course Policies

6.1 Use of AI Tools

AI coding assistants (autocomplete tools such as GitHub Copilot, and agentic tools such as Claude Code or Cursor Agent) are permitted and encouraged throughout this course, including on the project. Responsible use is part of what this course teaches, and you will be expected to follow these ground rules on the project and any individual work:

- **Understand before you ship.** Never commit code you cannot explain to a teammate.
- **Write your own tests.** Do not outsource verification to the same tool that wrote the code.
- **Disclose AI usage.** Note where AI tools contributed, as you would cite any other source.
- **Keep architecture decisions human.** Use AI for implementation, not for deciding what to build or why.

Using AI tools does not relieve you of responsibility for the correctness, security, and quality of the work you submit, nor does it change the expectations of academic integrity described below.

6.2 Student Accessibility Support Center Statement

If you have a physical, psychological, medical, or learning disability that may impact your course work, please contact the Student Accessibility Support Center, Stony Brook Union Suite 107, (631) 632-6748, or at `sasc@stonybrook.edu`. They will determine with you what accommodations are necessary and appropriate. All information and documentation is confidential.

6.3 Academic Integrity Statement

Each student must pursue his or her academic goals honestly and be personally accountable for all submitted work. Representing another person's work as your own is always wrong. Faculty is required to report any suspected instances of academic dishonesty to the Academic Judiciary. For more comprehensive information on academic integrity, including categories of academic dishonesty, please refer to the academic judiciary website at `http://www.stonybrook.edu/commcms/academic_integrity/index.html`.

6.4 Critical Incident Management

Stony Brook University expects students to respect the rights, privileges, and property of other people. Faculty are required to report to the Office of Student Conduct and Community Standards any disruptive behavior that interrupts their ability to teach, compromises the safety of the learning environment, or inhibits students' ability to learn. Further information about most academic matters can be found in the Undergraduate Bulletin, the Undergraduate Class Schedule, and the Faculty-Employee Handbook.

7 Student Presentation Topics

Each of the ten project groups delivers one class presentation on a topic below, in the order shown in the schedule (Section 5). Each presentation should combine the general software-engineering concept with a concrete tie-in to the group's own project, per the guidance in each subsection. The presentation slides should be submitted to the course instructor at least 72 hours prior to the scheduled presentation time. The instructor may provide feedback.

Every presentation must:

- **Include all group members.** Every member of the group is expected to participate in delivering the presentation, not just prepare the slides.
- **Include a hands-on component.** A live demo, a walkthrough of real code or configuration, or a short in-class exercise for the rest of the class; not slides alone.
- **Compare tools, methods, or techniques.** Presenters should introduce the different tools, methods, or techniques relevant to the topic, compare them against each other, and recommend which ones are best suited to a project like theirs (and why).

7.1 Presentation 1: Project Management Tools & Practices [09/22]

Goal: Coordinating a small team against a fixed set of deadlines

This talk is the applied, tool-focused counterpart to the course lecture on development methodologies: rather than comparing Scrum and Kanban in the abstract, it should cover how a small team actually runs a project day to day using tools like Jira, Trello, Linear, or GitHub Projects. Useful topics include maintaining a backlog, running a lightweight planning or stand-up routine, and estimating effort using techniques like story points or simple time-boxing.

For a small team working toward six fixed milestones, the practical challenge is less about picking the "correct" methodology and more about keeping the whole team aligned on who is doing what, noticing when a task is falling behind early enough to react, and avoiding the common student-project failure mode where most of the work happens in the final 48 hours before a deadline.

The group can show their project board or backlog as it exists today, explain how they divide and track work

among group members, and share one thing about their process that has actually worked and one that has not. Concrete, self-critical examples will be far more useful to their classmates than a generic overview of tools. If not possible, showing examples from practical projects would be fine.

7.2 Presentation 2: Software Architecture & Design Patterns [09/24]

Goal: Structuring a system so it can grow without collapsing under its own weight

Software architecture is the set of high-level decisions about how a system is structured: layered vs. microservices vs. event-driven styles, how components communicate, and where state lives. Design patterns (creational, structural, and behavioral, in the classic Gang-of-Four sense) are reusable solutions to recurring design problems within that structure, such as factories, observers, and adapters. The goal of this presentation is to explain both levels: the big-picture architecture of a system, and the smaller, recurring patterns used to implement its parts cleanly.

Blockchain applications add a specific architectural wrinkle worth highlighting: a split between on-chain logic (smart contracts, which are expensive to change once deployed) and off-chain logic (the application layer, which is cheap to iterate on). Patterns like the proxy pattern for upgradeable contracts, or the factory pattern for deploying many similar contract instances, come up constantly in this domain and are worth walking through with a concrete example.

The group should present the actual architecture diagram of their own project, explain why they split responsibilities the way they did, and name at least one design pattern they used or considered and the trade-off involved in that choice. This ties directly into their Milestone 2 design deliverable, so it doubles as a chance to pressure-test their own decisions in front of the class.

7.3 Presentation 3: Prompt Engineering & AI-Assisted Development [10/06]

Goal: Writing effective instructions for AI coding tools, and using them responsibly

Prompt engineering is the practice of writing clear, well-scoped instructions that get useful output from an AI coding assistant, whether that is an inline autocomplete tool like GitHub Copilot or an agentic tool like Claude Code or Cursor Agent. Good prompts specify the goal, the relevant context (existing code, constraints, edge cases), and the expected output format; vague prompts tend to produce vague or subtly wrong code. Techniques worth covering include giving the model concrete examples, breaking a large task into smaller verifiable steps, and iterating on a first draft rather than expecting a perfect result on the first try.

This matters because AI-assisted development is now a normal part of professional software engineering and the gap between a team that uses these tools well and one that uses them carelessly shows up directly in code quality and velocity. Poor prompting leads to hallucinated APIs, insecure defaults, and code nobody on the team actually understands, which becomes a liability during testing, debugging, and code review later in the semester.

For the presentation, the group should go beyond definitions and actually demonstrate prompting on a real piece of their project: show a weak prompt and the problems it causes, then a stronger version of the same request and why it works better. The talk should end with a short, concrete policy for how their own team plans to use AI tools for the rest of the semester, including how they will review and test anything AI-generated before it is merged.

7.4 Presentation 4: API Design & Documentation [10/08]

Goal: Designing interfaces other people and other systems can actually use

An API is a contract: it defines what a caller can ask a system to do and what it can expect back, independent of how the system is implemented internally. This talk should cover the major styles (REST, GraphQL, RPC), core design principles (consistent resource naming, sensible error codes, versioning so you can change an API without breaking existing callers, idempotency for operations that might be retried), and documentation tooling such as the OpenAPI/Swagger specification and Postman collections.

In a blockchain project, the "API" often has two layers worth discussing: the smart contract's own interface (its ABI, and the functions and events it exposes on-chain) and a conventional REST or GraphQL layer that wraps those on-chain calls for a web or mobile frontend. Good documentation matters especially here, since other developers, auditors, or even automated tools may need to interact with a contract's interface without reading its full source code.

The presenting group should walk through the actual interface of their own project, whether that is a contract ABI, a REST endpoint list, or both, and show what their documentation looks like today. A short discussion of how they would version or change that interface later in the project, without breaking whatever already depends on it, makes a strong closing point.

7.5 Presentation 5: CI/CD & DevOps [10/15]

Goal: Automating the path from a code change to a running system

Continuous integration and continuous deployment (CI/CD) automate the steps between committing code and having it running somewhere real: building the project, running the test suite, and deploying the result, usually via a pipeline tool such as GitHub Actions, GitLab CI, or Jenkins. This talk should explain what a pipeline actually does stage by stage, and the broader DevOps idea that infrastructure and deployment steps should themselves be defined as code rather than done by hand.

Deployment in a blockchain context has its own checklist worth covering: choosing between a local network, a public testnet, and mainnet; scripting contract deployment so it is repeatable; and verifying deployed contract source code on a block explorer so others can inspect what was actually deployed. These are meaningfully different steps from deploying a typical web service, even though the underlying CI/CD philosophy is the same.

The group should show their actual pipeline configuration (even a simple one), explain what is automated today versus what is still done by hand, and be honest about what breaks or gets skipped under time pressure. A short "what we would automate next" closing point works well here.

7.6 Presentation 6: Software Testing [10/27]

Goal: Building confidence that the system works, and keeps working

This presentation should cover the testing pyramid (unit tests for individual functions, integration tests for components working together, and end-to-end tests for full user flows), the distinction between functional and regression testing, and the basic idea of test-driven development. The emphasis should be practical: what makes a test useful versus a test that just inflates a coverage number, and how a small, fast test suite that runs on every change is worth more than a large one nobody runs.

Testing a blockchain application has some genuinely different challenges worth calling out: smart contracts typically run against local test networks (using tools like Hardhat or Anvil) rather than a live chain, transactions cost gas even in test scenarios, and once deployed, contract logic cannot simply be patched the way a web server can. These constraints push teams toward heavier upfront testing of contract logic than a typical web application would need.

The group should demonstrate a real test suite from their own project, not a toy example, including at least one test that caught a real bug, and report a coverage or pass-rate number if they track one. If their project has a smart contract component, they should specifically address how they test it differently from their off-chain code.

7.7 Presentation 7: Debugging & Observability [10/29]

Goal: Finding out what actually went wrong, and building systems that tell you

Debugging is the skill of narrowing down the cause of unexpected behavior, using techniques from simple print statements and breakpoints to systematic bisection of when a bug was introduced. Observability is the related but broader practice of building a system so that its internal behavior is visible from the outside, typically through the three pillars of logs, metrics, and traces. This talk should connect the two: good observability makes debugging dramatically faster because you are not guessing blind.

Blockchain systems complicate both practices in a specific way: transactions are immutable once confirmed, so you cannot "just fix it in production," and visibility into what actually happened on-chain usually comes from reading event logs or using a block explorer rather than attaching a debugger to a running process. Local simulation environments exist precisely to let developers reproduce and step through failures before anything touches a real network.

The group should walk through at least one real bug they hit in their project, the tools and steps they used to track it down, and whatever logging or monitoring they have put in place since. If they have not yet built meaningful observability into their system, this is a good place to say so honestly and propose what they would add before the next milestone.

7.8 Presentation 8: Software Security & Secure Coding Practices [11/10]

Goal: Writing code that holds up against someone actively trying to break it

Secure coding starts from a small set of durable principles: validate and sanitize all input, follow the principle of least privilege, never trust data from outside the system, and keep secrets like API keys and private keys out of source code. The OWASP Top Ten is a useful reference point for the most common vulnerability classes in conventional web applications, such as injection attacks and broken access control, and this talk should ground itself in a few concrete examples rather than a long abstract list.

Blockchain and smart contract security introduces a distinct set of vulnerability classes that deserve specific attention: reentrancy attacks, integer overflow and underflow, access control mistakes in contract functions, and manipulation of external data feeds (oracles). Because deployed contract code is very difficult to change, the cost of a missed vulnerability is often much higher than in conventional software, which is worth emphasizing.

The presenting group should pick one or two vulnerability classes most relevant to their own project, show a minimal example of the flaw, and explain the specific mitigation they applied or plan to apply. A short, practical checklist the rest of the class could reuse for their own project reviews makes a strong, concrete takeaway.

7.9 Presentation 9: Software Maintenance, Technical Debt & Refactoring [11/12]

Goal: Keeping a codebase workable as it grows past its first working version

Software maintenance is commonly split into four types: corrective (fixing bugs), adaptive (adjusting to a changed environment or dependency), perfective (improving something that already works), and preventive

(reducing future risk before it becomes a problem). Technical debt is the accumulated cost of shortcuts taken to move faster now at the expense of harder changes later; refactoring is the disciplined practice of improving a codebase's internal structure without changing its external behavior.

Technical debt tends to accumulate especially fast in short, high-velocity projects like this one, and AI-assisted code generation can make it worse if generated code is accepted without review: duplicated logic, inconsistent patterns, and code nobody fully understands all pile up quickly under deadline pressure. Static analysis tools and linters are useful for catching some of this automatically, but they do not replace a team actually noticing and prioritizing what to clean up.

The group should identify one or two real examples of technical debt in their own codebase, walk through a before-and-after refactor if they have done one, and describe how they intend to manage debt for the remainder of the semester given their fixed milestone deadlines.

7.10 Presentation 10: Responsible & Ethical Tech [11/24]

Goal: The obligations that come with building software that touches real people and real money

AI-assisted coding raises its own set of ethical questions worth addressing directly: bias inherited from training data, the risk of skill atrophy from over-relying on generated code, unresolved questions about the intellectual property status of AI-generated snippets, and the basic question of who is accountable when AI-assisted code causes harm. These are live, unsettled debates in the industry, not settled facts, and the presentation should reflect that honestly rather than presenting one side as obviously correct.

Blockchain technology carries its own distinct ethical tensions: the trade-off between decentralization and accountability when something goes wrong, the environmental footprint of proof-of-work networks compared to proof-of-stake alternatives, the fact that a smart contract bug or exploit is often irreversible once deployed, and the gap between blockchain's promise of financial inclusion and its real-world association with speculation and scams. A good presentation names these tensions rather than picking a side for the whole class.

The talk should also connect to broader responsible-technology practice: data privacy and user consent, algorithmic fairness, open-source licensing and attribution, and professional codes of conduct such as the ACM or IEEE codes of ethics. The group should close by applying at least one of these frameworks to their own project specifically: what is the most significant ethical consideration in what they are building, and how are they addressing it?

8 Blockchain Project Ideas

Groups are highly encouraged to propose their own project in the blockchain domain. We also provide 20 options below, spanning core protocol work (consensus, cross-chain, storage, compute) and applications built on top (identity, finance, governance, marketplaces), several of which integrate AI or a conventional database as a first-class component. Each project should be scoped to require a full semester of work even with AI coding assistance; the description, challenges, and stack notes below are a starting point for early team discussions and for scoping Milestone 1, not a fixed spec. Groups should adapt scope, formats, or mechanisms as needed and justify those choices in their proposal.

8.1 Zero-Knowledge Voting System

An anonymous but verifiable voting platform where zero-knowledge proofs let a voter prove their eligibility without revealing their identity, while the final tally remains publicly checkable by anyone.

Key Challenges:

- Integrating a ZK proving library (e.g., circom/snarkjs) and designing an eligibility circuit that is actually correct
- Preventing double-voting without ever linking a proof back to a specific identity
- Verifying proofs on-chain within reasonable gas and cost limits
- Building a registration and voting flow that a non-technical voter can actually use

Suggested Building Blocks: Solidity + Hardhat/Foundry; circom/snarkjs or a higher-level DSL (Noir); a Merkle-tree eligibility registry; a lightweight web frontend for proof generation in-browser.

Hint: correctly specifying and debugging a ZK circuit (constraints, trusted setup, edge cases in the eligibility logic) is slow, unglamorous work that AI tools routinely get subtly wrong; a working, exploit-free circuit takes iteration a single prompt cannot shortcut.

8.2 Escrow & Dispute Resolution Marketplace

A freelance or goods marketplace where contract-held funds are released on milestone completion, with disagreements resolved by a randomly selected jury of token-holders instead of a platform administrator.

Key Challenges:

- Designing jury selection and voting that is incentive-compatible (jurors are rewarded for voting honestly)
- Handling partial disputes, where some milestones are agreed and others are not
- Preventing collusion between a juror and one of the disputing parties
- Covering edge cases: non-responsive parties, missed deadlines, and appeals

Suggested Building Blocks: Solidity escrow contracts with commit-reveal jury voting; a relational database (Postgres) for off-chain marketplace listings/messaging; a job/queue system for deadline-triggered state transitions.

Hint: the hard part is the game-theoretic mechanism design (juror incentives, collusion resistance, appeals) plus the marketplace product around it; AI can draft a naive escrow contract in minutes, but a mechanism that survives adversarial testing requires a full semester of modeling, simulation, and iteration.

8.3 Supply Chain Provenance Tracker

A system that records a product's chain of custody on-chain from origin to consumer, paired with a scanning app that shows the verified history behind a QR code.

Key Challenges:

- Establishing trust for off-chain events (a shipment scan, a quality check) entering the chain
- Supporting multiple, sometimes non-cooperative participants writing to one shared history
- Detecting or preventing conflicting and duplicate records from different parties

- Designing a low-friction, scannable verification experience for end consumers

Suggested Building Blocks: permissioned or public chain for custody events; a mobile-friendly QR scanning frontend; a database mirroring on-chain events for fast queries; role-based access control per participant type (producer, shipper, retailer).

Hint: multi-party write conflicts and the off-chain-to-on-chain trust boundary are a distributed-systems problem, not a code-generation problem; getting the data model and multi-organization access control right takes real design iteration across milestones.

8.4 Decentralized Identity & Verifiable Credentials

A DID-based system where institutions (universities, licensing boards, employers) issue verifiable credentials, such as diplomas, transcripts, or professional licenses, that holders store and selectively share, with a revocation registry so an issuer can invalidate a credential later. Groups may pick a specific vertical (academic credentials, professional licensing, employment verification) to ground the design.

Key Challenges:

- Implementing selective disclosure: proving one fact from a credential without exposing the rest
- Designing revocation that does not leak who was revoked or when
- Handling key management and recovery when a user loses their credential-holding device
- Supporting interoperability across multiple issuers with different credential schemas

Suggested Building Blocks: W3C Verifiable Credentials / DID standards; a credential wallet (mobile or browser-extension); a revocation registry (Merkle accumulator or on-chain bitmap); institution-side issuance API and admin console.

Hint: selective disclosure and privacy-preserving revocation are genuinely unsolved-feeling problems at the undergraduate level; wiring together issuers, holders, and verifiers into a coherent, standards-compliant system is a multi-component integration effort AI cannot assemble correctly end to end.

8.5 Cross-Chain Asset Bridge

A bridge that moves tokens between two test networks via lock-and-mint or burn-and-release, secured by a relayer or light-client verification mechanism.

Key Challenges:

- Designing a security model that survives a malicious or offline relayer
- Handling reorgs and differing finality guarantees between the two chains
- Preventing double-spends across the bridge during network partitions
- Recovering gracefully from a failed or partially completed cross-chain transfer

Suggested Building Blocks: two EVM testnets (or one EVM + one non-EVM chain for extra depth); a relayer service (Node/Go) with signed attestations; light-client or optimistic-verification logic; a monitoring dashboard for in-flight transfers.

Hint: bridges are widely regarded as the highest-stakes, most exploited component in production blockchain systems; reasoning correctly about finality, reorgs, and partial-failure recovery requires distributed-systems thinking that has to be worked through, tested, and broken by the team itself.

8.6 Milestone-Gated Crowdfunding Platform

A Kickstarter-style platform where a campaign's funds are held in escrow and released to the creator only as backers vote to approve each milestone, with refunds available if a campaign stalls.

Key Challenges:

- Designing a fair voting and quorum mechanism across many small backers
- Handling partial refunds when a campaign is only partially completed
- Preventing creators from gaming vague or shifting milestone definitions
- Managing many concurrent campaigns, each with different rules, cleanly

Suggested Building Blocks: a factory contract pattern for per-campaign deployment; proportional-refund accounting logic; a database-backed campaign discovery/search frontend; off-chain notification system for voting deadlines.

Hint: quorum design, partial-refund accounting, and anti-gaming rules for milestone definitions interact in ways that only surface once you have real campaigns and real (simulated) bad actors to test against, which takes sustained iteration across the milestone schedule.

8.7 On-Chain Reputation for Undercollateralized Lending

A lending protocol that builds a borrower's credit score from their on-chain transaction and repayment history, enabling loans that require less collateral than typical DeFi lending.

Key Challenges:

- Designing a scoring algorithm resistant to manipulation, such as wash transactions used to inflate a score
- Handling the cold-start problem for new addresses with no history
- Balancing under-collateralization risk against realistic default incentives
- Wiring the score into actual loan terms, interest rates, and liquidation logic

Suggested Building Blocks: an off-chain indexer/database (e.g., Postgres + The Graph-style event indexing) computing scores from historical events; a scoring model (rule-based or a lightweight ML model as an AI-integration option); Solidity lending pool with score-adjusted collateral ratios.

Hint: a manipulation-resistant scoring model is a data science and mechanism-design problem layered on top of a lending protocol; validating that the score actually predicts default risk, and that it cannot be gamed, requires building simulated transaction histories and adversarial testing over many iterations.

8.8 Decentralized Prediction Market

A platform where users bet on the outcome of real-world events, priced by an automated market maker and resolved through an oracle or dispute process once the event concludes.

Key Challenges:

- Designing AMM pricing math that stays stable as a market approaches resolution
- Handling disputed or genuinely ambiguous real-world outcomes
- Preventing oracle manipulation or bribery around resolution time
- Maintaining usable liquidity for low-interest or long-tail markets

Suggested Building Blocks: constant-function or LMSR-style AMM contract; an oracle module (committee-based or optimistic-oracle pattern); a dispute/challenge window with staked disputers; frontend charting of live odds.

Hint: getting AMM pricing math right (and provably not exploitable near resolution) plus building a credible, bribery-resistant resolution process are two hard, interacting subsystems; each needs its own design, implementation, and adversarial test pass.

8.9 Anti-Scalping NFT Ticketing System

An event ticketing platform where tickets are NFTs carrying built-in resale rules, such as price caps or organizer royalties, alongside a legitimate resale marketplace.

Key Challenges:

- Encoding resale rules that are actually hard to circumvent through off-chain side deals
- Handling refunds and cancellations when an event changes or is called off
- Preventing bot-driven bulk minting during a popular ticket launch
- Balancing organizer control against a resale market people will actually use

Suggested Building Blocks: ERC-721/1155 tickets with transfer hooks enforcing price caps; a mint-rate-limiting mechanism (allowlists, commit-reveal, or ZK proof-of-personhood as a stretch goal); an event-organizer dashboard backed by a database; a resale marketplace UI.

Hint: enforceable resale rules and bot resistance are an adversarial cat-and-mouse problem; a naive implementation is trivially circumvented, and building one that actually holds up requires rounds of red-teaming your own system, which is inherently iterative work.

8.10 Incentivized Decentralized File Storage

A simplified Filecoin/Storj-style network where storage nodes stake tokens, store file chunks, and earn payment by passing periodic proof-of-storage challenges, with slashing for nodes that fail.

Key Challenges:

- Designing an efficient proof-of-storage or proof-of-retrievability scheme
- Handling node churn, including data that needs re-replication when a node leaves
- Setting slashing parameters that deter cheating without over-punishing honest failures

- Building a working chunked storage and retrieval layer, not just the incentive contract

Suggested Building Blocks: a P2P networking library (libp2p) for node-to-node transfer; erasure coding or simple replication for chunked storage; a challenge-response proof scheme (Merkle proofs over chunks); an on-chain staking/slashing contract coordinating payments.

Hint: this is a real distributed-systems project (networking, replication, fault tolerance) with a cryptoeconomic layer on top; building and load-testing the actual storage/retrieval path, not just the contract, is exactly the kind of systems work that resists one-shot AI generation.

8.11 DAO Governance Platform with Quadratic Voting

A platform for spinning up DAOs with treasury management and a choice between token-weighted and quadratic voting, letting a community pick the governance model that fits it.

Key Challenges:

- Preventing Sybil attacks against quadratic voting via many small identities
- Designing safe treasury execution using timelocks and multi-signature safeguards
- Handling low voter turnout and governance apathy realistically
- Supporting proposal types beyond a simple yes/no, such as budget allocation

Suggested Building Blocks: a governance framework pattern (OpenZeppelin Governor as a reference, reimplemented/extended rather than used as-is); a Sybil-resistance layer (proof-of-personhood, stake-weighted identity, or a bonding mechanism); a proposal/voting frontend; timelock + multisig treasury execution.

Hint: Sybil resistance for quadratic voting is an open problem in the field, not a solved library call; the team has to choose, justify, and stress-test a specific mitigation, then integrate it with treasury safety mechanisms across the whole semester.

8.12 Peer-to-Peer Insurance Protocol

A mutual insurance pool where members pay premiums into a shared fund, submit claims, and have other members vote on whether a claim should be paid out.

Key Challenges:

- Designing claims-voting that resists both collusion and voter apathy
- Sizing premiums and the pool so payouts remain solvent under realistic claim rates
- Handling disputed claims and the evidence submitted to support them
- Preventing adverse selection, where only high-risk members choose to join

Suggested Building Blocks: an actuarial-style premium model (simple risk pooling math, extendable with a data-driven/AI risk model); a claims-evidence submission and voting contract; a database for member risk profiles and claim history; solvency-monitoring dashboard.

Hint: this is simultaneously a mechanism-design problem (claims voting) and a quantitative one (pool solvency under realistic assumptions); validating that the pool doesn't collapse under adversarial or unlucky

claim patterns requires simulation and iteration a single AI pass cannot substitute for.

8.13 Carbon Credit Marketplace

A marketplace that tokenizes verified carbon offset credits and tracks each one from issuance through trading to permanent retirement.

Key Challenges:

- Preventing double-counting or double-selling of a credit that has already been retired
- Connecting real-world verification and audits to on-chain issuance trustworthily
- Designing a market that fairly handles credits of differing quality and vintage
- Ensuring a retired credit is provably and permanently unusable again

Suggested Building Blocks: ERC-1155 or fungible token-per-vintage design; a verifier/registrar onboarding flow with role-based issuance rights; an order book or AMM for trading; a public retirement ledger with burn proofs.

Hint: the interesting problem is designing a market and data model that correctly represents heterogeneous, real-world-verified assets (differing quality/vintage) while making double-counting structurally impossible, which takes real domain modeling, not a quick contract template.

8.14 Multi-Signature Treasury Wallet with Social Recovery

A smart contract wallet requiring M-of-N signer approval for transactions, with spending limits, timelocks, and guardian-based recovery if a key is lost.

Key Challenges:

- Designing guardian-based recovery that resists collusion among the guardians themselves
- Handling signer changes, such as adding or removing a signer, without opening a security gap
- Balancing timelock delays against usability for legitimate urgent transactions
- Preventing a malicious minority of signers from blocking or draining the wallet

Suggested Building Blocks: account-abstraction patterns (ERC-4337 as a stretch goal) or a classic multisig contract (Gnosis Safe-style, reimplemented); guardian recovery module with time-delayed execution; a wallet frontend with transaction simulation before signing.

Hint: wallet security is unforgiving; every edge case around signer changes, recovery, and timelocks is a potential fund-draining bug, so the team has to reason through and test adversarial scenarios exhaustively, which is precisely the slow, careful work AI-generated code tends to skip.

8.15 Decentralized Compute Marketplace

A marketplace where providers rent out spare compute capacity; jobs are submitted, executed, and verified before payment is released to the provider. Groups looking for an AI angle may target ML training/inference jobs specifically.

Key Challenges:

- Verifying a job actually executed correctly without simply re-running the whole thing
- Handling provider failures or maliciously incorrect results
- Matching jobs to providers with adequate capacity fairly and efficiently
- Designing payment release that is tied strictly to verified completion, not just submission

Suggested Building Blocks: a job-submission and containerized-execution pipeline; a verification scheme (redundant execution with majority voting, or spot-checking with slashing); a scheduler/matching service backed by a database; escrowed payment contract.

Hint: verifying computation without full re-execution is an active research area; the team must choose a pragmatic verification strategy (redundancy, spot-checks, staking), justify the trade-off, and build the actual execution pipeline around it, which is substantial systems work.

8.16 Multi-Format On-Chain Auction House

An auction platform supporting English, Dutch, and sealed-bid formats for both digital and physical assets, with anti-sniping protection and delivery escrow for physical goods.

Key Challenges:

- Implementing multiple auction formats correctly, including keeping sealed bids private until reveal
- Preventing last-second sniping without unfairly extending every auction
- Handling escrow and dispute resolution for confirming physical-item delivery
- Managing failed payments or non-delivery after an auction has closed

Suggested Building Blocks: commit-reveal scheme for sealed bids; per-format auction contracts sharing a common escrow module; oracle or attestation flow for physical-delivery confirmation; auction-discovery frontend.

Hint: three auction formats each have distinct correctness and fairness properties to get right (privacy of sealed bids, anti-sniping without unfair extensions, escrow for physical goods), and each interacts with the shared payment/escrow layer, making this a genuine multi-subsystem integration project.

8.17 Privacy-Preserving Health Record Sharing

A platform where patients store encrypted health records with access controlled on-chain, granting doctors or institutions temporary, revocable access without a central data custodian.

Key Challenges:

- Managing encryption keys without relying on a central custodian
- Designing fine-grained, revocable access control that still works in emergencies
- Handling large medical files, such as imaging, that cannot live on-chain directly

- Meeting realistic privacy expectations, including an audit trail that does not expose content

Suggested Building Blocks: proxy re-encryption or threshold key management; off-chain encrypted storage (IPFS/S3-compatible) referenced by on-chain pointers; an access-control contract with break-glass emergency logic; an append-only, privacy-preserving audit log.

Hint: key management without a trusted custodian, combined with an emergency-access exception path, is a genuinely hard applied-cryptography and access-control problem; getting both the security properties and the emergency usability right requires careful, iterative design review, not a single generated contract.

8.18 Decentralized Exchange with Hybrid AMM/Order Book

A trading venue for tokenized assets combining an automated market maker for baseline liquidity with an on-chain (or hybrid on/off-chain) limit order book for price-sensitive traders, including basic MEV/front-running mitigations.

Key Challenges:

- Designing AMM invariant math (constant-product or concentrated-liquidity style) that resists common exploits
- Building and matching a limit order book efficiently without prohibitive gas costs
- Mitigating front-running and sandwich attacks against pending trades
- Reconciling AMM and order-book liquidity into one consistent, arbitrage-resistant price

Suggested Building Blocks: Solidity AMM pool contracts; an off-chain matching engine with on-chain settlement (hybrid design) to keep gas costs manageable; a mempool-aware frontend warning users of slippage/MEV exposure; a database-backed order book and trade history.

Hint: DEX design sits at the intersection of financial engineering, distributed systems, and adversarial security (MEV); reconciling two liquidity models while defending against front-running is a rich, interacting set of problems that takes a full team a full semester to get right, let alone make fast and cheap.

8.19 AI-Assisted Smart Contract Security & Bug-Bounty Marketplace

A platform where developers submit smart contracts for review, an AI-assisted static/dynamic analysis pipeline produces an initial vulnerability report, and human auditors (staking reputation) verify, dispute, or extend those findings for a bounty paid out on-chain.

Key Challenges:

- Building or integrating real static/dynamic analysis tooling (not just prompting an LLM) to catch classes like reentrancy, access-control, and arithmetic bugs
- Designing an auditor reputation and staking system that rewards correct findings and penalizes false positives or missed bugs
- Handling disputes between the AI pipeline, the submitting developer, and human auditors over whether a finding is real
- Structuring bounty payout and escrow tied to verified, reproducible vulnerabilities rather than mere claims

Suggested Building Blocks: static analysis via Slither/Semgrep-style tooling plus an LLM-based triage layer for developer-facing explanations; a sandboxed execution environment (Foundry/Hardhat fork testing) to reproduce reported exploits; an on-chain reputation/staking contract for auditors; a submission and dispute-resolution web app.

Hint: this project explicitly forces the team to confront what AI tools are and are not good at in security analysis, building the human-verification, staking, and dispute layers around an imperfect AI pipeline is real systems and mechanism-design work, and directly a full-semester exercise in using AI responsibly rather than being replaced by it.

8.20 Decentralized Data Marketplace with Verifiable Query Execution

A marketplace where dataset owners list access to their data (for analytics or AI training), buyers submit queries or training jobs, and the platform returns cryptographically or economically verifiable proof that the query ran correctly over the real, committed dataset before payment is released, without necessarily revealing the raw data.

Key Challenges:

- Committing to a dataset on-chain (e.g., via a Merkle root) so results can later be checked against it
- Choosing and implementing a verification strategy for query correctness (verifiable computation, challengeable results, or trusted-execution attestation) appropriate for a semester project
- Preserving useful data privacy for the owner while still letting a buyer trust the result
- Pricing and licensing data access, including repeated queries, subsets, and usage-based billing

Suggested Building Blocks: a relational or columnar database as the actual query engine, with a Merkle-committed snapshot; a verification layer (start with an optimistic, challenge-based scheme; ZK/TEE verification as a stretch goal); a query-submission and billing API; a marketplace/discovery frontend for datasets.

Hint: verifiable computation over real datasets is an open, actively researched problem; the team must scope a tractable verification strategy, build a real query engine around it, and handle the privacy/pricing product layer on top, which is far more than any single AI-generated component can cover.