The UNIVERSITY of OKLAHOMA

NFT-Based Decentralized Ticketing System

Team Members

Bhanu Chaitanya Jasti (113651003)

Arava Dhanushwi (113647452)

May 4th, 2025

Contents

Project Repository and Video Demo

- **GitHub Repository:** Github link
- **Demo Video:** Video

1 Introduction

1.1 Background

Blockchain technology has introduced a decentralized framework for managing digital transactions and asset ownership. Unlike traditional systems that rely on centralized authorities, blockchain provides a trustless environment powered by consensus algorithms and cryptographic proofs. At the forefront of this innovation are smart contracts self executing programs that run on blockchain networks and automate processes based on pre-defined conditions.

One of the most impactful applications of blockchain and smart contracts is the emergence of Non-Fungible Tokens (NFTs). NFTs represent unique, verifiable digital assets. and have found utility in diverse fields such as digital art, gaming, identity, and asset tokenization. While NFTs are most commonly associated with digital collectibles, their potential to serve as secure, verifiable access tokens has made them an attractive solution for modernizing event ticketing.

Traditional ticketing systems often suffer from inefficiencies, fraud, and lack of transparency, making them vulnerable to manipulation. This project investigates the application of Ethereum-compatible smart contracts to reimagine event ticketing using decentralized technologies. Our prototype, deployed on the Sepolia Ethereum testnet, utilizes standardized NFT protocols to create a transparent and programmable platform for issuing, managing, and validating event tickets. The full system supports end-to-end ticket lifecycle management, including minting by organizers, purchasing by users via MetaMask, secure metadata storage via IPFS, secondary marketplace transactions, and real-time ticket validation using QR code scanning and smart contract-based verification.

1.2 Motivation

The motivation behind this project stems from the need to address ongoing challenges in conventional ticketing platforms, where centralized control often leads to fraud, poor user experiences, and limited ownership rights. The project draws inspiration from the principles of decentralization, verifiable ownership, and programmable logic, all foundational elements of blockchain systems.

We were particularly interested in understanding how NFT-based ticketing could enforce transfer rules, royalty enforcement, and dynamic metadata updates directly through smart contract logic. By leveraging the programmability of blockchain, our goal was to explore new ways to bring fairness, traceability, and flexibility to the ticketing process. The system is intended not only as a functional tool but as a research prototype that highlights both the potential and the current limitations of blockchain-based ticketing.

1.3 Real-World Problem: Ticket Scalping, Fraud & Resale Control

Event ticketing whether for concerts, conferences, or sports games has long faced problems caused by centralized control and a lack of transparency. One of the most pressing issues is ticket scalping, where automated bots or individuals purchase tickets in bulk and resell them at inflated prices. This not only locks out genuine fans but also creates an unfair market driven by profit rather than access.

In addition, counterfeit tickets remain a serious concern, especially in peer-to-peer resale scenarios where the buyer cannot easily verify authenticity. Furthermore, most current systems offer little to no transparency regarding how tickets are distributed, transferred, or resold. The lack of verifiable audit trails creates loopholes that can be exploited, with organizers losing control over their own ticketing policies.

There is also no standard mechanism to reward original artists or organizers from secondary sales. These real-world problems form the core rationale for building a decentralized system that allows for programmable rules, verifiable ownership, and traceable resale paths all of which can be enforced via smart contracts on a public blockchain.

1.4 Objectives of the Project

The primary objective of this project is to develop and evaluate a decentralized ticketing system built on Ethereum-compatible smart contracts, using the Sepolia testnet for deployment. To achieve this, we defined a series of specific goals:

- Implement multiple ticketing models using NFT standards, including standard, soulbound, royalty-enforced, and dynamic metadata variants.
- Provide event organizers with a dashboard interface to deploy event-specific contracts and mint tickets.
- Enable ticket buyers to connect their wallets, purchase NFTs, and interact with tickets through a user-friendly frontend.
- Enforce transfer restrictions and royalty logic directly at the contract level.
- Store ticket metadata securely and immutably using IPFS.
- Evaluate and compare the strengths and trade-offs of each ticketing model in terms of decentralization, enforceability, flexibility, and real-world usability.
- Integrate QR code-based verification at event gates to verify ownership and, in the case of dynamic tickets, mark tickets as "used" on-chain.

This combination of practical implementation and comparative analysis forms the foundation for a prototype that can inform future research and real-world adoption of blockchain-based event ticketing systems.

2 Problem Statement

2.1 Challenges in Traditional Ticketing Systems

Traditional ticketing platforms are predominantly centralized, giving a few entities total control over how tickets are issued, priced, and resold. This lack of transparency makes the system susceptible to several longstanding issues:

- Centralized control with limited auditability
- Exploitation by bots and scalpers during initial sales

- Absence of enforceable resale rules or price limits
- Difficulty in verifying ticket authenticity in secondary markets
- Limited compatibility with decentralized identity or ownership systems
- No standard framework for sharing revenue from resales with artists or organizers

These limitations create an unfair landscape where genuine users are disadvantaged, and organizers lack meaningful control once tickets are distributed. This centralization not only reduces trust but also limits interoperability with decentralized identity systems and Web3-native access control, which are increasingly important in global digital ecosystems.

2.2 What We Initially Proposed

As outlined in our original project proposal, we aimed to develop an NFT-based decentralized ticketing system powered by Ethereum smart contracts. The core design involved a factory contract that would enable event organizers to deploy independent ERC-721-based ticket contracts. Each ticket would be stored as an NFT, with its metadata hosted on IPFS to ensure decentralization and immutability.

Additionally, we intended to implement EIP-2981 .royalty support and a QR or wallet-based check-in mechanism to verify ticket ownership during event entry. The frontend would be built using React with MetaMask integration via Ethers.js, and deployments would be made on a public Ethereum testnet.

2.3 What Was Missing in the Initial Proposal

After submitting our initial proposal, we received feedback highlighting key areas where our system lacked depth and differentiation. While our concept aligned with known blockchain practices, it did not sufficiently distinguish itself from existing platforms like GET Protocol or YellowHeart.

Specifically, we had not addressed real-world challenges such as transfer restrictions, scalping prevention mechanisms, or the technical nuances involved in various NFT behaviors. The original plan focused largely on minting and verifying standard NFTs, without exploring how programmable ownership rules could be enforced differently across ticket types.

2.4 Goals of the Revised Implementation

In response to this feedback, we expanded the scope of our implementation to adopt a more research-oriented and comparative framework. The revised goals include:

- Introducing four distinct types of NFT-based tickets: Standard, Soulbound, Royalty-Enforced, and Dynamic Metadata Tickets
- Implementing logic to control or restrict ticket transferability, resale royalties, and metadata updates
- Comparing the advantages and limitations of each model through hands-on deployment and evaluation

- Demonstrating how smart contract design choices directly impact security, flexibility, and usability in a decentralized ticketing context

This revised approach allows us to go beyond a basic NFT application and evaluate how different smart contract mechanisms perform under realistic use cases.

Proposal vs. Implementation Summary

Initially Proposed	Actually Implemented
Single ERC-721 Ticket Model	Four Custom Models (Standard, Soulbound, Royalty, Dynamic)
Basic resale royalty logic	EIP-2981 fully implemented in RoyaltyTicket
IPFS Metadata for tickets	IPFS integration with dynamic updates in DynamicTicket
Check-in via QR/wallet (planned)	QR-based ticket validation with status update in DynamicTicket
Basic mint/view flow	Full dashboard for minting, listing, and displaying user-owned tickets

Table 1: Comparison of Initial Proposal vs Final Implementation

3 Related Work

3.1 Existing NFT Ticketing Platforms

In exploring this project, we reviewed several platforms that attempt to modernize ticketing using blockchain and NFTs. These systems inspired aspects of our design but also revealed key limitations.

- **GET Protocol:** Among the earliest NFT ticketing solutions, integrating blockchain with traditional ticketing infrastructure. However, it still relies on centralized metadata storage and off-chain resale enforcement, undermining full decentralization.
- **YellowHeart:** Focused on NFT tickets in the music industry, with royalty support. However, enforcement is not fully on-chain, and modular event logic is limited.
- **Seatlab:** Offers royalty-enabled NFT tickets but lacks dynamic behavior such as marking tickets as used or expiring them post-entry, which limits its real-time applicability .
- **NFT.TiX:** Fully decentralized ticket issuance but does not support metadata updates or transfer restrictions like soulbound behavior, which are essential for identity-linked access control .

- **POAP (Proof of Attendance Protocol):** Issues post-event NFTs for attendance but doesn't support pre-sale, ownership enforcement, or resale controls .

3.2 Limitations in Current Systems

These platforms, while innovative, face challenges that limit their real-world scalability:

- **Centralized metadata storage:** Many platforms still rely on centralized servers or cloud storage for NFT metadata, introducing a single point of failure .
- **Lack of dynamic ticket states:** Most systems cannot update metadata to reflect ticket status (e.g., used, expired), making them unsuitable for real-time validation.
- **Weak or optional resale enforcement:** While some platforms support resale logic, it is often optional or not enforced on-chain, allowing scalping to persist .
- **Rigid ticket templates:** Organizers have limited ability to configure ticket logic to suit their event needs .
- **Single-chain dependency:** Many platforms are tied to a single blockchain, restricting cross-chain accessibility .
- **Centralized verification:** Ticket validation often depends on web apps or APIs, which contradicts decentralization goals .
- **Complex UX:** Non-technical users face friction from wallet setup, gas fees, and contract interactions .
- **Incomplete royalty compliance:** Even platforms that support EIP-2981 cannot enforce royalties if resale happens on non-compliant marketplaces.

4 Proposed Solution

4.1 How Our Project Addresses These Gaps

In response, we developed a modular NFT ticketing system built on Ethereum-compatible smart contracts, with a focus on decentralization, flexibility, and enforceability.

- **Factory-based architecture:** We use a Factory contract to deploy event-specific ticketing contracts. Each event can choose a suitable model (e.g., transferable, soulbound) .
- **Four custom ticket types:**
  - **StandardTicket** – Fully transferable, basic ERC-721 .
  - **SoulboundTicket** – Non-transferable credential model that reflects the ideas proposed by Vitalik Buterin on identity-bound tokens.
  - **RoyaltyTicket** – Implements EIP-2981 royalties enforced on-chain .
  - **DynamicTicket** – Supports metadata updates (e.g., status: used), inspired by dynamic NFT models .

- **Decentralized metadata via IPFS:** All ticket data is stored on IPFS, eliminating reliance on centralized servers and ensuring immutability .
- **Resale control via EIP-2981:** Royalty logic is enforced directly in smart contracts to ensure that artists or organizers receive resale revenue .
- **QR-based real-time verification:** At check-in, tickets are scanned via QR codes that encode contract, owner, and token ID. DynamicTickets can be marked as used post-verification .
- **User-friendly frontend:** Built with React and Ethers.js, the UI provides MetaMask connectivity, minting tools, and resale features with minimal friction for end-users .
- **Multi-chain potential via EVM support:** By deploying to Sepolia (an Ethereum testnet), we maintain full compatibility with MetaMask, Hardhat, and Ethers.js—setting the foundation for broader EVM-chain deployments.

4.2 Why Our Project Is Unique

Our system goes beyond building a functional NFT ticketing dApp—it introduces a comparative research prototype that explores the trade-offs between NFT models. Unlike existing platforms that rely on rigid templates or partial decentralization, our implementation allows event organizers to:

- Choose custom ticket behaviors aligned to their use case (ID-bound, resellable, or status-aware)
- Deploy fully isolated, per-event contracts via a factory pattern for event-level autonomy
- Store all metadata on IPFS to maintain decentralization even after event completion
- Support both ticket lifecycle enforcement and wallet-based verification

Our work demonstrates how smart contract design patterns, such as modular contract deployment and metadata mutability, can be leveraged to build resilient, transparent, and user-centric ticketing infrastructure.

5 Methodology

5.1 Technical Implementation

5.1.1 Frontend: React + Ethers.js

We built the user interface using React to ensure a responsive and modular design. Ethers.js was integrated for all blockchain interactions, enabling wallet connectivity, real-time ticket listings, event creation by admins, and QR or wallet-based ticket verification for check-in.

5.1.2 Backend/Contracts: Solidity + Hardhat

All ticket logic was written in Solidity and compiled/tested using the Hardhat framework. The project includes four main smart contracts (`StandardTicket`, `SoulboundTicket`, `RoyaltyTicket`, `DynamicTicket`) along with a `Factory` contract for deploying event-specific instances. Deployment scripts were automated to streamline contract deployment to the Sepolia testnet.

Ticket Minting Logic For each event, the deployed ticket contract exposes a `mintTicket(address buyer)` function, callable only by the admin. Upon minting, ownership is recorded using the standard `ownerOf` function from the ERC-721 interface. This ensures that each minted NFT is uniquely assigned and queryable via its token ID.

5.1.3 Wallet Connection & Integration

Users interact with the dApp by connecting their MetaMask wallet. Once connected, Ethers.js enables them to invoke contract methods such as purchasing tickets, viewing owned tickets, and verifying status. All transactions are signed and executed on-chain, simulating real usage on Ethereum.

5.1.4 Marketplace Functionality

The `RoyaltyTicket` model supports listing and resale while automatically enforcing royalty distribution using EIP-2981.. Other ticket types such as `Soulbound` or `Dynamic` restrict transfers based on internal logic. These enforcement mechanisms demonstrate the flexibility of programmable transfer rules within NFTs.

5.1.5 Metadata Storage & Dynamic Update Logic

Ticket metadata including event name, description, image URL, and usage status is uploaded to the InterPlanetary File System (IPFS) during ticket creation. We integrated **Pinata**, a trusted IPFS gateway and pinning service, through our backend server to automate this upload process. This ensures that metadata remains immutable, verifiable, and publicly accessible even if the backend is offline.

The DynamicTicket model includes a setUsed() method, callable by the event organizer or verifier, which updates the associated metadata to reflect the ticket's current status (e.g., "used", "active", or "invalid"). By coupling smart contract logic with IPFS-based updates, our system enables real-world access control and usage tracking in a decentralized manner.

5.2 Smart Contract Design Details

Why These Four Ticket Types?

At the beginning of the project, our initial plan focused on building a working decentralized ticketing prototype. We proposed basic capabilities such as deploying event-specific contracts using a factory pattern, minting ERC-721 NFTs, storing metadata on IPFS, adding optional resale royalties (EIP-2981), enabling QR or wallet-based check-ins, and building a React + Ethers.js frontend. These formed the technical scope of our original proposal.

However, during our review phase, our professor encouraged us to explore a broader, research-oriented approach by implementing and comparing multiple NFT-based ticket models. After researching various blockchain ticketing techniques and their practical implications, we selected four distinct and meaningful types to develop: **StandardTicket**, **SoulboundTicket**, **RoyaltyTicket**, and **DynamicTicket**.

Each of these serves a unique purpose:

- **StandardTicket:** A baseline transferable ticket to model conventional NFT behavior.
- **SoulboundTicket:** Implements transfer restrictions to simulate personal credentials.
- **RoyaltyTicket:** Enables enforced royalty payments on resale, tackling scalping.
- **DynamicTicket:** Allows metadata to be updated post-mint for features like check-in tracking.

We also considered but ultimately did not implement the following advanced models:

- **Time-Locked Tickets:** Conceptually useful but narrow in scope; better handled off-chain.
- **Multi-use Tickets:** Required complex state tracking beyond the academic timeline.
- **Batch Minting Contracts:** Useful at scale, but unnecessary for our comparative focus.
- **ERC-6551 / Composable Tickets:** Innovative but too experimental and complex.
- **Multi-Signature and Loyalty Tickets:** Attractive features with high overhead.

Given time constraints and our focus on comparative evaluation, we implemented only those four models that best reflected trade-offs in decentralization, programmability, enforceability, and usability.

5.2.1 StandardTicket

This contract follows a minimal ERC-721 interface with no overrides or restrictions. It supports open minting and free transferability. We used this model as a baseline to compare the behaviors of constrained alternatives.

5.2.2 SoulboundTicket

`SoulboundTicket` overrides the `_beforeTokenTransfer` hook in ERC-721 to restrict transfers post-minting. It essentially locks the NFT to the wallet that minted or received it initially.

5.2.3 RoyaltyTicket

`RoyaltyTicket` uses EIP-2981 to define a `royaltyInfo` function. This returns the address of the royalty recipient and the royalty amount. When tickets are resold on compliant marketplaces, a percentage of the sale is routed to the organizer.

This approach discourages scalping and creates sustainable revenue streams for creators. It also reflects current trends in digital content resale economics.

5.2.4 DynamicTicket

This contract introduces a `setUsed()` function, callable by the admin or event verifier. It updates the ticket's metadata (hosted on IPFS) to reflect status like `"used"`, `"invalid"`, or `"active"`.

5.2.5 Ticket Comparison Table

To evaluate the trade-offs between different ticket types, we compare their design characteristics based on decentralization, enforceability, flexibility, and real-world applicability. Each type was implemented as a custom ERC-721 smart contract with unique behavior tailored to specific use cases.

Ticket Type	Transferable	Enforceability	Royalty Support	Best Use Case
StandardTicket	Yes	Low	No	General admission; flexible resale
SoulboundTicket	No	Very High	No	Certificates, identity-based access
RoyaltyTicket	Yes	Medium	Yes (EIP-2981)	Artist or organizer-focused resale with royalties
DynamicTicket	Yes (initially)	Medium	No	Event entry tracking; usage-based states

Table 2: Comparison of NFT Ticket Models

6 Challenges & Solutions

This project was a hands-on application of several core concepts learned in CS-5833, such as Ethereum smart contract development, NFT standards (ERC-721, EIP-2981) .S, decentralization using IPFS, and wallet-based authentication. Below, we describe the major technical challenges we encountered during the development process and how we used classroom knowledge to address them effectively.

6.1 Resale Control and Anti-Scalping

Challenge: One of the key issues in traditional ticketing systems is ticket scalping — where buyers hoard tickets and resell them at highly inflated prices. In blockchain-based

systems, this could happen more easily due to pseudonymity and automation via bots.

Solution: Leveraging our understanding of the `EIP-2981` royalty standard, we designed the *RoyaltyTicket* contract to enforce a fixed royalty fee on secondary sales. This ensured that event organizers receive a portion of resale profits on-chain. Furthermore, we extended the contract logic to allow organizers to define a `maxResalePrice`, directly embedding anti-scalping policies into the smart contract layer.

6.2 Non-Transferable Soulbound Tickets

Challenge: For use cases like certificates, scholarships, or identity-linked entry, we needed to create NFTs that were permanently bound to a user's wallet and could not be transferred or sold.

Solution: Building on discussions around identity in decentralized systems, we implemented the *SoulboundTicket* contract. We overrode ERC-721's `transferFrom` and `approve` functions to revert all transfer attempts. This mimics the "soulbound" concept described in Web3 research, ensuring the NFT remains tied to the original owner.

6.3 Marking Tickets as Used

Challenge: In real-world events, once a ticket is scanned for entry, it should be marked as used to prevent reuse. However, ERC-721 tokens are immutable after minting unless extended.

Solution: Drawing from lectures on dynamic metadata and mutable state, we built a *DynamicTicket* contract that allows event organizers to update metadata post-mint. Using metadata pinned to IPFS, the ticket's status can be changed from "valid" to "used" upon successful verification at the gate, reflecting real-time usage.

6.4 Secure IPFS Metadata Uploads

Challenge: Ticket metadata (event name, seat, date) needed to be stored in a decentralized manner. Directly uploading from the frontend introduced security risks due to API key exposure and CORS errors.

Solution: We followed best practices discussed in class around decentralized storage and privacy. We built a backend Express server that securely uploads metadata to IPFS via Pinata using JWT authentication. The backend returns the IPFS hash to the frontend, enabling seamless and secure integration with the blockchain.

6.5 Wallet Ownership and Access Control

Challenge: During development, only one wallet (the deployer) had access to administrative functions such as minting. This posed problems when trying to test or demo with multiple team accounts.

Solution: We used the `Ownable` pattern from OpenZeppelin to implement fine-grained access control. During deployment, we passed the correct `initialOwner` to the constructor and added wallet-based checks in the frontend to allow only the contract owner to access minting features.

6.6 JSX Parsing Errors and Custom Webpack

Challenge: We chose not to use `create-react-app` and instead set up a custom React + Webpack build for better control. This led to JSX parsing errors due to misconfigured Babel settings.

Solution: We reviewed module bundling and JSX compilation concepts and fixed the issue by installing `babel-loader` and configuring `@babel/preset-react`. This ensured that JSX syntax was correctly transpiled into browser-compatible JavaScript.

6.7 TokenID and Metadata Errors

Challenge: The frontend occasionally queried for `tokenId`s that had not been minted, causing `ERC721NonexistentToken` errors and breaking metadata fetch logic.

Solution: To resolve this, we implemented a `totalSupply()` function in each contract and used it to determine the range of valid `tokenId`s. We also added error handling in the frontend to gracefully handle cases where metadata could not be fetched.

6.8 Unified Marketplace Logic

Challenge: Not all ticket types supported resale (e.g., Soulbound and Dynamic tickets), which made it difficult to list all available tickets on one unified marketplace page.

Solution: Inspired by contract modularity principles covered in class, we filtered and grouped ticket types based on capabilities. The frontend dynamically loaded only resellable tickets (Standard and Royalty) and adjusted the purchase flow accordingly.

6.9 QR-Based Ticket Verification

Challenge: At event entry points, we needed a real-time way to verify that a scanned ticket was valid and belonged to the presenter.

Solution: We implemented a *VerifyTicket* page that scans QR codes containing the contract address, `tokenId`, and wallet address. Using `ethers.js`, the system queries on-chain ownership and either confirms the ticket or rejects it. For DynamicTickets, it also marks the ticket as "used," combining identity verification with real-time status updates.

6.10 Metadata Sync Delays Across Frontend and IPFS

Challenge: After updating ticket metadata (e.g., status from "valid" to "used"), our frontend sometimes showed outdated data due to IPFS propagation delays or caching.

Solution: We instructed users to manually refresh metadata via UI actions, and used toast messages to indicate when new metadata was available. This highlighted the challenges of working with decentralized, eventually-consistent systems.

6.11 Lack of User Feedback During Blockchain Actions

Challenge: Blockchain operations like minting or reselling can take several seconds, and users had no feedback, leading to confusion or repeated clicks.

Solution: We improved user experience by implementing spinners, toast messages, and modals to indicate when actions were pending or completed. These UI improvements helped bridge the gap between async blockchain behavior and user expectations.

6.12 Managing Multiple Ticket Variants Cleanly

Challenge: Each ticket type had different behavior (e.g., transfer rules, royalty logic, dynamic metadata), which made code reuse and maintenance difficult.

Solution: We applied object-oriented design principles discussed in class and created a modular contract structure. All ticket types inherit from a shared ERC-721 base, allowing us to implement their unique features while reusing core logic like minting and metadata handling.

7 Key Design Decisions

7.1 Why Multiple Ticket Types?

From the beginning, we realized that building just one type of NFT-based ticket would not do justice to the range of real-world ticketing needs. Traditional ticketing systems aren't one-size-fits-all some events require tickets to be resold, others must prevent transfer altogether, and some demand post-purchase state updates (e.g., used vs. unused). Taking this into account, and based on guidance from our professor, we deliberately implemented four distinct smart contracts: `StandardTicket`, `SoulboundTicket`, `RoyaltyTicket`, and `DynamicTicket`.

This decision allowed us to explore a variety of ticketing paradigms within the same platform. For example, with `SoulboundTicket`, we enforced non-transferability, simulating credentials or non-shareable access. With `RoyaltyTicket`, we implemented EIP-2981 to ensure fair compensation during resale. `DynamicTicket` helped us demonstrate how metadata could reflect real-time status updates such as check-in validation. The `StandardTicket` acted as our control model, providing a transferable ticket without constraints.

Having multiple ticket types enabled us to compare their technical behavior side by side how they differed in terms of transfer rules, resale logic, gas costs, and user interaction. It also made the project more versatile for future scalability and served as a meaningful research prototype to investigate the strengths and limitations of each NFT-based model.

7.2 Why Sepolia Testnet?

We evaluated several blockchain networks for development and testing, each offering different trade-offs in terms of tooling, developer experience, compatibility, and ecosystem support. Some of the alternatives we considered included:

- **Mumbai (Polygon Testnet):** Known for its low gas fees and fast confirmation times, Mumbai is a popular choice among developers. However, we experienced intermittent congestion and found that its EVM compatibility sometimes diverged from Ethereum mainnet standards, particularly in event indexing and contract behavior.
- **Avalanche Fuji:** Avalanche's testnet offers quick finality and supports Solidity smart contracts via the C-Chain. While attractive in terms of speed, the ecosystem requires different tooling and infrastructure setup, making it less seamless for Ethereum-first development.

- **BNB Chain Testnet:** Binance's test network offers decent throughput and developer support but leans heavily on centralized validators and lacks the nuanced tooling provided by Ethereum's ecosystem.
- **Solana Devnet:** Solana offers high throughput and fast transaction processing. However, it is not EVM-compatible, requiring contracts to be written in Rust and deployed using entirely different frameworks like Anchor. This made it incompatible with our Solidity-based architecture.

After evaluating these options, we selected **Sepolia**, an official Ethereum testnet. Sepolia is lightweight and specifically optimized for developer testing while retaining full compatibility with the Ethereum Virtual Machine (EVM) . It integrates seamlessly with industry-standard tools such as Hardhat, MetaMask, and Ethers.js, which are core to our project's infrastructure .

Sepolia also offers a reliable faucet for acquiring test ETH, public block explorers for transaction tracing, and network behavior that closely mimics mainnet conditions without the cost and risk . This made it an ideal environment for testing our contract deployments, simulating user flows, and debugging edge cases in a safe and reproducible way.

7.3 Scalability & Modularity

Scalability and flexibility were at the heart of our design. Instead of creating one large monolithic contract to manage all ticket types or events, we adopted a factory contract pattern. This allowed us to deploy separate smart contracts for each event, each with its own ticketing logic and supply constraints.

The modular approach ensured that event organizers could launch and manage their own ticketing contract with the flexibility to choose the type of ticket they wanted—be it transferable, royalty-enabled, or check-in-trackable. It also provided strong isolation: changes made to one event's contract would never impact another.

In terms of long-term design, this pattern is scalable and maintainable. It allows for future upgrades, where additional ticket models (e.g., time-locked or multi-use tickets) can be added without disrupting the existing system. Furthermore, this structure simplifies debugging and testing since each contract has clear boundaries and responsibilities.

7.4 Trade-offs

While implementing multiple contract types provided deep insight and flexibility, it also introduced certain trade-offs. Each contract had to be separately developed, tested, and deployed, increasing the overall complexity of our project. Gas consumption also varied between models especially for contracts like `DynamicTicket` that involve metadata updates.

Maintaining IPFS links and ensuring that metadata updates correctly reflected ticket status across all models required additional logic and off-chain coordination. Compatibility and royalty compliant marketplaces had to be manually tested to verify if EIP-2981 was working as expected.

Another trade-off was user experience. For instance, preventing transfers in `SoulboundTicket` worked from a technical standpoint but raised questions about what would happen if a

user lost access to their wallet. These edge cases, while outside the scope of our build, highlighted the complexities of designing for the real world.

Nonetheless, we accepted these trade-offs because they aligned with our goal of building a research-oriented prototype. Our objective was not just to demonstrate a working product but to reflect on how different contract designs shape real-world outcomes in decentralized ticketing systems.

8 Real-World NFT Ticketing Behavior Analysis

To evaluate our NFT ticketing prototype against real-world usage, we conducted an on-chain analysis of YellowHeart, a live NFT ticketing platform deployed on Ethereum Mainnet. Using Ethers.js and the verified ERC-721 contract at 0x7ba0A79eC30259E2792A43989EdD97c6E40Bb336, we extracted and analyzed 1,146 Transfer events. We then contrasted this with transactions from our own deployed system on Sepolia.

8.1 Comparative Insights

Aspect	YellowHeart (Mainnet)	Our System (Sepolia)
Total Transactions	1,146	34
Total Mints	350	23
Total Transfers	796	11
Unique Tokens	350	8
Unique Owners	696	3
Top Receiver	40 NFTs	26 NFTs
Most Active Token	Token ID 193 (10 transfers)	Token IDs 0, 2 (7 transfers each)
Peak Mint Day	Oct 4, 2021 (120 mints)	May 3, 2025 (9 mints)
Peak Resale Day	Nov 6, 2021 (100 transfers)	May 4, 2025 (6 transfers)
Avg. Mint Gas	226,691 gas	183,244 gas
Avg. Transfer Gas	197,389 gas	59,142 gas
Scalping Prevention	Not enforced	Enforced via Soulbound and resale caps
Metadata Updates	Not supported	Supported via DynamicTicket
Royalty Enforcement	Not enforced	Enforced via EIP-2981 in RoyaltyTicket
Verification Logic	Not implemented	QR-based with on-chain ownership check
Ticket Types	Single ERC-721	Four (Standard, Soulbound, Royalty, Dynamic)

Table 3: Comparison of YellowHeart and Our NFT Ticketing System

The results highlight the technical and behavioral limitations of current NFT ticketing implementations like YellowHeart. While their system exhibited significant resale activity and signs of scalping, it lacked on-chain enforcement for royalties, verification, and metadata updates. In contrast, our prototype introduced multiple smart contract models to prevent abuse, enforce resale royalties, and support post-mint metadata updates. Additionally, our system demonstrated significantly lower gas usage, making it more efficient and suitable for high-volume scenarios.

9 Results and Evaluation

9.1 Functionality Testing

Each component of the system was tested end-to-end on the Sepolia testnet. All smart contracts were successfully deployed, and interaction with them was verified via the MetaMask-connected frontend. The following key functions worked as expected:

- Event organizers could deploy new event-specific contracts using the Factory.
- Tickets could be minted with IPFS-hosted metadata and viewed in user wallets.
- Transferability was enforced correctly per ticket type (e.g., `SoulboundTicket` blocked transfers).
- `RoyaltyTicket` routed resale royalties correctly using EIP-2981 on test listings.
- `DynamicTicket` metadata could be updated via the `setUsed()` method and reflected in the UI.
- QR-based verification correctly decoded wallet and ticket data, and triggered on-chain validation.

9.2 Ticket Type Evaluation

We evaluated each model against a set of qualitative criteria: decentralization, enforceability, flexibility, and user experience. The previously introduced Table 2 summarizes the technical distinctions. Key observations include:

- **StandardTicket:** Worked well for general-purpose access. Minimal gas cost, but lacked enforcement features.
- **SoulboundTicket:** Successfully enforced non-transferability. Raises practical concerns if a user loses their wallet.
- **RoyaltyTicket:** Royalties were honored in test listings; however, enforcement depends on marketplace compliance.
- **DynamicTicket:** Metadata update flow worked, though syncing metadata took a few minutes.

9.3 Performance and Usability

- The React frontend worked reliably across tested browsers (Chrome, Firefox). MetaMask connection was seamless.
- QR verification was functional but may benefit from UI enhancements for production use.
- Gas usage was highest for `DynamicTicket` (due to metadata updates) and lowest for `StandardTicket`.
- IPFS integration via Pinata was stable, but updating metadata required careful coordination with token URIs.

9.4 Summary

Overall, the system achieved its intended goals. All four ticket types functioned as designed, and the platform provided a flexible, decentralized framework for managing event ticket lifecycles. Future improvements could include multi-chain support, off-chain metadata caching, and enhanced error handling for production use.

For a full walkthrough of the system functionality including contract deployment, ticket minting, resale listing, and QR-based verification, please refer to the accompanying demo video submitted alongside this report.
Demo Video: `https://drive.google.com/file/d/1QOZqo9eZN7EVMDbJ0a_bIsXA-CJsik6v/view?usp=drive_link`

10 GitHub Activity

We used Git and GitHub extensively throughout the development of this project to manage collaboration, track changes, and maintain version control across different components: smart contracts, backend, and frontend.

Project Structure Overview

- `contracts/` – Contains all Solidity contracts: `StandardTicket.sol`, `SoulboundTicket.sol`, `RoyaltyTicket.sol`, and `DynamicTicket.sol`.
- `scripts/` – Deployment and interaction scripts using Hardhat (e.g., `deploy.js`, `mint-standard.js`, `mark-used.js`).
- `backend/` – Node.js Express server handling IPFS uploads through Pinata integration.
- `frontend/src/components/` – React components for the UI (e.g., `AdminDashboard.js`, `Marketplace.js`, `VerifyTicket.js`).
- `frontend/src/utils/` – Utility scripts like `abiMap.js`, `pinata.js`, `walletConnect.js`.

Contribution Activity

- More than 40 commits were made during the development cycle across multiple branches.
- Active use of `git pull`, `git push`, and pull requests to merge features and fix issues.
- Merge conflicts were resolved manually, especially during IPFS integration, metadata syncing, and frontend route structuring.
- Continuous testing and bug fixing reflected in commit messages such as:
  - `fix: ERC721NonexistentToken bug in VerifyTicket.js`
  - `feat: implement resale logic for RoyaltyTicket`
  - `refactor: abstract walletConnect and IPFS logic`

```
- update: added QR scan verification and modal UX
```

- GitHub Issues and commits were used to track key milestones such as completing the dashboard, verifying QR scans, and integrating EIP-2981 royalty logic.

Final Repository

The full commit history and source code can be accessed from the public GitHub repository.
Repository URL: `https://github.com/jbhanuchai/blockchain_project`

11 Collaboration Team Contributions

This project was developed collaboratively by a two-member team: Bhanu Chaitanya Jasti and Arava Dhanushwi. The work was divided to leverage each member's strengths while ensuring shared responsibility in decision-making and debugging.

Bhanu Chaitanya Jasti

- Designed and implemented the entire React-based frontend, including pages such as `Home`, `Marketplace`, `MyTickets`, `AdminDashboard`, and `VerifyTicket`.
- Integrated MetaMask wallet connection and smart contract interactions using Ethers.js.
- Developed the QR-based ticket verification flow and dynamic status updates.
- Handled Pinata integration on the backend to upload and retrieve metadata from IPFS.
- Led the debugging of token URI issues, resale listing logic, and frontend modal UX.

Arava Dhanushwi

- Developed the Solidity smart contracts: `StandardTicket`, `SoulboundTicket`, `RoyaltyTicket`, `DynamicTicket`, and the `Factory` pattern.
- Implemented EIP-2981 royalty logic and enforced resale restrictions in `RoyaltyTicket`.
- Wrote Hardhat deployment and interaction scripts for minting, marking usage, and contract deployment.
- Set up the backend Express server to support Pinata integration and metadata uploads.
- Collaborated on testing smart contracts on the Sepolia testnet and validating different ticket behaviors.

Research Contributions

- Conducted an on-chain analysis of YellowHeart's NFT ticketing contract on Ethereum Mainnet by parsing 1,146 `Transfer` events, revealing behavioral patterns such as repeated resale activity and lack of royalty enforcement.
- Compared our system with YellowHeart across critical metrics including stats and metadata flexibility, resale logic, and transfer enforcement highlighting how our approach addressed limitations in current real-world platforms.
- Analyzed and implemented relevant Ethereum standards such as **EIP-2981** (royalty enforcement), **EIP-5192** (soulbound NFTs), and **ERC-721 metadata extensions** to ensure standard-compliant and extensible contract design.
- Investigated the role of IPFS in decentralized metadata storage and implemented a full integration using the Pinata API. This allowed immutable token metadata and dynamic updates (e.g., usage status) in our `DynamicTicket`.
- Compared all four ticket types `StandardTicket`, `SoulboundTicket`, `RoyaltyTicket`, and `DynamicTicket` across dimensions such as enforceability, decentralization, gas efficiency, metadata behavior, and user experience.
- Studied leading NFT ticketing platforms (e.g., NFT.TiX, GET Protocol) and identified their limitations in on-chain enforcement and decentralized metadata, which informed our decision to use smart contract logic and IPFS-based architecture.

Joint Responsibilities

- Finalized project architecture and selected appropriate blockchain tools and libraries.
- Reviewed each other's code via GitHub pull requests.
- Worked together on debugging cross-functional issues (e.g., wallet access control, IPFS syncin).
- Created demo video, documentation, and finalized the report layout in Overleaf.

12 Learnings & Reflections

This project provided a hands-on opportunity to apply blockchain concepts learned in the course, particularly those related to smart contracts, NFT standards, decentralized storage, and Ethereum-based development.

Technical Skills Acquired

- Deepened our understanding of the ERC-721 standard and its extensibility to support real-world use cases.
- Gained proficiency in writing and testing Solidity smart contracts using Hardhat.

- Learned how to enforce advanced constraints such as non-transferability (soulbound), royalty enforcement (EIP-2981), and dynamic metadata logic.
- Built a full-stack dApp using React and Ethers.js with seamless MetaMask integration.
- Implemented decentralized metadata management using IPFS and Pinata APIs.
- Debugged on-chain and off-chain issues across different environments (frontend, backend, and smart contracts).

Conceptual Takeaways

- Understood the importance of decentralized ticketing in combating fraud, scalping, and unfair resale practices.
- Realized the trade-offs between flexibility, decentralization, and user experience when designing smart contracts.
- Learned how blockchain principles: immutability, transparency, verifiability can reshape traditional systems like event ticketing.
- Experienced the complexity of coordinating metadata updates between smart contracts, and IPFS.

Personal Reflections

- This project emphasized the importance of modularity and clear separation of concerns between frontend, backend, and contract layers.
- Working as a team helped us simulate real-world collaborative development and improved our GitHub workflow, including issue tracking, branching, and pull request reviews.
- Although we initially underestimated the complexity of managing multiple ticket types, the process taught us how to design scalable, maintainable systems.
- The project has motivated us to explore more real-world blockchain applications and consider further work on enhancing this system post-course.

13 Future Work

While our project successfully implements a modular and functional NFT-based ticketing system, there remain several areas for future enhancement and exploration. These directions aim to increase scalability, real-world applicability, and user accessibility.

13.1 Event Capacity Management

To prevent overselling and enable more granular control, future versions can incorporate seat limits per ticket category (e.g., VIP, General, Early Bird). Contracts can be extended to track supply per category and reject minting once limits are reached, simulating real-world event capacity constraints.

13.2 Secure and Expirable QR Verification

Enhance QR-based verification by embedding expiration timestamps and off-chain digital signatures within the payload. This would reduce risks of QR reuse and support faster validation at event gates without requiring full on-chain queries during high-traffic periods.

13.3 Anti-Scalping and Bot Mitigation

Introduce minting constraints such as per-wallet limits, allowlist enforcement, and CAPTCHA-style Proof-of-Humanity checks. These measures would reduce bot-based bulk purchasing and scalping behavior, enhancing ticket fairness for real users.

13.4 Marketplace Enhancements

Future versions of the resale marketplace can incorporate advanced filters (by date, event, ticket type) and dynamic pricing mechanisms such as bidding or Dutch auctions. These features would reflect real-world market dynamics and improve user experience.

13.5 Fiat On-Ramp Integration

To improve accessibility for non-crypto users, the platform could integrate fiat payment systems like Stripe, MoonPay, or Coinbase Pay. This would allow users to purchase NFT tickets using credit/debit cards, reducing onboarding friction.

13.6 Cross-Chain NFT Ticketing

Extend functionality to other EVM-compatible blockchains such as Polygon, Arbitrum, or Optimism. Cross-chain minting and resale can be enabled using interoperability solutions or NFT bridging protocols, expanding the system's reach.

13.7 Post-Event Soulbound Badge System

Automatically mint Soulbound achievement badges for users who successfully check in with valid tickets. These badges can incentivize user loyalty and gamify participation across multiple events.

14 Conclusion

This project demonstrated how Ethereum smart contracts and NFTs can be used to build a decentralized ticketing system that addresses common issues like scalping, fraud, and resale control. By implementing four specialized ticket types and integrating IPFS, MetaMask, and QR-based verification, we created a functional and flexible prototype. Overall, the work highlights how blockchain principles can enhance transparency and programmability in real-world ticketing scenarios.

12 References

[1] Ethereum.org. "ERC-721 Token Standard." [Online]. Available: https://ethereum.org/en/developers/docs/standards/tokens/erc-721/

[2] Alchemy. "How to Create an NFT." [Online]. Available: https://www.alchemy.com/blog/how-to-create-an-nft

[3] Buterin, V. "Soulbound." [Online]. Available: https://vitalik.eth.limo/general/2022/01/26/soulbound.html

[4] Ethereum Improvement Proposal 5192. "Minimal Soulbound NFTs." [Online]. Available: https://eips.ethereum.org/EIPS/eip-5192

[5] Ethereum Improvement Proposal 2981. "NFT Royalty Standard." [Online]. Available: https://eips.ethereum.org/EIPS/eip-2981

[6] Alchemy. "What Are Dynamic NFTs?" [Online]. Available: https://www.alchemy.com/overviews/dynamic-nfts

[7] OpenZeppelin Docs. "ERC721URIStorage and Mutable Metadata." [Online]. Available: https://docs.openzeppelin.com/contracts/4.x/api/token/erc721#ERC721URIStorage

[8] MintGate. "How QR Codes Unlock NFT Access." [Online]. Available: https://mintgate.io/blog/how-qr-codes-unlock-nft-access

[9] Pinata. "Working with IPFS for NFTs." [Online]. Available: https://blog.pinata.cloud/working-with-ipfs-for-nfts/

[10] Ethereum Improvement Proposal 721 Metadata. "Metadata JSON Format." [Online]. Available: https://eips.ethereum.org/EIPS/eip-721#metadata

[11] LearnWeb3. "How NFT Royalties Work." [Online]. Available: https://learnweb3.io/degrees/essential-nfts/lessons/nft-royalties/