Cihan Aygün
211401010

Technical Design Document: Secure Event Ticketing DApp

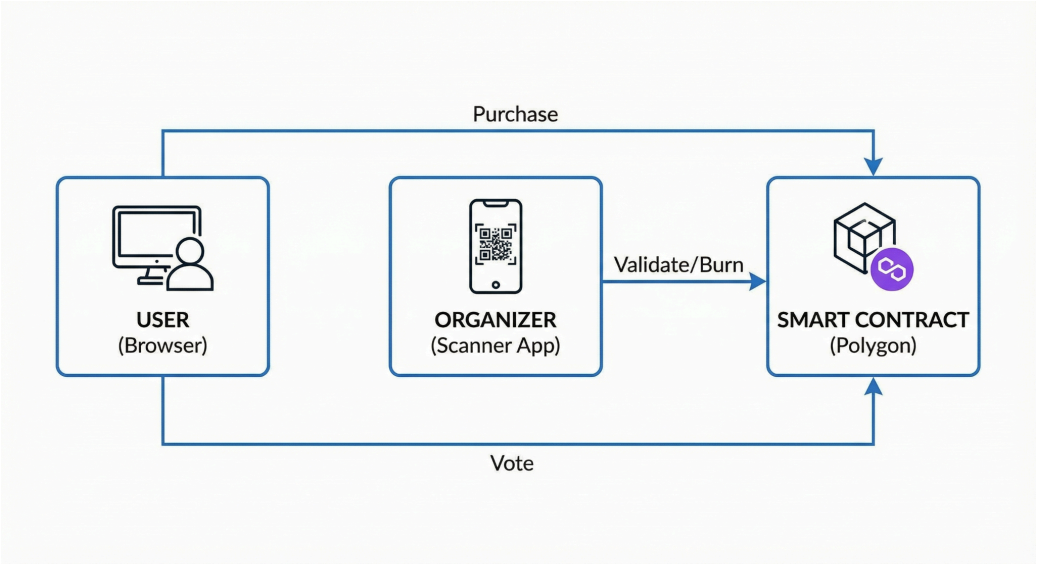
Network: Polygon (Amoy Testnet) **Standard:** ERC-721 (Soulbound Implementation)
Version: 1.0

1. Abstract

This application is a decentralized ticketing platform designed to ensure fairness, prevent scalping, and enforce organizer accountability. It utilizes the ERC-721 standard with modified transfer rules (implementing a **'Refundable Soulbound'** mechanism). While tickets are bound to the purchaser's wallet and cannot be traded between users, they explicitly allow transfers back to the Smart Contract to enable the trustless refund process. The system features a distinct "Trust Mechanism" where funds are locked in the smart contract until attendees vote on the event's quality.

2. System Architecture

The system consists of a Smart Contract deployed on Polygon, a Frontend Client for users, and a ‘Gatekeeper’ interface for the organizer to scan tickets and mark them as BURNED (status update, no transfer).



3. Smart Contract Specifications

3.1. Core Counters (EventStats)

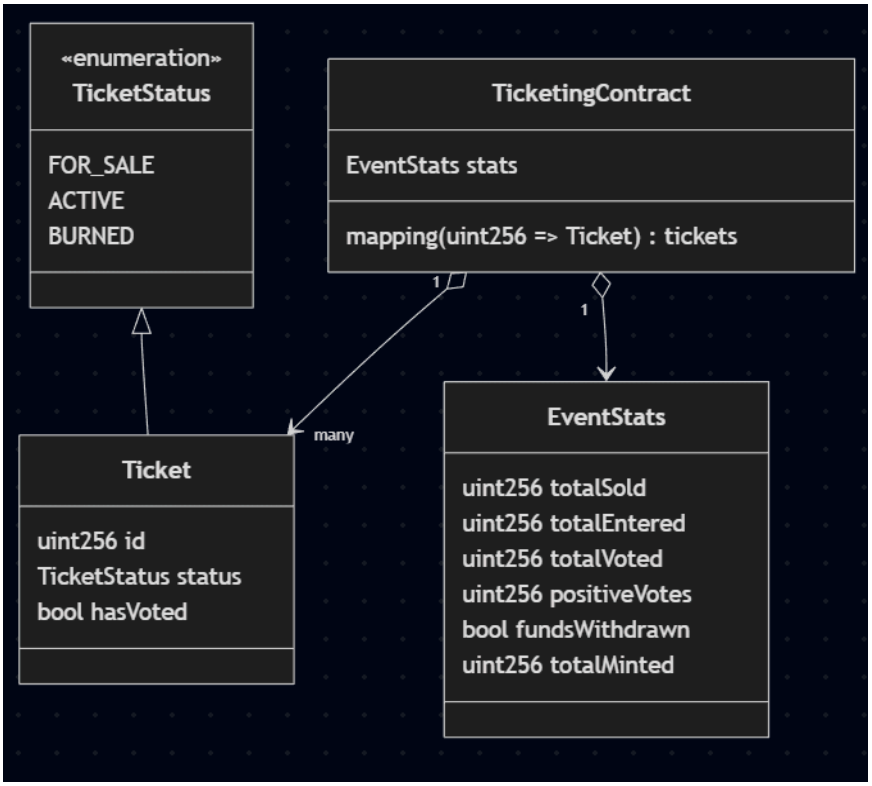
The contract tracks all event-level counters inside a single `EventStats` struct to keep state centralized and easy to query.

`EventStats public stats;` exposes the following fields:

- `stats.totalMinted`: Total number of tickets ever minted (monotonically increasing, bounded by `MAX_SUPPLY`).
- `stats.totalSold`: Number of currently sold, non-refunded tickets (can go up and down as tickets are minted, refunded, or re-sold).
- `stats.totalEntered`: Number of tickets that have been scanned for entry.
- `stats.totalVoted`: Number of **unique** tickets that have cast a vote.
- `stats.positiveVotes`: Number of votes where the user voted positively.
- `stats.fundsWithdrawn`: Boolean flag indicating whether event funds have already been finalized (either withdrawn by the organizer or burned).

3.2. Data Structures

A.Class Diagram



Note: *Since `TicketingContract` inherits from the ERC-721 standard, core methods like `balanceOf(owner)` are available by default. They are omitted from the diagram for brevity but are used to enforce wallet limits.*

B. Mappings & Global State

- mapping(uint256 => Ticket) public tickets; Maps each tokenId to its corresponding Ticket struct (status and hasVoted are stored per token).

  struct EventStats {
  uint256 totalMinted;
  uint256 totalSold;
  uint256 totalEntered;
  uint256 totalVoted;
  uint256 positiveVotes;
  bool fundsWithdrawn;}

  EventStats public stats;

  EventStats is the single source of truth for all aggregate counters. The contract does not duplicate these counters in separate state variables.

3.3. Core Function Interfaces (API Specs)

This section defines the public interface of the smart contract, detailing how the Frontend Client and Scanner App interact with the blockchain logic.

A. User Actions

1. `mintTicket()`

- **Description:** Allows a user to purchase a ticket. Accepts payment (`msg.value`) and assigns a valid ticket to the user.
- **Requirements:**
  - msg.value must equal TICKET_PRICE.
  - If the purchase is through the New Mint Route:
    - totalMinted must be strictly less than MAX_SUPPLY before the call.
    - The call will increment totalMinted and totalSold by one..
  - If the purchase is through the Resale Route:
    - No new tokens are minted; totalMinted MUST remain unchanged.
    - MAX_SUPPLY is NOT checked for this route, because the ticket already exists.
    - totalSold is still incremented, since a new sale has occurred.
  - User's balance must not exceed MAX_PER_WALLET after purchase.

- **State Transition:** Changes status from FOR_SALE (Default/Resale) to ACTIVE.

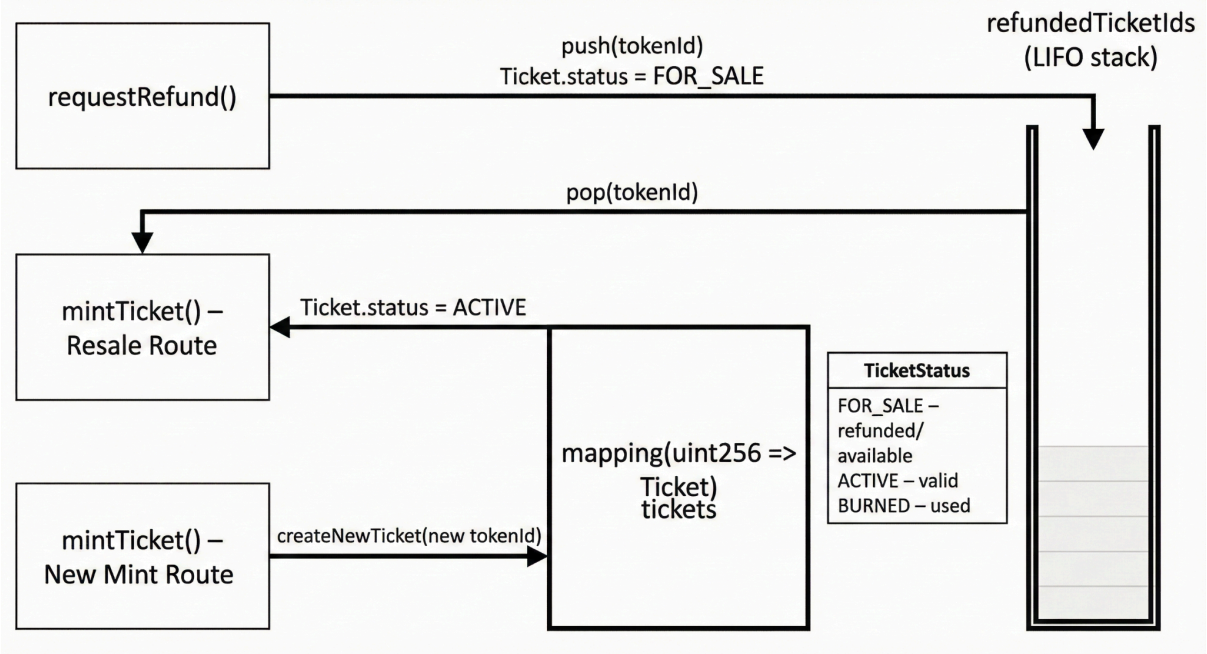
- **Logic:**
  - **Check Resale Pool:** Checks if there are any tickets in the "Refunded/Resale Stack" (tickets previously returned to the contract).
  - **Resale Route:** If yes, pops the last returned `tokenId`, transfers it from the Contract to the User, and updates status to `ACTIVE`.
  - **New Mint Route:** If no **(and totalMinted < MAX_SUPPLY)**, increments the `totalMinted` counter, mints a brand new ticket to the User, and sets status to ACTIVE.
- **Signature:** function mintTicket() external payable returns (uint256 tokenId);

2. `requestRefund(uint256 tokenId)`

- **Description:** Allows users to request a refund up to 6 hours before the event (`refundDeadline`).
- **Requirements**:
  - `msg.sender` must be the current owner (`ownerOf(tokenId)`)
  - `tickets[tokenId].status` MUST be `ACTIVE`; refunded or burned tickets are not refundable
  - Reverts with `RefundPeriodExpired` if `block.timestamp>=refundDeadline`
- **Logic:**
  - Verifies the refund window is open.
  - Transfers the token from the User to the Contract address.
  - Resets the ticket status to `FOR_SALE`.
  - Pushes the `tokenId` into the `refundedTicketIds` array (Making it available for the next buyer).
  - Refunds `TICKET_PRICE` to the user.
  - Decrement `totalSold` by 1 (`totalMinted` remains unchanged)
- **Signature:** function requestRefund(uint256 tokenId) external;

3. `vote(uint256 tokenId, bool isPositive)`

- **Description:** Casts a vote on the event quality within the post-event voting window.
- **Requirements:**
  - msg.sender must be the owner of tokenId.
  - The ticket's status for tokenId must be BURNED (proof of attendance).
  - hasVoted for this tokenId must be false.
  - block.timestamp <= VOTING_DEADLINE (within the allowed voting window).
- **Signature:** function vote(uint256 tokenId, bool isPositive) external;



B. Organizer & System Actions

1. `scanTicket(uint256 tokenId)`

- **Description:** Validates and consumes a ticket at the venue entrance. This function is used by the Organizer's Gatekeeper system to confirm that a ticket is valid for entry and within the allowed time window.
- **Access Control:** Can only be called by the Organizer or a designated Gatekeeper address. Unauthorized callers MUST revert according to the contract's access control rules.

- **Effects (on success)**
  - Confirms that the ticket is valid and eligible for entry.
  - Changes the ticket status from ACTIVE to BURNED (Soft Burn), marking it as entered.
  - Increments the global counter totalEntered.
  - Token ownership remains with the user; the ticket is not transferable, but is logically "consumed" for event entry.
- **Failure Conditions (reverts):**
  - If block.timestamp > ENTRY_DEADLINE, the function MUST revert with EntryPeriodExpired().
  - If the given tokenId does not correspond to an existing ticket, the function MUST revert with InvalidTicketId(tokenId).
  - If the ticket has already been scanned / entered before, the function MUST revert with TicketAlreadyUsed(uint256 tokenId).
- **Signature:** function scanTicket(uint256 tokenId) external returns (bool valid);

2. `withdrawFunds()`

- **Description:** Sets stats.fundsWithdrawn to true and transfers the contract balance to the Organizer if the event was successful.
- **Condition:** (totalEntered >= (totalSold * 30 / 100)) AND ((totalVoted == 0) OR (positiveVotes > totalVoted/2)) AND (stats.fundsWithdrawn == false)
- **Signature:** function withdrawFunds() external onlyOwner;

3. `burnFunds()`

- **Description:** Triggers the "Game Theory" penalty mechanism if the event fails.
- **Condition:**((totalEntered < (totalSold * 30 / 100)) OR (totalVoted > 0 && positiveVotes <= (totalVoted/2))) AND (stats.fundsWithdrawn == false).
- **Effect:** Sets stats.fundsWithdrawn to true and sends the entire contract balance to the 0xDead address.
- **Signature:** function burnFunds() external;

C. Events & Errors

Type	Name	Description
Event	**TicketMinted(address indexed to, uint256 tokenId)**	Emitted upon successful purchase.
Event	**TicketBurned(uint256 tokenId, uint256 timestamp)**	Emitted when a ticket is scanned/used.
Error	**SoulboundTransferBlocked()**	Reverts for any transferFrom / safeTransferFrom call where neither from nor to is the contract itself
Error	**WalletLimitExceeded(uint256 held)**	Reverts if mintTicket causes the user's balance to exceed MAX_PER_WALLET (2).

Error	**SupplyExhausted()**	Reverts if totalMinted reaches MAX_SUPPLY.
Error	**RefundPeriodExpired(uint256 current, uint256 deadline)**	Reverts if requestRefund is called less than 6 hours before the event.
Error	**TicketAlreadyUsed(uint256 tokenId)**	Reverts if scanTicket is called on a ticket with BURNED status.
Error	**UnauthorizedAccess()**	Reverts if a restricted function (e.g., scanTicket, withdrawFunds) is called by a non-organizer address.
Error	**FundsAlreadyProcessed()**	Reverts if `withdrawFunds` or `burnFunds` is called when `stats.fundsWithdrawn` is already true.
Error	**`AlreadyVoted(uint256 tokenId)`**	Reverts if the specific ticket has already been used to cast a vote (`hasVoted` is true).
Error	**`VotingPeriodExpired(uint256 current, uint256 deadline)`**	Reverts if `vote()` is called after the `VOTING_DEADLINE`.
Error	**`VoteEligibilityFailed( uint256 tokenId)`**	Reverts if the caller is not the owner of the token OR if the ticket status is not `BURNED` (proof of attendance).

Error	`EntryPeriodExpired()`	Thrown when scanTicket is called after the ENTRY_DEADLINE has passed.
Error	`InvalidTicketId(uint256 tokenId)`	Thrown when `scanTicket` is called with a tokenId that does not exist.
Error	`AttendanceThresholdNotMet(uint256 entered, uint256 required)`	Reverts if `withdrawFunds` is called when `totalEntered` is less than 30% of `totalSold`.

3.4. Token Metadata & Storage Standards

Since storing large amounts of data (images, descriptions) directly on the Polygon blockchain is prohibitively expensive, we utilize a decentralized off-chain storage solution. This ensures that ticket visuals and details are immutable and permanent, aligning with the project's trustless ethos.

A. Storage Strategy: IPFS (InterPlanetary File System)

- **Provider:** We will use IPFS (via pinning services like Pinata or NFT.Storage) to host ticket metadata and assets.
- **Immutability:** Once the `tokenURI` is set during minting, the content on IPFS cannot be altered by the organizer, preventing "rug pulls" or metadata manipulation.

B. Metadata JSON Schema (ERC-721 Compliant)

The contract implements `ERC721URIStorage`. Each ticket will point to a JSON file on IPFS adhering to the OpenSea metadata standard.

Example structure for a Ticket:

```
{
  "name": "Summer Tech Fest 2024 - General Admission #1023",
  "description": "Valid entry ticket for the Summer Tech Fest. This ticket is Soulbound and c
  "image": "ipfs://QmYourImageHashHere/ticket_visual.png",
  "external_url": "https://your-dapp-url.com/event/1",
  "attributes": [
    {
      "trait_type": "Event Date",
      "display_type": "date",
      "value": 1715600000
    },
    {
      "trait_type": "Venue",
      "value": "Convention Center Hall A"
    },
    {
      "trait_type": "Ticket Type",
      "value": "Standard"
    },
    {
      "trait_type": "Status",
      "value": "Valid"
    }
  ]
}
```

C. Dynamic vs. Static Metadata

- **Initial State:** Upon minting, the metadata `image` displays a generic "Valid Ticket" QR/Artwork.
- **Post-Burn (Optional):** While the on-chain status changes to `BURNED` immediately upon scanning, the metadata remains static to preserve the "souvenir" aspect of the NFT in the user's wallet (e.g., "I was there").

3.5. Error Handling & Gas Optimization

To optimize gas usage and reduce deployment costs, the smart contract replaces traditional string-based error messages (e.g., `require(..., "Error Message")`) with **Solidity Custom Errors**. Custom errors are significantly cheaper to deploy and execute because they do not require storing and emitting string data on-chain.

A. Error Code Definition Table

The custom errors summarized in Section 3.3.C are defined in the contract interface to handle exceptions efficiently and reduce gas costs. Section 3.3.C is the single source of truth for the error list and their high-level descriptions; this section focuses on how and where these errors are used within the contract functions.

B. Usage Overview

• WalletLimitExceeded(uint256 held)

- Thrown by `mintTicket()` when the caller attempts to exceed `MAX_PER_WALLET`.

• SupplyExhausted()

- Thrown by `mintTicket()` when totalMinted reaches `MAX_SUPPLY`.

• RefundPeriodExpired(uint256 current, uint256 deadline)

- Thrown by `requestRefund()` when called after the refund cutoff time.

• SoulboundTransferBlocked()

- Thrown by any transferFrom/ safeTransferFrom call that is not part of an allowed internal flow (e.g., refund or contract-initiated resale).

• TicketAlreadyUsed(uint256 tokenId)

- Thrown by `scanTicket()` when the ticket status is already `BURNED`.

• UnauthorizedAccess()

- Thrown by organizer-only functions (e.g., `scanTicket`, `withdrawFunds`) when called by a non-organizer address.

• FundsAlreadyProcessed()

- Thrown by `withdrawFunds()` or `burnFunds()` if the contract balance has already been distributed or burned (preventing double-spending or reentrancy).

• AlreadyVoted(uint256 tokenId)

- Thrown by `vote()` when checking the `hasVoted` boolean flag for the given ticket ID.

• VotingPeriodExpired(uint256 current, uint256 deadline)

- Thrown by `vote()` if `block.timestamp` is greater than `VOTING_DEADLINE`.

• VoteEligibilityFailed(uint256 tokenId)

- Thrown by `vote()` if `msg.sender` is not the token owner (`ownerOf` check) OR if the ticket's internal status is not `BURNED` (ensuring only actual attendees can vote).

• **AttendanceThresholdNotMet(uint256 entered, uint256 required)**

- Thrown by `withdrawFunds()` if the organizer attempts to withdraw before scanning at least 30% of the sold tickets.

C. Implementation Example

By using custom errors, we reduce the opcode overhead. Below is a comparison of the legacy approach vs. our optimized implementation:

Legacy Approach (High Gas):

```
// Uses expensive storage for the string

require(block.timestamp < refundDeadline, "Refund period has expired");
```

Optimized Approach (Our Implementation):

```
// Uses efficient 4-byte selector

if (block.timestamp >= refundDeadline) {

   revert RefundPeriodExpired(block.timestamp, refundDeadline);

}
```

4. Feature Implementation Logic

4.1. Ticketing & Supply (The "One-Time" Rule)

- **Logic:** The contract utilizes a `constructor` to define the `MAX_SUPPLY`. There will be **no** administrative `mint` function exposed after deployment.
- **Supply Counters Definitions:**
  - **`MAX_SUPPLY`:** The hard cap on unique Token IDs (e.g., 100). ID #101 can never exist.
  - **`totalMinted`:** Starts at 0, increments only when a *new* ticket is created. Checked against `MAX_SUPPLY`.
  - **`totalSold`:** Tracks active holders. Decrements on refund, increments on purchase.
  - *Example:* If 100 tickets are minted (`totalMinted`=100) and 1 user refunds (`totalSold`=99), a new buyer receives the refunded ID. No new ID is minted (`totalMinted` remains 100), preserving the hard cap.
- **Fairness:** To ensure everyone has an equal chance, we avoid whitelists for the general sale.
- **Restriction:** The contract enforces limits using the standard ERC-721 `balanceOf(msg.sender)` function. If the user's current balance plus the mint amount exceeds MAX_PER_WALLET, the transaction reverts.

4.2. Conditional Soulbound Mechanics (Non-Transferable)

***Note**: Unlike a pure SBT which is permanently locked, this implementation permits transfer **only** if the recipient (`to` address) is the Smart Contract itself. This exception is strictly for processing refunds.*

- **Logic:** Standard transfer functions (transferFrom, safeTransferFrom) are overridden to enforce soulbound behavior for tickets.
- **Exceptions (Critical)**: Transfers are allowed only in the following cases:
    - from address is the Smart Contract (reselling a refunded ticket from the contract back to a new buyer).
    - to address is the Smart Contract (processing a refund by transferring the ticket back to the contract).
    - All other transfers MUST revert with the SoulboundTransferBlocked() error.
- **Implementation:**
    - The contract overrides the following ERC-721 transfer functions to enforce the soulbound rule:
        - transferFrom(address from, address to, uint256 tokenId)
        - safeTransferFrom(address from, address to, uint256 tokenId)
        - safeTransferFrom(address from, address to, uint256 tokenId, bytes data)
    - Each of these functions only allows a transfer when`to == address(this)` or `from == address(this)`.For all other (from, to) combinations, the call MUST revert with `SoulboundTransferBlocked()`.
    - In addition, the approval functions are also overridden:
        - approve(address to, uint256 tokenId)
        - setApprovalForAll(address operator, bool approved)

    Note: These functions also revert with `SoulboundTransferBlocked()`, ensuring that no external operator can obtain transfer rights over the ticket.

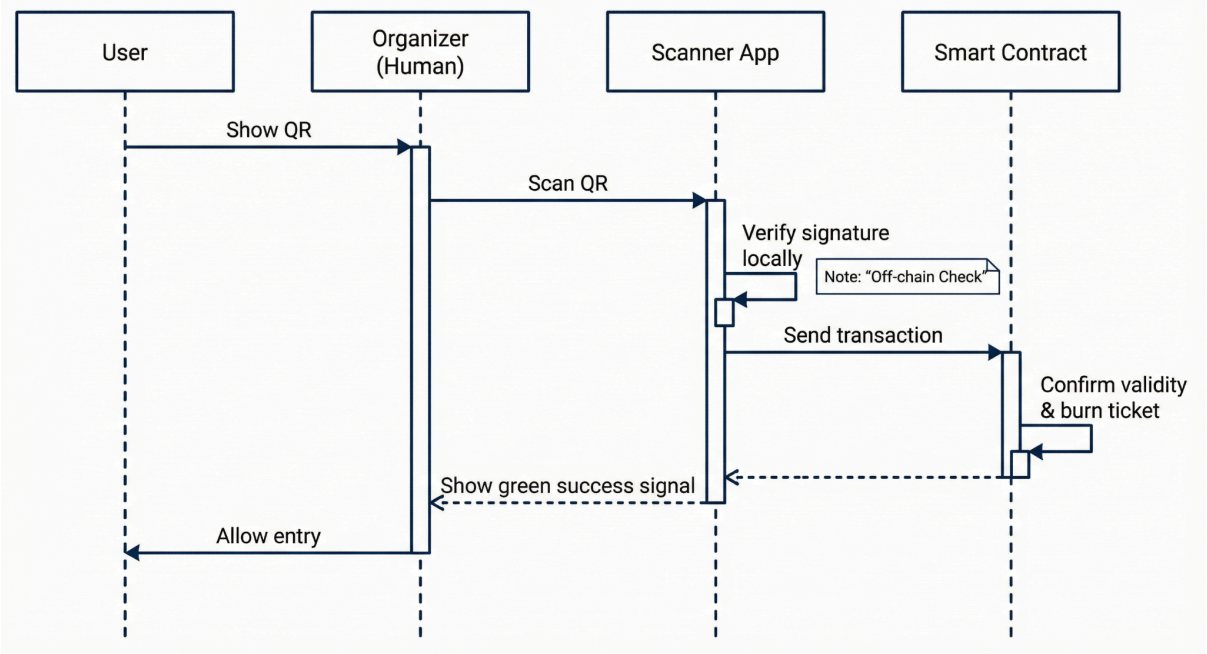
    - As a result, tickets cannot be transferred between externally owned accounts; they can only be "returned" to the contract as part of the refund flow.

4.3. Refund Mechanism

- **Condition:** `block.timestamp < refundDeadline`.
- **Action:** User calls `requestRefund(tokenId)`.
- **Result:** The token ownership is transferred back to the contract, making it available for future minting/purchase. TICKET_PRICE is refunded.

4.4. Entry & Soft Burning

- **Condition:** `block.timestamp <= ENTRY_DEADLINE`.
- **Role:** Only the **Organizer** (or a designated `Gatekeeper` role) can call the `scanTicket(tokenId)` function.
- **Logic:**
    1. First, the contract enforces the entry time window:
        - If block.timestamp > ENTRY_DEADLINE, the call MUST revert with the EntryPeriodExpired() error.
        - In this case, no state changes are performed (the ticket is not marked as entered, and counters are not incremented).
    2. Next, the contract validates that the ticket actually exists:
        - If the underlying ERC-721 token does not exist for the given tokenId (e.g., _exists(tokenId) == false), the call MUST revert with the InvalidTicketId(tokenId) error.
        - This prevents arbitrary or malformed QR codes from being accepted for non-existent tickets.
    3. Verify that tokenId corresponds to an existing ticket and that its status is ACTIVE (i.e., not already BURNED).
    4. Burn the ticket by updating its status from ACTIVE to BURNED in the internal mapping.
    5. Increment the totalEntered counter (crucial for the voting logic).



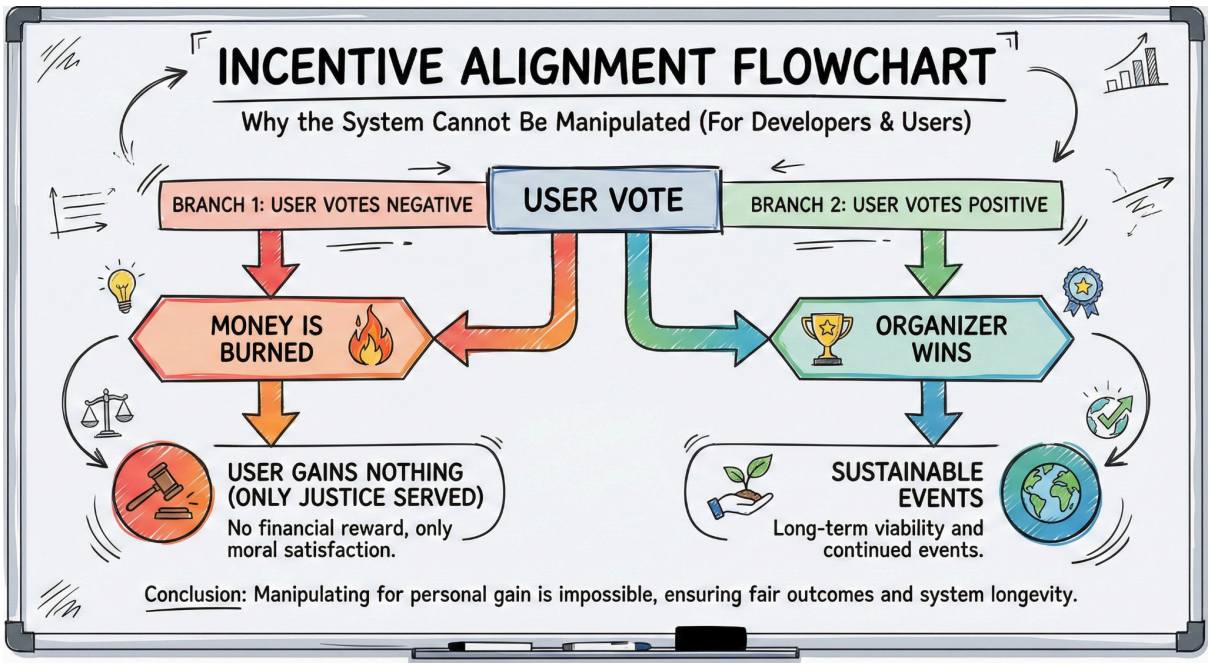
4.5. Post-Event Voting (The Trust Mechanism)

- **Voter Eligibility:** Only addresses holding a `BURNED` ticket (proven attendees) can vote.
- **Voting Window:** Votes are accepted only within the post-event voting window, i.e., while block.timestamp <= VOTING_DEADLINE (typically EVENT_TIMESTAMP + 24 hours).
- **Vote Weight:** 1 Ticket = 1 Vote.
- **Sybil Resistance:** The contract tracks the `hasVoted` boolean for each `tokenId`. Attempting to vote again with the same ticket reverts the transaction.

4.6. Fund Distribution Logic (Game Theory Security)

To prevent "profit-seeking" negative voting by attendees (colluding to get a refund after attending), the contract enforces a **"Burn on Failure"** policy.

- **Success Condition:**
  - **`If:`**`totalEntered >= (totalSold * 30 / 100)` AND (`totalVoted == 0` OR `positiveVotes > totalVoted/2`):
  - **Action:** Organizer calls `withdrawFunds()`.
  - **Requirement:** The organizer acts as the gatekeeper. To prevent them from blocking attendees to manipulate the vote, they MUST scan and admit at least 30% of the sold tickets. If they fail to do so, they cannot withdraw funds.
- **Failure Condition:**
  - **If:** `totalEntered < (totalSold * 30 / 100)` OR (`totalVoted > 0` AND `positiveVotes <= totalVoted/2`):
  - **Action:** Anyone can trigger `burnFunds()`.
  - **Logic:** The funds are burned if the event quality is poor (negative votes) OR if the organizer acted maliciously/negligently by failing to admit at least 30% of the ticket holders (Low Attendance Penalty).
  - **Result:** The entire MATIC balance held by the contract is sent to the `0x0000...dEaD` (Burn Address).
  - **Outcome:** Neither the Organizer nor the Attendees get the money. This ensures negative votes are cast only for genuine dissatisfaction, not financial gain.



5. Security Considerations

1. **Reentrancy Protection:**The contract inherits from OpenZeppelin's ReentrancyGuard and applies the nonReentrant modifier to all functions that move funds out of the contract or update user-facing balances. In particular, requestRefund(...), withdrawFunds(), and burnFunds() are marked as nonReentrant. This ensures that any attempt to re-enter these functions in the same transaction will revert, reducing the attack surface around refund and withdrawal flows. The implementation follows the Checks-Effects-Interactions pattern: all state checks and updates are performed before any external value transfer.
2. **Timestamp Manipulation:** Miners / validators can manipulate block timestamps slightly, within a narrow range of seconds. The contract only uses `block.timestamp` to enforce three coarse-grained time windows: `ENTRY_DEADLINE` for ticket purchases, `REFUND_DEADLINE` for refunds, and `VOTING_DEADLINE` for closing the voting phase (e.g., 90-minute, 6-hour, or similar multi-hour windows). This small possible drift is negligible for these use cases and is considered acceptable. The contract does not use `block.timestamp` for randomness, price discovery, or any high-sensitivity logic where small manipulation would be critical.
3. **Gas Limits and Complexity:** All critical user-facing functions (including mintTicket, scanTicket, requestRefund, withdrawFunds, and burnFunds) are designed with O(1) time complexity and do not iterate over unbounded lists of users or tickets. Aggregated counters (such as totalMinted, totalSold, totalEntered, and vote-related counters) are updated with constant-time operations, avoiding any risk of running out of gas due to large loops. Organizer-only operations such as withdrawFunds and burnFunds also perform a bounded, constant amount of work per call, ensuring they remain executable even as the number of tickets grows.

4. **Anti-Spoofing & Replay Protection:** To prevent attackers from guessing sequential Ticket IDs (e.g., generating a fake QR for ID #11), the system requires a **Cryptographic Signature**. The QR code is dynamic; it includes a timestamp and is signed by the wallet that owns the ticket. The Organizer App verifies this signature off-chain before interacting with the contract, ensuring that the person presenting the QR code is the actual owner of the wallet. Even if an attacker can guess a valid tokenId, they cannot produce a valid signature without the private key of the owning wallet.

6. Test Strategy & Quality Assurance

This section outlines the comprehensive testing strategy used to validate the system's integrity, ensuring that the Smart Contract, Frontend DApp, and the Organizer's Scanner App function correctly under both normal and malicious conditions.

6.1. Smart Contract Unit Testing (Hardhat/Foundry)

The core logic deployed on the Polygon Amoy Testnet is tested using a suite of automated unit tests to ensure compliance with the specifications defined in Section 3.

- **Supply & Minting Logic:**
  - **Cap Enforcement:** Verify that `mintTicket()` reverts with `SupplyExhausted()` once `totalMinted` reaches `MAX_SUPPLY`.
  - **Wallet Limits:** Verify that a single wallet cannot mint more than `MAX_PER_WALLET` tickets.
  - **LIFO Resale Queue:** Test that minting prioritizes popping from the `refundedTicketIds` stack before incrementing the `totalMinted` counter.
- **Soulbound Enforcement:**
  - **Transfer Blocking:** Attempt `transferFrom` and `safeTransferFrom` between two regular user wallets. Assert that the transaction reverts with `SoulboundTransferBlocked()`.
  - **Allowed Exceptions:** Verify that transfers are **only** successful when the `from` or `to` address is the Contract Address (handling Refunds and Resales).
- **Refund Mechanics:**
  - **Time Constraints:** Attempt `requestRefund()` after `REFUND_DEADLINE` to ensure it reverts with `RefundPeriodExpired`.
  - **State Updates:** Confirm that a successful refund resets the ticket status to `FOR_SALE` and pushes the ID to the `refundedTicketIds` array.
- **Game Theory & Fund Distribution:**
  - **Withdrawal Condition:** Mock a scenario where `totalEntered` < 30% of `totalSold`. Verify `withdrawFunds()` reverts with `AttendanceThresholdNotMet`.
  - **Burning Logic:** Simulate a scenario with majority negative votes. Verify `burnFunds()` sends the entire balance to the `0xDead` address.

6.2. QR Scanner & Off-Chain Verification Tests

The Organizer's Scanner App (Section 7.3) relies on a hybrid off-chain/on-chain validation model. This is critical for preventing entry fraud.

- **Cryptographic Signature Verification (Off-Chain):**
  - **Valid Signature:** Generate a QR payload signed by the ticket owner. Verify the scanner correctly recovers the signer's address.
  - **Spoofing Attack:** Create a QR payload with valid JSON data but sign it with a different wallet (non-owner). Assert that the scanner rejects the entry locally (Red Signal).
  - **Data Integrity:** Modify the `tokenId` in the JSON payload after signing. Verify that the signature recovery fails or mismatches the owner.
- **Replay Attack Prevention (Timestamp Check):**
  - **Expired QR:** Present a valid QR code where the `timestamp` is older than the allowed threshold (e.g., > 600 seconds). Assert the scanner rejects it as a "Stale/Replayed QR" to prevent screenshots from being shared.
- **On-Chain Interaction (`scanTicket`):**
  - **Status Check:** Scan a ticket that has already been scanned (`status == BURNED`). Verify the transaction reverts with `TicketAlreadyUsed`.
  - **Non-Existent ID:** Attempt to scan a `tokenId` that has not been minted. Verify it reverts with `InvalidTicketId`.
  - **Success Flow:** Scan a valid `ACTIVE` ticket. Confirm the transaction succeeds, the status changes to `BURNED`, and `totalEntered` increments by 1.

6.3. Frontend & Integration Testing

These tests ensure the User DApp (Browser) interacts correctly with the wallet and the blockchain.

- **Network Guard:**
  - Connect with a wallet set to Ethereum Mainnet. Verify that the UI triggers the `useSwitchChain` hook, prompting the user to switch to Polygon Amoy.
- **Voting Eligibility:**
  - Connect a wallet that holds a ticket with status `ACTIVE` (not scanned/burned). Verify that the voting UI is disabled or reverts if the user attempts to call `vote()`.
  - Connect a wallet with a `BURNED` ticket. Verify the user can cast a vote and that the `hasVoted` flag updates to `true` immediately to prevent double voting.

6.4. Security & Attack Vector Simulation

- **Reentrancy:** Attempt a reentrancy attack on the `requestRefund` function. Verify that the `nonReentrant` modifier blocks the recursive call.

- **Organizer Malpractice:** Simulate an organizer attempting to call `withdrawFunds()` before the event ends or without scanning attendees. Ensure the contract locks the funds.
- **Front-Running:** Verify that ticket metadata is pinned to IPFS immutably, ensuring the organizer cannot swap the "ticket image" for malicious content after the sale.

7. Frontend & Integration Architecture

This section details the client-side architecture for the two distinct interfaces defined in the system: the **User DApp** (for purchasing/voting) and the **Organizer Gatekeeper App** (for validating tickets).

7.1. Core Tech Stack

To ensure a responsive, type-safe, and mobile-friendly experience, the following libraries and frameworks will be utilized:

- **Framework: Next.js (React)** – Chosen for its server-side rendering capabilities and routing efficiency.
- **Styling: Tailwind CSS** – For rapid UI development and responsive design (crucial for the mobile scanner view).
- **Blockchain Interaction:**
    - **Wagmi:** A set of React Hooks for Ethereum. It handles the lifecycle of blockchain transactions (loading states, error handling) much more efficiently than raw Ethers.js.
    - **Viem:** Used as the low-level interface for JSON-RPC requests (replacing Ethers.js for lighter bundle size and speed).
- **Wallet UI: RainbowKit** – Provides a polished, plug-and-play wallet connection modal that supports Metamask, Coinbase Wallet, and WalletConnect.

7.2. Wallet Connection Strategy

Since the Smart Contract is deployed on the **Polygon Amoy Testnet**, the frontend must enforce specific network constraints:

1. **Connection Provider:** We utilize **WalletConnect** protocol via RainbowKit to ensure users can connect mobile wallets (like MetaMask Mobile or Trust Wallet) easily by scanning a QR code on the desktop DApp.
2. **Network Switching:**
    - The application implements a "Wrong Network" guard.
    - If a user connects via Ethereum Mainnet, the `wagmi` hook `useSwitchChain()` is automatically triggered to prompt the user to switch to **Polygon Amoy**.

7.3. The "Gatekeeper" Scanner App (Organizer View)

The Organizer interface is a specialized mobile-web view designed for speed at the venue entrance.

- **QR Scanning Library:** `react-qr-reader` (or `html5-qrcode`).
- **Workflow:**
    1. The library accesses the device camera and decodes the QR JSON payload.
    2. Off-Chain Verification (Anti-Spoofing):
        - The App recovers the signer's address from the `signature` using the `id` and `timestamp`.
        - It compares the `timestamp` with the current time (e.g., must be generated within the last 600 seconds) to prevent **Replay Attacks** (screenshots)
    3. On-Chain Ownership Check:
        - The App queries the smart contract: ownerOf(id).
        - It verifies that Recovered_Address == Owner_Address.
    4. Transaction Execution:
        - ONLY IF the signature is valid and matches the owner, the app constructs and sends the `scanTicket(tokenId)` transaction to the blockchain.
    5. Feedback UI: Returns Green (Valid) or Red (Invalid Signature / Already Used).

7.4. QR Code Data Protocol

To generate the ticket visual for the user:

- **Library:** `qrcode.react`.
- **Payload:** The QR code contains a JSON object signed by the user's wallet. The signature is **MANDATORY** to prevent ID spoofing.

```
{
  "id": 102,
  "timestamp": 1715601234,
  "signature": "0xabc123... (signed message hash)"
}
```

Note: *The signature is generated by the user's wallet signing a hash of* `(id, timestamp, contractAddress)`*. This proves ownership without revealing the private key.*