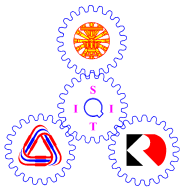

SCHOOL OF INFORMATION, COMPUTER AND COMMUNICATION TECHNOLOGY
SIRINDHORN INTERNATIONAL INSTITUTE OF TECHNOLOGY
THAMMASAT UNIVERSITY

Capstone Project

Nannapas Sitthimongkol	6322772102
Phatwasin Suksai	6322772402
Thritsadi Chukiatsiri	6322773175
Kednipa Prasongsri	6322774439
Methus Sukhon	6322775288

Presented to

Dr. Watthanasak Jeamwatthanachai

DES484 BLOCKCHAIN DEVELOPMENT

List of Figures

Table of Contents

Abstract

This report presents a comprehensive case study on the problems associated with traditional event ticketing systems and the transformative role of blockchain technology, particularly through the use of Smart Contracts and Non-Fungible Tokens (NFTs), in addressing these challenges. Traditional ticketing systems are fraught with issues like ticket black markets, lack of exchange protocols, and customer trust concerns, leading to fraud, inflated prices, and security risks. This study explores how smart contracts provide a secure, transparent, and tamper-proof solution by automating the ticketing process, while NFTs offer a unique, verifiable, and secure form of ticketing that reduces costs, prevents fraud, and opens up new revenue streams.

Key benefits of NFT ticketing, such as preventing fake tickets and scams, reducing costs, quick production, perpetual revenue, and new revenue opportunities, are discussed in detail. The report also delves into the functional and non-functional requirements of an e-ticketing system, including event management, ticketing, financial transactions, security, usability, and compliance. The design architecture covers the overall system flow, smart contract development on the Ethereum Goerli Testnet, and front-end development focusing on user interface design and user experience.

Finally, the report outlines the testing and deployment strategies, including unit testing for smart contracts, integration testing, and end-to-end testing, emphasizing the importance of these processes in ensuring the reliability and efficiency of the e-ticketing system. This case study provides valuable insights into the potential of blockchain technology in revolutionizing the event ticketing industry, offering a secure, efficient, and user-friendly alternative to traditional systems.

1. Case Study

Problems in Traditional Event Ticketing

1. **Ticket Black Market**

The ticketing industry is plagued by fraud, leading to inflated prices for genuine fans and posing security risks for event organizers. Technology misuse by ticket bots allows individuals to buy tickets in large quantities online, leading to reselling at higher rates on secondary markets.

2. **Lack of Exchange Protocol**

Existing systems lack the capability to track customers or transfer data seamlessly between ticket suppliers, allowing fraudulent activities to thrive in secondary markets. As tickets are resold, the original purchaser's ownership isn't retained, leaving event organizers unaware of attendees' identities or ticket origins.

3. **Customer Trust**

The authenticity of purchased tickets poses a significant concern for guests. Fake tickets, often obtained from deceptive websites posing as authorized sellers, not only lead to financial losses but also result in disappointment and distrust among consumers.

Smart Contract

Smart Contracts are self-executing contracts with the terms of the agreement directly written into lines of code. They operate on blockchain technology, ensuring that they are secure, transparent, and tamper-proof. In the context of ticketing, smart contracts automate the ticketing process, ensuring that once a ticket is purchased, the contract is executed automatically, issuing the ticket to the buyer without the need for an intermediary.

Non-fungible tokens

Instead of traditional paper or digital tickets, NFTs are used to represent ownership or access to events, concerts, shows, or experiences. Each NFT ticket is unique and stored on a blockchain, providing proof of ownership and authenticity. NFT ticketing aims to revolutionize ticketing by offering verifiable, secure, and scarce digital tickets that cannot be replicated or counterfeited. These tokens often allow holders to access specific events and can also serve as collectible digital memorabilia, potentially holding value beyond the event itself.

Benefits of NFT Ticketing

1. **Preventing fake tickets and scams**

Blockchain provides a trusted source for both ticket holders and organizers. The transfer of NFTs from the initial sale to resale is stored on the blockchain immutably so that all parties can prove the ticket's authenticity. In cases where the resale of tickets is forbidden, NFTs can be developed as nontransferable, not to be moved to another buyer.

2. **Reduce costs**

Costs associated with selling and minting NFTs are negligible compared to the traditional ticketing system. You can produce an unforgeable ticket for less production cost, and customers and organizers can validate the authenticity of every ticket on the chain and track the history of ownership.

3. **Quick production**

Contrary to the traditional way of digitizing tickets, the approach to creating and minting NFTs in minimal time. NFT can be minted and ready to put for sale in less than a minute.

4. **Perpetual revenue**

Because programmable NFTs can have built-in rules for merchandise, content, resale, and royalty splits, it means that the organizer can analyze profit sharing percentages for future resale or creative content on secondary markets and receive funds knowing the tickets are unalterable within the NFT's coding.

5. **New revenue opportunities**

NFT-based tickets act as programmable money, providing unlimited potential for new revenue opportunities, for example, the resale of NFT tickets as collectibles, using NFT tickets to provide food and drink deals, and rewarding fans who have gathered many event tickets.

2. Key Requirements

2.1 Functional Requirements

Event Management

- **Create Events:** Allow the contract owner to create events with details like name, date, location, ticket limit, and ticket price.
- **Store Event Images on IPFS:** Use Pinata to store event cover images on IPFS and record their URIs in the event details.

Ticketing

- **Ticket Purchase:** Enable users to purchase tickets for available events, ensuring the payment matches the ticket price.
- **Issue Tickets as NFTs:** Mint NFT tickets using the ERC721 standard upon purchase.
- **Ticket Usage:** Allow the contract owner to mark tickets as used.

Event and Ticket Visibility

- **View Events:** Provide functions to view details of all events, open events, and specific event details.
- **View User Tickets:** Enable users to view their owned tickets.

Financial Transactions

- **Withdrawal of Funds:** Enable the contract owner to withdraw ETH collected from ticket sales.

Event Status Control

- **Toggle Event Status:** Allow the contract owner to open or close events for ticket sales.

Data Retrieval

- **Retrieve Event Images:** Facilitate retrieval of event images from IPFS for display in user interfaces.

2.2 Non-Functional Requirements

Security

- **Non-Reentrancy:** Implement non-reentrancy in functions involving financial transactions to prevent reentrancy attacks.
- **Access Control:** Restrict sensitive operations (like creating events, withdrawing funds) to the contract owner.

Usability

- **User-Friendly Interfaces:** Develop interfaces for easy interaction with the contract, including event creation, ticket purchasing, and viewing events and tickets.

Compliance

- **ERC721 Compliance:** Ensure that the NFT ticketing system fully adheres to the ERC721 token standard.

3. Design Architecture

Overall System Flow

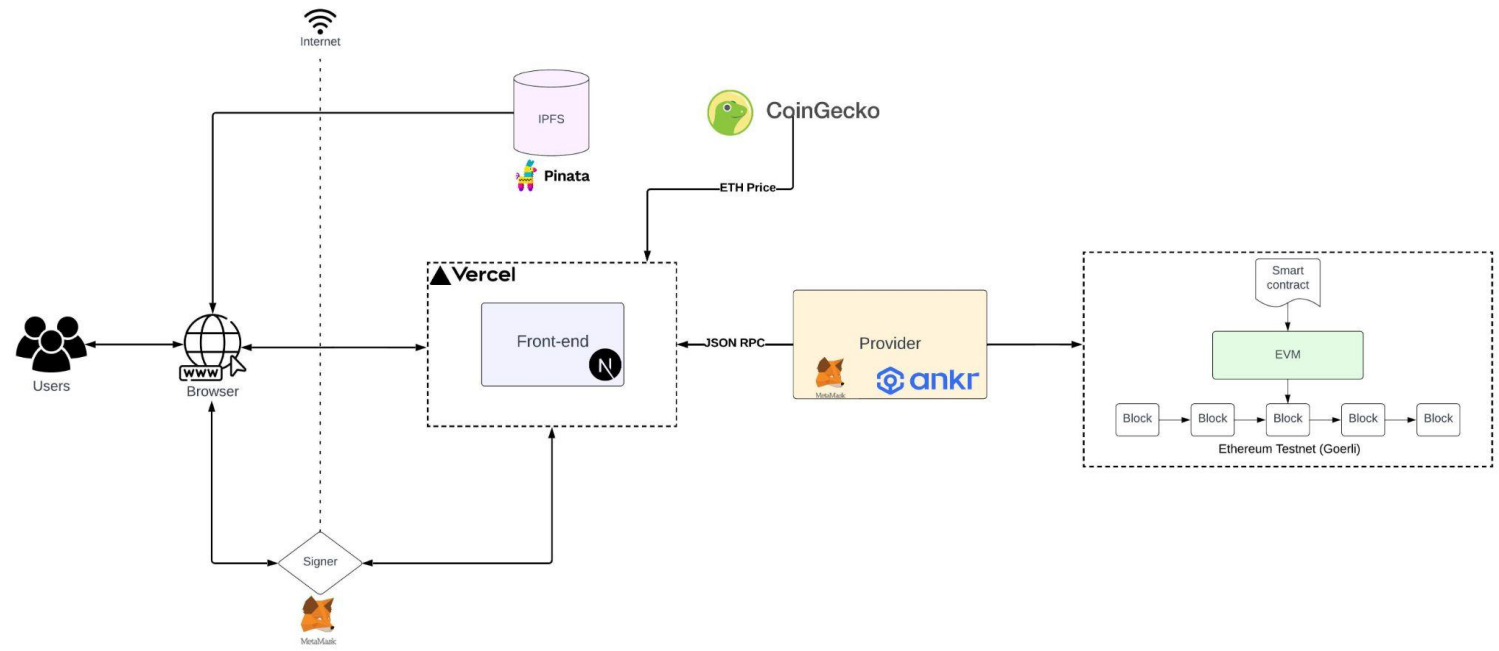

Figure 1. Overall System Flow

1. Users access the DApp through their web browser and use their wallet (MetaMask) to sign transactions.

2. The front-end, which users interact with, is served by Vercel and communicates with the Ethereum network via JSON RPC calls through a provider.

3. Data such as images stored and retrieved from IPFS using Pinata for decentralized storage.

4. For features like displaying the current price of ETH, the front-end fetches data from CoinGecko.

5. The smart contract on the Goerli testnet processes transactions sent from the user's browser and interacts with the blocks on the EVM.

4. Smart Contract Development

4.1 Blockchain Network

Our project leverages the **Ethereum Goerli Testnet**, a vital platform for the development and testing of smart contracts and decentralized applications. The blockchain's key specifications are as follows:

Network Specifications

- **PoA Engine:** clique
- **Network ID:** 5
- **Chain ID:** 5
- **EVM Version:** London
- **RPC:** ANKR (https://rpc.ankr.com/eth_goerli)

Resources

- **Status Dashboard:** The status and health of the Goerli Testnet can be monitored via Goerli Testnet Stats. This dashboard provides real-time information about network performance and conditions.

- **Block Explorers:** Transactions, blocks, and addresses can be explored through Goerli Etherscan, which is a web-based tool offering detailed insights into on-chain activities.

- **Faucets:** To obtain test Ether for deploying contracts or performing transactions on the Goerli Testnet, Goerli Faucet can be used. This service dispenses free test Ether, allowing developers to test their applications without real financial costs.

4.2 Tools & Library

- **Programming Languages**
    - Typescript
    - Solidity

- **IDE**
    - Remix: For developing, testing, and deploying smart contracts directly in a web browser. It's particularly useful for quick iterations and testing of smart contract logic
- **Web3 Framework**
    - Hardhat : The primary environment for compiling, running, testing, and deploying smart contracts
    - Ethers : A library used within this environment for interacting with the Ethereum blockchain and smart contracts
    - Chai : An assertion library that integrates with Hardhat for writing test cases for smart contracts, ensuring that they behave as expected.
- **IPFS**
    - Pinata : Makes it simple to upload to IPFS and to fetch content from the network with blazing speeds thanks to Dedicated Gateways. For storing images.

4.3 Smart Contract Components

Inheritance

- `ERC721Enumerable:` This is an extension of the ERC721 standard that allows for enumeration over all tokens in the contract, as well as tokens owned by a particular address.
- `Ownable:` A utility from OpenZeppelin that provides basic authorization control functions, simplifying the implementation of "user permissions".

Structs

- `Event:` Represents an event with properties like id, name, timestamp, location, image URI, ticket limit, tickets issued count, ticket price, and status (open or closed).
- `Ticket:` Represents a ticket with properties like id, the event it's associated with, and its usage status.

State Variables

- `locked:` A boolean that helps prevent reentrancy attacks.
- `nextEventId:` Tracks the next available ID for a new event.
- `events:` A mapping from an event ID to its corresponding Event struct.
- `tickets:` A mapping from a ticket ID to its corresponding Ticket struct.

Modifiers

- `nonReentrant:` Ensures that certain functions cannot be reentered while they are still executing.

Events

- `EventCreated:` Emitted when a new event is created.
- `TicketsPurchased:` Emitted when tickets are purchased.
- `TicketUsed:` Emitted when a ticket is marked as used.
- `Withdrawn:` Emitted when funds are withdrawn from the contract.
- `EventToggled:` Emitted when an event's status is toggled between open and closed.

4.4 Smart Contract Functions

Constructor

- Initializes the ERC721 token with a name and symbol and sets the nextEventId to 0.

Event Management

- `createEvent:` Allows the contract owner to create a new event.
- `toggleEvent:` Allows the contract owner to open or close an event for ticket sales.
- `viewOpenEvents:` Returns an array of all open events.
- `viewAllEvents:` Returns an array of all events.

Ticket Management

- `purchaseTickets:` Allows users to buy tickets for an open event if tickets are available.
- `useTicket:` Allows the contract owner to mark a ticket as used.
- `viewUserTickets:` Returns an array of tickets owned by the caller.

Financial Transactions

- `viewETHBalance:` Allows anyone to view the contract's balance of ETH.
- `withdrawAll:` Allows the contract owner to withdraw all ETH from the contract.

4.5 Smart Contract Security

- **SafeMath:** The smart contract is written in Solidity version **0.8.20**, which has built-in overflow and underflow protection. This means that the use of SafeMath, a library historically essential for safe arithmetic operations in Solidity, is no longer necessary for this version.

- **Non-Reentrancy Protection:** The contract employs a `nonReentrant` modifier to prevent reentrancy attacks. This is crucial for functions like `purchaseTickets` and `withdrawAll` that involve transferring Ether. The modifier ensures that no recursive calls can be made to these functions while they are executing.

- **Use of OpenZeppelin Contracts:** Importing `ERC721Enumerable` and `Ownable` from OpenZeppelin is a good security practice. OpenZeppelin contracts are widely used and audited, which reduces the risk of vulnerabilities in these aspects of the contract.

- **Access Control:** The contract uses the `onlyOwner` modifier for critical functions like `createEvent`, `useTicket`, `withdrawAll`, and `toggleEvent`. This ensures that only the owner of the contract can perform these sensitive operations, which is a fundamental security measure.

- **Data Validation:** The contract consistently validates inputs and states before proceeding with the logic. For example, in `purchaseTickets`, it checks if the event exists, if there are enough tickets available, if the event is not closed, and if the correct amount of Ether is sent. These checks prevent various logical errors and misuse.

- **Visibility and Access:** Functions and variables have explicit visibility sets (public, private), which is good practice in Solidity. This clear distinction helps avoid unintended exposure of functions or state variables.

- **Event Emission:** Emitting events for significant actions (`EventCreated`, `TicketsPurchased`, `TicketUsed`, `Withdrawn`, `EventToggled`) aids in tracking and auditing contract activities, which is beneficial for security and transparency.

5. Front End Development

5.1 User Interface design

The integration of user centric design concepts with technical innovation forms a fundamental basis. This is the process of creating a user interface for a Web3 ticket. Our design philosophy, which is based on enhancing user experience and seamlessly blending technological innovation. The interface contains 4 screens: Home screen, Ticket reservation details screen,Inventory screen which is divided into 2 types Used ticket and My Tickets.

5.1.1 Wireframe

Home screen

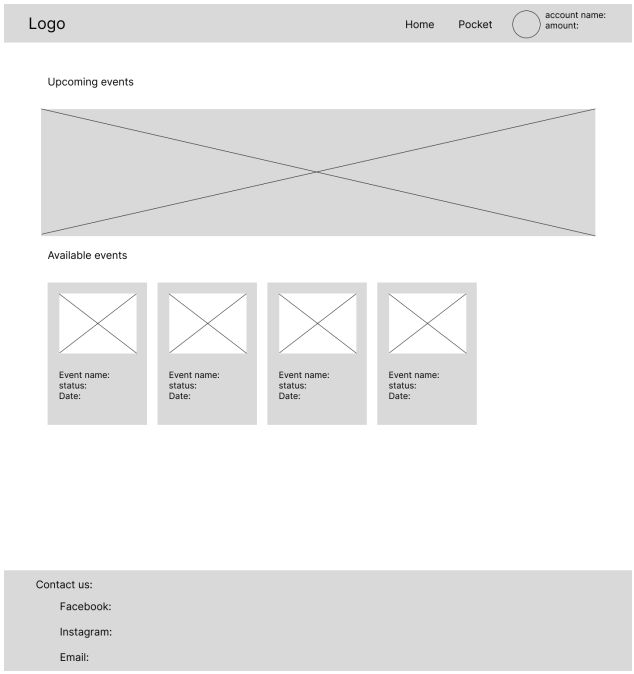

Figure 2. Home screen page

Connecting Pocket

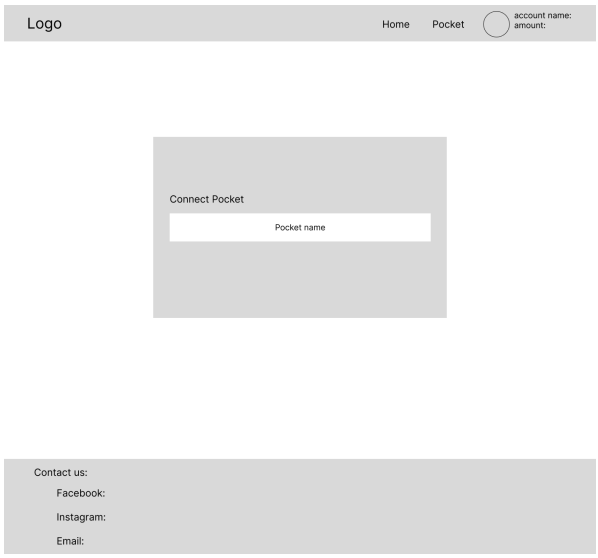

Figure 3. Connecting Pocket page

Pocket

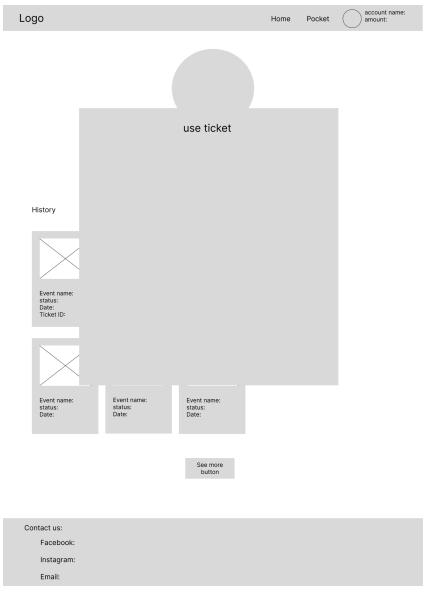

Figure 4. Pocket page

5.1.2 Mockup

Home screen

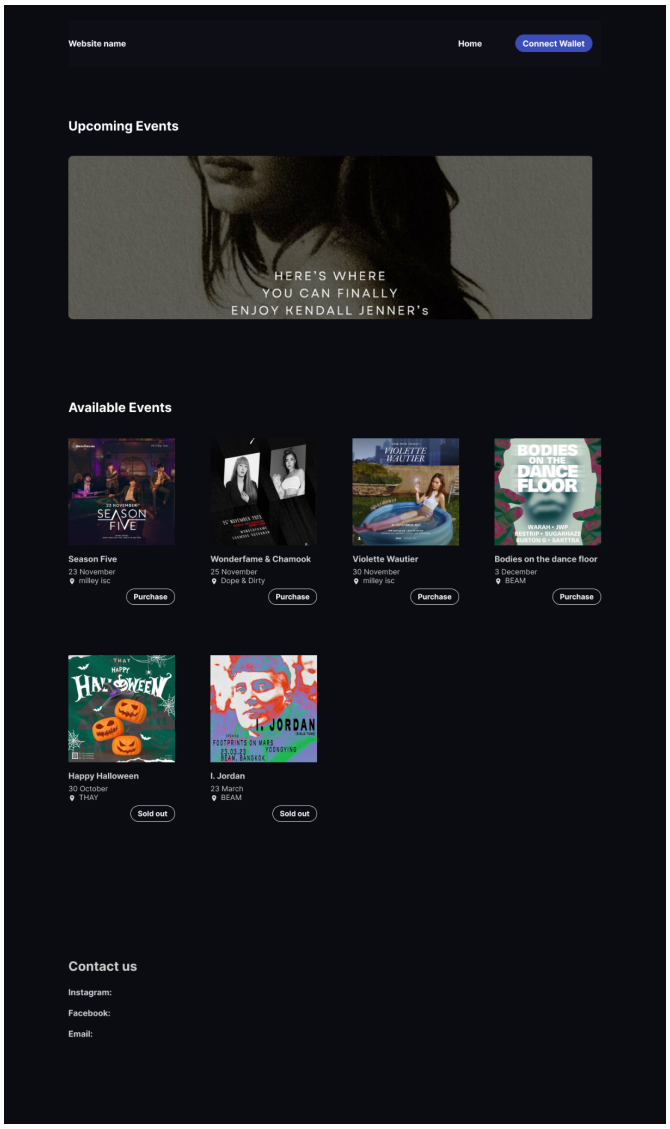

Figure 5. Home screen

Ticket reservation details screen

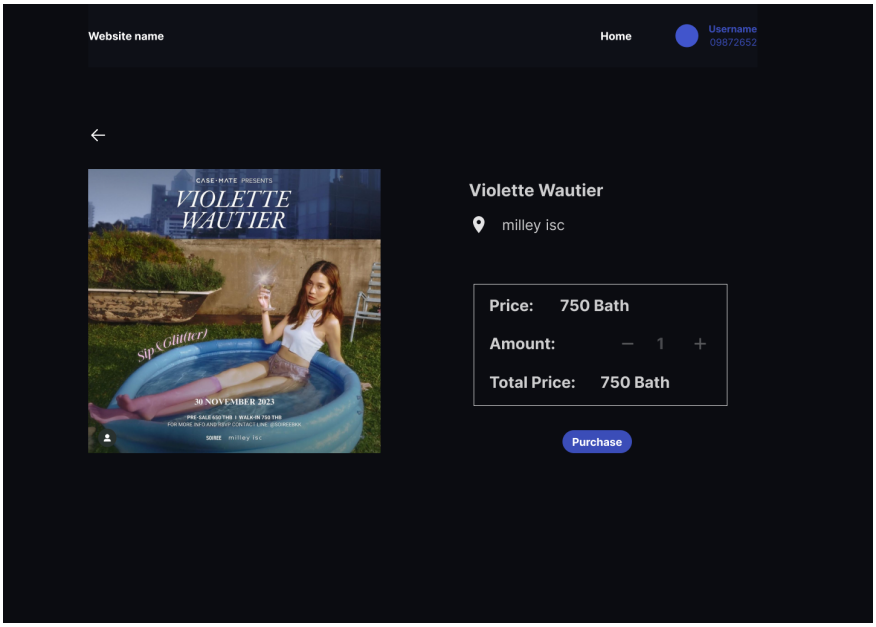

Figure 6. Ticket reservation details screen

Inventory screen (My Tickets)

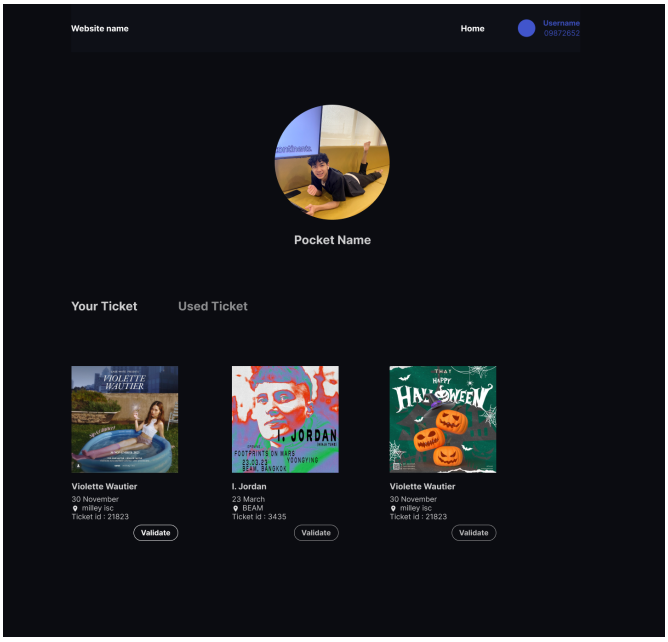

Figure 7. Inventory screen (My Tickets)

Inventory screen (Used Tickets)

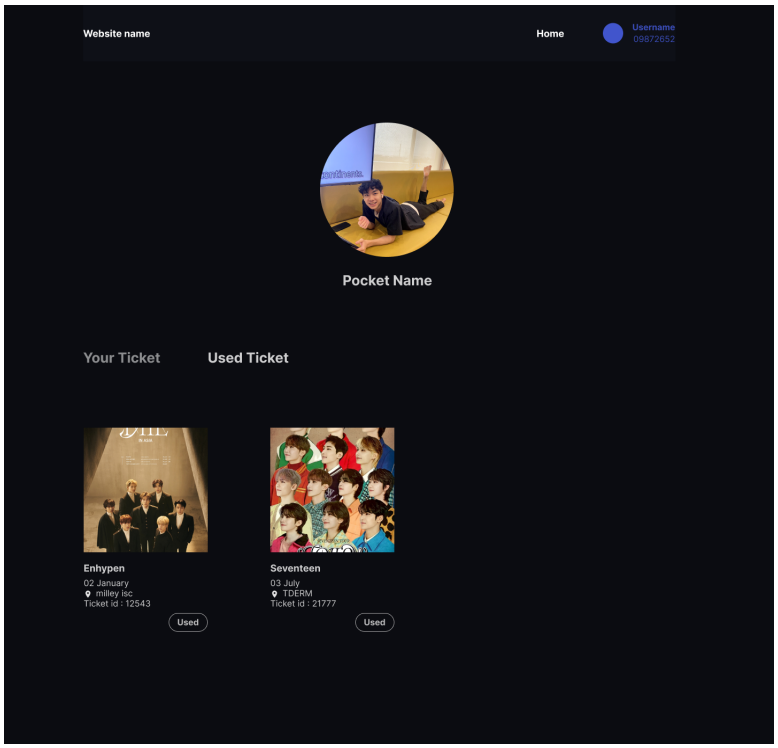

Figure 8. Inventory screen (Used Tickets)

5.2 User Manual

Admin

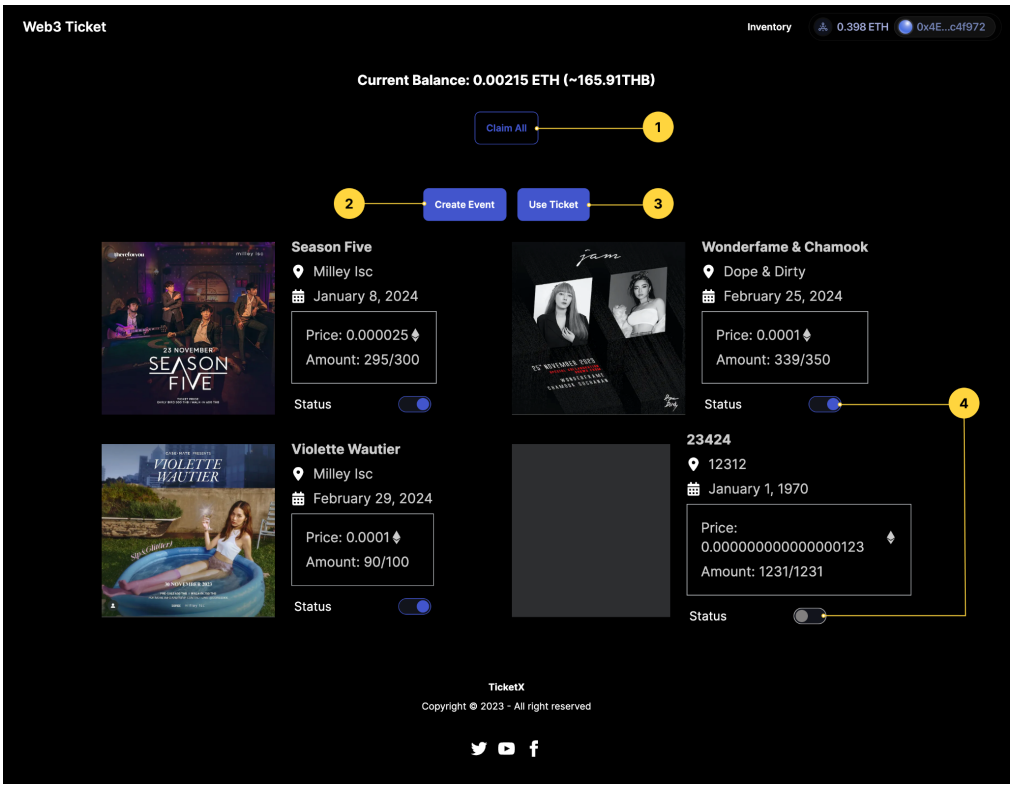

Figure 9. User manual of Admin page

- **Number 1** Claim All button: This button is used to collect the proceeds from ticket sales

- **Number 2** Create Event button: This button is used to initiate the event creation process, guiding users to the event setup screen for administrators

- **Number 3** Used Ticket button: This button is employed to verify purchased tickets. Clicking it will navigate to the ticket validation screen for user authentication

- **Number 4** Status Slider button: This button is utilized to discern the availability of tickets. Admins can click it to toggle between available and unavailable states, aiding in easy identification of ticket status.

  To access the admin page conveniently, append "/admin" to the end of the link. This hidden link is designed for ease of use, eliminating the need for a separate admin button to avoid user confusion.
  Link example: https://web3-e-ticket.vercel.app/admin

Admin Creating Event

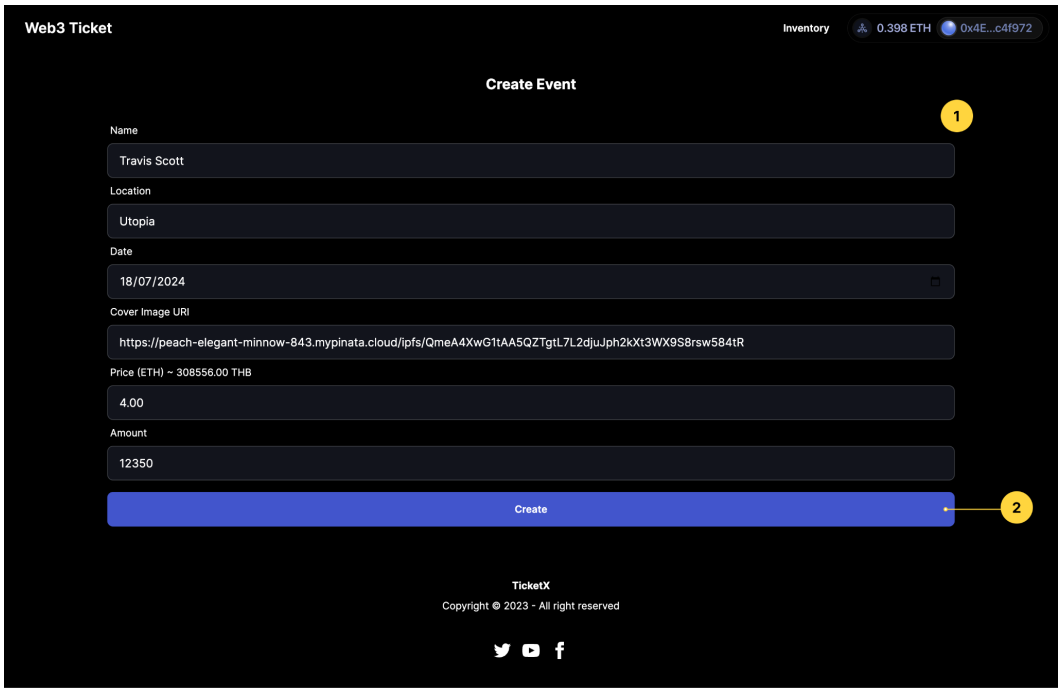

Figure 10. User manual of Admin Creating Event page

- **Number 1** Concert Detail Input: Administrators can input essential details for the concert, including:
    - Name: Specify the name of the concert
    - Location: Provide the venue or location of the event
    - Date: Select the date of the concert
    - Cover Image URL: Input the URL for the cover image
    - Ticket Price: Set the price for each ticket
    - Ticket Quantity: Specify the number of tickets available for sale
- **Number 2** Create Event Button: Administrators can use this button to initiate the event creation process. Once created, the event will be displayed on the Home page for visibility.

Admin Validate Ticket

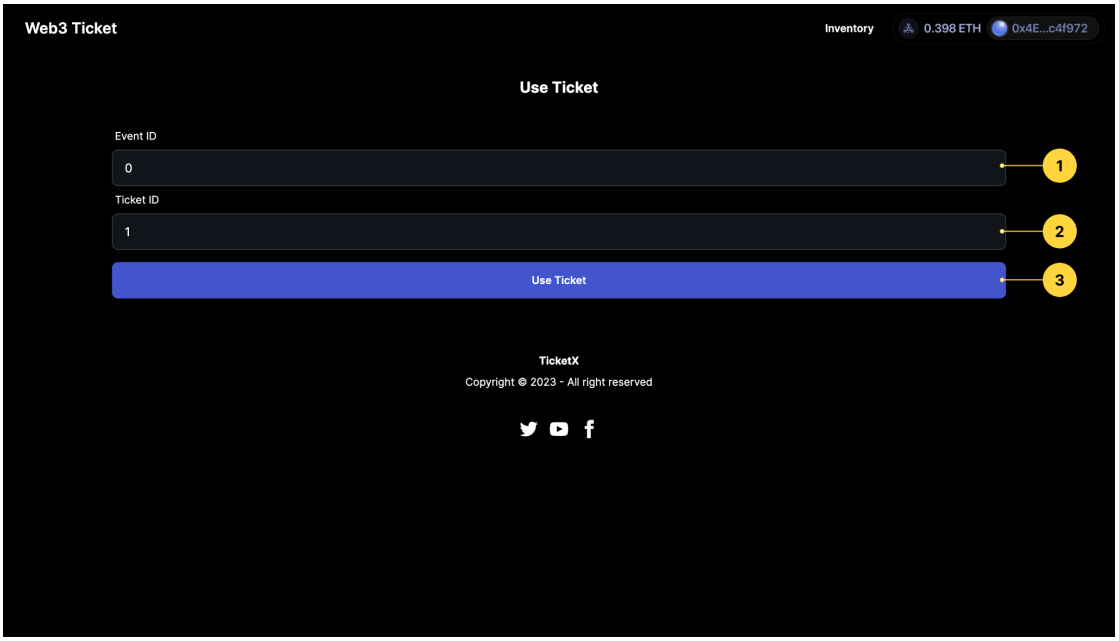

Figure 11. User manual of Admin Validate Ticket page

- **Number 1** Event ID input: The administrator will enter Event id
- **Number 2** Ticket ID input: The administrator will input Ticket id
- **Number 3** Use ticket: Clicking this button will mark the specified ticket as used, linking it to the corresponding event

Home screen I

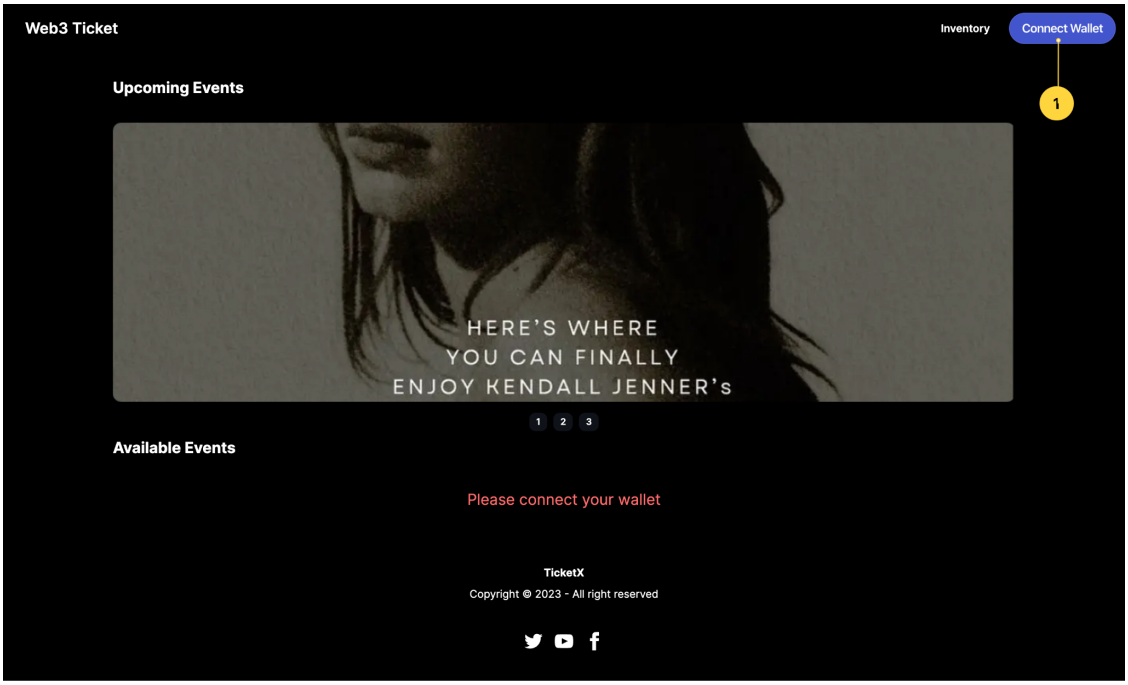

Figure 12. User manual of Home screen I

- **Number 1** Connect Wallet button: Click this button to initiate the wallet connection process.

Users are required to connect their wallet to proceed. After connecting, the available events section will be accessible.

Home screen II

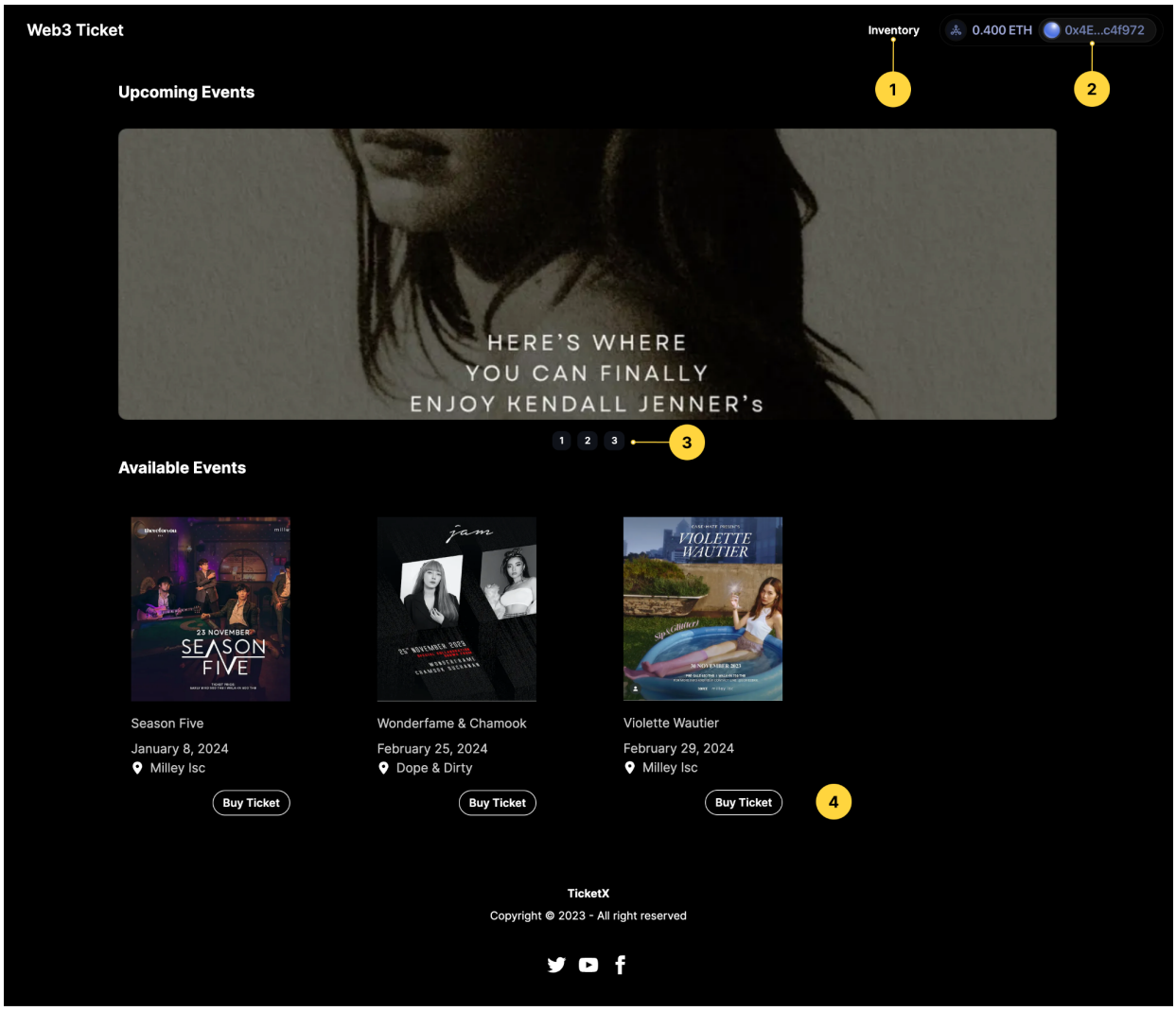

Figure 13. User manual of Home screen II

- **Number 1** View Inventory Button: Users can click this button to navigate to the inventory page and review the tickets they own.
- **Number 2** Wallet Balancing: Displays the current balance in the user's wallet.
- **Number 3** Upcoming Events button: Clicking this button (1, 2, 3) will display the corresponding poster for the upcoming events.
- **Number 4** Buy ticket button: On the card displaying concert details in the "Available" section, users can click this button to purchase tickets for the respective event.

Connecting Wallet

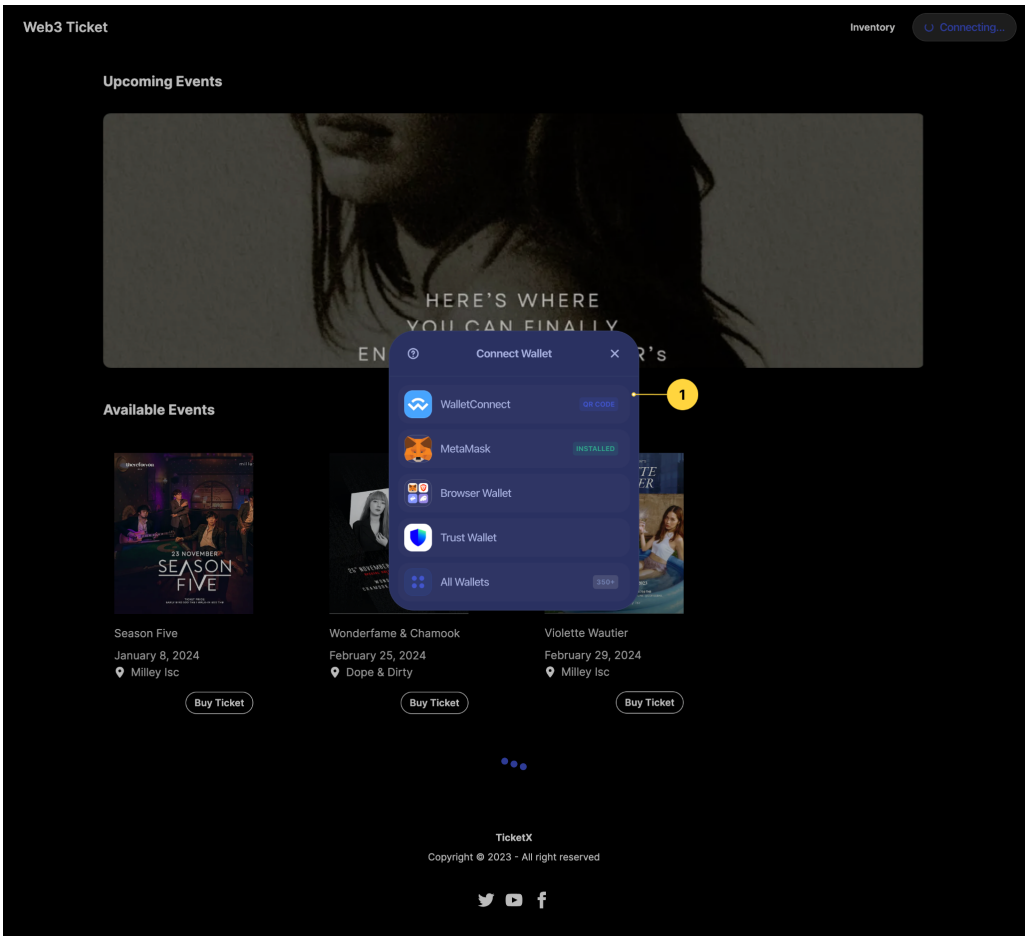

Figure 14. User manual of Connecting Wallet page

- **Number 1** Connect wallet panel: This panel represents the available wallet options on the website, including WalletConnect, Metamask, Browser Wallet, Trust Wallet, and others. Users can choose and connect the wallet of their preference

Following the user's click on the "Connect Wallet" button, a dedicated panel will appear, guiding users through the process of connecting their wallet. This panel offers options for various wallets, including WalletConnect, Metamask, Browser Wallet, Trust Wallet, and others. Users can seamlessly select and connect their preferred wallet from this interface

Connecting Wallet Successfully

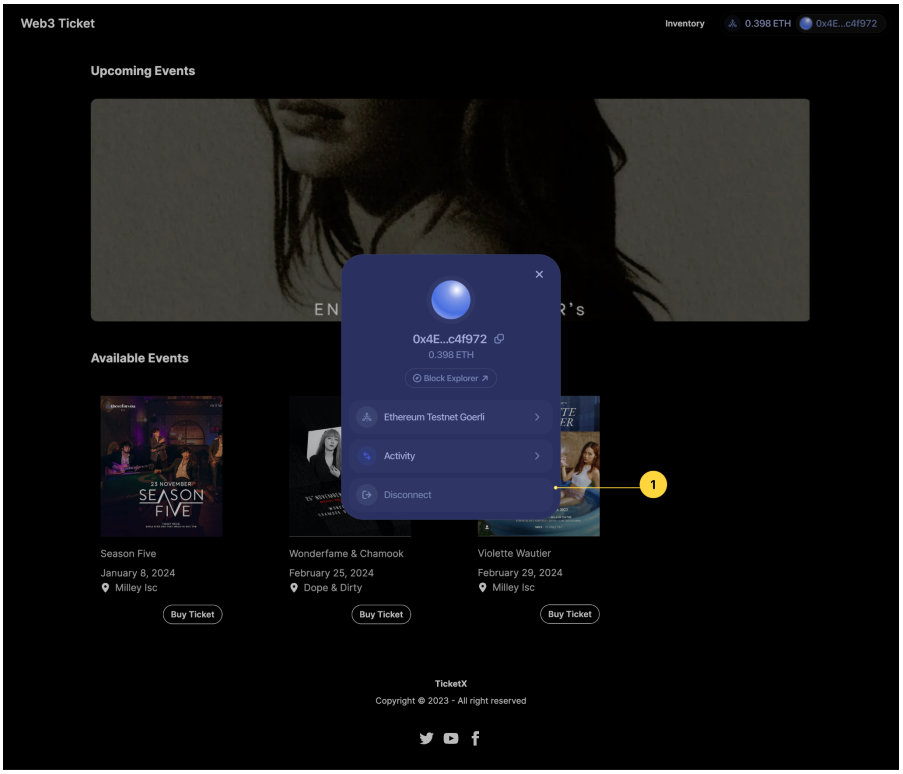

Figure 15. User manual of Connecting Wallet Successfully alert

- **Number 1** Wallet Control Panel: This control panel presents crucial wallet information, including Ethereum Testnet (Goerli) balance, recent activity, and a disconnect button for managing the wallet connection

Buying ticket

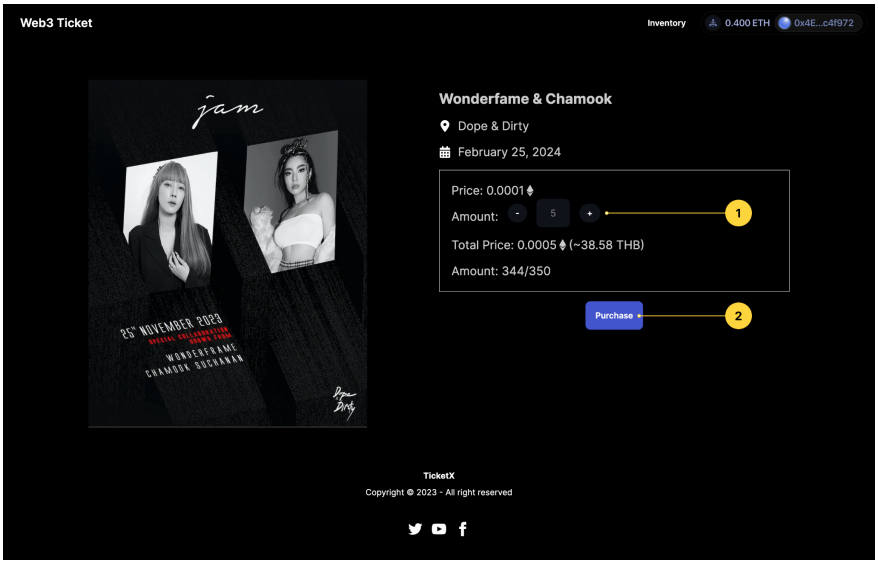

Figure 16. User manual of Buying ticket page

- **Number 1** Amount ticket button: Users can click to add or delete the quantity of tickets they wish to purchase. The total price dynamically adjusts according to the selected amount of tickets. Additionally, the total amount will be displayed in both Thai Baht and Ethereum, providing clarity on the cost in different currencies.
- **Number 2** Purchase button: Clicking this button finalizes the transaction, allowing users to complete the ticket purchase.

Clicking on the figure will direct users to the detailed ticket page, where they can proceed with the ticket purchase.

Transaction Successfully

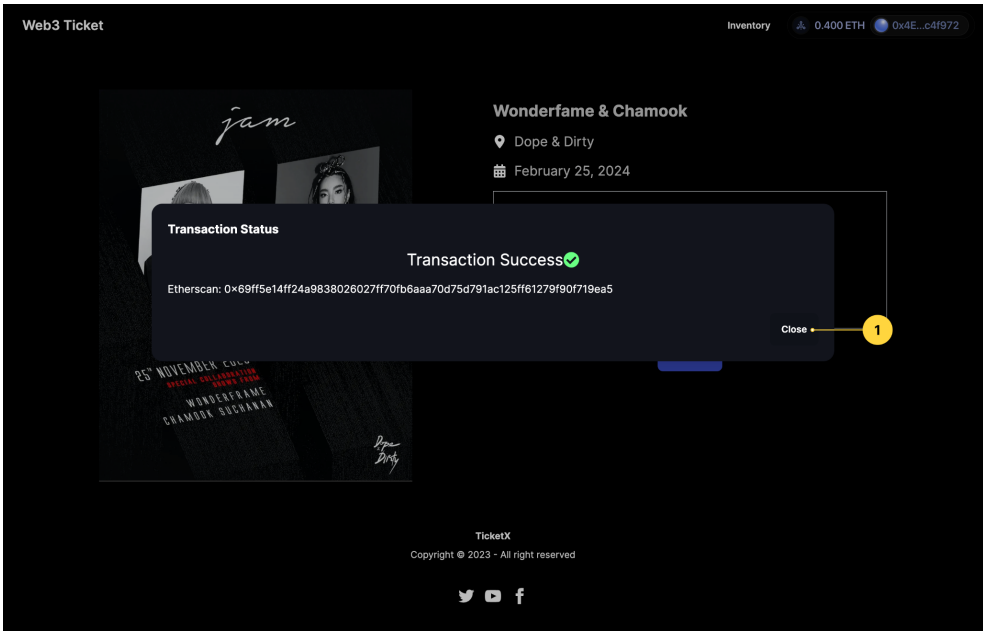

Figure 17. User manual of Transaction Successfully alert

- **Number 1** Close button: Close the transaction successfully panel

Inventory (My Ticket)

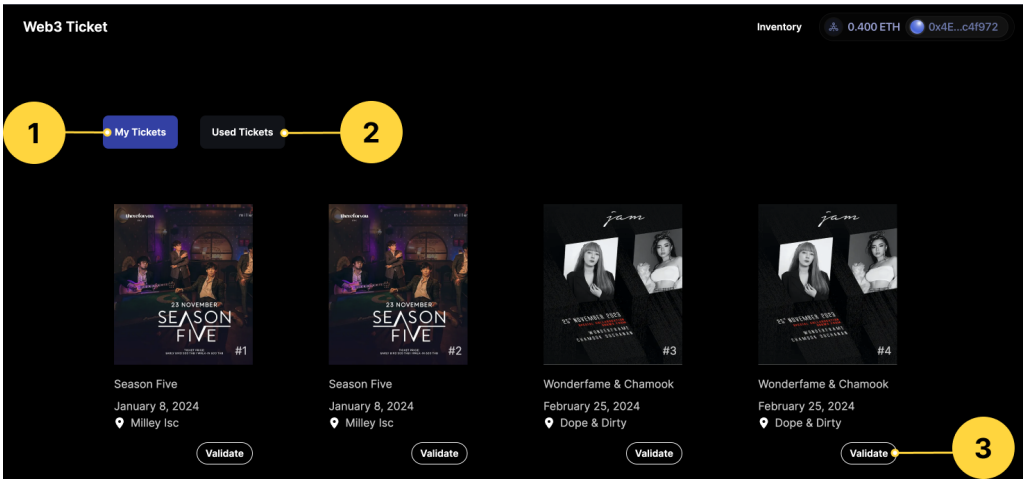

Figure 18. User manual of Inventory (My Ticket) page

- **Number 1** My Ticket button: Clicking this button will display the tickets that the user has already purchased.
- **Number 2** Used Ticket button: Clicking this button will navigate users to the section containing validated tickets, showcasing tickets that have been successfully authenticated.
- **Number 3** Validate button: Clicking this button will confirm and utilize the ticket, marking it as used in the process

Inventory (Used Ticket)

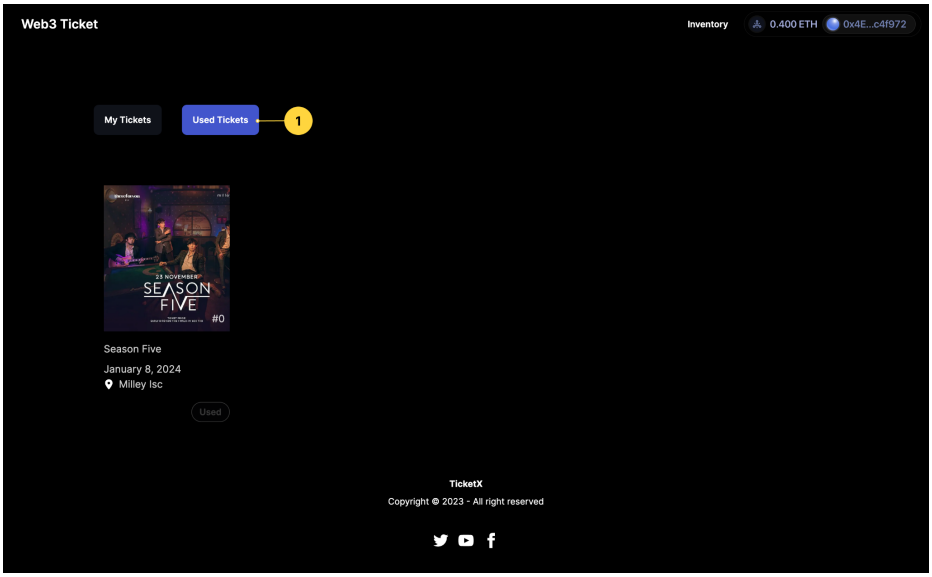

Figure 19. User manual of Inventory (Used Ticket) page

- **Number 1** Used Ticket button: Clicking this button will navigate users to the section containing validated tickets, showcasing tickets that have been successfully authenticated.

5.3 Front-end Technology

Our front-end technology stack is meticulously crafted to deliver a seamless and user-friendly experience for our Web3 Ticket. Key components of our front-end technology include

Programming Languages

- Typescript
- HTML
- CSS

UI Framework

- **NextJS:** Enhances React's development process by offering built-in features like routing, facilitating faster and more efficient web application development.
- **TailwindCSS:** A utility-first CSS framework that allows rapid UI development
- **DaisyUI:** Provides a plugin for TailwindCSS that adds component classes to speed up the design process

Web3 Integration

- **Web3Modal:** This integration is crucial for connecting the application to various Ethereum wallets, making the platform accessible and user-friendly.
- **TypeChain:** For interacting with smart contracts, TypeChain is used to generate TypeScript typings, which enhance the developer experience by providing type safety and autocompletion.
- **Ethers.js:** A library for interacting with Ethereum smart contracts and accounts, Ethers.js is employed due to its lightweight and modular nature, facilitating blockchain transactions and data queries.

External API:

- **CoinGecko API:** The application fetches real-time Ethereum prices in Thai Baht (THB) using the CoinGecko API, providing users with up-to-date cryptocurrency valuations.

6. Testing and Deployment

6.1 Frontend Deployment

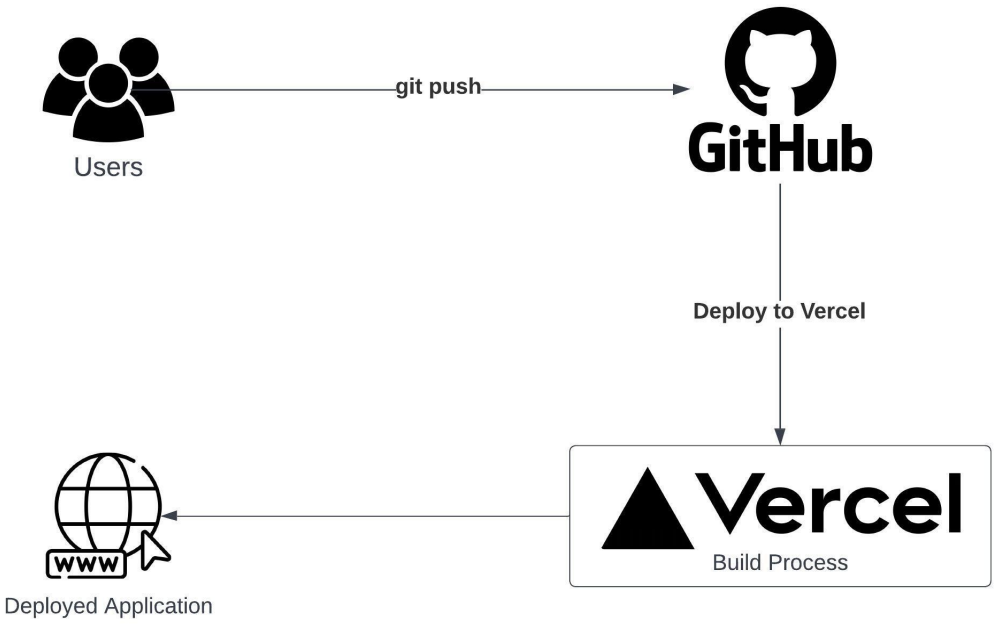

Figure 20. Fronted Deployment

Deployment Process

The diagram illustrates the deployment process of a frontend application to Vercel through GitHub. Upon a `git push` action to the repository by users, the updated code triggers an automatic deployment to Vercel.

Vercel's Role

Vercel is a cloud platform for static sites and Serverless Functions that automates deployment, scaling, and operations. When code is pushed to a connected GitHub repository, Vercel initiates its build process. This process involves installing dependencies, running build scripts, and generating a production version of the application.

Access to the Deployed Application

Once the build and tests pass, Vercel deploys the application, making it accessible to users via the web. The platform also provides automatic scaling to handle traffic as needed, ensuring the application remains responsive and available.

In essence, Vercel handles both Continuous Integration (CI) by running tests during the build process, and Continuous Deployment (CD) by deploying the application after a successful build and test phase, streamlining the entire process from code push to production.

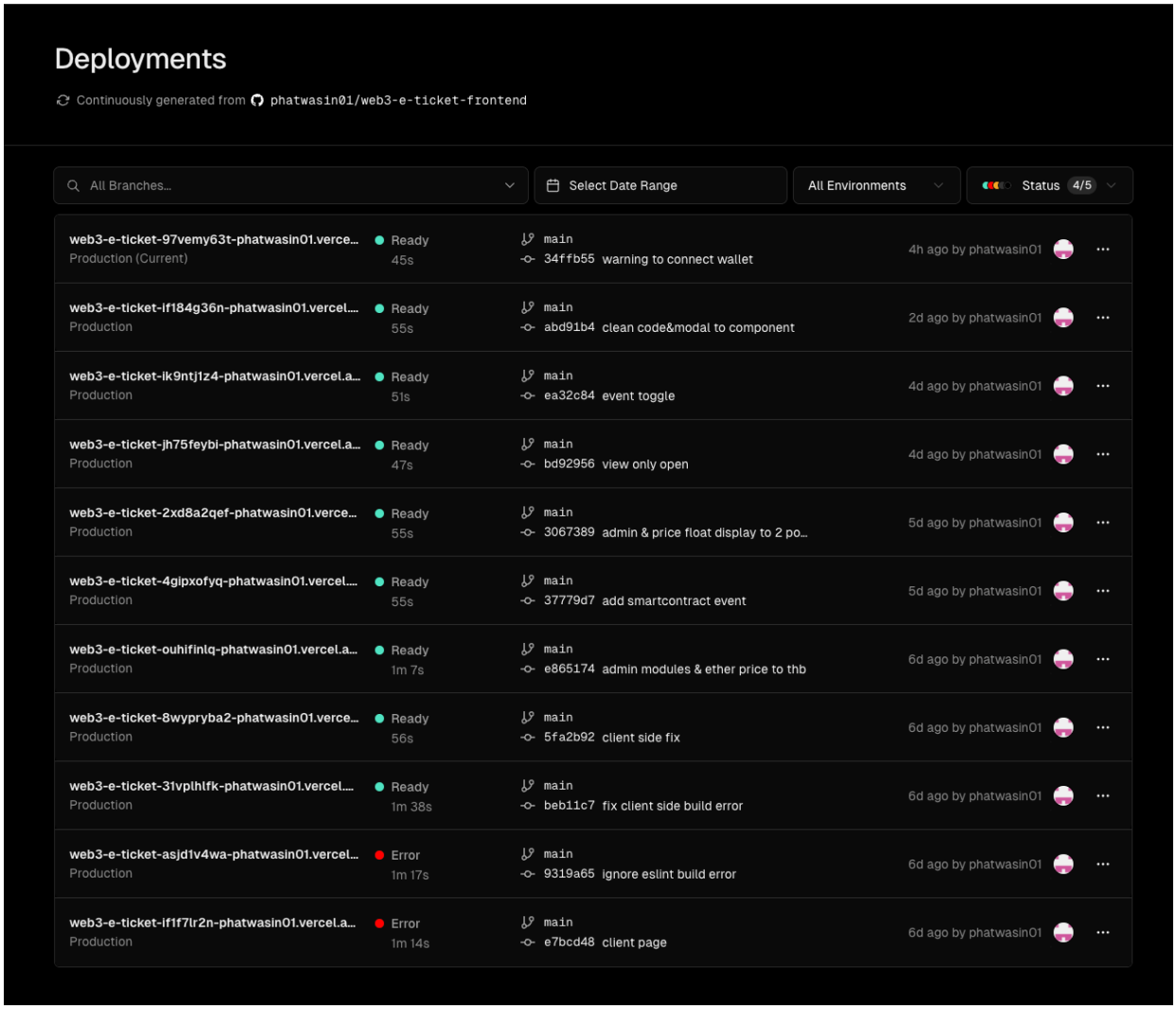

Figure 21.Vercel Deployment Results

6.2 Unit Testing for Smart Contracts

Importance in Smart Contracts

In the context of smart contracts, unit testing is particularly vital due to the immutable nature of blockchain. Once a smart contract is deployed, it cannot be altered, making any bugs or vulnerabilities potentially irreversible and exploitable. Therefore, unit testing is essential to ensure the correctness, security, and reliability of smart contracts before they are deployed to the blockchain

Structure of Unit Tests

Unit tests are typically grouped into test suites, which are collections of test cases that are related to a specific component or functionality. The unit tests are categorized into different sections, such as "**Deployment**," "**Event Management**," "**Ticket Purchasing**," "**Using Tickets**," and "**Contract's Ethers**". Each section contains several test cases that cover both the expected behavior and edge cases of the smart contract.

Test Cases and Assertions

Each test case uses assertions to compare the actual outcome with the expected outcome. Assertions are statements that check if a particular condition is true. If the assertion passes, it means the test is successful; if it fails, it means there is a bug in the code. The use of `.emit()` and `.revertedWith()` in the test cases checks whether certain events are emitted or certain errors are thrown, respectively, in response to various function calls.

```
pnpm run test

> @ test /Users/phatwasinsuksai/Desktop/web3-e-ticket-fullstack/solidity
> hardhat test


  EventTicketing
    Deployment
      ✔ Should deploy the contract correctly (898ms)
      ✔ Should set the right owner
    Event Management
      ✔ Should allow the owner to create an event
      ✔ Should fail when a non-owner tries to create an event
      ✔ Should allow the owner to toggle an event's status
    Ticket Purchasing
      ✔ Should allow users to purchase tickets after creating an event
      ✔ Should fail when a user tries to purchase tickets for an event that doesn't exist
      ✔ Should fail when a user sent less ethers than the total price
      ✔ Should fail when a user tries to purchase more tickets than the limit
      ✔ Should fail when a user tries to purchase when the event is closed
    Using Tickets
      ✔ Should allow the owner to mark a ticket as used
      ✔ Should fail when non-owner tries to mark a ticket as used
      ✔ Should fail when trying to use a ticket that has already been used
      ✔ Should fail when trying to use a ticket with non-existent event ID
      ✔ Should fail when trying to use a ticket that not belong to the event
    Viewing Events and Tickets
      ✔ Should allow users to view all events
      ✔ Should allow users to view only openning events
      ✔ Should allow users to view their own tickets
    Contract's Ethers
      ✔ should return 0 when contract has no balance
      ✔ should return correct balance when contract has Ether
      ✔ should only be callable by owner
      ✔ should withdraw all Ether to owner


  22 passing (1s)
```

Figure 22. Unit Testing Results

6.3 Integration Testing

Integration testing is a key stage in the Continuous Integration (CI) process. In the context of smart contract development with **GitHub** and **GitHub Actions**, it involves the following steps:

1. **Code Push:** Developers push their code updates to a GitHub repository, which serves as the version control system.

2. **GitHub Actions Trigger:** The push event triggers the CI pipeline defined in GitHub Actions, automating the subsequent testing process.

3. **Compilation:** The smart contract code is compiled first to ensure syntax correctness and to prepare it for testing.

4. **Unit Testing:** Unit tests are run to verify individual components of the smart contract for functional integrity.

5. **Local Deployment:** The contract is deployed in a local blockchain simulation (via Hardhat), verifying its operational readiness in an environment akin to the live blockchain.

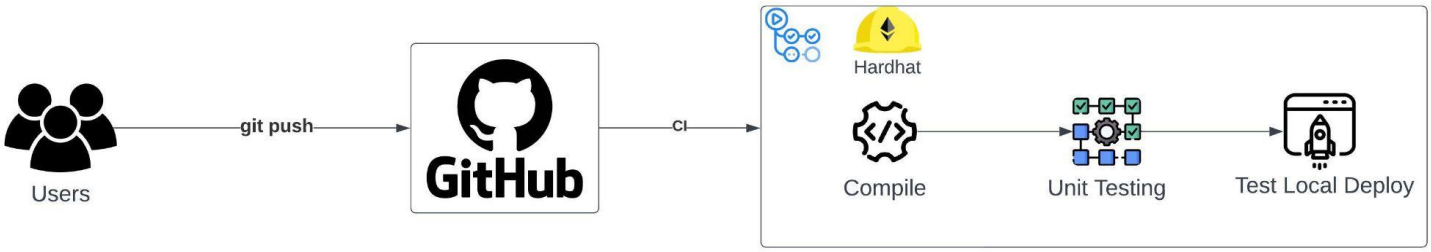

Figure 23. Integration Testing Workflow

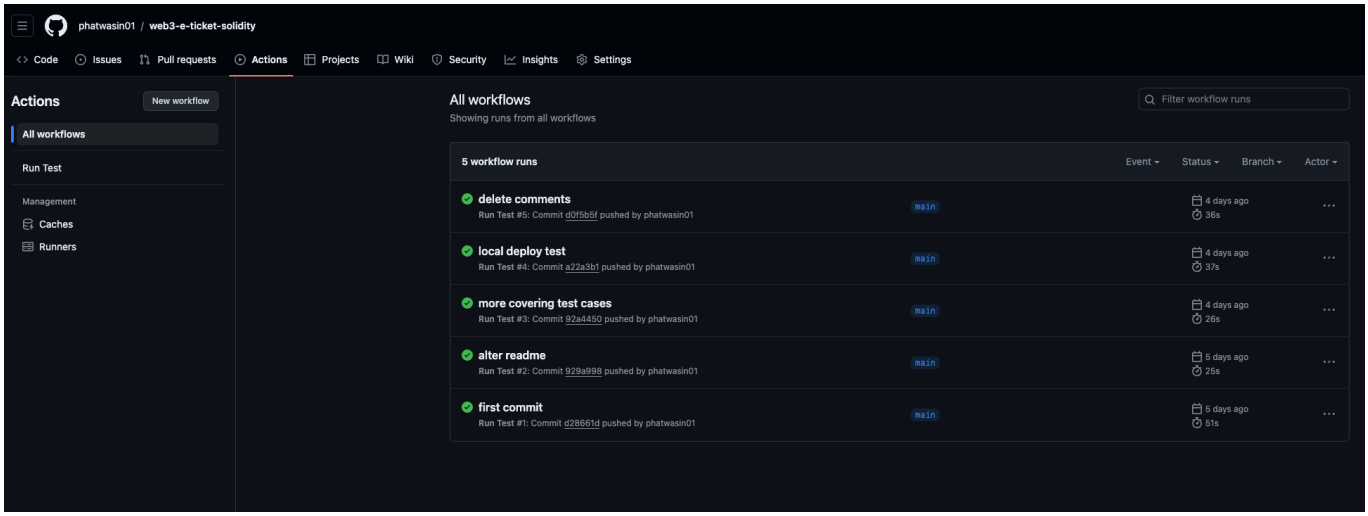

Figure 24.Github Action CI Results

6.4 End-to-end Testing

This testing approach focuses on evaluating the **system's behavior** from the **user's perspective** and aims to identify and address any issues that may arise across the entire application flow.

	Web3 Ticket E2E Testing				
No	**Acceptance Requirements**	**Test Steps**	**Expected Result**	**Test Results** (Passed/Failed)	**Test Date**
	Event Management				
1	Create Event	1. Log in as the contract owner. 2. Navigate to the event creation interface. 3. Enter valid event details (name, date, location, ticket limit, and ticket price). 4. Submit the event creation form.	The event is created, and details are stored in the smart contract.	Passed	17/12/2023
2	Store Event Images on IPFS	1. Create an event with an associated cover image. 2. Verify that the image URI is recorded in the event details. 3. Retrieve the image from IPFS using the recorded URI.	The image is retrieved successfully, and the URI matches the recorded value.	Passed	17/12/2023
	Ticketing				
3	Ticket Purchase	1. Log in as a user. 2. View available events. 3. Purchase a ticket for a chosen event. 4. Confirm the payment amount.	The user owns the purchased ticket, and the payment matches the ticket price.	Passed	17/12/2023
4	Issue Tickets as NFTs	1. Purchase a ticket. 2. Verify that the purchased ticket is an ERC721 token	The purchased ticket is an ERC721-compliant NFT.	Passed	17/12/2023
5	Manage Ticket Usage	1. Log in as the contract owner. 2. Mark a ticket as used.	The ticket status is updated, indicating it has been used.	Passed	17/12/2023
	Event and Ticket Visibility				
6	View Events	1. Log in as a user. 2. View all events, open events, and specific event details.	Event details are displayed correctly.	Passed	17/12/2023
7	View User Tickets	1. Log in as a user. 2. Access the ticket viewing interface.	User-owned tickets are displayed.	Passed	17/12/2023

Figure 25. E2E testing I

	Web3 Ticket E2E Testing				
No	**Acceptance Requirements**	**Test Steps**	**Expected Result**	**Test Results** (Passed/Failed)	**Test Date**
	Financial Transactions				
8	Withdrawal of Funds	1. Log in as the contract owner. 2. Initiate a withdrawal of funds.	ETH is withdrawn to the designated account.	Passed	17/12/2023
	Event Status Control				
9	Toggle Event Status	1. Log in as the contract owner. 2. Toggle the status of an event.	The event status is updated accordingly.	Passed	17/12/2023
	Data Retrieval				
10	Retrieve Event Images	1. Access the interface for retrieving event images. 2. Retrieve an event image using its URI.	The image is successfully retrieved and displayed.	Passed	17/12/2023

Figure 26. E2E testing II

6.5 Performance Issue and Feedback

Current Challenges

The application faces performance bottlenecks due to its current architecture, which relies heavily on fetching data directly from the blockchain. This not only leads to loading issues due to the inherent latency of blockchain interactions but also necessitates that users connect their wallets to view events on the homepage. These can detract the user experience, as the expectation in modern web applications is for immediate access and swift interaction.

Implemented Solutions and Their Limitations

To mitigate these issues, the system has implemented client-side caching, storing some responses in the user's browser storage. While this approach does provide some relief by reducing redundant data fetching, it has proven insufficient. The limitations of client-side caching are evident, as it does not entirely overcome the latency issues and still requires initial blockchain queries that are subject to network congestion and delays.

Proposed Enhancements

For future iterations, it is proposed that the system integrates an off-chain database to store event data. This architectural evolution would shift the data load away from the blockchain, allowing for more rapid retrieval of event information. An off-chain database could serve as a cache for the blockchain, providing the following benefits:

- **Enhanced Speed:** Off-chain databases can deliver content significantly faster than blockchain queries, resulting in an immediate performance boost.
- **Reduced Load:** By decreasing the dependency on real-time blockchain interactions, the system can alleviate the load on the Ethereum network, leading to cost savings and improved scalability.
- **User Experience:** Users would not be required to connect their wallets merely to browse events, thus streamlining the onboarding process and making the platform more accessible to a broader audience.
- **Data Integrity:** Periodic synchronization between the blockchain and the off-chain database would ensure data integrity, while still leveraging the blockchain for critical operations that require decentralization and security.