**AI-Enhanced Decentralized Event Ticketing Platform — Architecture, DB Schema & Smart Contract Flow**

This document provides a ready-to-implement architecture, database schema, and smart-contract flow for your AI‑Enhanced Decentralized Event Ticketing Platform (NFT + Chainlink + AI). Use this as a blueprint for development, deployment, and for writing the project case study on your resume/portfolio.

**1. High-level Architecture**

**Components:**

* Frontend (React) — Wallet integration (MetaMask), UI, Chatbot
* Backend (Node.js/Express) — REST APIs, Web3/Ethers bridge, AI API gateway
* Smart Contracts (Solidity on Ethereum) — TicketNFT, TicketMarket, Oracle integrations
* AI Services (Python: Flask/FastAPI) — Demand prediction, Fraud detection, Chatbot using LangChain/OpenAI
* IPFS — Store ticket metadata, QR codes, images
* MongoDB — Off-chain data (users, events, orders, logs)
* Chainlink — Price feeds (ETH/USD) and custom oracles for event-time verification

**Data/Control Flow (brief):**

1. User browses events on React frontend.
2. For purchase, frontend calls backend to request mint price.
3. Backend queries AI service (demand model) + Chainlink feed to compute dynamic price.
4. Frontend initiates wallet transaction to mint NFT (smart contract).
5. Smart contract emits events captured by backend; metadata stored on IPFS; MongoDB updated.
6. Post-purchase, AI fraud detector monitors transfers and flags anomalies.

**ASCII Diagram:**

[User Browser] --(1)--> [Frontend React]

[Frontend] --(2)--> [Backend API]

[Backend API] --(3)--> [AI Service: DemandPrediction / FraudDetection]

[Backend API] --(4)--> [Chainlink Oracle]

[Frontend] --(5)--> [Smart Contract on Ethereum]

[Smart Contract] --(6)--> [Event Emitted] --> [Backend Listener]

[Backend Listener] --(7)--> [IPFS] & [MongoDB]

**2. Component Responsibilities**

**Frontend (React)**

* Event listing and search UI (calls backend GET /events).
* Wallet connect & transaction signing (MetaMask + Ethers.js).
* Chatbot UI (LangChain-powered via backend /ai/chat).
* Ticket display & transfer UI (reads NFT metadata from IPFS or contract).

**Backend (Node.js/Express)**

* REST APIs for events, ticket operations, user management.
* Auth: JWT + wallet address binding (optional: sign-in with Ethereum signature).
* Web3/Ethers layer to read on-chain events and call contract view methods.
* API gateway to AI services (prediction, fraud detection, chat) and Chainlink interactions.
* Listener worker (websocket or polling) for contract events to update MongoDB.

**AI Services (Python)**

* **Demand Prediction**: scikit-learn model (saved .pkl) served via FastAPI.
  + Endpoint: /predict-demand -> returns demand score & recommended price multiplier.
* **Fraud Detection**: Isolation Forest or autoencoder model served via API.
  + Endpoint: /detect-fraud -> returns risk score and reason.
* **Chatbot**: LangChain orchestrates prompts and calls to OpenAI API for natural language queries.
  + Endpoint: /chat -> returns reply and optionally recommended events.

**Smart Contracts (Solidity)**

* TicketNFT.sol (ERC‑721): mintTicket, transferTicket, burnTicket, tokenURI.
* TicketMarketplace.sol: buyTicket (if selling off-chain), setResaleRules, royalties.
* PriceOracleIntegration.sol: fetchPrice() (from Chainlink) — used for on-chain checks or events.
* EventTimeVerifier.sol: verified via Chainlink or a trusted oracle to assert event timestamps.

**3. Smart Contract Flow & Functions**

**TicketNFT.sol (Core)**

**Key Functions:**

* mintTicket(address to, uint256 eventId, string metadataURI) — mints ERC‑721, stores eventId mapping.
* validateTicket(uint256 tokenId, address holder) — returns boolean (checks ownership & event match).
* setMetadataURI(uint256 tokenId, string uri) — admin only (set IPFS URI).

**Key Events:**

* TicketMinted(uint256 tokenId, uint256 eventId, address owner)
* TicketTransferred(uint256 tokenId, address from, address to)
* TicketBurned(uint256 tokenId)

**Marketplace / Resale Logic**

* listTicket(uint256 tokenId, uint256 price)
* buyListedTicket(uint256 tokenId)
* preventResale(uint256 tokenId) — if event organizer restricts resale.
* setRoyalty(address recipient, uint256 percentage)

**Oracle Integration (Chainlink)**

* Request Chainlink price feed: getETHPrice() — used in contract events or checks.
* For event time verification, use a custom Chainlink job to ping event provider APIs.

**Security Patterns:**

* Ownable/AccessControl for administrative functions.
* ReentrancyGuard for marketplace buy functions.
* Pausable for emergency stops.

**4. Database Schema (MongoDB Collections)**

**users**

{

\_id: ObjectId,

walletAddress: String, // primary identifier

email: String,

name: String,

role: String, // user | organizer | admin

createdAt: Date

}

**events**

{

\_id: ObjectId,

organizerId: ObjectId,

title: String,

description: String,

venue: String,

location: { city: String, lat: Number, lng: Number },

startTime: Date,

endTime: Date,

currency: String,

basePrice: Number,

totalTickets: Number,

remainingTickets: Number,

ticketMetadataSchema: Object,

createdAt: Date

}

**tickets**

{

\_id: ObjectId,

tokenId: String, // tokenId on-chain

eventId: ObjectId,

ownerWallet: String,

metadataURI: String, // ipfs://CID

pricePaid: Number,

purchasedAt: Date,

status: String // active | used | transferred | refunded

}

**transactions / orders**

{

\_id: ObjectId,

ticketId: ObjectId,

buyerWallet: String,

txHash: String,

chainId: Number,

amount: Number,

createdAt: Date

}

**audit\_logs**

* store smart contract events captured by backend for replay and debugging

**5. REST API Endpoints (suggested)**

**Auth**

* POST /auth/signin — signature-based login (wallet signature + nonce) -> returns JWT
* GET /auth/me — profile

**Events**

* GET /events — list events (filters: location, date, genre)
* GET /events/:id — event details with pricing suggestion
* POST /events — create event (organizer)

**Tickets**

* POST /tickets/mint — backend-assisted mint request (creates metadata & returns minting payload)
* GET /tickets/user/:wallet — list user tickets
* POST /tickets/transfer — record transfer after on-chain transfer

**AI**

* POST /ai/predict-demand — returns demand score & recommended multiplier
* POST /ai/detect-fraud — returns risk score
* POST /ai/chat — conversational search and recommendations

**6. Sequence Example: Purchase Flow**

1. Frontend GET /events/:id -> shows dynamic suggested price via /ai/predict-demand + Chainlink price.
2. User clicks Buy -> Frontend calls backend POST /tickets/mint to create metadata on IPFS and obtain metadataURI.
3. Frontend constructs transaction calling TicketNFT.mintTicket(to, eventId, metadataURI) and prompts MetaMask.
4. On chain confirmation: TicketMinted event emitted.
5. Backend listener picks up event, writes into tickets collection and creates transactions record.
6. Post-purchase: AI fraud detector analyzes wallet behavior asynchronously and flags if risky.

**7. Deployment & DevOps Checklist**

* Hardhat: configure networks for Goerli/Sepolia & add etherscan API keys for verification.
* IPFS: use Web3.Storage or Infura IPFS; store API keys securely in backend env.
* Chainlink: ensure access to required data feeds and oracle job specs for custom jobs.
* CI/CD: set up GitHub Actions to run tests, linting, and deploy frontend & backend.
* Monitoring: integrate Sentry for backend errors; use Alchemy/Alchemy Notify or Infura for reliable provider notifications.

**8. Testing Plan**

* Unit tests for smart contracts (Hardhat + Waffle / Mocha + Chai).
* Integration tests for backend (spin up local chain with Hardhat network).
* E2E tests for frontend (Cypress / Playwright) using mocked providers.
* AI tests: validate model outputs on synthetic datasets; set thresholds for fraud alarms.

**9. Security & Privacy Considerations**

* Validate and sanitize all metadata before uploading to IPFS.
* Use signature-based login (avoid storing private keys). JWT expiry & refresh strategy.
* Rate-limit AI endpoints to avoid abuse and cost spikes (OpenAI calls can be expensive).
* Smart contract audits or at least third-party review for marketplace and minting logic.
* Gas optimization: batch metadata writes off-chain; only essential data on-chain.

**10. Mock Data Samples**

* Provide small JSON files for events & transactions for initial testing (store in /docs/mock-data).
* Use Hardhat scripts to mint a handful of tickets and populate MongoDB with sample users/events.

**11. Next Steps (Priority)**

1. Create Hardhat project & write TicketNFT.sol with basic minting and events.
2. Scaffold backend with Express and MongoDB models.
3. Implement simple frontend that can connect wallet and read tokenURI.
4. Build AI prediction prototype on a small synthetic dataset and expose /predict-demand.
5. Integrate Chainlink price feed locally or via testnet.

**12. Appendix: Helpful Hardhat & Web3 Snippets**

(Include sample snippets file in /docs/snippets.md — minting call, event listener, IPFS upload examples.)

**AI-Enhanced Decentralized Event Ticketing Platform — Project Overview**

**1. Project Title**

**AI-Enhanced Decentralized Event Ticketing Platform using NFT & Chainlink**

**2. Project Summary**

The AI‑Enhanced Decentralized Event Ticketing Platform is a blockchain-based solution designed to address critical challenges in the ticketing industry, such as fraud, scalping, counterfeit tickets, and static pricing. It leverages **NFTs** to represent tickets uniquely on the Ethereum blockchain, **Chainlink oracles** to integrate real-world data, and **AI/ML models** to enable dynamic pricing, fraud detection, and conversational search capabilities.

This platform empowers event organizers to sell tickets transparently, securely, and efficiently while enhancing the buying experience for attendees.

**3. Problem Statement**

**Challenges in Traditional Ticketing Systems:**

1. **Fraud & Counterfeit Tickets:** Easy duplication or forgery leads to financial loss and trust issues.
2. **Scalping & Hoarding:** Bots and bulk purchases result in inflated resale prices.
3. **Static Pricing:** Ticket prices don’t adapt to demand, losing potential revenue.
4. **Centralized Systems:** Vulnerable to outages, hacking, and manipulation.
5. **Poor User Experience:** Limited search and event discovery features.

**4. Proposed Solution**

**How the Platform Solves These Problems:**

1. **Immutable Ownership (Blockchain/NFTs):** Each ticket is minted as an ERC-721 NFT, verifiable on-chain, ensuring authenticity.
2. **Resale Prevention:** Smart contracts enforce resale rules to curb scalping.
3. **Dynamic Pricing (AI + Chainlink):** AI-powered demand forecasting adjusts ticket prices in real-time, factoring in ETH/USD rates from Chainlink.
4. **Fraud Detection (AI):** ML algorithms monitor transactions for suspicious behavior.
5. **Decentralized File Storage (IPFS):** Ticket metadata and QR codes stored on IPFS for permanence.
6. **Natural Language Search (AI Chatbot):** LangChain + OpenAI API enables conversational event search.

**5. Objectives**

* **Security:** Ensure tickets cannot be forged or altered.
* **Transparency:** Allow users to verify ticket authenticity on-chain.
* **Fair Pricing:** Implement dynamic pricing to balance demand and affordability.
* **Scalability:** Support high transaction volumes for large events.
* **Enhanced UX:** Improve ticket discovery and purchase experience via AI.

**6. Target Audience**

* **Event Organizers:** Music festivals, sports events, conferences.
* **Ticket Buyers:** Individuals looking for secure, authentic, and fair-priced tickets.
* **Blockchain Enthusiasts:** Users interested in Web3-based services.

**7. Key Features**

1. **NFT Ticketing:** Unique, verifiable ERC‑721 tokens.
2. **Dynamic Pricing Engine:** AI adjusts prices based on demand.
3. **Fraud Detection Module:** Flags suspicious wallet activity.
4. **Chainlink Integration:** Real-time market data & event verification.
5. **Chatbot Search:** Conversational, AI-assisted event discovery.
6. **Decentralized Storage:** IPFS for ticket assets.

**8. Technology Stack**

* **Frontend:** React.js, TailwindCSS, Web3.js, Ethers.js
* **Backend:** Node.js, Express.js
* **Blockchain:** Ethereum, Solidity, Hardhat
* **Database:** MongoDB
* **File Storage:** IPFS
* **AI/ML:** scikit-learn, LangChain, OpenAI API
* **Oracles:** Chainlink
* **Wallet Integration:** MetaMask
* **Testing:** Mocha, Chai, Postman

**9. Expected Impact**

* **Reduced Fraud:** Blockchain immutability + AI fraud checks.
* **Optimized Revenue:** Real-time pricing to match demand.
* **Better User Experience:** Seamless, interactive ticket purchasing.
* **Trust Building:** Transparent system boosts confidence among buyers and organizers.

**10. Conclusion**

The AI-Enhanced Decentralized Event Ticketing Platform addresses long-standing issues in the ticketing industry with a fusion of blockchain, AI, and decentralized storage. It not only secures transactions but also revolutionizes the event discovery and purchase experience, making it a robust, future-proof solution for the global events market.

Smart Contract –

**AI-Enhanced Decentralized Event Ticketing Platform — Smart Contract Structure & Connectivity**

**Libraries & Standards Used**

* **OpenZeppelin ERC721**: For creating NFT-based tickets.
* **OpenZeppelin Ownable/AccessControl**: To manage admin privileges.
* **OpenZeppelin ReentrancyGuard**: To prevent reentrancy attacks.
* **Chainlink Price Feeds**: For ETH/USD conversion.
* **Custom Chainlink Oracle**: For event time verification.

**Core Contracts & Functions**

**1. TicketNFT.sol (ERC-721 Contract)**

* **mintTicket(address to, uint256 eventId, string metadataURI)** — Mints a unique NFT ticket for a specific event and assigns metadata.
* **validateTicket(uint256 tokenId)** — Checks if a ticket is valid for entry.
* **burnTicket(uint256 tokenId)** — Used by event organizers to invalidate tickets post-event.
* **setMetadataURI(uint256 tokenId, string uri)** — Updates the ticket’s IPFS metadata.
* **Events:** TicketMinted, TicketBurned, MetadataUpdated

**2. TicketMarketplace.sol**

* **listTicket(uint256 tokenId, uint256 price)** — Allows ticket holders to list their NFT for resale (if allowed).
* **buyTicket(uint256 tokenId)** — Transfers ownership and handles payment.
* **preventResale(uint256 tokenId)** — Organizer function to lock tickets against resale.
* **setRoyalty(address recipient, uint256 percentage)** — Distributes resale royalties to event organizers.

**3. PriceOracleIntegration.sol**

* **getETHPrice()** — Reads ETH/USD price from Chainlink.
* **updateDynamicPrice(uint256 eventId)** — Adjusts ticket price based on AI model output and Chainlink data.

**4. EventTimeVerifier.sol**

* **verifyEventTime(uint256 eventId)** — Confirms event schedule via Chainlink custom job.

**How the Contracts Connect**

1. **TicketNFT.sol** is the core ticket representation. Both TicketMarketplace and AI-based pricing logic interact with this contract to read/write ticket data.
2. **TicketMarketplace.sol** references TicketNFT.sol to verify ownership before allowing listing or sale.
3. **PriceOracleIntegration.sol** is called by the marketplace or backend to fetch up-to-date ETH prices before finalizing sales.
4. **EventTimeVerifier.sol** is optionally called before allowing ticket validation or entry.
5. Each contract can be deployed independently but interconnected via their constructor parameters (passing deployed contract addresses).
6. Access control ensures only the marketplace can trigger mint/burn functions in the NFT contract.

**Template .sol File Outlines (No Code, Only Structure)**

**TicketNFT.sol**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Imports: OpenZeppelin ERC721, Ownable

contract TicketNFT {

// State variables

// Constructor

// mintTicket()

// validateTicket()

// burnTicket()

// setMetadataURI()

// Events

}

**TicketMarketplace.sol**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Imports: ReentrancyGuard, Ownable, Interface of TicketNFT

contract TicketMarketplace {

// State variables

// Constructor with TicketNFT address

// listTicket()

// buyTicket()

// preventResale()

// setRoyalty()

}

**PriceOracleIntegration.sol**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Imports: Chainlink interfaces

contract PriceOracleIntegration {

// State variables

// Constructor with oracle feed address

// getETHPrice()

// updateDynamicPrice()

}

**EventTimeVerifier.sol**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Imports: Chainlink oracle client

contract EventTimeVerifier {

// State variables

// Constructor

// verifyEventTime()

}

This modular design keeps the system flexible, enabling independent upgrades of ticket logic, marketplace rules, or oracle integrations without redeploying the entire platform.