



UNIVERSITY OF PISA

MSc in Computer Engineering

Project for Distributed Systems and Middleware
Technologies

Distributed Betting System

Professor:

Prof. Alessio Bechini

Students:

Francesco Barcherini (645413)

Andrea Di Matteo (641388)

Francesco Scarrone (645328)

ACADEMIC YEAR 2025/2026

Contents

1	Project Specification	3
1.1	Introduction	3
1.2	Functional Requirements	3
1.2.1	Dynamic Odds Calculation	4
1.2.2	Betting Caps	4
1.3	Non-Functional Requirements	4
1.4	Synchronization and Coordination Challenges	5
2	System Architecture	6
2.1	Nginx Reverse Proxy	6
2.2	Spring Boot Server	7
2.3	MySQL Database	7
2.4	Erlang Cluster	8
2.4.1	REST API Specification	8
2.4.2	WebSocket Protocol	9
2.4.3	Files structure	9
2.5	Mnesia Database	10
2.6	Network Topology	11
3	Coordination and Communication	12
3.1	Frontend updates	12
3.1.1	Connection Lifecycle	12
3.1.2	Broadcast Dispatcher	13
3.1.3	Message Types	13
3.2	Cross-Service Authentication	14
3.2.1	Token Structure	14
3.2.2	Token Flow	14
3.3	Data Consistency	14
3.3.1	Cluster Initialization	14
3.3.2	Transactions and Fault Tolerance	15
3.4	Update race conditions in the frontend	15
3.5	Fairness in user bets	15
4	Web Application	17
4.1	Authentication and Session Management	17
4.2	Dashboard	19
4.3	Bet Details	20
4.4	My Bets	22
4.5	Deposit	23
4.6	Administrator Interface	23

1 Project Specification

1.1 Introduction

BetMarket is a distributed online betting platform. The system allows users to place bets on events with dynamically calculated odds, while ensuring data consistency across multiple backend nodes.

The platform supports both real events (such as football matches) and virtual events (such as simulated roulette spins). A key feature is the dynamic quoting system: betting odds are continuously updated based on the distribution of placed bets.

To avoid significant financial losses for the bookmaker, each bet has a maximum cap determined by the total amount currently wagered and its distribution across options. The system manages the complete lifecycle of betting events, from creation through result settlement and payout processing.

1.2 Functional Requirements

The system must support the following functional requirements organized by user role:

- **Unregistered User**
 - View the list of all available betting events with their names and betting options
 - View current odds for each available option in an event
 - View total amount wagered on each event
 - Perform user registration and login
- **Registered User**
 - Place bets on available events with specified amounts
 - Have bets instantly confirmed with odds locked at placement time
 - View their account balance
 - Deposit funds to increase the balance
 - View their betting history including all placed bets with amounts, odds, and outcomes
 - Receive real-time updates of odds changes when other users place bets
 - Automatically receive account creation with initial balance on first interaction
- **Administrator**
 - Create new betting events with two options

- Close betting on any active event
- Set event outcomes and trigger automatic payout processing
- View the bookmaker's current balance

1.2.1 Dynamic Odds Calculation

The odds system dynamically adjusts based on betting activity to manage the bookmaker's risk. For a game with total amounts T_1 and T_2 bet on options 1 and 2 respectively:

$$\text{odd}_1 = 1 + \frac{T_2 + V}{T_1 + V} \times (1 - c) \quad (1)$$

$$\text{odd}_2 = 1 + \frac{T_1 + V}{T_2 + V} \times (1 - c) \quad (2)$$

Where V is the virtual initial bet (configured in system config, default 100) to prevent division by zero, and c is the commission percentage (default 0.5) ensuring the bookmaker's profit margin.

1.2.2 Betting Caps

To limit potential losses, each option has a dynamically calculated maximum bet cap:

$$\text{cap}_i = \max \left(10, \frac{T_1 + T_2 + M - L_i}{\text{odd}_i - 1} \right) \quad (3)$$

Where M is the configured margin for each event (default 50) and L_i is the potential loss for option i (sum of all payouts if that option wins). The minimum cap of 10 ensures bets are always accepted.

1.3 Non-Functional Requirements

The system must satisfy the following non-functional requirements:

- System must continue operating even if one node fails (3-node cluster with 2-node fault tolerance)
- System must prevent inconsistencies or data losses due to node failures (e.g. on balance changes, payouts, bets, odds calculation etc.)
- WebSocket connections for live updates without polling
- Horizontal scalability
- JWT token-based authentication with expiration to separate the game logic from the login logic

1.4 Synchronization and Coordination Challenges

The distributed nature of the system introduces several critical coordination challenges:

- **Dynamic odds and caps propagation:** when a user places a bet, the odds and caps for that event change. These updates must be broadcast to all users currently viewing that event, regardless of which node they are connected to.
- **Odds locking for concurrent bets:** if two users place a bet concurrently, each bet must be accepted only if it matches the exact odd shown to that user at quote time. If the odd changes after the user submits the bet, the bet must be refused (and the client should refresh odds).
- **Event state synchronization:** when an administrator creates or closes an event, all nodes must immediately reflect this change, and all connected clients must be notified.
- **Payout for multiple bets on the same event:** if a user places multiple bets on the same event, the final balance shown after settlement must reflect the sum of all payouts (and losses) across all those bets.
- **Payout processing after event conclusion:** when an event concludes, all winning bets must be identified and winners must be paid, showing the updated balance.
- **Atomicity under node failures:** if a node faults during a critical operation (bet placement, payout settlement, balance update), the whole operation must not be completed: partial writes must be prevented and the system must remain consistent.
- **Cross-service authentication:** users authenticate with the Spring service but access betting operations on Erlang nodes. Tokens generated by one service must be validated by another without shared state with a token-based approach.

2 System Architecture

The architecture consists of three main components:

- **Nginx Reverse Proxy:** Acts as the entry point for all client requests to the system, performing load balancing across Erlang nodes and routing request to the appropriate service.
- **Spring Boot Server:** Handles user login and registration, and serves the web frontend application. User information is stored in a MySQL database.
- **Erlang Cluster:** A distributed cluster of three nodes that manages all betting operations, including game management, bet placement, odds calculation, and real-time updates via WebSocket. A Mnesia database is used for storing games, bets and user balance across all Erlang nodes.

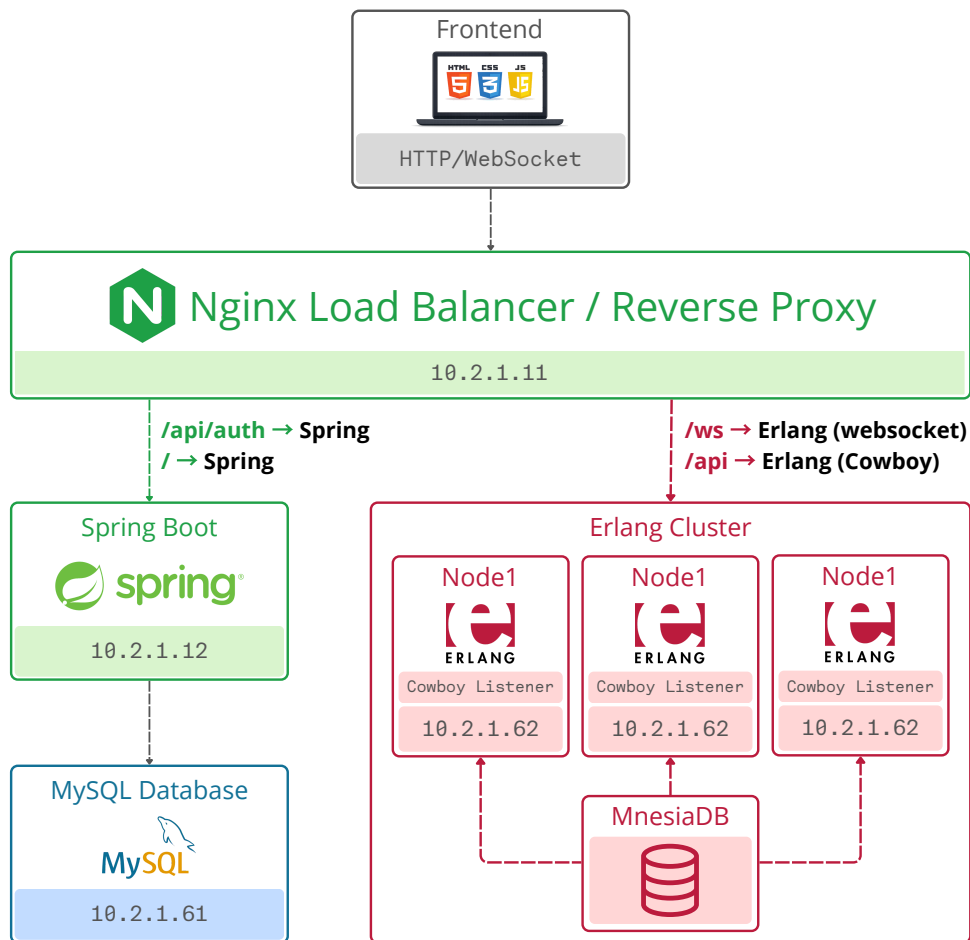


Figure 1: High-level architecture of the BetMarket system

2.1 Nginx Reverse Proxy

The Nginx server is deployed on a dedicated machine and serves as the single entry point for all client requests. It performs several critical functions:

- **Load Balancing:** Distributes requests across the three Erlang nodes using round-robin strategy, ensuring even load distribution.
- **Request Routing:** Routes `/api/auth/*` endpoints to the Spring server for authentication, while all other `/api/*` endpoints are directed to the Erlang cluster.
- **WebSocket Proxying:** Handles WebSocket upgrade requests via `/ws` endpoint and maintains persistent connections to Erlang nodes with 60 seconds timeout.
- **Static Content:** Forwards requests for the web application to the Spring server.

The Nginx configuration defines two upstream groups:

- **spring_backend:** Single Spring server at 10.2.1.12:8080
- **erlang_backend:** Three Erlang nodes at 10.2.1.62:8080, 10.2.1.27:8080, and 10.2.1.28:8080

From a security standpoint, placing Nginx in this position allows it to act as the single entry point for communication with the application, preventing direct external access to internal services. Since this is a purely educational project, the HTTP protocol was used. However, in a production scenario, HTTPS could be employed for external communication with the load balancer, while internal communication within the application network could still rely on HTTP to reduce overhead and improve performance.

2.2 Spring Boot Server

The Spring Boot application provides:

- **User Registration:** Creates new user accounts with hashed passwords stored in MySQL.
- **Authentication:** Validates credentials and issues JWT tokens containing user ID and admin status.
- **Web Frontend:** Serves HTML, CSS, and JavaScript files for the user interface.

The JWT tokens are signed using HMAC-SHA256 with a shared secret among the Spring Boot backend and the Erlang cluster, allowing the Erlang nodes to independently validate tokens without contacting the Spring server.

2.3 MySQL Database

MySQL stores user and authentication information, it only contains a single **User** table containing the following fields

- **username:** username of the user
- **is_admin:** boolean value that indicates if the user is a platform admin

- **password:** hashed password

The MySQL server is separate from the Erlang cluster and communicates only with the Spring Boot application, ensuring clear separation of authentication logic from game logic.

2.4 Erlang Cluster

The Erlang cluster is the core of the betting system, consisting of three nodes that communicate using Erlang's native distributed features:

- **betting_node1@10.2.1.62:** Master node responsible for initializing the Mnesia schema and coordinating cluster setup.
- **betting_node2@10.2.1.27:** Worker node that joins the cluster on startup.
- **betting_node3@10.2.1.28:** Worker node that joins the cluster on startup.

Each node runs:

- **Cowboy HTTP Server:** Handles REST API requests on port 8080.
- **WebSocket Handler:** Manages persistent WebSocket connections for real-time updates.
- **Broadcast Dispatcher:** Coordinates message distribution across nodes and to connected clients.
- **Mnesia:** Local replica of the distributed database.
- **OTP Supervisor:** Automatically restarts crashed processes to ensure system resilience.

2.4.1 REST API Specification

The system provides REST endpoints for user and administrative operations:

Method	Endpoint	Description
GET	/api/games	List all active betting events with current odds
GET	/api/games/{id}	Get specific game details including odds, caps, betting total

Table 1: Public REST endpoints

Method	Endpoint	Description
POST	/api/bet	Place a bet on a game
GET	/api/balance	Get user's current balance
POST	/api/balance	Deposit funds into the user's balance
GET	/api/user/bets	Get user's betting history
GET	/api/user/bets/{game_id}	Get user's bets on a specific game

Table 2: Authenticated user REST endpoints

Method	Endpoint	Description
POST	/api/admin/game	Create a new betting event
POST	/api/admin/stop_betting	Close betting on a game
POST	/api/admin/start_game	Set game result and process payouts
GET	/api/admin/profit	Get bookmaker's profit/loss

Table 3: Administrator REST endpoints

2.4.2 WebSocket Protocol

WebSocket connections provide real-time updates to connected clients:

The connection flow is the following:

1. Client connects to `ws://host/ws`, where `host` is the hostname of the load balancer, Nginx will route this request to an Erlang node.
2. Server registers connection in global registry
3. Client sends `{opcode:"authenticate", token:"jwt_token"}` message
4. Server validates token and re-registers connection with user ID (possibly as admin)

To maintain long-lived connections:

- Client sends `{opcode:"keepalive"}` every 30 seconds
- Server resets the 60-second idle timeout on receipt
- Connections that exceed idle timeout are closed

2.4.3 Files structure

The file structure is the following:

- **betting_app.erl**: Application behaviour callback that starts the supervisor during application startup
- **betting_sup.erl**: Top-level supervisor managing all critical child processes

- **betting_node_mnesia.erl**: Handles Mnesia cluster initialization and membership management
- **broadcast_dispatcher.erl**: GenServer process that distributes messages to connected clients
- **cowboy_listener.erl**: HTTP server listening on configured port
- **websocket_handler.erl**: WebSocket connection handler
- **jwt_validator.erl**: JWT token validation logic
- **odds_calculator.erl**: Dynamic odds and betting caps calculation
- Multiple HTTP handlers (bet_handler, games_handler, balance_handler, admin handlers) implementing Cowboy request processing

The OTP supervisor (**betting_node_sup**) manages a single child process: the Cowboy listener, thus if the HTTP server encounters an unrecoverable error, it crashes and the supervisor restarts it, ensuring the system remains available. It uses the **one_for_one** restart strategy, meaning that if a supervised child process crashes, only that specific process is restarted. The supervisor is configured with **intensity => 5** and **period => 10**, allowing up to 5 restarts within a 10-second window before the supervisor itself terminates.

2.5 Mnesia Database

Mnesia is Erlang's built-in distributed database, chosen for its tight integration with Erlang and support for transactions across distributed nodes. The database schema includes:

The system uses four Mnesia tables with **disc_copies** replication:

- **account**: Stores user balances keyed by user ID.
- **game**: Stores betting events with their current state, odds totals, and results.
- **bet**: Records individual bets with the odds at placement time.
- **counter**: Maintains auto-incrementing IDs for games and bets.

```
1 -record(account, {  
2     user_id,           % User ID from JWT  
3     balance            % Current balance (float)  
4 }).  
5  
6 -record(counter, {  
7     name,              % Counter name (game_id | bet_id | balance_seq)  
8     value              % Current value (integer)  
9 }).  
10  
11 -record(game, {  
12     game_id,           % Unique game ID (integer)
```

```

13     question_text,    % Question text
14     opt1_text,       % Option 1 text
15     opt2_text,       % Option 2 text
16     category,        % real | virtual
17     result,          % Result: undefined | opt1 | opt2
18     betting_open,    % Boolean
19     tot_opt1,         % Total amount bet on option 1
20     tot_opt2,         % Total amount bet on option 2
21     created_at       % Timestamp
22 }).
23
24 -record(bet, {
25     bet_id,           % Unique bet ID (integer)
26     user_id,          % User ID
27     game_id,          % Game ID
28     amount,           % Bet amount (float)
29     choice,           % opt1 | opt2
30     odd,              % Odd at the time of betting (float)
31     placed_at        % Timestamp
32 }).

```

Listing 1: Mnesia Table Definitions

All tables use `disc_copies` replication, ensuring data persists across node restarts and is synchronized across all cluster members.

2.6 Network Topology

The system is deployed on a private network with the following IP assignments:

Component	IP Address	Port
Nginx (Entry Point)	10.2.1.11	80
Spring Boot	10.2.1.12	8080
MySQL Database	10.2.1.61	3306
Erlang Node 1 (Master)	10.2.1.62	8080
Erlang Node 2	10.2.1.27	8080
Erlang Node 3	10.2.1.28	8080

Table 4: Network configuration of system components

3 Coordination and Communication

This section describes how the system addresses the main coordination and synchronization challenges.

The system must solve several coordination problems:

1. Frontend updates

- **Dynamic Odds Propagation:** When a bet is placed, odds and caps must be recalculated and broadcast to all connected clients in real time.
 - **New Event Notification:** When an administrator creates a new betting event, all nodes must be aware of it, and all clients must update the dashboard.
 - **Game Result Distribution:** When a game concludes, the result and its effects must be sent to all affected clients; this includes updating the category in the dashboard, the bet outcome (win/loss) and the user balance.
 - **Balance update after bet/deposit/win:** Several game events may cause a change in the user balance; this should be reflected in real time across all open sessions of the interested user to show the up-to-date value.
2. **Cross-Service Authentication:** Users authenticate with Spring but access game services on Erlang; tokens are necessary due to the stateless nature of the connection.
 3. **Data Consistency:** Users must see consistent balance and bet information regardless of which node handles their requests.
 4. **Update race conditions in the frontend:** The balance may change several times almost simultaneously, so the frontend must display the most recent update (based on the calculation order) received via WebSocket, even in the presence of race conditions in message delivery.
 5. **Fairness in user bets:** The user must be aware of the exact possible return when placing a bet. This can be problematic in the case of concurrent bets from different users, since one of them may use updated (potentially worse) odds after the other bet is placed; in this case, the system should warn the user and ask them to place the bet again if they wish to proceed.

3.1 Frontend updates

The system uses WebSocket connections to push real-time updates to clients.

3.1.1 Connection Lifecycle

1. Client connects to `/ws` endpoint; Nginx routes to an Erlang node.

2. Connection is registered in Erlang's global registry as `{ws, Node, Pid}`.
3. Client sends `authenticate` message with JWT token.
4. Upon successful authentication, connection is re-registered as `{ws_user, Node, UserId, Pid}` or `{ws_admin, Node, UserId, Pid}`.
5. Connection remains open with 60-second idle timeout; clients send periodic `keepalive` messages.

3.1.2 Broadcast Dispatcher

Each Erlang node runs a broadcast dispatcher process that:

- Registers itself globally as `{dispatcher, Node, Pid}`.
- Listens for broadcast messages from API handlers.
- Forwards messages to appropriate WebSocket handlers.

When a broadcast is triggered (e.g., odds update), the API handler sends a message to all dispatchers across the cluster. Each dispatcher then delivers the message to WebSocket connections on its node.

3.1.3 Message Types

The system broadcasts several message types:

Message	Description
<code>update_odds</code>	Sent when a bet is placed; contains new odds, caps, and total volume
<code>new_game</code>	Sent when admin creates a game and contains game details
<code>betting_closed</code>	Sent when admin stops betting on a game
<code>game_result</code>	Sent when admin sets game result and contains the winning option
<code>balance_update</code>	Sent to a specific user when their balance changes and includes the updated balance along with a strictly increasing sequence number (to prevent race conditions, see 3.4)
<code>bet_placed</code>	Confirms bet placement to the betting user (used to update "My bets" sections)
<code>profit_update</code>	Sent only to admins when there is a change in the bookmaker's profit (i.e., when an admin sets the results of a game)

Table 5: WebSocket broadcast message types

3.2 Cross-Service Authentication

The system uses JSON Web Tokens (JWT) to enable stateless authentication across services.

3.2.1 Token Structure

Each JWT contains:

- **id**: User identifier (string)
- **isAdmin**: Boolean flag indicating administrator privileges
- **iat**: Issued at timestamp (Unix seconds)
- **exp**: Expiration timestamp (Unix seconds)

3.2.2 Token Flow

1. User submits credentials to Spring's `/api/auth/login` endpoint.
2. Spring validates credentials against MySQL and generates a signed JWT.
3. Client stores the JWT in localstorage and includes it in subsequent requests to Erlang nodes.
4. Erlang nodes independently validate the JWT signature using the shared secret.
5. For WebSocket connections, clients send a message with opcode `authenticate` and the signed token.

The JWT secret is configured via environment variable (`JWT_SECRET`), ensuring both Spring and Erlang nodes use the same key without hardcoding secrets in the codebase.

3.3 Data Consistency

Mnesia distributed database provides the functionalities for data consistency across the Erlang cluster.

3.3.1 Cluster Initialization

The first time the system starts, the master node (`betting_node1`) orchestrates the cluster setup:

1. Waits for all worker nodes to connect.
2. Deletes any existing schema on all nodes.
3. Creates a fresh schema including all connected nodes.
4. Starts Mnesia on all nodes via RPC.
5. Creates tables on all nodes.

6. Initializes the bookmaker account with configured initial balance.

Worker nodes follow a simpler process:

1. Connect to the master node.
2. Wait for the master to initialize schema and tables.
3. Ensure local table copies are available.

3.3.2 Transactions and Fault Tolerance

All state-modifying operations use Mnesia transactions to ensure atomicity and fault tolerance:

- **Bet Placement:** Reading game state, validating balance, deducting amount, updating game totals, and creating bet record occur atomically.
- **Game Result:** Marking game closed, calculating payouts, and updating winner balances happen in a single transaction.

If a transaction fails (e.g., insufficient balance, game closed), it is automatically rolled back with no partial effects.

3.4 Update race conditions in the frontend

In this real-time betting system, a user's balance may change multiple times almost simultaneously, for example due to several bets placed on the same event. To ensure consistency, the frontend must always display the most recent balance update received via WebSocket, even when race conditions occur in message delivery.

This problem is addressed by using a shared counter table in `mnesia`. The table stores a counter value that is accessible from all nodes in the system. Each time a balance update occurs, the counter is queried and incremented by 1 within the same transaction that updates the balance. This approach guarantees that each balance change is associated with a unique, sequential identifier, allowing the frontend to correctly order updates and display the most recent balance, even under conditions of very rapid or concurrent updates.

3.5 Fairness in user bets

It is essential that users are informed of the exact potential return before confirming a bet. However, it is possible that multiple users place bets on the same event simultaneously. Each new bet can alter the total volume on a given outcome, which in turn may change the odds. As a result, a user might attempt to place a bet based on outdated odds, leading to a mismatch between the expected and actual potential return.

To address this, the frontend captures the current odds at the moment the user selects

a betting option and includes this value with the bet request sent to the backend. The backend then verifies that the odds submitted by the user match the actual current odds, possibly allowing a small tolerance (e.g., 0.01) to account for negligible differences. If the odds have changed beyond this threshold, the backend rejects the bet, requiring the user to review and confirm the updated odds before proceeding. This mechanism ensures that users always place bets with precise knowledge of the potential return even with rapidly changing odds.

4 Web Application

The BetMarket web application provides an interface for betting and account management. This section describes the frontend implementation and key interfaces.

4.1 Authentication and Session Management

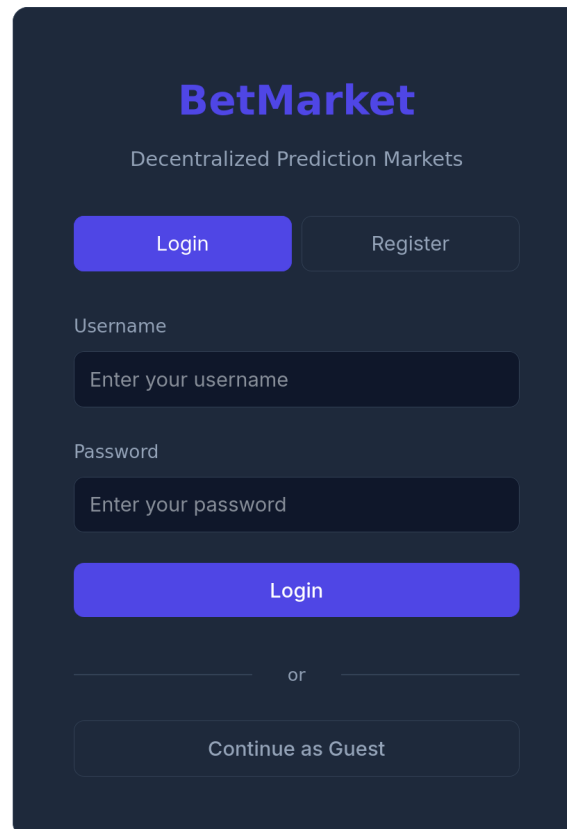
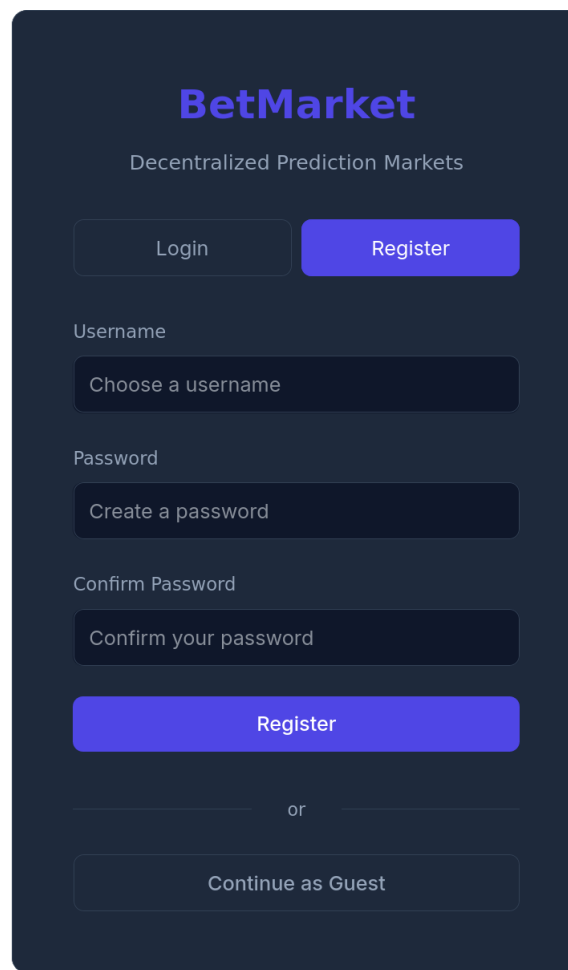
The image shows a dark-themed login page for 'BetMarket'. At the top, the logo 'BetMarket' is in a light blue font, with the tagline 'Decentralized Prediction Markets' below it. There are two buttons: a blue 'Login' button and a grey 'Register' button. Below these are input fields for 'Username' and 'Password', each with a placeholder text 'Enter your username' and 'Enter your password' respectively. A large blue 'Login' button is positioned below the password field. At the bottom, there is a horizontal line with the word 'or' in the center, and a grey button labeled 'Continue as Guest' below it.

Figure 2: User login page for authentication

Users access the system through a login page where credentials are submitted to the Spring Boot authentication service at `/api/auth/login`. Upon successful authentication, the server returns a JWT token containing user ID and admin status, which is stored in browser local storage. All subsequent requests to the Erlang backend include this token in the Authorization header.



The image shows a user registration form for 'BetMarket'. The form is set against a dark blue background. At the top, the 'BetMarket' logo is displayed in a light blue font, with the tagline 'Decentralized Prediction Markets' below it. The form includes a 'Login' button and a 'Register' button at the top. Below these are three input fields: 'Username' with a placeholder 'Choose a username', 'Password' with a placeholder 'Create a password', and 'Confirm Password' with a placeholder 'Confirm your password'. A large 'Register' button is positioned below the input fields. At the bottom, there is a horizontal line with the word 'or' in the center, and a 'Continue as Guest' button below it.

Figure 3: User registration form

New users register through a registration form asking for username, password and password confirmation. After registration, users are automatically logged in. The JWT authentication mechanism allows stateless validation: each Erlang node independently verifies tokens using the shared HMAC-SHA256 secret, eliminating the need for session state replication across the cluster.

4.2 Dashboard

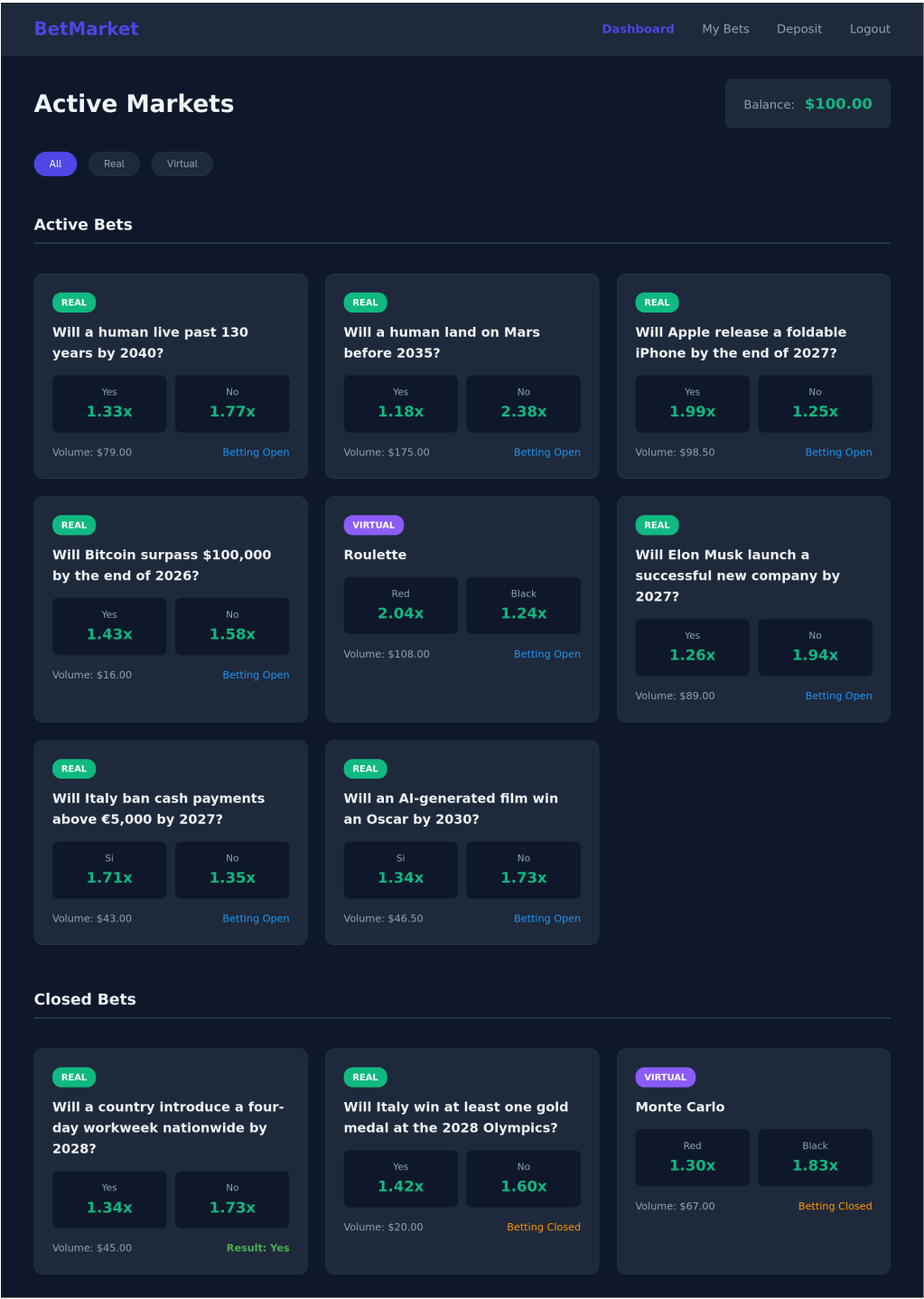


Figure 4: Main dashboard showing available betting events

The main dashboard displays all available betting events with their current odds, betting caps, and wagered totals. The interface establishes a WebSocket connection to the Erlang backend immediately after authentication, sending the JWT token for validation. Once authenticated over WebSocket, the client receives real-time

broadcasts whenever odds change, new events are created, or games conclude. This architecture ensures that multiple users viewing the same event see identical, synchronized odds. When one user places a bet, all connected clients receive an `update_odds` message containing the new values, which the JavaScript frontend applies instantly without page reload.

4.3 Bet Details

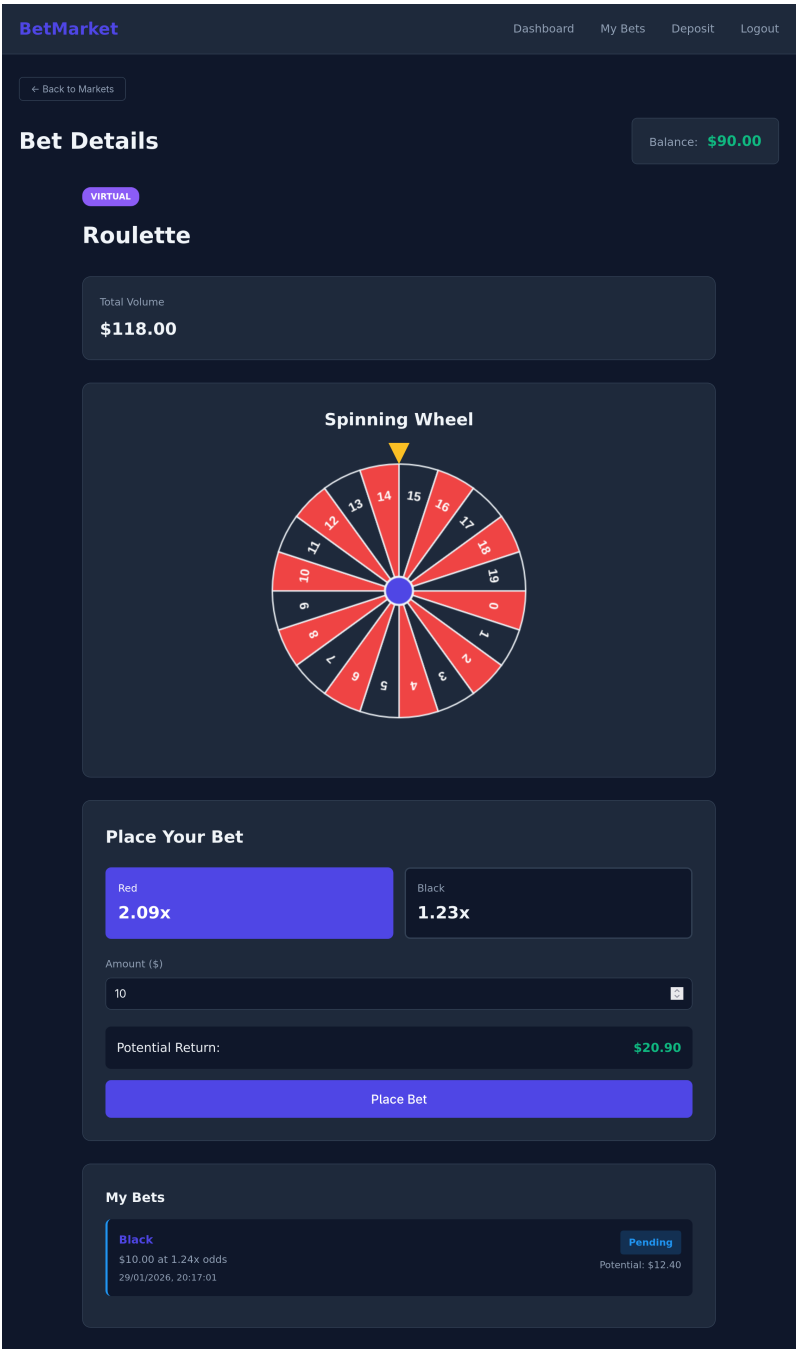


Figure 5: Dialog for placing a new bet

When placing a bet, users interact with a modal dialog that displays current odds, betting caps, and calculates projected winnings in real-time as they enter an amount. Client-side validation ensures that the bet amount does not exceed the user's balance or the current betting cap for the selected option. This pre-validation reduces failed requests and provides immediate feedback.

However, client-side validation is not sufficient for security: the Erlang backend performs authoritative validation within a Mnesia transaction. This transaction checks the latest odds, recalculates caps, verifies balance sufficiency, and atomically deducts the amount while recording the bet. The transaction ensures consistency even if multiple users place bets simultaneously or if the client's cached data is slightly stale.

After a successful bet, the user receives two WebSocket messages: a **bet_confirmed** with the locked-in odds and bet ID, and a **balance_update** reflecting their new balance. These updates appear as toast notifications, confirming the operation without requiring page navigation.

In the My Bets subsection there are old bets on the showed event with odd and status.

4.4 My Bets

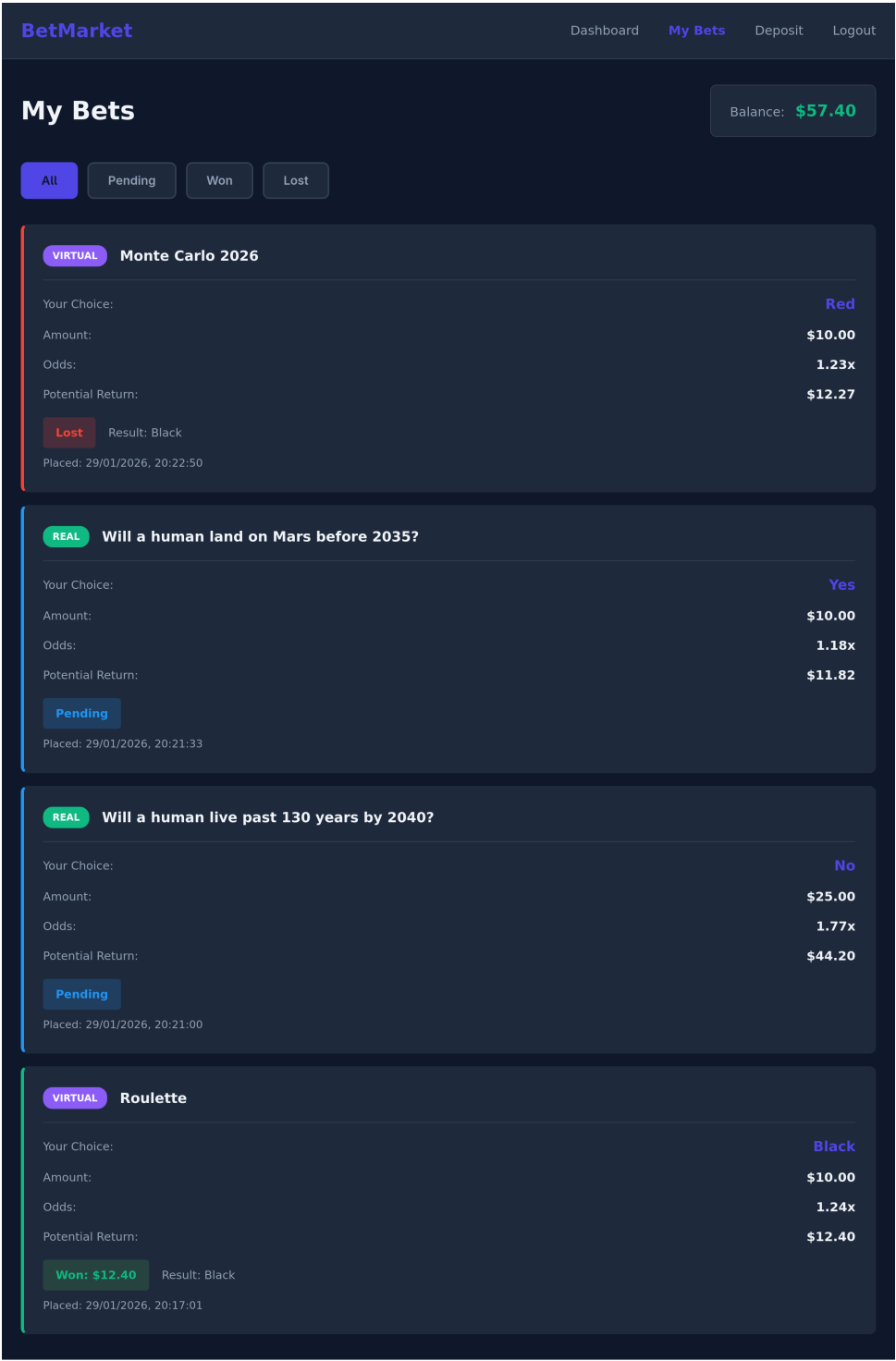


Figure 6: Detailed view of user’s bets

In the My Bets page, users can view their complete betting history, showing pending bets (events not yet concluded) and settled bets (won or lost). The history page reflects real-time updates: when an event concludes, pending bets immediately

transition to won/lost status and display actual winnings.

4.5 Deposit

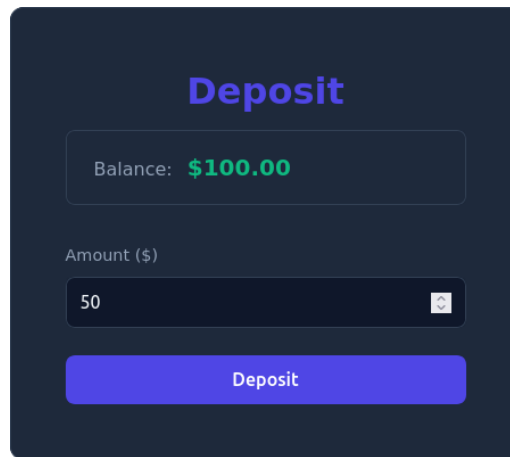
The image shows a dark-themed user interface for a deposit function. At the top, the word "Deposit" is written in a bold, light blue font. Below it, a rounded rectangle contains the text "Balance: \$100.00" in a light green font. Underneath the balance box, the label "Amount (\$)" is followed by a dark input field containing the number "50" and a small currency icon. At the bottom of the interface is a wide, rounded blue button with the word "Deposit" in white text.

Figure 7: Deposit interface

The deposit page allows users to see their current balance and add funds to their wallets. The application provides a simple form to insert the amount to add to the account. The deposit value is then sent to the backend to update the balance accordingly. In a real world scenario this operation would be performed through an external payment service provider.

4.6 Administrator Interface

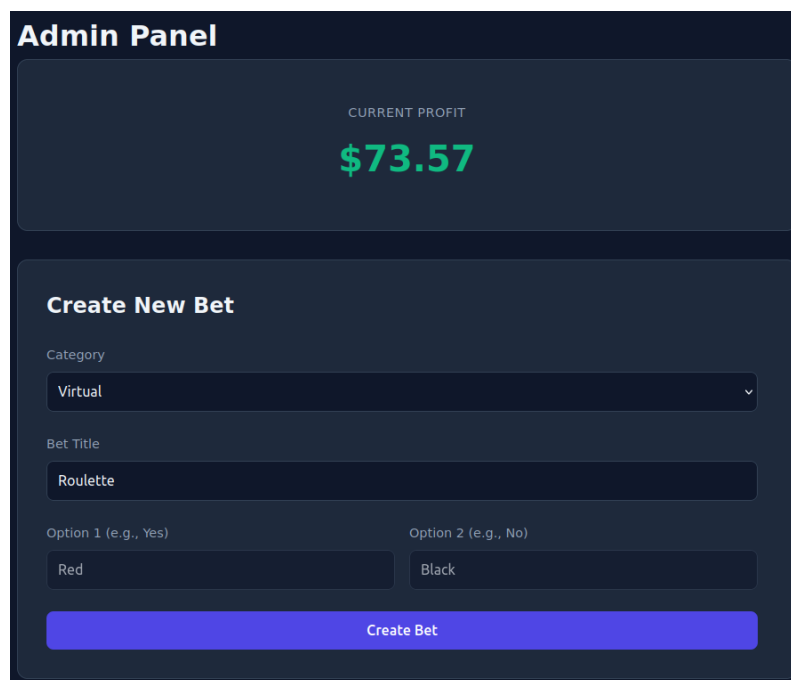
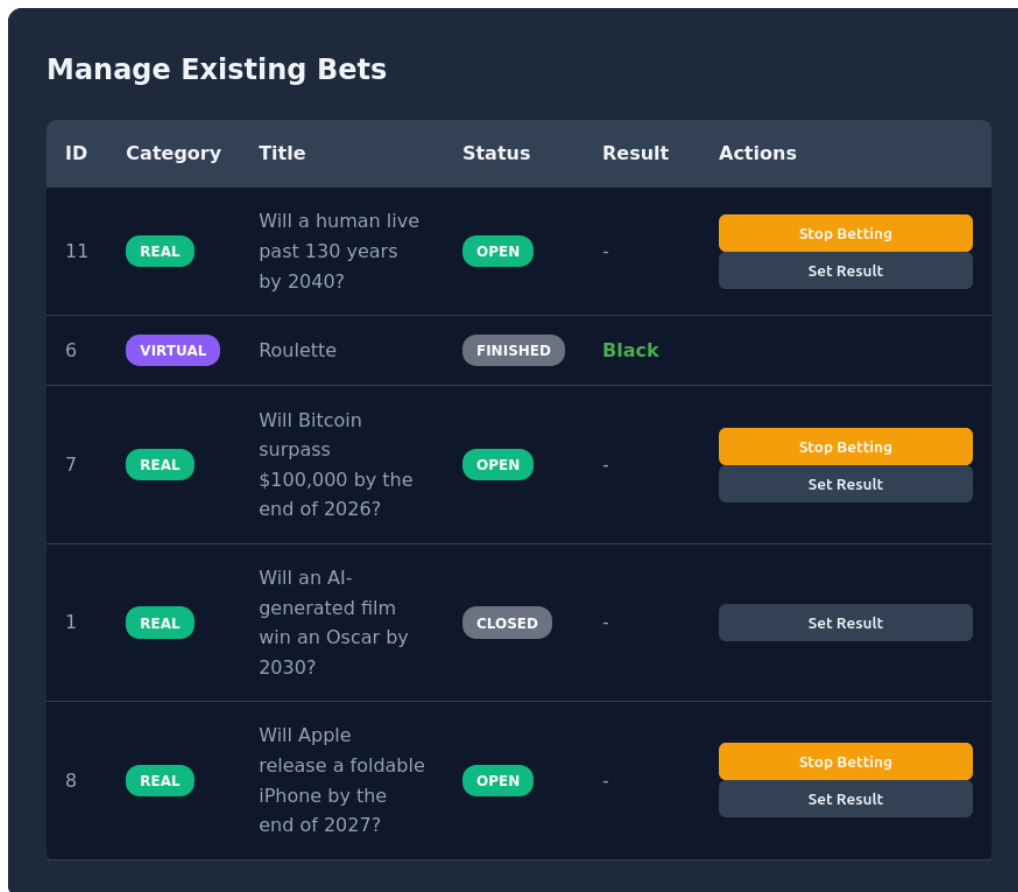
The image displays an "Admin Panel" with a dark background. At the top, the title "Admin Panel" is in white. Below it, a large rounded rectangle shows "CURRENT PROFIT" in small white text above a large, bold, light green "\$73.57". Underneath this is a section titled "Create New Bet" in white. This section contains a "Category" dropdown menu with "Virtual" selected, a "Bet Title" input field with "Roulette" entered, and two input fields for "Option 1 (e.g., Yes)" with "Red" and "Option 2 (e.g., No)" with "Black". At the bottom of the "Create New Bet" section is a wide, rounded blue button labeled "Create Bet" in white text.

Figure 8: Bookmaker balance and admin dashboard for event management

Administrators can see the current profit of the bookmaker and access additional functionalities through a dedicated dashboard that includes an event creation form and an event management table. When creating a new event, the admin specifies the event name and two betting options. The Erlang backend initializes the event with zero betting totals and calculates initial odds based on the configured virtual bet parameter.



ID	Category	Title	Status	Result	Actions
11	REAL	Will a human live past 130 years by 2040?	OPEN	-	<div>Stop Betting</div> <div>Set Result</div>
6	VIRTUAL	Roulette	FINISHED	Black	
7	REAL	Will Bitcoin surpass \$100,000 by the end of 2026?	OPEN	-	<div>Stop Betting</div> <div>Set Result</div>
1	REAL	Will an AI-generated film win an Oscar by 2030?	CLOSED	-	<div>Set Result</div>
8	REAL	Will Apple release a foldable iPhone by the end of 2027?	OPEN	-	<div>Stop Betting</div> <div>Set Result</div>

Figure 9: Dialog for setting game result

For each active event, the admin can stop bets and set the game result, triggering the payout process. Upon confirmation, the backend processes payouts within a transaction, crediting all winners' balances.