SkillWager Protocol - Specifiche Tecniche e Architetturali (Extended)

1. Visione del Prodotto e Value Proposition

SkillWager è un protocollo di *Decentralized Competitive Betting* progettato nativamente per la Avalanche C-Chain. La missione del protocollo è democratizzare la monetizzazione degli e-sports, permettendo a qualsiasi giocatore (non solo ai professionisti) di scommettere sulle proprie abilità in tempo reale.

A differenza delle piattaforme di wagering tradizionali (Web2), che richiedono depositi centralizzati, KYC invasivi e hanno il controllo totale sui fondi degli utenti, SkillWager opera come un **layer infrastrutturale trustless**. Il sistema risolve il "problema dell'oracolo" nel gaming (ovvero: come fa la blockchain a sapere chi ha vinto a FIFA?) senza richiedere integrazioni API complesse o fragili con i server di gioco proprietari (Sony, EA, Activision).

La soluzione si basa su un approccio socio-economico: un sistema di **"Optimistic Escrow"** rinforzato da **Security Bonds (Cauzioni)**. Questo disincentiva matematicamente i comportamenti scorretti, rendendo economicamente svantaggioso mentire sull'esito di una partita.

2. Stack Tecnologico e Scelte Architetturali

La scelta dello stack è ottimizzata per velocità (UX), sicurezza (Fondi) e rapidità di sviluppo (Hackathon).

- **Blockchain: Avalanche C-Chain (EVM)**
  - *Perché Avalanche:* La finalità *sub-second* è il requisito non negoziabile. Un giocatore che ha appena finito una partita adrenalinica non può attendere 15 secondi (o minuti) per la conferma della transazione. Le fee basse della C-Chain permettono micro-scommesse (es. 1-5 USDC) senza che il costo del gas eroda il margine di vincita.
  - *Scalabilità Futura:* Possibilità di migrare su una **Avalanche Subnet** dedicata (GameChains) per zero-gas fee per i giocatori (Gasless transactions via Account Abstraction).
- **Smart Contracts: Solidity 0.8.17+**
  - Framework: **Foundry** (preferito per test veloci in Rust) o **Hardhat**.
  - Sicurezza: Utilizzo di librerie OpenZeppelin per `ReentrancyGuard` (critico nelle funzioni di withdraw) e `Ownable`.
  - Pattern: *Pull-over-Push* per i prelievi per evitare attacchi DoS in cui un contratto malevolo rifiuta di ricevere ETH/AVAX bloccando lo stato del match.
- **Frontend: Next.js + RainbowKit + Wagmi**
  - *Librerie:* **viem** come interfaccia RPC leggera e performante.
  - *UX:* Design "Mobile-first" poiché la maggior parte dei giocatori console userà il telefono come "second screen" per gestire la scommessa mentre gioca.
- **Backend Leggero: Node.js (NestJS o Express)**
  - *Ruolo Limitato:* Il backend **non tocca mai le chiavi private**.

- *Funzioni:*
    1. **WebSocket Coordinator:** Gestisce la comunicazione realtime per la Lobby (es. "Player B è entrato nella stanza", "Player A sta digitando").
    2. **Evidence Storage:** In caso di disputa, agisce da gateway per caricare screenshot/video su **IPFS** o storage temporaneo, fornendo l'hash al contratto.
- *Database:* **Redis** per lo stato effimero delle lobby (TTL di 1 ora per match non on-chain).

3. Logica Economica Core e Teoria dei Giochi

Il cuore della sicurezza di SkillWager è il meccanismo di incentivi. Assumiamo che gli agenti siano razionali e mirino a massimizzare il profitto.

Struttura del Deposito (The "Skin in the Game")

Per partecipare, un utente non deposita solo la scommessa, ma mette a rischio un capitale addizionale. Ogni giocatore deposita `TotalLock = Wager (W) + Bond (B)`.

- **Wager (W):** L'importo della scommessa (es. 10 AVAX).
- **Bond (B):** 20% del Wager (es. 2 AVAX). Questo funge da collaterale anti-frode.
    - *Nota:* Il Bond deve essere sufficientemente alto da scoraggiare il "trolling" ma abbastanza basso da non alzare troppo la barriera d'ingresso.

Matrice dei Payoff e Scenari di Risoluzione

La tabella seguente illustra i flussi finanziari esatti per ogni scenario possibile. Le fee del protocollo sono applicate solo sulle vincite per non penalizzare i rimborsi.

Scenario	Input Giocatori (Voti)	Azione Smart Contract	Esito Economico Dettagliato
Vittoria Netta (Happy Path)	**A:** "Vince A" **B:** "Vince A"	`State -> RESOLVED` Payout Immediato	**Player A (Vincitore):** Riceve $2W + B_A$ (meno fee protocollo 5% su vincita). **Player B (Perdente Onesto):** Riceve indietro il suo B_B. *Il sistema premia l'onestà restituendo la cauzione anche al perdente.*

Pareggio (Fair Play)	**A:** "Pareggio" **B:** "Pareggio"	`State -> RESOLVED` Refund Totale	**Player A:** Riceve $(W + B) * 0.99$. **Player B:** Riceve $(W + B) * 0.99$. *Viene trattenuta una micro-fee (1%) per coprire i costi operativi e disincentivare lo spam di match nulli.*
Disputa (Tentativo di Truffa)	**A:** "Vince A" **B:** "Vince B"	`State -> DISPUTE` Timer Arbitro Attivo	I fondi ($2W + 2B$) restano bloccati nel contratto. Viene emesso un evento `DisputeOpened` che notifica il backend/dashboard admin.
Risoluzione Arbitro (Slashing)	L'Arbitro verifica le prove e chiama: `arbiterResolve(Winner=A)`	`State -> RESOLVED` Esecuzione Slashing	**Player A (Vittima):** Riceve il piatto pieno ($2W$) + il proprio Bond (B_A). **Player B (Baro):** Perde TUTTO (W e B_B). **Arbitro:** Incassa il Bond del Baro (B_B) come *Arbitration Fee*. **Protocollo:** Incassa eventuali residui/fee base.
Timeout (Abbandono)	**A:** "Vince A" **B:** *Nessuna risposta per 2h*	`State -> RESOLVED` Auto-Win	Il contratto assume che il silenzio di B sia un'ammissione di sconfitta o abbandono. Si applica lo scenario "Vittoria Netta".

4. Architettura Smart Contract (State Machine)

Il contratto deve gestire rigidamente il ciclo di vita del match per evitare stati inconsistenti.

Stati Immutabili (Enum):

1. **OPEN:** Il match è stato inizializzato da Player A. I fondi di A sono nel contratto. B non è ancora arrivato.
   - *Transizioni possibili:* -> `LOCKED` (se B entra), -> `CANCELLED` (se A annulla).
2. **LOCKED:** Player B ha depositato fondi corretti. Il match è "live". I fondi sono in escrow.
   - *Transizioni possibili:* -> `RESOLVED` (accordo), -> `DISPUTE` (disaccordo).
3. **DISPUTE:** Stato di eccezione. Richiede intervento esterno.
   - *Transizioni possibili:* -> `RESOLVED` (solo via Arbitro).

4. **RESOLVED:** Stato terminale. I fondi sono stati distribuiti.
    - *Transizioni possibili:* Nessuna.
5. **CANCELLED:** Stato terminale. A ha ritirato i fondi prima dell'inizio.

5. Interfacce Chiave e Firme Funzioni

Dettaglio tecnico delle funzioni Solidity per l'implementazione.

Funzioni Utente (Write)

- `function createMatch(uint256 _wager) external payable returns (bytes32 matchId)`
    - *Validation:* `msg.value` deve essere esattamente `_wager + (_wager * BOND_PERCENT / 100)`.
    - *Event:* `MatchCreated(bytes32 indexed matchId, address indexed playerA, uint amount)`.
- `function joinMatch(bytes32 _matchId) external payable`
    - *Validation:* Match deve essere `OPEN`. `msg.value` deve essere identico al deposito di A.
    - *Event:* `MatchJoined(bytes32 indexed matchId, address indexed playerB)`.
- `function submitResult(bytes32 _matchId, uint8 _result)`
    - *Params:* `_result` enum: 1 (Win), 2 (Loss), 3 (Draw).
    - *Logic:* Implementa il "commit-reveal" semplificato. Se è il primo voto, salva. Se è il secondo, confronta.
    - *Event:* `ResultSubmitted(bytes32 matchId, address player, uint8 result)`.

Funzioni Admin/Arbitro (Write)

- `function resolveDispute(bytes32 _id, address _winner)`
    - *Access Control:* `onlyArbiter` (o DAO in futuro).
    - *Logic:* Esegue lo slashing forzato del bond perdente e trasferisce la fee all'arbitro.
    - *Event:* `DisputeResolved(bytes32 matchId, address winner, address arbiter)`.

Funzioni View (Read)

- `function getMatchDetails(bytes32 _id) external view returns (MatchStruct memory)`
    - Restituisce stato, indirizzi, wager, bond e voti attuali.

6. Piano di Sviluppo Hackathon (Roadmap 48h)

Una guida passo-passo per il team di sviluppo.

Fase 1: Core & Foundation (Ore 0-12)

- **Smart Contract:** Setup repository Hardhat. Scrittura delle struct dati e delle funzioni `createMatch` e `joinMatch`. Unit test base per il deposito fondi.
- **Frontend:** Scaffold Next.js. Configurazione RainbowKit per Avalanche Fuji Testnet. Creazione componenti UI base (Navbar, Connect Button).

Fase 2: Game Loop Logic (Ore 12-24)

- **Smart Contract:** Implementazione logica `submitResult` e calcolo automatico payout (Happy Path). Implementazione `reentrancyGuard`.
- **Frontend:** Integrazione funzioni di scrittura contratto. Creazione pagina "Lobby" che ascolta eventi on-chain per mostrare partite aperte.
- **Backend:** Setup WebSocket server minimale per chat istantanea nella lobby del match.

Fase 3: Dispute & Edge Cases (Ore 24-36)

- **Smart Contract:** Implementazione logica `DISPUTE`, Slashing del Bond e funzione `resolveDispute`.
- **Frontend:** UI per caricamento "prove" (mockup o integrazione base IPFS). Dashboard "Admin" per l'Arbitro (pagina segreta per la demo).
- **Testing:** Test di integrazione su Testnet. Simulazione scenari di disaccordo.

Fase 4: Polish & Demo Prep (Ore 36-48)

- **UX/UI:** Aggiunta animazioni (Confetti per vittoria), stati di loading, gestione errori RPC.
- **Script Demo:** Preparazione dei due wallet (Alice e Bob) con AVAX di test.
- **Pitch:** Creazione slide che evidenziano il modello economico (Bonding) come differenziatore chiave.

7. Analisi dei Rischi e Mitigazioni

1. **Attacco Sybil / Collusione:**
   - *Rischio:* Un utente gioca contro se stesso (due wallet) per generare volume falso o farmare token futuri.
   - *Mitigazione:* Le fee di protocollo rendono questa attività a somma negativa (si perde denaro ogni volta).
2. **Arbitro Centralizzato (Hackathon limit):**
   - *Rischio:* Durante l'hackathon, l'arbitro è il team stesso (centralizzato).