Realtime On-Chain Chess Engine with Ephemeral Rollups

by Send Arcade

September 22, 2025

Abstract

This paper details the implementation of a fully on-chain chess engine, porting the core logic of the open-source chess engine, Stockfish, to a decentralized network to enable a transparent and verifiable framework for chess computation. Our system represents the 64-square chessboard with a byte-per-square encoding, allowing efficient on-chain storage and rapid validation of game states. Each square's byte value includes both piece type and color, facilitating efficient access and manipulation during gameplay. This design ensures the engine operates within blockchain constraints.

The engine supports all legal chess moves, including special rules like castling, en passant, and pawn promotion, and detects game-ending conditions such as check, checkmate, stalemate, the 50-move rule, threefold repetition, and insufficient material. It also supports various time control systems with increments, enabling diverse tournament formats, and offers single-player and two-player matchmaking with turn management and stake-based gameplay using escrows.

All the Complex computations are delegated to an ephemeral rollup via Cross-Program Invocation (CPI) calls, managed by MagicBlock infrastructure, ensuring high-computational tasks do not impact the the engine's efficiency. The runtime's core validation pipeline guarantees move integrity, verifying bounds, turns, piece-specific rules, and self-check validation. After successful validation, the system updates the on-chain game state, managing piece positions, captures, castling rights, en passant squares, turn switching, and pawn promotions. This implementation provides a verifiable, trustless foundation for competitive chess, AI agent training, and secure tournament play.

Contents

1 Technical Architecture

1.1 System Architecture Overview

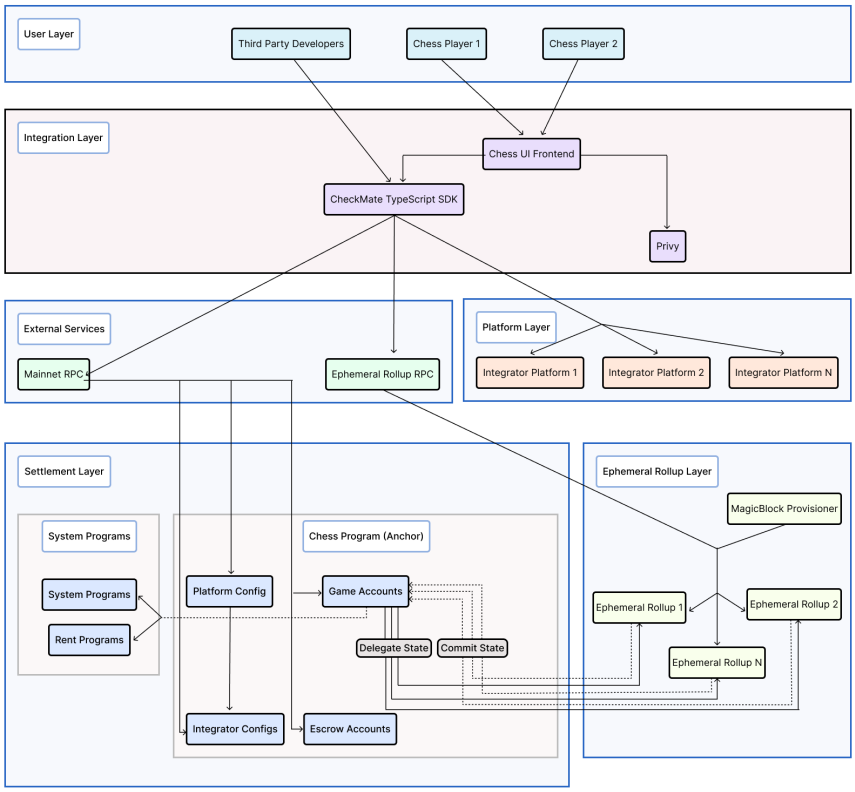

Figure 1: System Architecture Diagram

The architecture consists of five primary layers:

- **User Layer**: Chess players and third-party developers.
- **Integration Layer**: SDK, wallets, and frontend interfaces.
- **Platform Layer**: Multiple integrator platforms with custom configurations.

- **Ephemeral Rollup Layer**: High-speed execution environments for active games.
- **Settlement Layer**: Permanent state storage and payout settlement.

1.2 Core Program Structure

Our chess program implements a comprehensive on-chain gaming system built on Solana using the Anchor framework, with strategic integration points for MagicBlock's Ephemeral Rollup technology.

1.2.1 Game State Management

The `GameAccount` structure serves as the central state container, encompassing all aspects of a chess game:

Board Representation:

```
pub board_state: [u8; 64], // 8x8 board encoding
```

Listing 1: Board State Representation

Each square is encoded as a single byte containing both piece type and color information, enabling efficient storage and rapid access during move validation.

Player Management:

```
pub player_white: Pubkey,
pub player_black: Option<Pubkey>,
pub current_turn: PlayerColor,
```

Listing 2: Player Management

Supports both single-player game creation and two-player matchmaking with clear turn management.

Chess-Specific State:

```
pub castling_rights: CastlingRights,
pub en_passant_square: Option<u8>,
pub king_white_square: u8,
pub king_black_square: u8,
pub last_move: Option<LastMove>,
```

Listing 3: Chess-Specific State

Complete implementation of chess rules including special moves, position tracking, and move history.

Financial Integration:

```
pub token_mint: Pubkey,
pub token_vault: Pubkey,
pub entry_fee: u64,
pub total_prize_pool: u64,
```

Listing 4: Financial Integration

Native support for stake-based gameplay with secure escrow mechanisms.

Time Control System:

```
pub time_used_white: u32,
pub time_used_black: u32,
pub total_time_limit_white: Option<u32>,
pub total_time_limit_black: Option<u32>,
```

Listing 5: Time Control System

Flexible time control implementation supporting various tournament formats.

1.2.2 Instruction Set Architecture

The program exposes a comprehensive set of instructions covering the complete chess game lifecycle:

Game Lifecycle Management:

- `create_game`: Initialize new games with configurable parameters
- `join_game`: Enable second player to join open games
- `cancel_game`: Allow creators to cancel games before they start
- `forfeit_game`: Enable players to concede during active games

Move Execution:

- `make_move`: Core instruction handling move validation and state updates
- Comprehensive chess rule validation including check detection
- Special move support (castling, en passant, promotion)

Draw Mechanics:

- `offer_draw`: Propose draw to opponent
- `accept_draw`: Accept pending draw offers
- `reject_draw`: Decline draw proposals

Financial Operations:

- `claim_winnings`: Distribute prize pools to winners
- Automated fee calculation and distribution

1.3 Multi-Tenant Platform Design

The architecture supports multiple gaming platforms through a sophisticated integrator system:

1.3.1 Platform Configuration

```
pub struct PlatformConfig {
    pub fee_basis_points: u16, // 500 = 5%
    pub authority: Pubkey,
    pub fee_vault: Pubkey,
    pub bump: u8,
}
```

Listing 6: Platform Configuration Structure

1.3.2 Integrator System

```
pub struct IntegratorConfig {
    pub integrator_id: Pubkey,
    pub integrator: Pubkey,
    pub fee_basis_points: u16,
    pub fee_vault: Pubkey,
    pub total_games_created: u64,
    pub active_games_count: u32,
    pub total_volume: u64,
    pub next_game_id: u64,
}
```

Listing 7: Integrator Configuration Structure

This design enables:

- **White-label Integration**: Third-party platforms can integrate chess functionality with their own branding
- **Custom Fee Structures**: Each integrator can set their own fee rates
- **Analytics and Tracking**: Comprehensive statistics for business intelligence
- **Revenue Sharing**: Automated distribution between platform and integrators

1.4 Financial Architecture

Game Vaults

The system implements a secure escrow mechanism that holds player funds during games:

```
pub total_deposited: u64,
pub white_deposit: u64,
pub black_deposit: u64,
pub is_locked: bool,
```

Listing 8: Escrow Account Structure

Security Features:

- Funds are locked in program-controlled accounts.
- Automatic distribution based on game outcomes.
- Protection against any manipulation.
- Support for partial refunds in case of disputes.

Each game creates an escrow account that securely holds the combined entry fees:

```
let total_entry_fees = player1_fee + player2_fee;
let platform_fee = total_entry_fees * platform_fee_bps / 10000;
let integrator_fee = total_entry_fees * integrator_fee_bps / 10000;
let prize_pool = total_entry_fees - platform_fee - integrator_fee;
```

Listing 9: Fee Calculation

Winner Distribution:

- Winner receives 100% of prize pool.
- Draw results in 50/50 split.
- Timeout/abandonment follows predefined rules.

1.5 Fee Structure

The platform implements a dual-fee system:

- **Platform Fees**: 2.5% of entry fees as the Base fees collected by the SendRC for providing the engine; used to buyback and burn $SEND.
- **Integrator Fees**: Optional fees that can be added by third-party platforms who integrate Checkmate SDK.

2 Rollup Contract Integration

The engine is powered by Send Rollup Contract(SendRC) — ****a middleware contract built on the Magicblock infrastructure—that supports all our games, including Lana Roads.

SendRC natively defaults $SEND to be used as the playable currency in all of our games. It also automatically sends the revenues generated to sendstrategicreserve.sol.

Note: While default supporting SEND, the platform architecture accommodates any SPL token.

2.1 Ephemeral Rollup Fundamentals

MagicBlock's Ephemeral Rollups offer temporary, high-speed execution environments that maintain full compatibility with the base layer. Unlike traditional Layer-2 solutions, Ephemeral Rollups:

- **Spin up on-demand** for specific gaming sessions.
- **Disappear automatically** when no longer needed, leaving no permanent state bloat.
- **Maintain full SVM compatibility**, allowing existing Solana programs to run without modification.
- **Achieve sub-50ms latency** through optimized consensus-free execution.

2.2 State Delegation Architecture

2.2.1 Account Delegation Process

```
#[cfg(not(feature = "local"))]
#[commit]
#[derive(Accounts)]
pub struct MakeMove<'info> {
    // Account definitions with delegation support
}
```

Listing 10: MakeMove with Delegation Support

Delegation Lifecycle:

1. **Initiation:** An active game's account is delegated to an Ephemeral Rollup to enable rapid gameplay.
2. **Execution:** Chess moves are validated with sub-50ms latency within the high-speed rollup environment.
3. **Commitment:** Final score sheets are securely committed back to the Solana with notation transcript.

4. **Undelegation:** Control of the game account returns to the base layer once the active match is complete.

2.2.2 Parallel Read Access

During delegation, the base layer maintains read access to delegated accounts, ensuring:

- **Composability**: Other programs can still read game state for integrations.
- **Transparency**: Game state remains visible to all participants.
- **Compatibility**: Existing tooling and interfaces continue to function.

3 Implementation Details

3.1 Conditional Compilation

The program uses feature flags to support both local development and Ephemeral Rollup deployment:

```
#[cfg(feature = "local")]
#[derive(Accounts)]
pub struct MakeMove<'info> {
    // Local development version
}
#[cfg(not(feature = "local"))]
#[commit]
#[derive(Accounts)]
pub struct MakeMove<'info> {
    // Ephemeral Rollup version with commit annotation
}
```

Listing 11: Conditional Compilation for MakeMove

3.2 Commit Annotations

The `#[commit]` macro automatically handles state commitment to the base layer:

- **Automatic Batching**: Multiple moves are batched for efficient settlement.
- **Cryptographic Proofs**: All state changes include verification proofs.
- **Rollback Protection**: Invalid piece transitions are automatically reverted.

3.3 Session Management

Ephemeral Rollup sessions are managed through simple delegation calls:

```
    // Delegation initiated through CPI calls
    // Session automatically managed by MagicBlock infrastructure
    // Undelegation triggered by game completion or timeout
```

Listing 12: Session Management

3.4 Performance Benefits

3.4.1 Latency Reduction

- **Base Layer**: 400ms average block time on Solana.
- **Ephemeral Rollup**: 10-50ms execution time for chess moves.
- **User Experience**: Near-instantaneous move confirmation and board updates.

3.4.2 Throughput Scaling

- **Horizontal Scaling**: Multiple rollups can operate simultaneously.
- **Elastic Provisioning**: Rollups spin up automatically based on demand.
- **No Congestion**: Game moves don't compete with other Solana transactions during active sessions.

3.4.3 Cost Efficiency

- **Reduced Base Layer Load**: Only final state commitments consume Solana block space.
- **Batched Settlements**: Multiple moves can be settled in single transactions.
- **Dynamic Pricing**: Costs scale with actual usage rather than peak capacity.

4 Chess Game Logic Implementation

4.1 Board Representation

Our chess implementation uses an efficient 64-element array to represent the game board, with each element encoded as a single byte containing both piece type and color information.

4.1.1 Encoding Scheme

```
    // Piece encoding: 0xTC where T = type, C = color
    // Color: 1 = White, 2 = Black
    // Type: 1 = Pawn, 2 = Knight, 3 = Bishop, 4 = Rook, 5 = Queen, 6 = King
    // Examples:
    // 0x11 = White Pawn
    // 0x26 = Black King
    // 0x00 = Empty Square
```

Listing 13: Piece Encoding Scheme

4.1.2 Board Initialization

```
fn initialize_chess_board() -> [u8; 64] {
    let mut board = [0u8; 64];
    // White pieces (ranks 1-2)
    board[0] = 0x14; // White Rook on a1
    board[1] = 0x12; // White Knight on b1
    // ... complete setup
    // Black pieces (ranks 7-8)
    board[56] = 0x24; // Black Rook on a8
```

```
    // ... complete setup
    board
}
```

Listing 14: Chess Board Initialization

4.1.3 Square Access Methods

```
pub fn get_piece_at(&self, square: u8) -> u8 {
    if square < 64 {
        self.board_state[square as usize]
    } else {
        0
    }
}
pub fn set_piece_at(&mut self, square: u8, piece: u8) {
    if square < 64 {
        self.board_state[square as usize] = piece;
    }
}
```

Listing 15: Square Access Methods

4.2 Move Validation System

The chess engine implements comprehensive rule validation covering all aspects of legal chess play:

4.2.1 Core Validation Pipeline

```
pub fn handler(
    ctx: Context<MakeMove>,
    from_square: u8,
    to_square: u8,
    promotion_piece: Option<PieceType>,
) -> Result<()> {
    // 1. Basic validation
    ctx.accounts.validate(from_square, to_square)?;
    // 2. Turn validation
    require!(
        game_account.is_player_turn(&player.key()),
        ChessError::NotPlayerTurn
    );
    // 3. Piece validation
    let piece_at_from = game_account.get_piece_at(from_square);
    require!(piece_at_from != 0, ChessError::NoPieceAtSquare);
    // 4. Chess rule validation
    require!(
        utils::is_valid_chess_move(
            &game_account.board_state,
            from_square,
            to_square,
            piece_type,
            &game_account.castling_rights,
            game_account.en_passant_square,
            promotion_piece.clone()
```

```
        )?,
        ChessError::InvalidMove
    );
    // 5. Check validation
    require!(
        !utils::move_leaves_king_in_check(
            &game_account.board_state,
            from_square,
            to_square,
            &player_color,
            &game_account.castling_rights
        )?,
        ChessError::MoveExposesKing
    );
    // 6. Apply move
    utils::apply_move_to_board(/* ... */)?
}
```

Listing 16: Move Validation Pipeline

4.2.2 Special Move Handling

Castling Implementation:

- Validates king and rook positions.
- Checks for intervening pieces.
- Ensures neither piece has moved previously.
- Verifies king is not in check.

En Passant Capture:

- Tracks pawn double-moves for en passant opportunities.
- Validates capture conditions.
- Removes captured pawn from board.

Pawn Promotion:

- Detects pawns reaching the opposite end.
- Validates promotion piece selection.
- Updates board state with promoted piece.

4.3 Game State Transitions

The system manages complex game state transitions through a well-defined state machine:

4.3.1 Game Status Flow

```
pub enum GameStatus {
    WaitingForPlayer, // Initial state, awaiting second player
    InProgress, // Active game with both players
    Finished, // Game completed with result
    Cancelled, // Game cancelled before completion
    Disputed, // Under dispute resolution
    TimedOut, // Exceeded time limits
}
```

Listing 17: Game Status Enumeration

4.3.2 Turn Management

```
pub fn switch_turn(&mut self) {
    self.current_turn = match self.current_turn {
        PlayerColor::White => PlayerColor::Black,
        PlayerColor::Black => PlayerColor::White,
    };
}
pub fn is_player_turn(&self, player: &Pubkey) -> bool {
    match self.current_turn {
        PlayerColor::White => self.player_white == *player,
        PlayerColor::Black => {
            if let Some(black_player) = self.player_black {
                black_player == *player
            } else {
                false
            }
        }
    }
}
```

Listing 18: Turn Management Methods

4.3.3 Time Control Implementation

```
let time_taken = if let Some(last_move_time) =
   game_account.last_move_at {
    (clock.unix_timestamp - last_move_time) as u32
} else {
    0
};
match player_color {
    PlayerColor::White => {
        game_account.time_used_white += time_taken;
        if let Some(increment) = game_account.get_time_increment() {
            game_account.time_used_white =
                game_account.time_used_white.saturating_sub(increment);
        }
    }
    PlayerColor::Black => {
        // Similar logic for black player
    }
}
```

Listing 19: Time Control Logic

4.3.4 Result Determination

```
pub enum GameResult {
    None,
    Checkmate,
    Stalemate,
    Resignation,
    TimeOut,
    FiftyMoveRule,
    InsufficientMaterial,
    ThreefoldRepetition,
    DrawByAgreement,
    Abandoned,
}
```

Listing 20: Game Result Enumeration

The system automatically detects game-ending conditions:

- **Checkmate Detection**: Validates that the king is in check with no legal moves.
- **Stalemate Recognition**: Identifies positions where no legal moves exist but king is not in check.
- **Draw Conditions**: Implements fifty-move rule, insufficient material, and repetition detection.
- **Time Forfeit**: Automatically ends games when time limits are exceeded.

5 SDK and Integration Layer

5.1 CheckMate SDK Architecture

The CheckMate SDK provides a comprehensive TypeScript interface for interacting with the on-chain chess program. Built on top of Solana's @solana/kit framework, it abstracts complex blockchain interactions into intuitive game operations while maintaining full compatibility with Ephemeral Rollups.

```
import { ChessGameSDK } from "@sendarcade/checkmate";
import { createSolanaRpc } from "@solana/kit";
const rpc = createSolanaRpc("https://api.devnet.solana.com");
const erRpc = createSolanaRpc("https://devnet.magicblock.app");
const chessMateSDK = new ChessGameSDK();
```

Listing 21: SDK Initialization

5.2 Installation and Setup

Installation:

```
npm install @sendarcade/checkmate @solana/kit @solana-program/token
```

Listing 22: SDK Installation

Wallet Integration:
The SDK works with any wallet implementing the @solana/kit `TransactionSigner` interface:

```
import { createNoopSigner, address } from "@solana/kit";
const signer = createNoopSigner(address("YourPublicKeyHere"));
```

Listing 23: Wallet Integration

5.3 Core SDK Methods

Platform Setup:

```
const params = {
  integrator: signer,
  integratorId: address("YourIntegratorId"),
  feeBasisPoints: 500,
  feeVault: address("YourFeeVaultAddress")
};
const { instruction, integratorConfigPDA } =
  await chessMateSDK.initializeIntegratorIx(params);
```

Listing 24: Platform Setup

Game Management:

```
const createParams = {
  rpc,
  creator: signer,
  integratorId: address("YourIntegratorId"),
  tokenMint: address("EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v"),
  entryFee: BigInt(1_000_000),
  timeControl: {
    initialTime: 600,
    increment: 5,
    moveTimeLimit: null
  },
  ratedGame: true,
  allowDrawOffers: true
};
const { instruction, gameId, gameAccountAddress } =
  await chessMateSDK.createGameIx(createParams);
const joinParams = {
  player: signer,
  gameId: BigInt(1),
  integratorId: address("YourIntegratorId"),
  tokenMint: address("EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v")
};
const { instruction } = await chessMateSDK.joinGameIx(joinParams);
```

Listing 25: Game Management

Gameplay Operations:

```
const moveParams = {
  player: signer,
  gameId: BigInt(1),
  integratorId: address("YourIntegratorId"),
  fromSquare: "e2",
  toSquare: "e4",
  promotionPiece: undefined
```

```
};
const { instruction } = await chessMateSDK.makeMoveIx(moveParams);
const signature = await sendTransaction(erRpc, [instruction], signer);
const offerDrawParams = {
  player: signer,
  gameId: BigInt(1),
  integratorId: address("YourIntegratorId")
};
const { instruction } = await
    chessMateSDK.offerDrawIx(offerDrawParams);
const { instruction } = await
    chessMateSDK.acceptDrawIx(offerDrawParams);
const { instruction } = await
    chessMateSDK.rejectDrawIx(offerDrawParams);
const claimParams = {
  rpc,
  player: signer,
  gameId: BigInt(1),
  integratorId: address("YourIntegratorId"),
  tokenMint: address("EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v")
};
const { instruction } = await
    chessMateSDK.claimWinningsIx(claimParams);
```

Listing 26: Gameplay Operations

5.4 State Management and RPC Switching

A critical aspect of the CheckMate SDK is its intelligent RPC management for Ephemeral Rollup integration:

```
import { getGameAccountPDA, fetchGameAccount, GameStatus } from
    "@sendarcade/checkmate";
const [gameAccountPDA] = await getGameAccountPDA(
  address("YourIntegratorId"),
  BigInt(1)
);
const maybeGameAccount = await fetchGameAccount(rpc, gameAccountPDA);
if (maybeGameAccount.exists &&
     maybeGameAccount.data.gameStatus === GameStatus.InProgress) {
  const gameAccount = await fetchGameAccount(erRpc, gameAccountPDA);
} else {
  const gameAccount = await fetchGameAccount(rpc, gameAccountPDA);
}
```

Listing 27: RPC Switching

5.5 Chess Utilities and Game Logic

The SDK includes comprehensive chess utilities:

```
import { ChessUtils } from "@sendarcade/checkmate";
const utils = new ChessUtils();
const index = utils.squareToIndex("e4");
const notation = utils.indexToSquare(28);
const piece = utils.getPieceAt(gameData.board, index);
console.log(`Piece at e4: ${piece?.type} (${piece?.color})`);
utils.displayBoard(gameData.board, "Current Position");
```

```
const legalMoves = utils.getLegalMoves(gameData.board,
    gameData.currentTurn);
const e2Index = utils.squareToIndex("e2");
const legalMovesFromE2 = legalMoves.filter(move => move.from ===
    e2Index);
console.log("Legal moves from e2:",
  legalMovesFromE2.map(m => utils.indexToSquare(m.to)));
```

Listing 28: Chess Utilities

5.6 Error Handling and Validation

The SDK implements comprehensive error handling with specific error types:

```
import { ChessGameError, GameNotFoundError } from
    "@sendarcade/checkmate";
try {
  const { instruction } = await chessMateSDK.makeMoveIx({
    player: signer,
    gameId: BigInt(1),
    integratorId: address("YourIntegratorId"),
    fromSquare: "e2",
    toSquare: "e5"
  });
} catch (error) {
  if (error instanceof GameNotFoundError) {
    console.error("Game not found:", error.message);
  } else if (error instanceof ChessGameError) {
    console.error("Chess game error:", error.message);
  } else {
    console.error("Unknown error:", error);
  }
}
```

Listing 29: Error Handling

Error Categories:

- `InvalidMoveError`: Chess rule violations.
- `GameNotFoundError`: Invalid game references.
- `UnauthorizedPlayerError`: Permission violations.
- `InsufficientFundsError`: Wallet balance issues.
- `NetworkError`: RPC connectivity issues.

5.7 Integration Patterns

Event-Driven Architecture:

The SDK supports real-time game state monitoring through polling patterns:

```
const pollGameState = async (gameId: bigint) => {
  const [gameAccountPDA] = await getGameAccountPDA(integratorId,
    gameId);
  setInterval(async () => {
    try {
```

```
      const gameAccount = await fetchGameAccount(erRpc,
    gameAccountPDA);
      if (gameAccount.exists) {
        updateUI(gameAccount.data);
        if (gameAccount.data.gameStatus === GameStatus.Finished) {
          handleGameEnd(gameAccount.data);
        }
      }
    } catch (error) {
      console.error("Failed to fetch game state:", error);
    }
  }, 1000);
};
```

Listing 30: Event-Driven Monitoring

Multi-Platform Integration:

```
const platformConfigs = {
  chess_com: {
    integratorId: address("ChessComIntegratorId"),
    feeBasisPoints: 300,
    branding: { theme: "classic", logo: "chess_com_logo.png" }
  },
  lichess: {
    integratorId: address("LichessIntegratorId"),
    feeBasisPoints: 250,
    branding: { theme: "modern", logo: "lichess_logo.png" }
  }
};
const config = platformConfigs[platform];
const gameParams = {
  ...baseParams,
  integratorId: config.integratorId
};
```

Listing 31: Multi-Platform Configuration

5.8 Performance Optimizations

The SDK includes several performance optimizations:

Instruction Batching:

```
const instructions = [
  await chessMateSDK.makeMoveIx(moveParams1),
  await chessMateSDK.makeMoveIx(moveParams2)
];
const signature = await sendTransaction(erRpc, instructions, signer);
```

Listing 32: Instruction Batching

Caching and State Management:

- Local game state caching to reduce RPC calls.
- Optimistic UI updates with rollback on failure.
- Efficient board state serialization.
- Minimal account data fetching.

6 Throughput Metrics

Ephemeral Rollup integration delivers significant performance improvements:

Transaction Speed:

- Move confirmation: 50ms.
- Real-time game state updates.
- Batch settlement: Every 2-4 seconds.

Cost Efficiency:

- Per-move cost: $0.00001
- Game creation: $0.001.
- Settlement: Standard Solana fees.

Design Patterns used:

- **Result types** for error handling with custom ChessError variants
- **require! macros** for early returns on validation failures
- **Option<PieceType>** for optional pawn promotion
- **References (&)** to avoid unnecessary copying of game state

For technical implementation details, API documentation, and integration guides, please refer to the accompanying technical documentation and SDK reference materials.