Cash Markets: Centralized Backend Architecture and Implementation Guide

Table of Contents

1. System Architecture Overview

The architecture adopts a centralized design where the backend server maintains complete authority over game logic, user accounting, and state management. The Aptos blockchain serves solely as a settlement layer for monetary transactions and a source of verifiable randomness. This design prioritizes performance and development velocity while maintaining the core value proposition of provably fair gameplay through on-chain random seeds.

```
graph TB
    subgraph "Client Layer"
        C1[React Client 1]
        C2[React Client 2]
        CN[React Client N]
    end

    subgraph "Backend Server"
        WS[WebSocket Manager]
        GE[Game Engine]
        AM[Account Manager]
        TM[Trade Manager]
        PE[Price Engine]
        SM[Session Manager]
        API[REST API]
    end

    subgraph "Data Layer"
        PG[(PostgreSQL
Primary Storage)]
        RD[(Redis
Cache & Sessions)]
```

```
        TS[(TimescaleDB
Time Series)]
    end

    subgraph "Blockchain Layer"
        AC[Aptos Client]
        SC[Smart Contract]
        RNG[Randomness Module]
    end

    C1 <-->|WebSocket| WS
    C2 <-->|WebSocket| WS
    CN <-->|WebSocket| WS

    WS --> GE
    GE --> AM
    GE --> TM
    GE --> PE
    WS --> SM

    AM --> PG
    TM --> PG
    PE --> TS
    SM --> RD

    GE --> AC
    AC --> SC
    SC --> RNG

    API --> AM
    API --> TM
```

The server operates as the authoritative source for all game state, maintaining a complete audit trail of every action while providing sub-millisecond response times for trading operations. This centralized approach enables sophisticated features like position aggregation, real-time P&L calculations, and complex order types that would be prohibitively expensive to implement on-chain.

2. Backend Technology Stack and Rationale

2.1 Core Technology Selection

The backend utilizes TypeScript with Node.js as the primary runtime environment. This choice balances several critical factors:

TypeScript provides type safety crucial for financial calculations while maintaining the development velocity advantages of JavaScript. The extensive ecosystem of NPM packages accelerates development, particularly for WebSocket handling, database connectivity, and Aptos SDK integration. Modern Node.js versions with V8 optimizations deliver performance sufficient for our requirements when properly architected.

```
graph LR
    subgraph "Application Layer"
        TS[TypeScript 5.x]
        NODE[Node.js 20.x]
        WS[ws Library]
        EX[Express.js]
    end

    subgraph "Data Access Layer"
        PG[node-postgres]
        RD[ioredis]
        TS_CLIENT[TimescaleDB Client]
        TYPEORM[TypeORM]
    end

    subgraph "Blockchain Layer"
        APTOS[Aptos TS SDK]
        ETHERS[Ethers.js]
    end

    subgraph "Utility Layer"
        ZOD[Zod Validation]
        WINSTON[Winston Logging]
        BULL[Bull Queue]
        PM2[PM2 Process Manager]
    end
```

2.2 Database Technology Stack

The system employs a polyglot persistence strategy optimized for different data access patterns:

PostgreSQL 15 serves as the primary transactional database, storing user accounts, trade history, and game metadata. Its ACID compliance ensures financial data integrity while advanced features like partial indexes and materialized views optimize complex queries.

Redis 7 provides high-speed caching and session management. All active game state resides in Redis for microsecond access times, with write-through to PostgreSQL for durability. Redis Streams power the event sourcing system for game state changes.

TimescaleDB (PostgreSQL extension) handles time-series data for candles and price history. Its compression algorithms reduce storage by 90% while maintaining query performance for historical analysis.

2.3 Infrastructure Components

```
// Server architecture configuration
interface ServerConfig {
  // Core settings
  port: number;
  wsPort: number;
```

```
  // Database connections
  postgres: {
    host: string;
    port: number;
    database: string;
    username: string;
    password: string;
    maxConnections: number;
    statementTimeout: number;
  };

  redis: {
    host: string;
    port: number;
    password: string;
    db: number;
    keyPrefix: string;
  };

  timescale: {
    // Inherits from postgres
    chunkTimeInterval: string; // '1 hour'
    compressionAfter: string;   // '1 day'
  };

  // Performance tuning
  performance: {
    maxConcurrentGames: number;
    candleBufferSize: number;
    wsHeartbeatInterval: number;
    dbConnectionPoolSize: number;
  };
}
```

3. Database Design and Schema

3.1 Core Schema Design

The database schema optimizes for both transactional consistency and query performance:

```
erDiagram
    USERS ||--o{ ACCOUNTS : has
    USERS ||--o{ SESSIONS : creates
    ACCOUNTS ||--o{ TRADES : places
    ACCOUNTS ||--o{ TRANSACTIONS : performs
    GAMES ||--o{ ROUNDS : contains
    ROUNDS ||--o{ TRADES : includes
    ROUNDS ||--o{ CANDLES : generates
    TRADES ||--o{ TRADE_EVENTS : tracks

    USERS {
```

```
        uuid id PK
        string wallet_address UK
        string username UK
        jsonb metadata
        timestamp created_at
        timestamp last_login
        boolean is_active
        int total_rounds_played
        decimal total_volume_traded
    }

    ACCOUNTS {
        uuid id PK
        uuid user_id FK
        decimal balance
        decimal locked_balance
        decimal total_deposited
        decimal total_withdrawn
        decimal total_pnl
        timestamp last_transaction
        int withdrawal_lock_until
    }

    TRADES {
        uuid id PK
        uuid account_id FK
        uuid round_id FK
        enum direction
        decimal entry_price
        decimal exit_price
        decimal size
        decimal pnl
        timestamp opened_at
        timestamp closed_at
        int candle_index_open
        int candle_index_close
        jsonb metadata
    }

    ROUNDS {
        uuid id PK
        uuid game_id FK
        int round_number
        string blockchain_seed
        timestamp started_at
        timestamp ended_at
        enum status
        int total_trades
        decimal total_volume
    }
```

3.2 Performance Optimization Tables

The system maintains denormalized tables for frequently accessed data:

```sql
-- Real-time player statistics (updated via triggers)
CREATE TABLE player_stats_realtime (
    user_id UUID PRIMARY KEY REFERENCES users(id),
    current_balance DECIMAL(20,8),
    active_trades_count INTEGER DEFAULT 0,
    today_pnl DECIMAL(20,8) DEFAULT 0,
    today_trades_count INTEGER DEFAULT 0,
    win_rate_7d DECIMAL(5,4),
    avg_trade_size DECIMAL(20,8),
    last_updated TIMESTAMP DEFAULT NOW()
);

-- Aggregated round statistics for leaderboards
CREATE MATERIALIZED VIEW round_statistics AS
SELECT
    r.id as round_id,
    COUNT(DISTINCT t.account_id) as unique_traders,
    COUNT(t.id) as total_trades,
    SUM(t.size) as total_volume,
    AVG(t.pnl) as average_pnl,
    MAX(t.pnl) as best_trade,
    MIN(t.pnl) as worst_trade
FROM rounds r
LEFT JOIN trades t ON r.id = t.round_id
GROUP BY r.id;

-- High-frequency trade tracking table (partitioned by day)
CREATE TABLE trade_events (
    id BIGSERIAL,
    trade_id UUID REFERENCES trades(id),
    event_type VARCHAR(20),
    price DECIMAL(20,8),
    timestamp TIMESTAMP,
    metadata JSONB
) PARTITION BY RANGE (timestamp);
```

3.3 Redis Data Structures

Redis maintains hot data for active gameplay:

```typescript
// Redis key patterns and data structures
interface RedisSchema {
  // User session data
  session: {
    key: `session:${userId}`;
    data: {
      userId: string;
      accountId: string;
```

```
      wsConnectionId: string;
      lastActivity: number;
      currentRoundId?: string;
    };
  };

  // Active round state
  round: {
    key: `round:${roundId}`;
    data: {
      id: string;
      seed: string;
      startTime: number;
      candlesGenerated: number;
      activeTrades: Trade[];
      status: 'active' | 'settling' | 'completed';
    };
  };

  // Real-time candle buffer
  candles: {
    key: `candles:${roundId}`;
    type: 'stream'; // Redis Stream
    data: {
      price: number;
      timestamp: number;
      volume: number;
    };
  };

  // Trade execution queue
  tradeQueue: {
    key: `queue:trades:${accountId}`;
    type: 'list';
    data: TradeOrder;
  };
}
```

4. Server Architecture Components

4.1 Core Service Architecture

The server implements a service-oriented architecture with clear separation of concerns:

```
graph TB
    subgraph "API Gateway Layer"
        REST[REST API Gateway]
        WSG[WebSocket Gateway]
        AUTH[Auth Middleware]
    end
```

```
    subgraph "Business Logic Layer"
        US[User Service]
        AS[Account Service]
        TS[Trade Service]
        GS[Game Service]
        PS[Price Service]
    end

    subgraph "Infrastructure Layer"
        CACHE[Cache Manager]
        QUEUE[Queue Manager]
        EVENTS[Event Bus]
        MONITOR[Health Monitor]
    end

    subgraph "Data Access Layer"
        DAO[Data Access Objects]
        REPO[Repositories]
        MAPPER[Object Mappers]
    end

    REST --> AUTH
    WSG --> AUTH
    AUTH --> US
    AUTH --> AS

    US --> CACHE
    AS --> CACHE
    TS --> QUEUE
    GS --> EVENTS
    PS --> EVENTS

    CACHE --> DAO
    QUEUE --> DAO
    EVENTS --> DAO
```

4.2 Service Implementation Details

Each service encapsulates specific business logic and maintains its own internal state:

```
// Account Service manages all financial operations
class AccountService {
  private db: PostgresClient;
  private redis: RedisClient;
  private eventBus: EventBus;

  async getBalance(userId: string): Promise<Balance> {
    // Try cache first
    const cached = await this.redis.get(`balance:${userId}`);
    if (cached) return cached;
```

```
    // Fetch from database
    const balance = await this.db.query(
      'SELECT balance, locked_balance FROM accounts WHERE user_id = $1',
      [userId]
    );

    // Update cache with TTL
    await this.redis.setex(`balance:${userId}`, 300, balance);
    return balance;
  }

  async lockFundsForTrade(userId: string, amount: Decimal): Promise<void>
{
    // Use PostgreSQL advisory lock for consistency
    await this.db.query('SELECT pg_advisory_lock($1)',
[userId.hashCode()]);

    try {
      // Check available balance
      const account = await this.db.query(
        'SELECT balance, locked_balance FROM accounts WHERE user_id = $1
FOR UPDATE',
        [userId]
      );

      if (account.balance.minus(account.locked_balance).lt(amount)) {
        throw new InsufficientBalanceError();
      }

      // Update locked balance
      await this.db.query(
        'UPDATE accounts SET locked_balance = locked_balance + $1 WHERE
user_id = $2',
        [amount, userId]
      );

      // Emit event for real-time updates
      this.eventBus.emit('account.balance.locked', { userId, amount });

    } finally {
      await this.db.query('SELECT pg_advisory_unlock($1)',
[userId.hashCode()]);
    }
  }
}
```

4.3 Game Engine Architecture

The game engine orchestrates all game-related operations:

```
class GameEngine {
  private priceEngine: PriceEngine;
  private tradeManager: TradeManager;
  private roundManager: RoundManager;
  private wsManager: WebSocketManager;

  async processRound(roundId: string): Promise<void> {
    const round = await this.roundManager.getRound(roundId);
    const seed = await this.fetchBlockchainSeed(round.seedBlockHeight);

    // Initialize price generation
    this.priceEngine.initialize(seed);

    // Generate and stream candles
    for (let i = 0; i < CANDLES_PER_ROUND; i++) {
      const candle = this.priceEngine.generateCandle(i);

      // Store in time series database
      await this.storeCandle(roundId, candle);

      // Update all active trades
      const trades = await this.tradeManager.getActiveTrades(roundId);
      for (const trade of trades) {
        await this.updateTradePnL(trade, candle.close);
      }

      // Broadcast to connected clients
      await this.wsManager.broadcastToRound(roundId, {
        type: 'candle',
        data: candle
      });

      // Precise timing
      await this.preciseSleep(CANDLE_INTERVAL_MS);
    }
  }

  private async preciseSleep(ms: number): Promise<void> {
    const start = process.hrtime.bigint();
    const target = start + BigInt(ms * 1_000_000);

    while (process.hrtime.bigint() < target) {
      // Busy wait for last few microseconds
      if (target - process.hrtime.bigint() > 1_000_000n) {
        await new Promise(resolve => setImmediate(resolve));
      }
    }
  }
}
```

5. WebSocket Protocol and Real-time Communication

5.1 WebSocket Message Protocol

The WebSocket protocol uses a binary format for efficiency:

```
sequenceDiagram
    participant Client
    participant WSGateway
    participant GameEngine
    participant Redis

    Client->>WSGateway: Connect + Auth Token
    WSGateway->>Redis: Validate Session
    Redis-->>WSGateway: Session Valid
    WSGateway-->>Client: Connected + Session ID

    Client->>WSGateway: Subscribe to Round
    WSGateway->>GameEngine: Add Subscriber

    loop Every 65ms
        GameEngine->>Redis: Get Next Candle
        Redis-->>GameEngine: Candle Data
        GameEngine->>WSGateway: Broadcast Candle
        WSGateway-->>Client: Binary Candle Message
    end

    Client->>WSGateway: Open Position
    WSGateway->>GameEngine: Process Trade
    GameEngine-->>WSGateway: Trade Confirmation
    WSGateway-->>Client: Position Update
```

5.2 Message Format Specification

Messages use a compact binary protocol with MessagePack encoding:

```
// Message type definitions
enum MessageType {
  // Client -> Server
  AUTH = 0x01,
  SUBSCRIBE_ROUND = 0x02,
  UNSUBSCRIBE_ROUND = 0x03,
  OPEN_POSITION = 0x04,
  CLOSE_POSITION = 0x05,
  GET_BALANCE = 0x06,

  // Server -> Client
  AUTH_SUCCESS = 0x81,
  AUTH_FAILURE = 0x82,
  CANDLE_DATA = 0x83,
  POSITION_UPDATE = 0x84,
  BALANCE_UPDATE = 0x85,
```

```
  ROUND_END = 0x86,
  ERROR = 0xFF
}

// Binary message structure
interface BinaryMessage {
  version: uint8;      // Protocol version
  type: MessageType;   // Message type
  timestamp: uint64;   // Server timestamp
  sequence: uint32;    // Message sequence number
  payload: Buffer;     // MessagePack encoded payload
}

// Candle data payload (most frequent message)
interface CandlePayload {
  roundId: string;
  index: uint16;
  open: float64;
  high: float64;
  low: float64;
  close: float64;
  volume: float64;
  timestamp: uint64;
}
```

5.3 Connection Management

The WebSocket manager handles thousands of concurrent connections efficiently:

```
class WebSocketManager {
  private connections: Map<string, WebSocketConnection>;
  private roundSubscriptions: Map<string, Set<string>>;
  private messageQueue: PriorityQueue<OutboundMessage>;

  constructor() {
    // Initialize worker threads for message serialization
    this.initializeWorkerPool();

    // Start message pump
    this.startMessagePump();
  }

  private async startMessagePump(): Promise<void> {
    while (true) {
      const batch = await this.messageQueue.dequeueBatch(100);

      // Group messages by connection
      const grouped = this.groupMessagesByConnection(batch);

      // Send in parallel
      await Promise.all(
```

```
        Array.from(grouped.entries()).map(([connId, messages]) =>
          this.sendBatch(connId, messages)
        )
      );

      // Prevent CPU spinning
      if (batch.length === 0) {
        await new Promise(resolve => setTimeout(resolve, 1));
      }
    }
  }

  async broadcastCandle(roundId: string, candle: Candle): Promise<void> {
    const subscribers = this.roundSubscriptions.get(roundId) || new Set();

    // Serialize once
    const serialized = this.serializeCandle(candle);

    // Queue for all subscribers
    for (const connectionId of subscribers) {
      this.messageQueue.enqueue({
        connectionId,
        priority: MessagePriority.HIGH,
        data: serialized
      });
    }
  }
}
```

6. State Management and Synchronization

6.1 State Architecture

The system maintains state across multiple layers with careful synchronization:

```
graph TB
    subgraph "Authoritative State"
        PG[(PostgreSQL
Source of Truth)]
    end

    subgraph "Cache Layer"
        REDIS[(Redis
Hot State)]
        LOCAL[Local Memory
Ultra-Hot State]
    end

    subgraph "Client State"
        C1[Client 1 State]
        C2[Client 2 State]
```

```
        CN[Client N State]
    end

    PG -->|Write-Through| REDIS
    REDIS -->|Load on Start| LOCAL
    LOCAL -->|Stream Updates| C1
    LOCAL -->|Stream Updates| C2
    LOCAL -->|Stream Updates| CN

    C1 -->|Commands| LOCAL
    C2 -->|Commands| LOCAL
    CN -->|Commands| LOCAL

    LOCAL -->|Batch Write| REDIS
    REDIS -->|Periodic Flush| PG
```

6.2 State Synchronization Strategy

The system employs eventual consistency with strict ordering guarantees:

```
class StateManager {
  private postgres: PostgresClient;
  private redis: RedisClient;
  private localCache: LRUCache<string, GameState>;
  private writeBuffer: WriteBuffer;

  async updateTradeState(trade: Trade): Promise<void> {
    // 1. Update local cache immediately
    const gameState = await this.getGameState(trade.roundId);
    gameState.trades.set(trade.id, trade);

    // 2. Queue Redis update
    await this.redis.hset(
      `round:${trade.roundId}:trades`,
      trade.id,
      JSON.stringify(trade)
    );

    // 3. Buffer PostgreSQL write
    this.writeBuffer.add({
      table: 'trades',
      operation: 'upsert',
      data: trade,
      priority: WritePriority.HIGH
    });

    // 4. Emit state change event
    this.eventBus.emit('trade.updated', {
      roundId: trade.roundId,
      tradeId: trade.id,
      accountId: trade.accountId,
```

```
        pnl: trade.currentPnl
      });
    }

    // Flush write buffer periodically or when full
    private async flushWriteBuffer(): Promise<void> {
      const batch = this.writeBuffer.drain();
      if (batch.length === 0) return;

      // Group by table for efficient bulk operations
      const grouped = this.groupByTable(batch);

      await this.postgres.transaction(async (trx) => {
        for (const [table, operations] of grouped) {
          await this.executeBulkOperations(trx, table, operations);
        }
      });
    }
  }
```

6.3 Consistency Guarantees

The system provides different consistency levels for different operations:

```
enum ConsistencyLevel {
  EVENTUAL = 'eventual',       // Default for non-critical updates
  READ_YOUR_WRITES = 'ryw',    // For user's own data
  STRONG = 'strong'            // For financial operations
}

class ConsistencyManager {
  async read<T>(
    key: string,
    consistency: ConsistencyLevel = ConsistencyLevel.EVENTUAL
  ): Promise<T> {
    switch (consistency) {
      case ConsistencyLevel.EVENTUAL:
        // Try cache first, fallback to database
        return await this.readEventual(key);

      case ConsistencyLevel.READ_YOUR_WRITES:
        // Check write buffer, then cache, then database
        return await this.readYourWrites(key);

      case ConsistencyLevel.STRONG:
        // Always read from primary database
        return await this.readStrong(key);
    }
  }
}
```

7. Aptos Integration Layer

7.1 Blockchain Integration Architecture

The Aptos integration layer handles all blockchain interactions:

```
graph LR
    subgraph "Game Server"
        BC[Blockchain Client]
        EM[Event Monitor]
        TM[Transaction Manager]
        SM[Seed Manager]
    end

    subgraph "Aptos Network"
        RPC[RPC Node]
        WS[WebSocket Events]
        SC[Smart Contract]
    end

    BC --> RPC
    EM --> WS
    TM --> RPC
    SM --> SC

    SC -->|Seeds| SM
    SC -->|Events| EM
```

7.2 Smart Contract Interface

The minimal smart contract handles only money and randomness:

```
// Smart contract interface (Move pseudocode)
interface CashMarketsContract {
  // Money management
  deposit(user: address, amount: u64): void;
  withdraw(user: address, amount: u64): void;
  getBalance(user: address): u64;

  // Randomness
  requestSeed(): {seed: u256, blockHeight: u64};
  verifySeed(blockHeight: u64, seed: u256): bool;

  // Settlement
  settleRound(
    roundId: u64,
    winnersAndAmounts: Array<{address, amount}>
  ): void;
}
```

7.3 Blockchain Client Implementation

The blockchain client manages all Aptos interactions:

```
class AptosClient {
  private client: Aptos;
  private contractAddress: string;
  private serverAccount: AptosAccount;

  async requestRandomSeed(): Promise<SeedData> {
    const payload = {
      function: `${this.contractAddress}::game::request_seed`,
      type_arguments: [],
      arguments: []
    };

    const txn = await this.client.generateTransaction(
      this.serverAccount.address(),
      payload
    );

    const signedTxn = await this.client.signTransaction(
      this.serverAccount,
      txn
    );

    const result = await this.client.submitTransaction(signedTxn);

    // Extract seed from events
    const seedEvent = result.events.find(e => e.type === 'SeedGenerated');
    return {
      seed: seedEvent.data.seed,
      blockHeight: seedEvent.data.block_height,
      timestamp: Date.now()
    };
  }

  async batchSettle(settlements: Settlement[]): Promise<void> {
    // Group settlements to minimize gas
    const batches = this.createOptimalBatches(settlements);

    for (const batch of batches) {
      await this.submitSettlement(batch);

      // Rate limiting to avoid congestion
      await this.rateLimiter.wait();
    }
  }
}
```

8. Performance Architecture

8.1 Performance Optimization Strategy

The system implements multiple layers of performance optimization:

```
graph TB
    subgraph "Request Path Optimization"
        REQ[Incoming Request]
        CF[CloudFlare Cache]
        LB[Load Balancer]
        CACHE[Application Cache]
        HOT[Hot Data Path]
        COLD[Cold Data Path]
    end

    REQ --> CF
    CF -->|Cache Miss| LB
    LB --> CACHE
    CACHE -->|Hit| HOT
    CACHE -->|Miss| COLD

    subgraph "Data Path Optimization"
        MEM[In-Memory Store]
        REDIS_OPT[Redis Pipeline]
        PG_POOL[PG Connection Pool]
        BATCH[Batch Processor]
    end

    HOT --> MEM
    COLD --> REDIS_OPT
    REDIS_OPT --> PG_POOL
    PG_POOL --> BATCH
```

8.2 Caching Strategy

The multi-layer caching system minimizes database load:

```
class CacheManager {
  private l1Cache: NodeCache;      // In-process memory cache
  private l2Cache: RedisClient;     // Distributed Redis cache
  private l3Cache: PostgresClient; // Database with query cache

  async get<T>(key: string, options: CacheOptions): Promise<T | null> {
    // L1: Check in-memory cache (microseconds)
    const l1Result = this.l1Cache.get<T>(key);
    if (l1Result) {
      this.metrics.increment('cache.l1.hit');
      return l1Result;
    }

    // L2: Check Redis cache (sub-millisecond)
```

```
    const l2Result = await this.l2Cache.get(key);
    if (l2Result) {
      this.metrics.increment('cache.l2.hit');
      // Promote to L1
      this.l1Cache.set(key, l2Result, options.l1Ttl);
      return JSON.parse(l2Result);
    }

    // L3: Load from database (milliseconds)
    const l3Result = await options.loader();
    if (l3Result) {
      this.metrics.increment('cache.l3.hit');
      // Populate all cache levels
      await this.populateCaches(key, l3Result, options);
      return l3Result;
    }

    return null;
  }

  private async populateCaches<T>(
    key: string,
    value: T,
    options: CacheOptions
  ): Promise<void> {
    // Update caches in parallel
    await Promise.all([
      this.l1Cache.set(key, value, options.l1Ttl),
      this.l2Cache.setex(key, options.l2Ttl, JSON.stringify(value))
    ]);
  }
}
```

8.3 Database Query Optimization

PostgreSQL queries are optimized for the access patterns:

```sql
-- Optimized indexes for common queries
CREATE INDEX CONCURRENTLY idx_trades_account_round
  ON trades(account_id, round_id, closed_at DESC);

CREATE INDEX CONCURRENTLY idx_trades_open_positions
  ON trades(account_id, closed_at)
  WHERE closed_at IS NULL;

CREATE INDEX CONCURRENTLY idx_accounts_balance_check
  ON accounts(user_id, balance, locked_balance);

-- Partial index for active rounds only
CREATE INDEX CONCURRENTLY idx_rounds_active
  ON rounds(game_id, started_at)
```

```
  WHERE status = 'active';

-- Query optimization example
EXPLAIN ANALYZE
SELECT
  t.id,
  t.direction,
  t.size,
  t.entry_price,
  c.close as current_price,
  (c.close - t.entry_price) * t.size as unrealized_pnl
FROM trades t
JOIN LATERAL (
  SELECT close
  FROM candles
  WHERE round_id = t.round_id
  ORDER BY timestamp DESC
  LIMIT 1
) c ON true
WHERE t.account_id = $1
  AND t.closed_at IS NULL;
```

8.4 Connection Pooling and Resource Management

The system maintains optimal resource utilization:

```
class ResourceManager {
  private pgPool: Pool;
  private redisPool: RedisPool;
  private workerPool: WorkerPool;

  constructor() {
    // PostgreSQL connection pooling
    this.pgPool = new Pool({
      max: 20,                    // Maximum connections
      min: 5,                     // Minimum connections
      idleTimeoutMillis: 30000,   // Close idle connections
      connectionTimeoutMillis: 2000,
      statement_timeout: 5000,
      query_timeout: 10000
    });

    // Redis connection pooling
    this.redisPool = new RedisPool({
      minIdle: 10,
      maxTotal: 50,
      maxWaitMillis: 100,
      enableAutoPipelining: true,
      enableOfflineQueue: false
    });
```

```
    // Worker thread pool for CPU-intensive tasks
    this.workerPool = new WorkerPool({
      size: os.cpus().length,
      taskTimeout: 1000,
      recycleAfter: 1000  // Recycle workers after N tasks
    });
  }

  async executeWithRetry<T>(
    operation: () => Promise<T>,
    options: RetryOptions
  ): Promise<T> {
    let lastError: Error;

    for (let attempt = 0; attempt < options.maxAttempts; attempt++) {
      try {
        return await operation();
      } catch (error) {
        lastError = error;

        // Exponential backoff with jitter
        const delay = Math.min(
          options.baseDelay * Math.pow(2, attempt) + Math.random() * 100,
          options.maxDelay
        );

        await new Promise(resolve => setTimeout(resolve, delay));
      }
    }

    throw lastError;
  }
}
```

9. Error Handling and Recovery

9.1 Error Classification and Handling

The system implements comprehensive error handling with appropriate recovery strategies:

```
graph TB
    subgraph "Error Types"
        E1[Network Errors]
        E2[Database Errors]
        E3[Business Logic Errors]
        E4[Blockchain Errors]
        E5[System Errors]
    end

    subgraph "Recovery Strategies"
        R1[Retry with Backoff]
```

```
        R2[Circuit Breaker]
        R3[Fallback Service]
        R4[Graceful Degradation]
        R5[Emergency Shutdown]
    end

    E1 --> R1
    E2 --> R2
    E3 --> R4
    E4 --> R3
    E5 --> R5
```

9.2 Error Recovery Implementation

Each error type has specific handling logic:

```
class ErrorHandler {
  private circuitBreakers: Map<string, CircuitBreaker>;
  private errorMetrics: MetricsCollector;

  async handleError(error: Error, context: ErrorContext): Promise<void> {
    // Classify error
    const errorType = this.classifyError(error);

    // Log with context
    logger.error({
      error: error.message,
      stack: error.stack,
      type: errorType,
      context,
      timestamp: new Date().toISOString()
    });

    // Update metrics
    this.errorMetrics.increment(`errors.${errorType}`);

    // Apply recovery strategy
    switch (errorType) {
      case ErrorType.DATABASE_CONNECTION:
        await this.handleDatabaseError(error, context);
        break;

      case ErrorType.BLOCKCHAIN_TIMEOUT:
        await this.handleBlockchainError(error, context);
        break;

      case ErrorType.INSUFFICIENT_BALANCE:
        await this.handleBusinessError(error, context);
        break;

      case ErrorType.SYSTEM_CRITICAL:
```

```
        await this.handleCriticalError(error, context);
        break;
      }
    }

  private async handleDatabaseError(
    error: Error,
    context: ErrorContext
  ): Promise<void> {
    const breaker = this.circuitBreakers.get('database');

    if (breaker.isOpen()) {
      // Use cache only mode
      await this.switchToCacheOnlyMode();
      return;
    }

    // Attempt reconnection
    try {
      await this.databaseReconnect();
      breaker.recordSuccess();
    } catch (reconnectError) {
      breaker.recordFailure();

      if (breaker.isOpen()) {
        // Alert operations team
        await this.alertOps('Database circuit breaker opened', {
          error,
          context,
          failures: breaker.getFailureCount()
        });
      }
    }
  }
}
```

9.3 Data Recovery Procedures

The system maintains data integrity through multiple recovery mechanisms:

```
class DataRecoveryManager {
  async recoverFromCrash(): Promise<void> {
    // 1. Restore Redis state from PostgreSQL
    await this.restoreRedisState();

    // 2. Replay incomplete transactions
    await this.replayIncompleteTransactions();

    // 3. Reconcile blockchain state
    await this.reconcileBlockchainState();
```

```
    // 4. Validate account balances
    await this.validateAllBalances();
  }

  private async replayIncompleteTransactions(): Promise<void> {
    // Find transactions without completion events
    const incomplete = await this.postgres.query(`
      SELECT t.*
      FROM trades t
      WHERE t.closed_at IS NULL
        AND t.opened_at < NOW() - INTERVAL '1 hour'
        AND NOT EXISTS (
          SELECT 1 FROM trade_events te
          WHERE te.trade_id = t.id
            AND te.event_type = 'CLOSED'
        )
    `);

    for (const trade of incomplete) {
      try {
        // Attempt to find final state
        const finalState = await this.reconstructTradeState(trade);

        if (finalState) {
          await this.completeTrade(trade, finalState);
        } else {
          // Mark as abandoned and refund
          await this.abandonTrade(trade);
        }
      } catch (error) {
        logger.error('Failed to replay trade', { trade, error });
      }
    }
  }
}
```

10. Implementation Details

10.1 Project Structure

The TypeScript backend follows a modular architecture:

```
cash-markets-backend/
├── src/
│   ├── api/
│   │   ├── rest/
│   │   │   ├── controllers/
│   │   │   ├── middleware/
│   │   │   └── routes/
│   │   └── websocket/
│   │       ├── handlers/
```

```
│   │       ├── protocol/
│   │       └── sessions/
│   ├── core/
│   │   ├── game-engine/
│   │   ├── price-engine/
│   │   ├── trade-manager/
│   │   └── account-manager/
│   ├── data/
│   │   ├── repositories/
│   │   ├── entities/
│   │   └── migrations/
│   ├── infrastructure/
│   │   ├── database/
│   │   ├── cache/
│   │   ├── queue/
│   │   └── monitoring/
│   └── shared/
│       ├── types/
│       ├── utils/
│       └── constants/
├── tests/
├── scripts/
└── config/
```

10.2 Development Workflow

The implementation follows test-driven development:

```
// Example test for trade execution
describe('TradeManager', () => {
  let tradeManager: TradeManager;
  let mockDb: MockDatabase;
  let mockRedis: MockRedis;

  beforeEach(() => {
    mockDb = new MockDatabase();
    mockRedis = new MockRedis();
    tradeManager = new TradeManager(mockDb, mockRedis);
  });

  describe('executeTrade', () => {
    it('should lock funds before opening position', async () => {
      const trade = {
        accountId: 'user123',
        roundId: 'round456',
        size: new Decimal('100'),
        direction: TradeDirection.LONG
      };

      await tradeManager.executeTrade(trade);
```

```
      expect(mockDb.queries).toContainEqual({
        sql: expect.stringContaining('UPDATE accounts SET
locked_balance'),
        params: expect.arrayContaining([trade.size.toString()])
      });
    });

    it('should handle insufficient balance gracefully', async () => {
      mockDb.setQueryResponse('SELECT balance', { balance: '50' });

      const trade = {
        accountId: 'user123',
        size: new Decimal('100'),
        direction: TradeDirection.LONG
      };

      await expect(tradeManager.executeTrade(trade))
        .rejects.toThrow(InsufficientBalanceError);
    });
  });
});
```

10.3 Deployment Configuration

The system uses environment-based configuration:

```yaml
# docker-compose.yml
version: '3.8'

services:
  app:
    build: .
    ports:
      - "3000:3000"
      - "8080:8080"
    environment:
      - NODE_ENV=production
      - DATABASE_URL=postgresql://user:pass@postgres:5432/cashmarkets
      - REDIS_URL=redis://redis:6379
      - APTOS_NETWORK=mainnet
      - APTOS_NODE_URL=https://fullnode.mainnet.aptoslabs.com
    depends_on:
      - postgres
      - redis
      - timescale
    deploy:
      replicas: 3
      resources:
        limits:
          cpus: '2'
          memory: 4G
```

```
  postgres:
    image: postgres:15
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=cashmarkets
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
    deploy:
      resources:
        limits:
          cpus: '4'
          memory: 8G

  redis:
    image: redis:7-alpine
    command: redis-server --appendonly yes
    volumes:
      - redis_data:/data
    deploy:
      resources:
        limits:
          cpus: '1'
          memory: 2G

  timescale:
    image: timescale/timescaledb:latest-pg15
    volumes:
      - timescale_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=timeseries
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
```

10.4 Monitoring and Observability

The system implements comprehensive monitoring:

```
class MonitoringService {
  private prometheus: PrometheusClient;
  private healthChecks: Map<string, HealthCheck>;

  constructor() {
    this.setupMetrics();
    this.setupHealthChecks();
  }

  private setupMetrics(): void {
    // Business metrics
    this.prometheus.registerGauge({
```

```
      name: 'active_rounds_total',
      help: 'Total number of active rounds'
    });

    this.prometheus.registerHistogram({
      name: 'trade_execution_duration_seconds',
      help: 'Time taken to execute trades',
      buckets: [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1]
    });

    this.prometheus.registerCounter({
      name: 'candles_generated_total',
      help: 'Total number of candles generated',
      labelNames: ['round_id']
    });

    // System metrics
    this.prometheus.registerGauge({
      name: 'websocket_connections_active',
      help: 'Number of active WebSocket connections'
    });

    this.prometheus.registerHistogram({
      name: 'database_query_duration_seconds',
      help: 'Database query execution time',
      labelNames: ['query_type'],
      buckets: [0.001, 0.01, 0.1, 1, 10]
    });
  }

  private setupHealthChecks(): void {
    // Database health
    this.healthChecks.set('database', {
      name: 'PostgreSQL Connection',
      check: async () => {
        const result = await this.postgres.query('SELECT 1');
        return { healthy: true, latency: result.duration };
      }
    });

    // Redis health
    this.healthChecks.set('redis', {
      name: 'Redis Connection',
      check: async () => {
        const start = Date.now();
        await this.redis.ping();
        return { healthy: true, latency: Date.now() - start };
      }
    });

    // Blockchain health
    this.healthChecks.set('blockchain', {
      name: 'Aptos RPC Connection',
      check: async () => {
```

```
          const start = Date.now();
          const height = await this.aptos.getBlockHeight();
          return {
            healthy: true,
            latency: Date.now() - start,
            blockHeight: height
          };
        }
      });
    }
  }
```

10.5 Performance Benchmarks

Expected performance characteristics based on the architecture:

Metric	Target	Implementation Detail
Candle Generation	< 0.1ms	Pure computation in memory
Trade Execution	< 5ms	Redis + async PostgreSQL write
WebSocket Broadcast	< 1ms	Binary protocol with pipelining
Position Update	< 10ms	Cached balance + Redis update
Database Write	< 50ms	Batched writes with connection pooling
Cache Hit Rate	> 95%	Multi-layer caching strategy
Concurrent Users	10,000+	Horizontal scaling with sticky sessions
Rounds per Second	100+	Limited by blockchain seed generation

The architecture scales horizontally by adding more server instances behind the load balancer. Each server can handle approximately 3,000 concurrent WebSocket connections with the specified hardware requirements. The bottleneck shifts to the blockchain for seed generation at scale, which can be mitigated by pre-generating seeds during low-activity periods.

Conclusion

This centralized backend architecture provides a robust foundation for Cash Markets that prioritizes performance and development velocity while maintaining the core value proposition of provably fair gaming. The layered approach with PostgreSQL for durability, Redis for performance, and carefully managed state synchronization ensures the system can handle the demanding requirements of real-time trading with real money.

The modular service architecture enables rapid iteration and testing while the comprehensive error handling and monitoring infrastructure ensures production reliability. By offloading all complex logic to the backend while using Aptos purely for monetary settlement and randomness, the system achieves an optimal balance between decentralization where it matters and centralization where it provides tangible benefits.