Cash Markets: Verifiably Random Candlestick Trading Simulator

Technical Architecture Document for Aptos Integration with Shelby Protocol

Version 1.0 | July 2025

Table of Contents

---

1. Executive Summary

Cash Markets generates candlestick charts that update every 65 milliseconds while maintaining provable randomness through on-chain seeds and deterministic off-chain expansion.

The system solves the challenge of combining sub-second visual updates with blockchain-based trust guarantees. By leveraging Aptos's randomness API for unpredictable seeds and Shelby Protocol for high-performance data distribution, it provides fast updates with transparency.

Key Technical Achievements

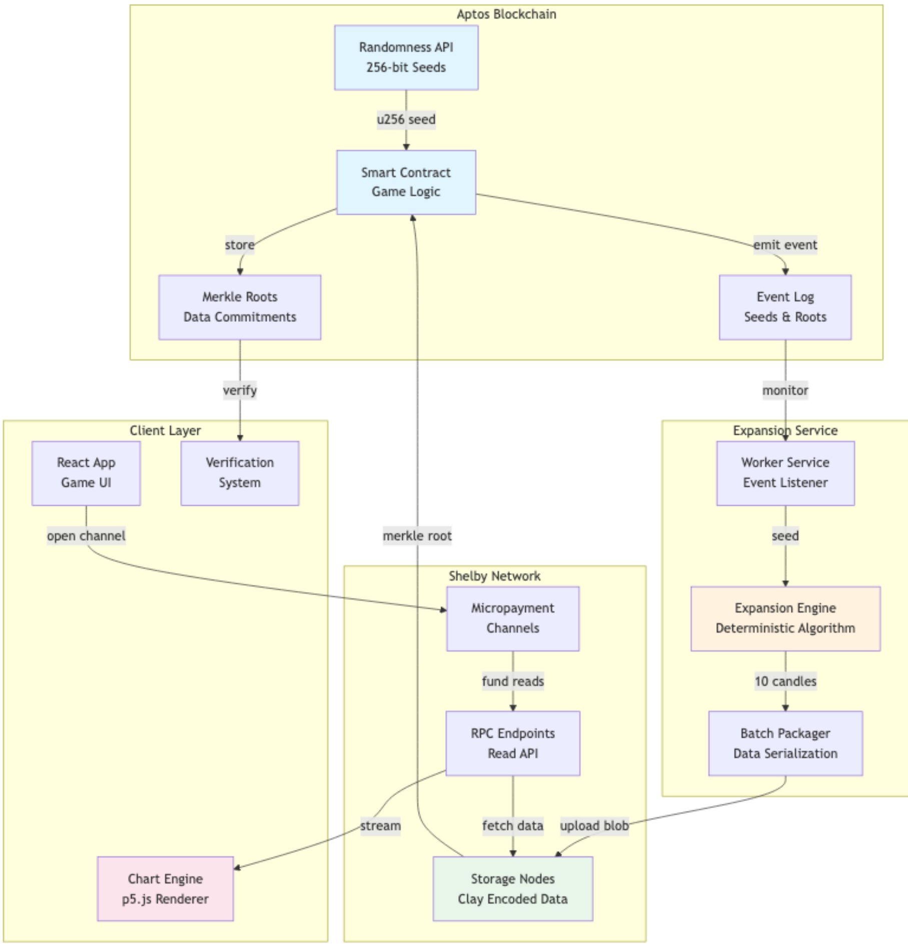
Challenge	Traditional Approach	Our Solution
Update Frequency	Compromise on speed or trust	65ms updates with on-chain proof

Challenge	Traditional Approach	Our Solution
Gas Economics	High costs limit functionality	Batched operations reduce costs 10x
Data Distribution	Centralized servers or slow chains	Shelby's fiber network ensures CDN-like performance
House Edge	Hidden algorithms	Transparent, verifiable edge through liquidation events

---

2. System Overview and Architecture

The architecture separates on-chain and off-chain components, optimizing each for its strengths while maintaining cryptographic links.



Component Responsibilities

On-Chain Components (Aptos) - Generate cryptographically secure random seeds every 650ms (10 candles × 65ms) - Store Merkle roots of generated batches for verification - Handle game state, round management, and trade settlement - Emit events for off-chain services to monitor

Off-Chain Expansion Service - Monitor blockchain for new seed events - Expand seeds into candlestick data using deterministic algorithms - Package data and upload to Shelby for distribution - Submit Merkle roots back to chain for verification

Shelby Data Layer - Store candle batches with sub-2x replication overhead - Provide 40+ Mbps read throughput - Handle micropayment channels for gasless data access - Maintain permanent archive for replay functionality

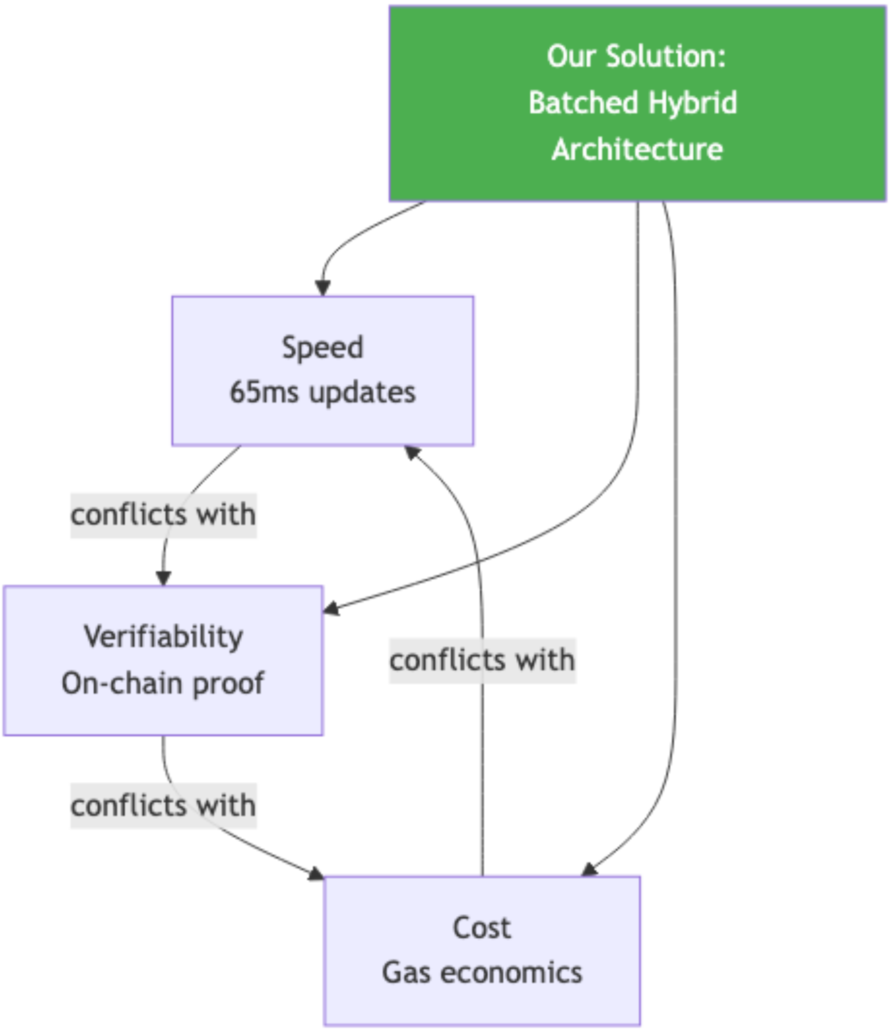
Client Application - Render charts at 65ms intervals with smooth interpolation - Prefetch upcoming batches to prevent stuttering - Verify data integrity using on-chain roots - Handle position management and P&L calculations

---

3. The Randomness Challenge

Building a verifiably random trading simulator requires reconciling three competing requirements:

3.1 The Trilemma



Speed Requirements: Mobile games demand responsive visuals. Our 65ms update rate creates continuous price movement for engaging gameplay.

Verifiability Requirements: Players must prove that charts weren't manipulated based on their positions. This requires on-chain anchoring of randomness.

Cost Requirements: With 460 candles per 30-second round, naive on-chain generation would cost 920,000 octas (0.0092 APT) just for randomness calls.

3.2 The Batching Solution

By grouping candles into batches of 10, we achieve a 650ms window for each on-chain operation. This provides time for:

1. **Blockchain finality** (~600ms on Aptos)
2. **Seed expansion** (~1ms for computational work)
3. **Network propagation** (~49ms buffer for latency variations)

This batching reduces randomness calls from 460 to 46 per round, bringing costs down to 92,000 octas—within budget while maintaining unpredictability.

---

4. Aptos Randomness API Deep Dive

The Aptos randomness module (AIP-41) provides cryptographically secure, publicly verifiable randomness.

4.1 API Capabilities

The randomness module offers several key functions:

```
randomness::u256_integer() → 256-bit random value
randomness::u64_range(min, max) → value in [min, max)
randomness::bytes(n) → n random bytes
```

4.2 Security Properties

Unpredictability: The randomness is generated using a threshold VRF (Verifiable Random Function) run by validators. Even a malicious minority cannot predict future values.

Unbiasability: No single party can influence the random output, ensuring fair gameplay.

Public Verifiability: All random values are public and can be verified by anyone.

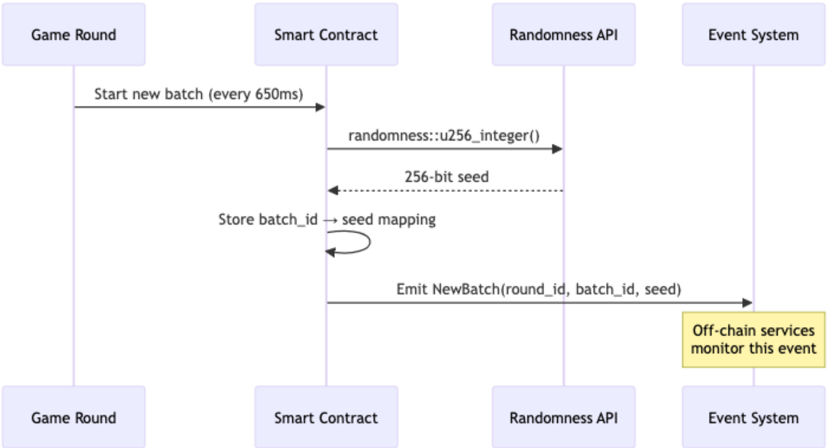
4.3 Integration Considerations

Gas Costs: Each randomness call consumes 2,000 octas. This fixed cost drives our batching strategy.

Entry Function Requirements: Randomness can only be called from functions marked with `#[randomness]`, ensuring proper transaction handling.

Undergasing Protection: The API includes protections against undergasing attacks where adversaries might try to abort unfavorable outcomes.

4.4 Our Implementation Strategy



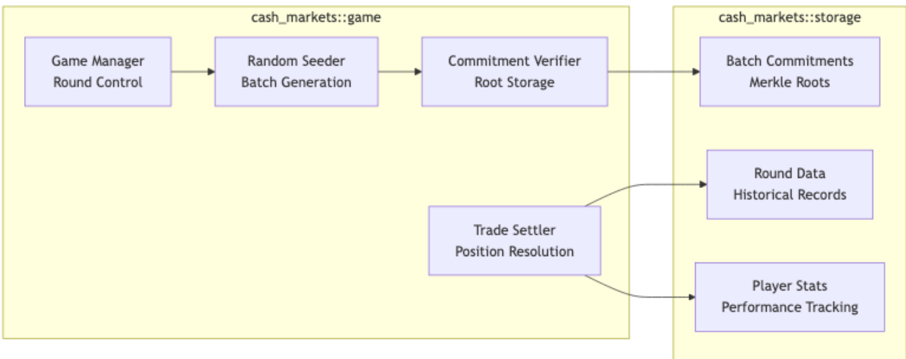
The 256-bit seed provides sufficient entropy for generating 10 candles with complex price dynamics. Each seed is permanently associated with its batch, enabling historical verification.

---

5. Smart Contract Architecture

The Move smart contracts on Aptos handle critical on-chain operations while remaining lightweight to minimize gas costs.

5.1 Core Contract Modules



5.2 Key Data Structures

Round State

```
struct Round {
    id: u64
    start_time: u64
    end_time: u64
    total_batches: u8
    status: RoundStatus
}
```

Batch Commitment

```
struct BatchCommitment {
    round_id: u64
    batch_index: u8
    seed: vector<u8>      // 256-bit seed
    merkle_root: vector<u8>  // 32-byte root
    timestamp: u64
}
```

5.3 Contract Functions

Seed Generation (Called every 650ms during active rounds)

```
#[randomness]
entry fun generate_batch_seed(round_id, batch_index)
    - Call randomness::u256_integer()
    - Emit NewBatch event with seed
    - Update round state
```

Root Commitment (Called by authorized worker after batch generation)

```
entry fun commit_batch_root(round_id, batch_index, merkle_root)
    - Verify caller authorization
    - Store root in BatchCommitment
    - Emit BatchCommitted event
```

Trade Settlement (Called at round end or position close)

```
entry fun settle_trade(player, round_id, entry_price, exit_price)
    - Verify prices against committed roots
    - Calculate P&L
    - Update player balance
    - Emit TradeSettled event
```

5.4 Event System

Events enable off-chain services to react to on-chain state changes:

```
event NewBatch {
    round_id: u64
    batch_index: u8
    seed: vector<u8>
    timestamp: u64
}
```

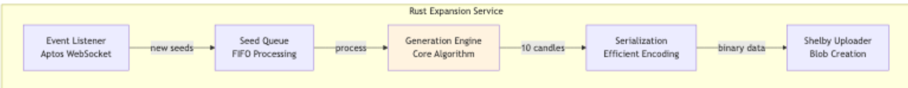
```
event BatchCommitted {
    round_id: u64
    batch_index: u8
    merkle_root: vector<u8>
}

event TradeSettled {
    player: address
    round_id: u64
    pnl: i64
    final_balance: u64
}
```

---

6. Off-Chain Expansion Engine

The expansion engine transforms on-chain seeds into candlestick data using deterministic algorithms that anyone can reproduce.

6.1 Architecture Overview



6.2 Why Rust Over Node.js

The expansion service is implemented in Rust for these reasons:

Performance: Rust generates candlesticks 10x faster than Node.js, crucial for keeping up with the 650ms batch window under load.

Determinism: Rust's strict typing and lack of garbage collection ensure identical outputs across different machines and times.

WASM Compatibility: The algorithm can be compiled to WebAssembly for client-side verification without reimplementation.

Resource Efficiency: Lower memory footprint and CPU usage reduce operational costs at scale.

6.3 The Generation Algorithm

The algorithm models cryptocurrency price action with house edge:

```
function expand_seed(seed: [u8; 32], batch_index: u8) ->
Vec<Candle>
    Initialize ChaCha20 PRNG with seed

    for i in 0..10:
```

```
// Base price movement using modified GBM
base_return = sample_normal(drift, volatility)

// Jump detection (2% probability per candle)
if random() < 0.02:
    jump_size = sample_jump_distribution()
    base_return += jump_size

// Liquidation events (0.15% base probability)
if candle_count > 80 and random() < 0.0015:
    return create_liquidation_candle()

// Generate OHLC with realistic wicks
open = previous_close
close = open * exp(base_return)
high = close + sample_wick_up()
low = close - sample_wick_down()

candles.push(Candle { open, high, low, close })
```

6.4 Market Dynamics Modeling

The algorithm incorporates several layers of market behavior:

Base Volatility: Log-normal distribution with 2% standard deviation per candle creates price movement.

Trend Cycles: Multiple sine waves of different periods create trends and reversals.

Event Simulation: Occasional large moves simulate news events or whale trades.

Consolidation Periods: Reduced volatility periods mimic real market behavior.

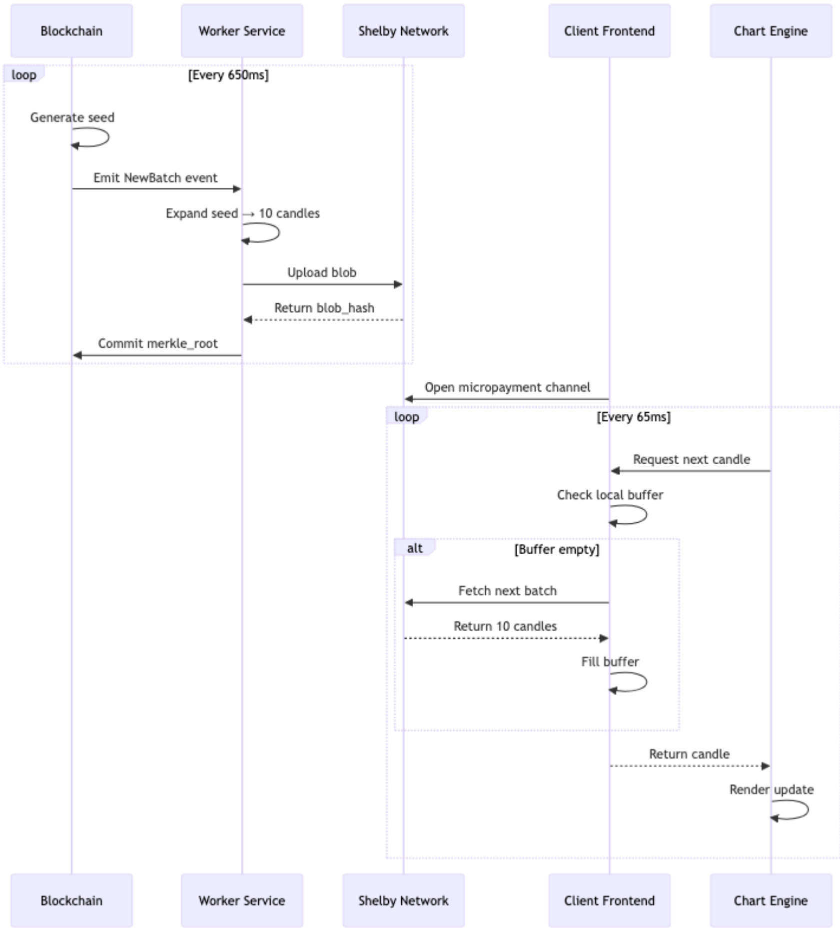
Liquidation Cascades: Rare but dramatic crashes to zero ensure house profitability.

---

7. Data Flow and Synchronization

Data flows through the system to maintain consistency and performance.

7.1 Complete Data Journey



7.2 Synchronization Strategy

Blockchain State: The source of truth for seeds and commitments. All other components derive their state from chain events.

Worker Service: Maintains a queue of unprocessed seeds to handle temporary Shelby outages.

Client Buffering: Prefetches upcoming batches when buffer drops below 5 candles, ensuring smooth playback.

Verification Checkpoints: Clients periodically verify Merkle roots against on-chain data to detect tampering.

7.3 Handling Edge Cases

Network Latency: The 650ms batch window includes a 49ms buffer for network variations.

Worker Failures: Multiple workers can operate simultaneously; the contract accepts the first valid commitment.

Shelby Outages: Clients can fall back to requesting data from alternative workers if Shelby is temporarily unavailable.

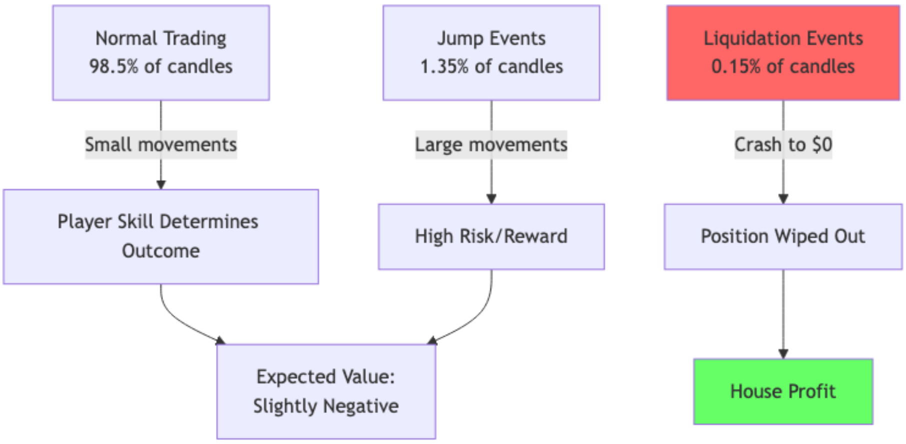
Chain Reorganizations: Event monitoring includes block confirmation to avoid processing seeds from orphaned blocks.

---

8. House Edge Mathematics

The house edge is implemented transparently through the candlestick generation algorithm, primarily via liquidation events.

8.1 Probability Model



8.2 Expected Value Calculation

Per-round expected value for a player with average skill:

```
Base drift per candle: -0.01%
Candles per round: 460
Base expected loss: -4.6%

Jump events (positive): +2% probability × +8% size = +0.16%
Jump events (negative): +2% probability × -15% size = -0.30%

Liquidation risk: 0.15% × 460 candles × 20% position size = -13.8%
(when it occurs)
Liquidation probability per round: 1 - (1 - 0.0015)^460 ≈ 50%

Expected value per round: -4.6% + 0.16% - 0.30% - (50% × 13.8%) ≈
-11.6%
```

8.3 Psychological Factors

The algorithm creates an engaging experience despite negative expected value:

Near Misses: Liquidations often occur after profitable runs, creating "almost won" scenarios.

Win Frequency: 42% of rounds are profitable before considering liquidations.

Skill Element: Better timing can improve outcomes, giving players a sense of control.

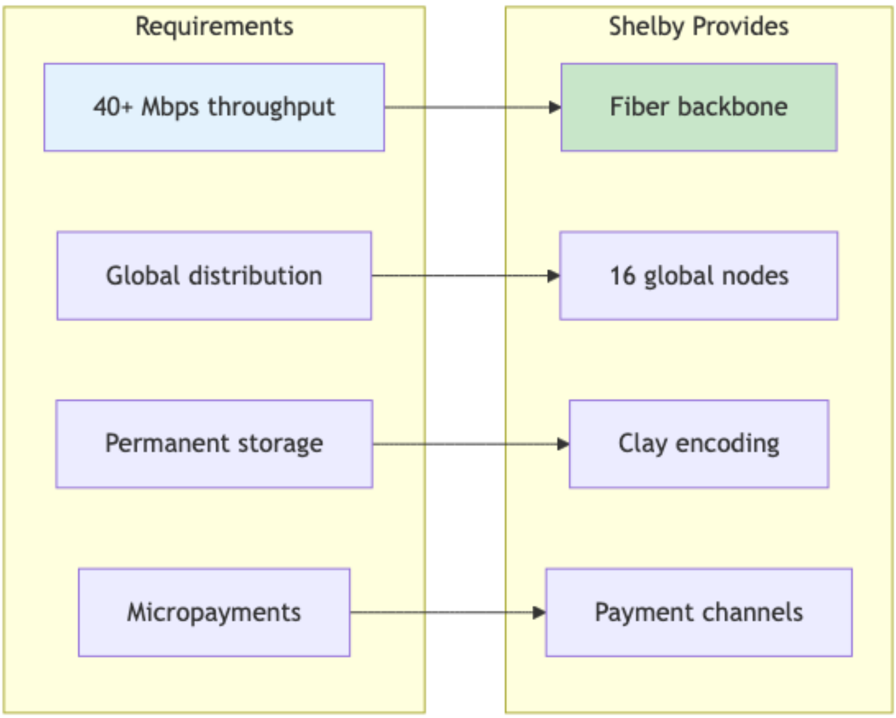
Visual Excitement: Dramatic price movements create dopamine responses regardless of P&L.

---

9. Shelby Integration Details

Shelby Protocol provides the high-performance data layer that makes real-time chart streaming possible.

9.1 Why Shelby Is Essential



9.2 Data Encoding Strategy

Each batch of 10 candles is encoded efficiently:

```
Batch Structure:
- Header (8 bytes): round_id, batch_index
- Candles (320 bytes): 10 × 32 bytes per candle
- Checksum (4 bytes): CRC32 for integrity
Total: 332 bytes per batch

Shelby Encoding:
```

```
- Clay codes with k=10, m=6
- Results in 16 chunks across providers
- Each chunk: ~33 bytes
- Total storage with coding: ~528 bytes (<2x overhead)
```

9.3 Read Performance

Shelby's architecture ensures consistent performance:

Dedicated Fiber: Eliminates public internet congestion **Strategic Caching**: Hot data stays in memory at edge nodes **Parallel Retrieval**: Clients can fetch from multiple nodes simultaneously **Micropayment Efficiency**: No on-chain transactions for reads

9.4 Cost Structure

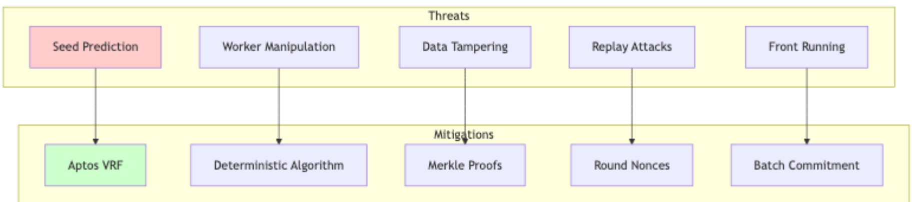
```
Write costs (per batch):
- Upload: 50 octas
- Storage: 0.00001 APT/day
- Total per round: ~0.00023 APT

Read costs (per player per round):
- 46 batches × 332 bytes = 15.3 KB
- Cost: 15.3 KB × 2x10^-7 APT/KB = 0.00000306 APT
- Negligible compared to house edge revenue
```

---

10. Security and Verification

The system implements multiple layers of security to ensure fairness and prevent exploitation.

10.1 Threat Model and Mitigations



10.2 Verification Protocol

Players can verify any historical round:

```
Verification Process:
1. Fetch round data from blockchain
   - All batch seeds
   - All Merkle roots
```

```
2. For each batch:
   - Run expansion algorithm with seed
   - Generate expected candles
   - Compute Merkle root
   - Compare with on-chain root

3. Fetch actual data from Shelby
   - Verify blob hash matches
   - Decode and compare candles

4. Verify trade outcomes
   - Confirm entry/exit prices
   - Recalculate P&L
```

10.3 Cryptographic Guarantees

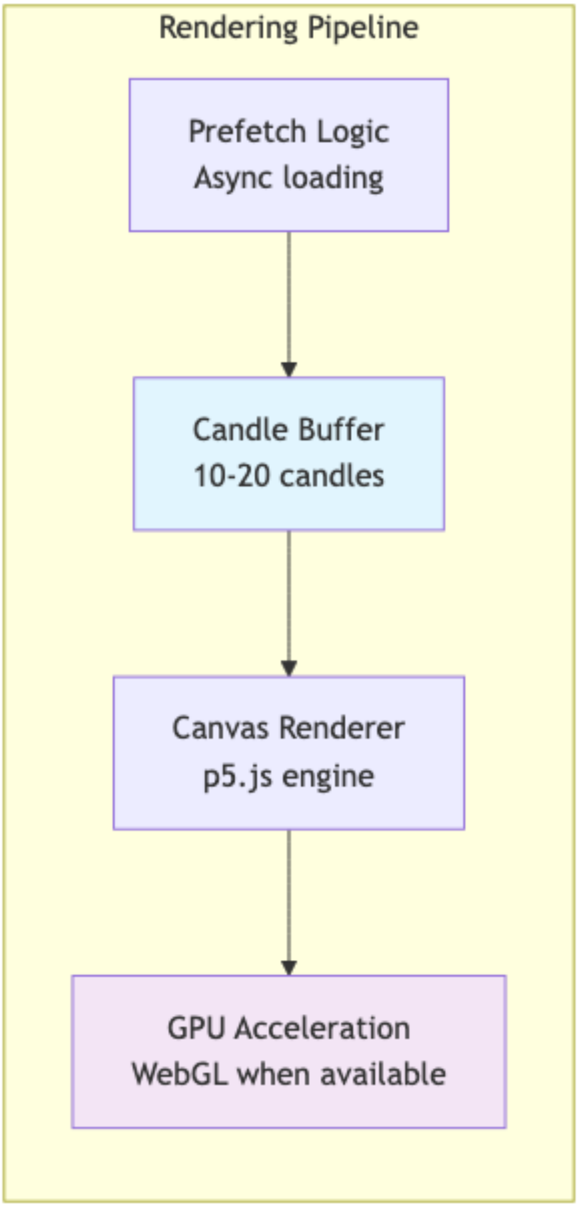
Randomness: 256-bit seeds provide 2^256 possible states per batch **Collision Resistance**: SHA-256 Merkle roots prevent tampering **Non-repudiation**: On-chain commitments create permanent audit trail

---

11. Performance Optimization

Achieving 65ms update rates requires optimization across all system components.

11.1 Client-Side Optimizations



Double Buffering: Maintain 10-20 candles in memory to absorb network jitter

Predictive Prefetching: Start fetching next batch when buffer drops below 50%

Efficient Rendering: Update only changed portions of canvas, batch draw calls

Hardware Acceleration: Utilize WebGL for smooth animations on capable devices

11.2 Network Optimizations

Batch Compression: Use MessagePack for 30% size reduction over JSON

Connection Pooling: Maintain persistent connections to Shelby nodes

Geographic Routing: Connect to nearest Shelby node based on latency

Adaptive Quality: Reduce update rate on poor connections while maintaining fairness

11.3 Blockchain Optimizations

Event Filtering: Subscribe only to relevant contract events

Batch Transaction Submission: Worker submits multiple roots in one transaction when possible

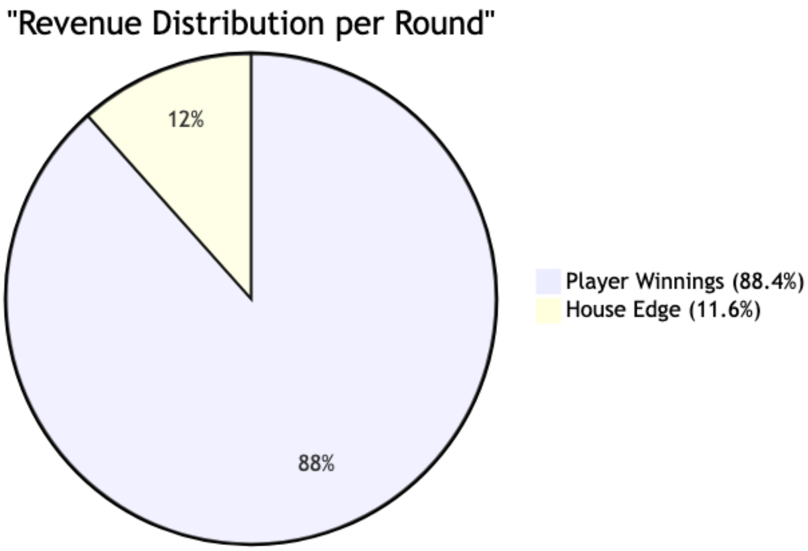
Gas Price Optimization: Use Aptos gas price oracle to minimize costs

---

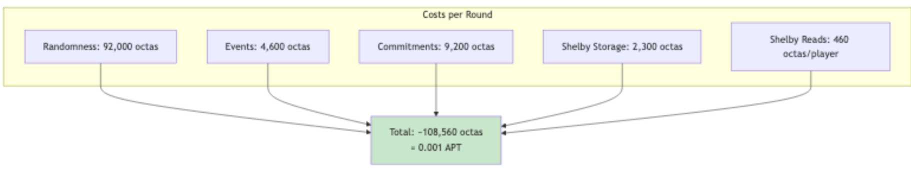
12. Economic Model

The economic sustainability of Cash Markets depends on balancing player engagement with operational costs.

12.1 Revenue Streams



12.2 Cost Structure



12.3 Break-Even Analysis

```
Assumptions:
- Average bet size: 0.1 APT (20% of starting balance)
```

```
- House edge: 11.6%
- Cost per round: 0.001 APT

Break-even players per round:
0.001 APT ÷ (0.1 APT × 11.6%) = 0.86 players

With 100 concurrent players:
Revenue: 100 × 0.1 × 11.6% = 1.16 APT
Costs: 0.001 + (100 × 0.00000306) = 0.00131 APT
Profit: 1.15869 APT per round
```