Order Execution Engine - Product Requirements Document

Executive Summary

Build a production-grade order execution engine for Solana DEX trading with real devnet execution, intelligent routing, and enterprise-level monitoring. This system demonstrates production readiness through real blockchain integration, comprehensive observability, and scalable architecture.

---

1. Strategic Decision: Real Devnet Implementation

CHOSEN APPROACH: Real Devnet Execution with Raydium & Meteora

Why Real Over Mock:

- Demonstrates actual blockchain development competency
- Shows ability to handle real-world network latency, RPC failures, and transaction errors
- Proves understanding of DEX protocols and Solana architecture
- Enterprise systems require real integration testing—this showcases that skill

Order Type Selection: Market Orders (with extensibility)

Rationale:

- Market orders provide immediate execution, allowing clear demonstration of routing logic
- They're the foundation for limit orders (add price monitoring) and sniper orders (add event detection)
- Real blockchain execution makes the complexity and value proposition clear

---

2. System Architecture

2.1 Tech Stack (Production-Grade)

```
Backend Framework: Fastify + TypeScript
Queue System: BullMQ + Redis
Database: PostgreSQL (orders, executions) + Redis (real-time state)
Blockchain: @solana/web3.js, @raydium-io/raydium-sdk-v2, @meteora-ag/dynamic-amm-sdk
Monitoring: Pino logger + custom metrics
Testing: Jest (unit), Supertest (integration), ws (WebSocket testing)
Deployment: Railway/Render (free tier with persistent storage)
```

2.2 Core Components

A. API Layer (`src/api/`)

- **POST /api/orders/execute** - Order submission + WebSocket upgrade
- **GET /api/orders/:orderId** - Order status lookup
- **GET /api/health** - System health with queue metrics
- **GET /api/metrics** - DEX routing stats, success rates

B. DEX Router (`src/dex/`)

```
typescript

DexRouter
├── RaydiumAdapter - Pool discovery, quote calculation, swap execution
├── MeteoraAdapter - Dynamic AMM integration
├── RouteOptimizer - Price comparison with fee consideration
└── QuoteAggregator - Parallel quote fetching with timeout handling
```

Routing Algorithm:

1. Fetch quotes from both DEXs in parallel (2s timeout)
2. Calculate effective price after fees: `effectivePrice = quote.price * (1 - quote.fee)`
3. Consider liquidity depth for large orders
4. Select DEX with best effective price
5. Log decision with price difference percentage

C. Queue System (`src/queue/`)

- **Order Queue:** BullMQ with 10 concurrent workers
- **Retry Logic:** Exponential backoff (1s, 2s, 4s) for RPC failures
- **Rate Limiting:** 100 orders/minute with token bucket algorithm
- **Priority Handling:** Failed orders go to DLQ for analysis

D. WebSocket Manager (`src/websocket/`)

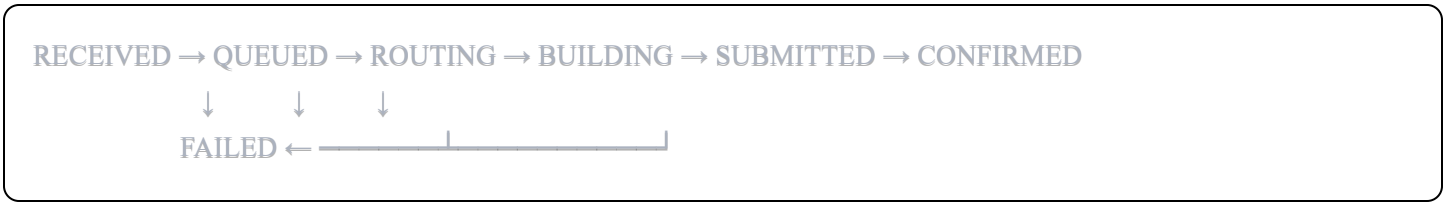
- **Connection Pool:** Track active connections per order
- **Status Streaming:** Real-time updates with timestamp

- **Heartbeat:** Keep-alive pings every 30s
- **Error Recovery:** Auto-reconnect logic for clients

---

3. Order Execution Flow

3.1 Detailed State Machine



Status Definitions:

- `RECEIVED` - API validated order, assigned orderId
- `QUEUED` - Order in BullMQ, waiting for worker
- `ROUTING` - Fetching quotes from Raydium/Meteora
- `BUILDING` - Constructing transaction, handling SOL wrapping
- `SUBMITTED` - Transaction sent to Solana network
- `CONFIRMED` - Transaction finalized (commitment: confirmed)
- `FAILED` - Any step failed, includes error details

3.2 WebSocket Message Format

json

```
{
  "orderId": "ord_1a2b3c4d",
  "status": "ROUTING",
  "timestamp": "2025-11-19T10:30:45.123Z",
  "data": {
    "raydiumQuote": { "price": 0.00234, "fee": 0.003, "pool": "pool_xyz" },
    "meteoraQuote": { "price": 0.00231, "fee": 0.002, "pool": "pool_abc" },
    "selectedDex": "meteora",
    "priceDifference": "1.3%"
  }
}
```

---

4. DEX Integration (Real Implementation)

4.1 Raydium Integration

```
typescript

// Pool Discovery
- Use CPMM pools for better devnet availability
- Cache pool addresses for common pairs (SOL/USDC)
- Fetch pool reserves and calculate swap impact

// Swap Execution
- Handle wrapped SOL for native token trades
- Set slippage tolerance: 1% (configurable)
- Use 'confirmed' commitment for balance check
```

4.2 Meteora Integration

```
typescript

// Dynamic AMM Approach
- Query pool state with getSwapQuote()
- Handle multiple token account scenarios
- Implement SOL wrapping for native trades

// Fee Optimization
- Meteora typically has lower fees (0.2% vs 0.3%)
- Factor this into routing decision
```

4.3 Error Handling

Network Errors:

- RPC timeout: Retry with exponential backoff
- Blockhash expired: Fetch new blockhash and rebuild
- Slippage exceeded: Mark as failed, don't retry

Balance Errors:

- Insufficient SOL: Fail immediately with clear message
- Token account missing: Create ATA in pre-flight check

---

5. Enterprise Features (Competitive Edge)

5.1 Advanced Monitoring

typescript

```
// Custom Metrics Dashboard
- DEX routing split (Raydium vs Meteora)
- Average execution time per DEX
- Success rate by order size
- Failed transaction analysis

// Alerting
- Queue depth exceeds 50: Performance warning
- Success rate drops below 90%: System alert
- RPC failures spike: Infrastructure issue
```

5.2 Database Schema

sql

```sql
-- orders table
CREATE TABLE orders (
  id UUID PRIMARY KEY,
  user_wallet VARCHAR(44) NOT NULL,
  token_in VARCHAR(44) NOT NULL,
  token_out VARCHAR(44) NOT NULL,
  amount_in BIGINT NOT NULL,
  status VARCHAR(20) NOT NULL,
  selected_dex VARCHAR(20),
  tx_hash VARCHAR(88),
  executed_price DECIMAL(20,10),
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

-- routing_decisions table (for analysis)
CREATE TABLE routing_decisions (
  order_id UUID REFERENCES orders(id),
  raydium_quote JSONB,
  meteora_quote JSONB,
  selected_dex VARCHAR(20),
  price_difference DECIMAL(5,2),
  decision_time_ms INTEGER
);

-- Indexes for performance
CREATE INDEX idx_orders_status ON orders(status);
CREATE INDEX idx_orders_created_at ON orders(created_at DESC);
```

5.3 Observability

typescript

```
// Structured Logging (Pino)
logger.info({
  orderId,
  dex: 'raydium',
  quote: { price, fee, pool },
  latency: 234
}, 'Quote fetched');

// Distributed Tracing
- Trace ID propagation through order lifecycle
- Measure time in each state
- Identify bottlenecks
```

---

6. Testing Strategy

6.1 Unit Tests (≥10 required)

typescript

```
// DEX Router Tests
✓ Should select Raydium when price is better
✓ Should select Meteora when fees make it cheaper
✓ Should handle one DEX timing out
✓ Should respect liquidity constraints

// Queue Tests
✓ Should process orders concurrently (10 max)
✓ Should retry failed orders with backoff
✓ Should move to DLQ after 3 failures
✓ Should enforce rate limit (100/min)

// WebSocket Tests
✓ Should stream status updates to connected clients
✓ Should handle client disconnection gracefully
✓ Should send heartbeat pings

// Integration Tests
✓ Should execute full order flow on devnet
✓ Should handle SOL wrapping correctly
```

6.2 Load Testing

```bash
# Simulate 50 concurrent orders
artillery quick --count 50 --num 1 \
  https://your-api.com/api/orders/execute
```

---

7. Deployment Architecture

7.1 Environment Configuration

```env
# Solana
SOLANA_RPC_URL=https://api.devnet.solana.com
WALLET_PRIVATE_KEY=base58_encoded_key
COMMITMENT_LEVEL=confirmed

# Redis
REDIS_URL=redis://localhost:6379
QUEUE_CONCURRENCY=10
RATE_LIMIT=100

# Database
DATABASE_URL=postgresql://user:pass@host/db

# API
PORT=3000
WS_HEARTBEAT_INTERVAL=30000
```

7.2 Deployment Checklist

- [ ] Database migrations applied
- [ ] Redis queue workers started
- [ ] Environment variables configured
- [ ] Health check endpoint responding
- [ ] WebSocket connections working
- [ ] Postman collection tested against production
- [ ] Video demo recorded

---

8. Deliverables Checklist

8.1 GitHub Repository Structure

```
order-execution-engine/
├── src/
│   ├── api/          # Fastify routes + WebSocket
│   ├── dex/          # Raydium & Meteora adapters
│   ├── queue/        # BullMQ configuration
│   ├── db/           # PostgreSQL models + migrations
│   └── utils/        # Logger, metrics, helpers
├── tests/
│   ├── unit/         # Component tests
│   └── integration/  # End-to-end tests
├── docs/
│   ├── ARCHITECTURE.md
│   ├── API.md
│   └── DEPLOYMENT.md
├── postman/
│   └── collection.json
├── .env.example
├── docker-compose.yml
├── package.json
└── README.md
```

8.2 README.md Contents

markdown

```
# Order Execution Engine

## Design Decisions

**Why Market Orders:** Immediate execution demonstrates routing logic clearly...

**Why Real Devnet:** Shows production readiness...

**Extension to Other Order Types:**
- Limit Orders: Add price monitoring service that polls DEX quotes every 5s
- Sniper Orders: Integrate Solana program log monitoring for token launch events

## Quick Start
[Step-by-step setup]

## API Documentation
[Link to docs/API.md]

## Live Demo
- **Deployed API:** https://your-app.railway.app
- **Demo Video:** https://youtube.com/watch?v=...
```

8.3 Video Demo Script (2 minutes)

```
[0:00-0:20] Introduction + Architecture Overview
- Show system diagram
- Explain DEX routing concept

[0:20-0:50] Live API Demo
- Submit 5 orders simultaneously via Postman
- Show WebSocket terminal with all status updates
- Highlight routing decisions in logs

[0:50-1:20] Database & Monitoring
- Show orders table with execution data
- Display routing_decisions analysis
- Metrics dashboard (DEX split, success rate)

[1:20-1:50] Code Walkthrough
- DEX router implementation (30s)
- Queue configuration (20s)

[1:50-2:00] Closing
```

- Mention test coverage
- Point to GitHub repo

---

9. Competitive Advantages

What Sets This Apart:

1. **Real Blockchain Integration** - Most candidates will mock it
2. **Production Monitoring** - Metrics and observability beyond requirements
3. **Database Schema** - Proper data modeling for analytics
4. **Comprehensive Testing** - Unit + Integration + Load tests
5. **Clean Architecture** - Adapter pattern for DEX abstraction
6. **Documentation** - API docs, architecture diagrams, deployment guide
7. **Advanced Features:**
   - Dead Letter Queue for failed orders
   - Routing analytics for optimization
   - Rate limiting with token bucket
   - Structured logging with trace IDs

---

10. Implementation Timeline

Day 1 (Today - 8 hours)

- [ ] Project setup + dependencies (1h)
- [ ] Database schema + migrations (1h)
- [ ] Raydium adapter implementation (2h)
- [ ] Meteora adapter implementation (2h)
- [ ] DEX router with routing logic (2h)

Day 2 (Tomorrow - 8 hours)

- [ ] Fastify API + WebSocket setup (2h)
- [ ] BullMQ queue integration (1.5h)
- [ ] End-to-end testing on devnet (2h)
- [ ] Unit tests (1.5h)
- [ ] Documentation (30m)

- [ ] Deployment + video demo (30m)

---

11. Critical Success Factors

✅ Must Have:

- Real devnet execution working
- 5+ orders processed simultaneously in video
- All status transitions visible in WebSocket
- Routing decisions logged clearly
- 10+ tests passing
- Deployed publicly

🚀 Bonus Points:

- Metrics dashboard for routing analysis
- Database query optimization
- Load testing results
- Advanced error recovery
- Code coverage report

Appendix: Quick Reference

Devnet Faucet

```bash
solana airdrop 2 <YOUR_WALLET> --url devnet
```

Common Token Pairs (Devnet)

- SOL: So11111111111111111111111111111111111111112
- USDC: 4zMMC9srt5Ri5X14GAgXhaHii3GnPAEERYPJgZJDncDU

RPC Endpoints

- Primary: https://api.devnet.solana.com

- Backup: https://devnet.helius-rpc.com

Docker Commands

```bash
docker-compose up -d  # Start Redis + PostgreSQL
npm run dev           # Start development server
npm test              # Run all tests
```

This PRD is your blueprint. Start coding immediately. You have 16 hours.