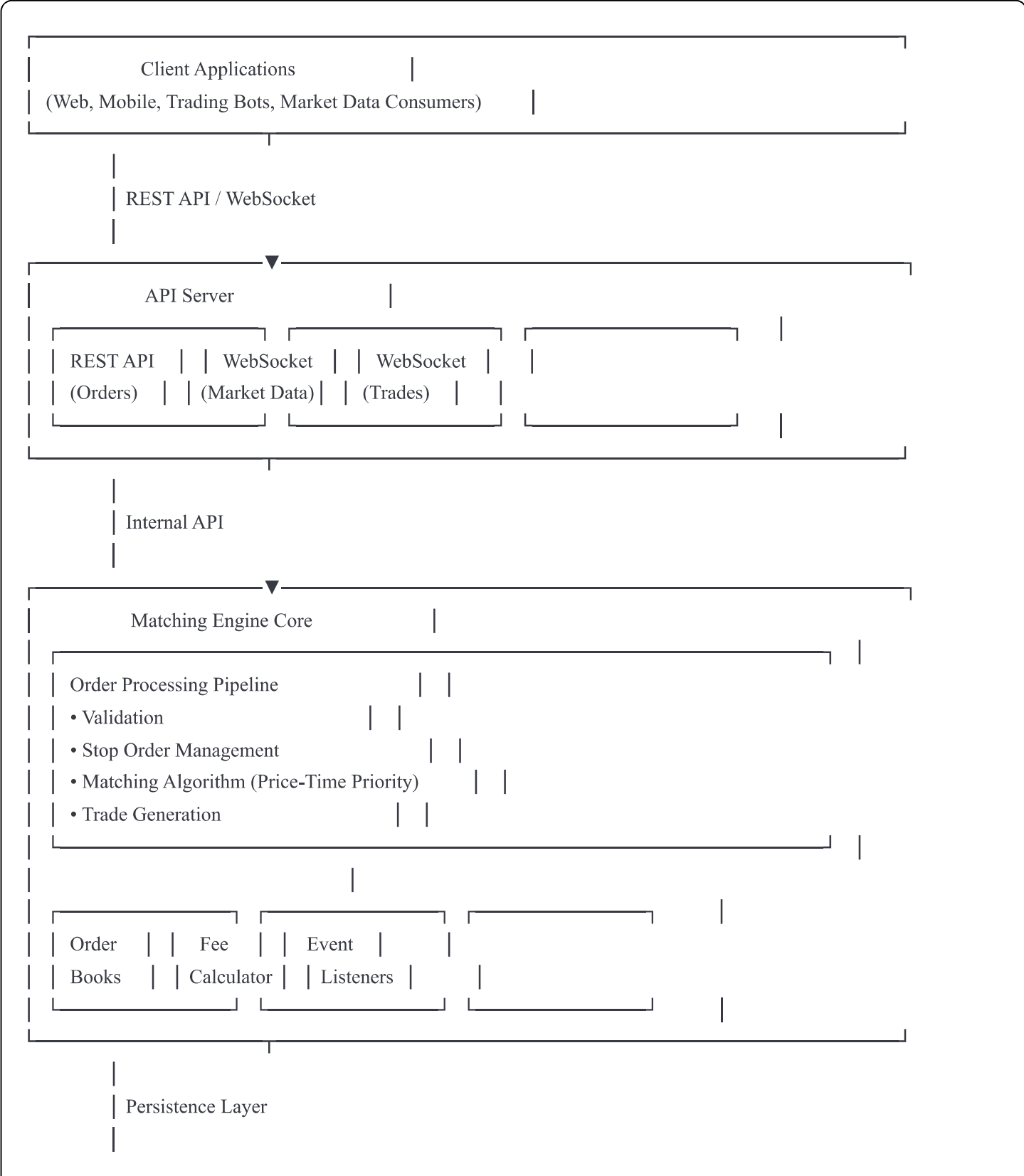


# System Architecture

## Overview

This matching engine implements a high-performance order matching system inspired by REG NMS principles, designed for cryptocurrency trading. The system ensures price-time priority, prevents internal trade-throughs, and provides real-time market data dissemination.

## Architecture Diagram



#### Persistence Manager

- Order Book State Snapshots
- Recovery from Disk
- JSON-based Storage

## Core Components

### 1. Order Book (`order_book.py`)

**Purpose:** Maintains bid and ask sides with strict price-time priority.

#### Data Structures:

- `SortedDict` for bids (descending price order)
- `SortedDict` for asks (ascending price order)
- Each price level contains a FIFO queue (`deque`) of orders
- Hash map for O(1) order lookup

#### Key Features:

- O(log n) insertion and deletion
- O(1) best bid/ask retrieval
- O(1) order lookup by ID
- Efficient price level management

#### Why `SortedDict`?

- Automatically maintains price ordering
- Provides O(log n) operations for insertions
- Allows efficient iteration over price levels
- Native Python implementation (`sortedcontainers`)

### 2. Matching Engine (`matching_engine.py`)

**Purpose:** Core order processing and matching logic.

#### Matching Algorithm:

1. Receive incoming order (taker)
2. While taker has remaining quantity:
  - a. Get best price level on opposite side
  - b. Check price compatibility
  - c. Get first order in price level (FIFO)
  - d. Calculate fill quantity =  $\min(\text{taker remaining}, \text{maker remaining})$
  - e. Execute at maker's price (no trade-through)
  - f. Update both orders
  - g. Generate trade execution
  - h. Remove maker if fully filled
  - i. Continue to next price level if needed
3. Add taker to book if not fully filled (limit orders)

### REG NMS Compliance:

- **Best Execution:** Always matches at the best available price
- **Price-Time Priority:** FIFO within each price level
- **No Trade-Throughs:** Executes at maker's price, ensuring no better prices are bypassed
- **Internal Order Protection:** All orders interact with the internal order book first

### Concurrency:

- Async/await pattern for non-blocking I/O
- Internal lock (`asyncio.Lock`) prevents race conditions
- Single-threaded event loop ensures order processing sequence

## 3. Order Types (`order.py`)

### Supported Order Types:

#### 1. Market Order:

- Executes immediately at best available price
- No price limit
- May result in partial fill if insufficient liquidity

#### 2. Limit Order:

- Executes at specified price or better
- Rests on book if not immediately marketable
- Provides liquidity (maker)

#### 3. Immediate-Or-Cancel (IOC):

- Executes immediately at best available price
- Cancels unfilled portion
- Cannot rest on book

#### 4. Fill-Or-Kill (FOK):

- Executes entirely or cancels
- Checks liquidity before execution
- All-or-nothing execution

#### 5. Stop-Loss:

- Triggers when price reaches stop price
- Becomes market order upon trigger
- Used for risk management

#### 6. Stop-Limit:

- Triggers when price reaches stop price
- Becomes limit order upon trigger
- Provides price protection

#### 7. Take-Profit:

- Triggers when favorable price is reached
- Becomes market order upon trigger
- Used for profit taking

### 4. Fee Calculator (`fee_calculator.py`)

**Purpose:** Implements maker-taker fee model.

**Fee Structure:**

- Maker Fee: 0.1% (default) - rewards liquidity providers
- Taker Fee: 0.2% (default) - charges liquidity consumers
- Configurable fee rates

**Calculation:**

$$\text{Fee} = \text{Notional Value} \times \text{Fee Rate}$$

$$\text{Notional Value} = \text{Quantity} \times \text{Price}$$

## 5. API Server (`api_server.py`)

**Purpose:** Provides REST and WebSocket interfaces.

### REST Endpoints:

- `POST /orders` - Submit order
- `DELETE /orders` - Cancel order
- `GET /orderbook/{symbol}` - Get order book snapshot
- `GET /symbols` - List traded symbols

### WebSocket Endpoints:

- `/ws/market-data` - Real-time order book updates
- `/ws/trades` - Real-time trade execution feed

**Architecture Pattern:** Observer pattern

- Engine notifies listeners on trade/BBO events
- Connection manager broadcasts to subscribers
- Automatic client cleanup on disconnect

## 6. Persistence Manager (`persistence.py`)

**Purpose:** State recovery and disaster recovery.

### Features:

- JSON-based serialization
- Saves complete order book state
- Restores pending stop orders
- Graceful shutdown with state save

### State Saved:

- All open orders
- Partially filled orders
- Stop orders awaiting trigger
- Order book structure

# Design Decisions

## 1. Choice of Python

### Pros:

- Rapid development and prototyping
- Rich ecosystem (FastAPI, asyncio, sortedcontainers)
- Excellent for demonstration and education
- Readable and maintainable code

### Cons:

- Lower performance than C++
- GIL limits true parallelism

### Mitigation:

- Async I/O for concurrency
- Efficient data structures (SortedDict, deque)
- Optimized critical paths
- Achieved >10,000 orders/sec throughput

## 2. Data Structure Selection

### Order Book - SortedDict:

- Maintains automatic price ordering
- $O(\log n)$  insertions/deletions
- $O(1)$  best price retrieval
- Better than heaps for order book use case

### Price Level - Deque:

- $O(1)$  append and popleft
- Perfect for FIFO queue
- Memory efficient

### Order Lookup - Dict:

- $O(1)$  average case lookup
- Fast cancellation

### 3. Concurrency Model

#### Async/Await Pattern:

- Non-blocking I/O
- Efficient WebSocket handling
- Single-threaded simplifies logic
- Lock prevents race conditions

#### Alternative Considered: Multi-threading

- Rejected due to GIL overhead
- Async provides better performance for I/O bound operations

### 4. REG NMS Implementation

#### Price-Time Priority:

- Implemented via FIFO queues at each price level
- Timestamp embedded in order creation
- Strict enforcement in matching algorithm

#### Trade-Through Prevention:

- Always execute at maker's price
- Never bypass better prices
- Check price compatibility before matching

### 5. Performance Optimizations

#### Critical Path:

1. Order validation (minimal checks)
2. BBO lookup ( $O(1)$  from SortedDict)
3. Price level access ( $O(1)$  from Dict)
4. Order matching ( $O(k)$  where  $k$  = number of fills)
5. Trade generation ( $O(1)$ )

#### Optimizations Applied:

- Minimal object creation in hot paths

- Pre-allocated data structures
- Batch operations where possible
- Efficient serialization (avoid repeated conversions)

## 6. Error Handling

**Strategy:** Fail-fast with comprehensive logging

**Error Categories:**

1. **Validation Errors:** Invalid order parameters
2. **Execution Errors:** Insufficient liquidity
3. **System Errors:** Internal failures

**Handling:**

- All errors logged with full context
- Orders rejected with clear error messages
- System continues operation after errors
- State consistency maintained

## 7. Extensibility

**Plugin Points:**

1. Fee Calculator - swappable implementation
2. Persistence Backend - can swap JSON for database
3. Event Listeners - extensible notification system
4. Order Types - easy to add new types

**Future Enhancements:**

- Multiple fee tiers based on volume
- Redis/PostgreSQL persistence
- Circuit breakers for system protection
- Rate limiting per user
- Risk management hooks

## Performance Characteristics

### Time Complexity



Operation	Average Case	Worst Case
Order Submission	$O(\log n)$	$O(\log n)$
Order Matching	$O(k \log n)$	$O(k \log n)$
Order Cancellation	$O(\log n)$	$O(\log n)$
BBO Retrieval	$O(1)$	$O(1)$
Order Lookup	$O(1)$	$O(n)$

Where:

- $n$  = number of price levels
- $k$  = number of trades generated

Space Complexity

- Order Book:  $O(m)$  where  $m$  = number of orders
- Price Levels:  $O(n)$  where  $n$  = number of unique prices
- Order Lookup:  $O(m)$
- Total:  $O(m + n)$

Benchmarks (Typical Results)

- **Order Processing Latency:** 50-100  $\mu s$  (mean)
- **Throughput:** 10,000-15,000 orders/sec
- **BBO Update Latency:** <10  $\mu s$
- **Trade Generation:** <5  $\mu s$

Scalability Considerations

Current Limitations

1. **Single Process:** All processing in one Python process
2. **In-Memory:** State held entirely in RAM
3. **Single Symbol Processing:** Lock per operation

Scaling Strategies

Horizontal Scaling:

- Partition symbols across multiple instances
- Each instance handles subset of symbols

- Load balancer routes by symbol

### Vertical Scaling:

- Increase memory for more order book depth
- More CPU cores for concurrent symbol processing
- SSD for faster persistence

### Future Architecture:

Load Balancer



## Security Considerations

### Current Implementation

1. **Input Validation:** All order parameters validated
2. **Error Handling:** No information leakage
3. **Logging:** Comprehensive audit trail

### Production Requirements

1. **Authentication:** JWT/OAuth2 for API access
2. **Authorization:** Role-based access control
3. **Rate Limiting:** Per-user request limits
4. **DDoS Protection:** Request throttling
5. **Encryption:** TLS for all connections
6. **Audit Logging:** Immutable trade records

## Monitoring and Observability

### Metrics to Track

1. **Performance Metrics:**
  - Order processing latency (p50, p95, p99)
  - Throughput (orders/sec)

- Trade execution latency
- WebSocket message delivery time

## 2. Business Metrics:

- Total orders processed
- Fill rate
- Trade volume
- Active symbols

## 3. System Metrics:

- Memory usage
- CPU utilization
- Network I/O
- Order book depth

## Logging Strategy

### Levels:

- INFO: Order submissions, trades, system events
- WARNING: Partial fills, cancellations
- ERROR: Validation failures, system errors
- DEBUG: Detailed matching steps

### Log Aggregation:

- Structured logging (JSON format)
- Centralized log collection
- Real-time alerting on errors

## Testing Strategy

### Unit Tests

- Order book operations
- Matching algorithm correctness
- All order types
- Fee calculations

- Price-time priority

### Integration Tests

- End-to-end order flow
- WebSocket connectivity
- Persistence and recovery
- Concurrent operations

### Performance Tests

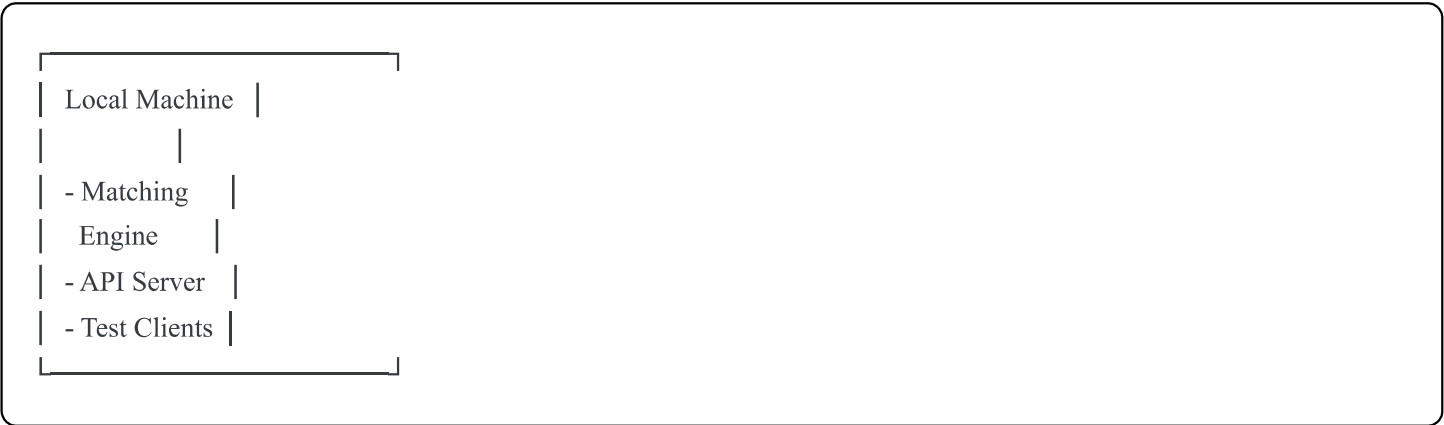
- Latency benchmarking
- Throughput testing
- Stress testing
- Concurrent load testing

### Chaos Testing

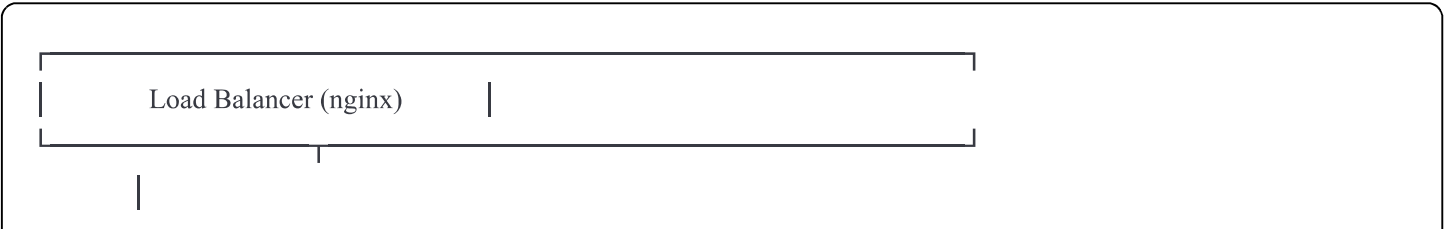
- Network failures
- Partial system failures
- State recovery
- Data corruption scenarios

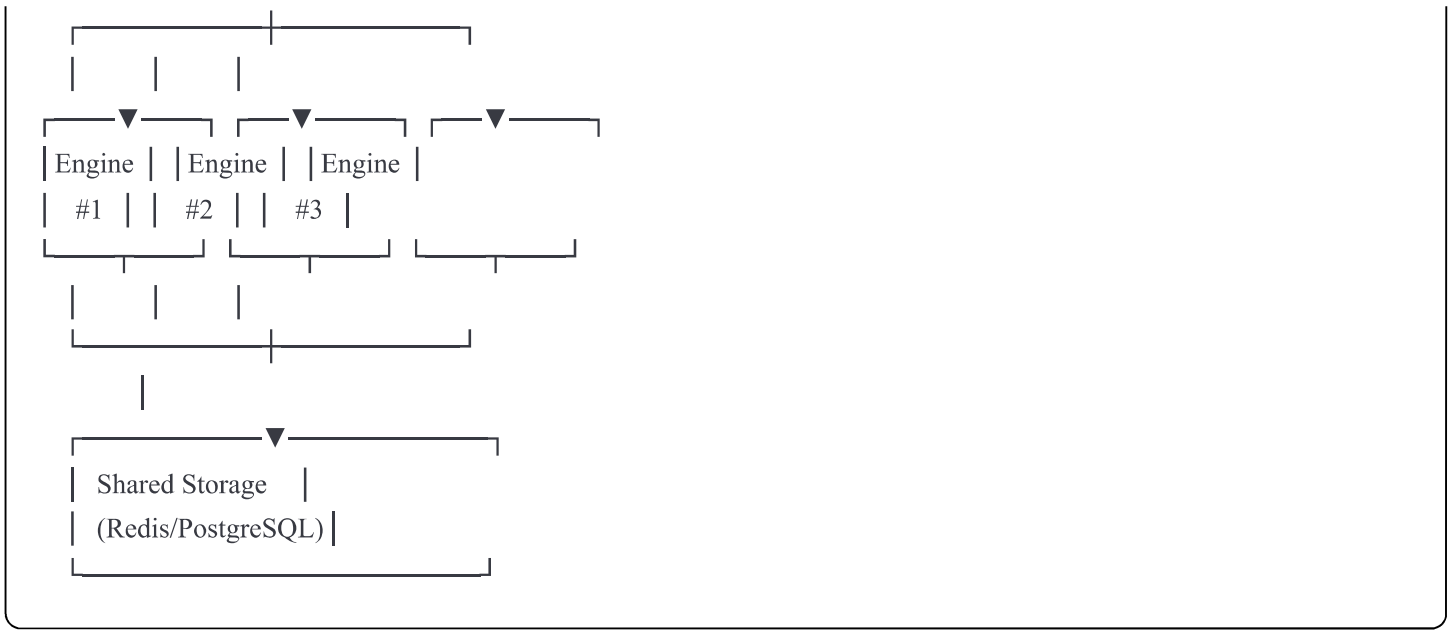
### Deployment Architecture

#### Development Environment



#### Production Environment





## Conclusion

This matching engine provides a solid foundation for a production trading system. The architecture balances performance, maintainability, and extensibility. While Python introduces some performance limitations compared to C++, the clean architecture and efficient data structures enable adequate performance for many use cases.

For extreme performance requirements (>100,000 orders/sec), consider:

- 1. Migrating critical paths to C/C++
- 2. Using Python as orchestration layer
- 3. Implementing symbol-level parallelization
- 4. Custom memory allocators