Fullstack Developer Technical Assignment

Low-Latency Order Matching Engine

Time Allocation: 8 hours (1 working day)

Language: Go or Rust (your choice)

Table of Contents

1. Problem Statement

Your task is to build a high-performance order matching engine capable of handling stock or cryptocurrency trading orders with extremely low latency. This component forms the core of any trading exchange and requires careful attention to correctness, concurrency management, and performance optimization. The matching engine must process orders efficiently while maintaining strict guarantees about trade execution order and price priority.

Understanding the Order Matching Engine

An order matching engine serves as the heart of a trading system. It receives buy and sell orders from traders, maintains an order book containing all active orders, matches compatible orders based on price-time priority rules, and executes trades when the buy price meets or exceeds the sell price. The system must guarantee correctness even under high concurrency while achieving low-latency performance that meets the demands of modern trading systems.

Important Note: You do not need to understand real-world trading, financial markets, or investment strategies to complete this assignment. This is purely a technical systems engineering challenge. The matching engine is simply a system that connects buyers and sellers based on price and time rules. Think of it like a sophisticated priority queue problem with concurrent access.

Trading systems in production environments process millions of orders daily with microsecond-level latencies. While we don't expect microsecond performance in an eight-hour assignment, we do expect you to demonstrate a solid understanding of performance-critical system design and make thoughtful optimization choices that show your ability to reason about system bottlenecks and trade-offs.

2. Technical Requirements

Core Functionality

Your implementation must support two fundamental order types. The first is the limit order, which specifies an exact price and quantity. For buy limit orders, execution occurs at the specified price or better (meaning lower), while sell limit orders execute at the specified price or better (meaning higher). If a limit order cannot be immediately matched, it remains in the order book awaiting a compatible counterparty. The second type is the market order, which does not specify a price and instead executes at the best available price in the current market. Market buy orders match against the lowest available sell price, while market sell orders match against the highest available buy price. Market orders must execute immediately or be rejected if insufficient liquidity exists.

Order Structure and Data Representation

Each order in your system must contain a unique identifier generated server-side, a trading symbol represented as a string (like "AAPL" or "BTC"), a side indicating whether it is a buy or sell order, a type specifying limit or market order, a price represented as an integer (to avoid floating-point precision issues), a quantity as an integer representing the number of shares, and a timestamp in Unix milliseconds.

Example Order Structure:

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "symbol": "AAPL",
  "side": "BUY",
  "type": "LIMIT",
  "price": 15050,
  "quantity": 100,
  "timestamp": 1698567890123
}
```

Critical: Price Representation

It is critical that you use integer arithmetic for all price calculations, storing prices in the smallest meaningful unit such as cents for dollar-denominated assets. Never use floating-point numbers for financial calculations as they introduce precision errors that are unacceptable in trading systems.

For example:

- A price of $150.50 should be stored as `15050` (cents)

- A price of $100.00 should be stored as `10000` (cents)

- A price of $99.99 should be stored as 9999 (cents)

When you need to display prices to users, divide by 100 to convert back to dollars. This approach eliminates floating-point precision issues entirely.

How Share Prices Work in This System

You are NOT responsible for determining or tracking the "real" market price of shares. Your system doesn't need to know that Apple stock trades at $150 or Bitcoin at $40,000. Here's how it works:

1. **Traders determine prices:** When someone submits a limit order, they specify what price they're willing to pay (for buys) or accept (for sells). That's their personal valuation.

2. **Your system just matches:** Your job is to match orders where the buyer's price ≥ seller's price. You don't care about external market prices.

3. **The order book determines current prices:** At any moment, the "best bid" (highest buy price) and "best ask" (lowest sell price) in your order book represent the current trading prices in your system. This emerges naturally from the orders submitted.

4. **Example flow:**

- Trader A submits: BUY 100 shares of AAPL at $150.00

- Trader B submits: SELL 100 shares of AAPL at $149.50

- Your system sees: buy price ($150.00) ≥ sell price ($149.50) → Execute trade at $149.50

- You don't need to know or care what AAPL "should" cost

5. **Price discovery:** The process of traders submitting orders and your system matching them naturally discovers prices. This is how real markets work too.

Bottom line: You're building a matching engine, not a pricing engine. You match existing orders based on their stated prices. You don't generate, predict, or track "correct" prices for any asset.

3. Matching Logic and Priority Rules (DETAILED)

This section explains in detail how your matching engine should work. Even if you've never used a trading platform, these rules are straightforward once you understand them.

3.1 The Order Book Concept

Think of the order book as two priority queues:

BUY side (also called "Bids"): All the pending buy orders, sorted from highest price to lowest price. These represent people willing to buy at various prices.

SELL side (also called "Asks"): All the pending sell orders, sorted from lowest price to highest price. These represent people willing to sell at various prices.

Visual Example of an Order Book:

```
╔═══════════════════════════════════════════════════════════════╗
║                    ORDER BOOK - AAPL                          ║
╠═══════════════════════════════════════════════════════════════╣
║  SELL SIDE (Asks) - People wanting to sell                    ║
║  Price     | Quantity | Order ID    | Timestamp               ║
║  $150.55   |   500    | order-789   | 10:30:15.123            ║
║  $150.52   |   300    | order-456   | 10:29:45.678            ║
║  $150.50   |  1000    | order-123   | 10:28:30.456            ║ ← Best Ask (lowest sell price)
╠═══════════════════════════════════════════════════════════════╣
║  ← SPREAD: $0.05 (difference between best bid & ask)          ║
╠═══════════════════════════════════════════════════════════════╣
║  $150.45   |   500    | order-321   | 10:27:12.345            ║ ← Best Bid (highest buy price)
║  $150.40   |   800    | order-654   | 10:26:50.789            ║
║  $150.35   |  1000    | order-987   | 10:25:22.123            ║
║  BUY SIDE (Bids) - People wanting to buy                      ║
╚═══════════════════════════════════════════════════════════════╝
```

Key Terms:

- **Best Bid:** The highest price someone is willing to pay (top of buy side)

- **Best Ask:** The lowest price someone is willing to accept (top of sell side)

- **Spread:** The difference between best bid and best ask

- **Price Level:** All orders at the same price

3.2 Price-Time Priority Rules

Your matching engine must follow these priority rules strictly:

Rule 1: Price Priority

Orders match based on the best price first. For buy orders, "best" means highest price. For sell orders, "best" means lowest price.

Rule 2: Time Priority (FIFO)

When multiple orders exist at the same price level, they match in the order they arrived (First In, First Out). The timestamp determines the order.

Rule 3: Partial Fills Allowed

If an incoming order's quantity exceeds what's available at the best price, it can partially fill and then continue matching at the next price level.

3.3 Detailed Matching Examples

Example 1: Simple Full Match

Initial Order Book State:

Side	Price	Quantity	Order ID	Timestamp
SELL	$150.50	1000	order-001	10:00:00.000
BUY	$150.45	500	order-002	10:01:00.000

New Order Arrives:

```
{
  "side": "BUY",
  "type": "LIMIT",
  "price": $150.50,
  "quantity": 500
}
```

Matching Process:

1. Check if buy price ($150.50) ≥ best sell price ($150.50): **YES ✓**

2. Match 500 shares at $150.50

3. Generate trade: Buyer pays $150.50 × 500 = $75,250

Result:

- Trade executed: 500 shares at $150.50

- Order-001 remaining: 500 shares still at $150.50

- New buy order: Fully filled, removed from book

Final Order Book State:

Side	Price	Quantity	Order ID	Timestamp
SELL	$150.50	500	order-001	10:00:00.000
BUY	$150.45	500	order-002	10:01:00.000

Example 2: Multiple Price Levels (Walking the Book)

Initial Order Book State:

Side	Price	Quantity	Order ID	Timestamp
SELL	$150.50	300	order-003	10:00:00.000
SELL	$150.52	400	order-004	10:00:05.000
SELL	$150.55	600	order-005	10:00:10.000
BUY	$150.45	500	order-006	10:01:00.000

New Order Arrives:

```
{
  "side": "BUY",
  "type": "LIMIT",
  "price": $150.53,
  "quantity": 800
}
```

Matching Process (Step by Step):

Step 1: Check best ask ($150.50) against buy price ($150.53)

- $150.53 ≥ $150.50? **YES ✓**

- Available at $150.50: 300 shares

- Match 300 shares at $150.50

- Remaining to fill: 800 - 300 = 500 shares

Step 2: Check next ask ($150.52) against buy price ($150.53)

- $150.53 ≥ $150.52? **YES ✓**

- Available at $150.52: 400 shares

- Match 400 shares at $150.52

- Remaining to fill: 500 - 400 = 100 shares

Step 3: Check next ask ($150.55) against buy price ($150.53)

- $150.53 ≥ $150.55? **NO ✗**

- Cannot match at this price

- Stop matching

Result:

- Trade 1: 300 shares at $150.50 (cost: $45,150)

- Trade 2: 400 shares at $150.52 (cost: $60,208)

- Total filled: 700 shares (average price: $150.51)

- Remaining: 100 shares added to book at $150.53

Final Order Book State:

Side	Price	Quantity	Order ID	Timestamp
SELL	$150.55	600	order-005	10:00:10.000
BUY	$150.53	100	order-NEW	10:02:00.000
BUY	$150.45	500	order-006	10:01:00.000

Example 3: Time Priority at Same Price (FIFO)

Initial Order Book State:

Side	Price	Quantity	Order ID	Timestamp	Order Type
SELL	$150.50	200	order-007	10:00:00.000	First
SELL	$150.50	300	order-008	10:00:05.000	Second
SELL	$150.50	400	order-009	10:00:10.000	Third

New Order Arrives:

```
{
  "side": "BUY",
  "type": "LIMIT",
  "price": $150.50,
  "quantity": 500
```

```
}
```

Matching Process:

All three sell orders are at the same price ($150.50), so we match based on time priority (FIFO):

Step 1: Match against order-007 (oldest)

- Match 200 shares at $150.50

- Remaining: 500 - 200 = 300 shares

Step 2: Match against order-008 (second oldest)

- Match 300 shares at $150.50

- Remaining: 300 - 300 = 0 shares

- Fully filled!

Result:

- Trade 1: 200 shares at $150.50 with order-007

- Trade 2: 300 shares at $150.50 with order-008

- Order-009 remains untouched (never reached)

Final Order Book State:

Side	Price	Quantity	Order ID	Timestamp
SELL	$150.50	400	order-009	10:00:10.000

Key Learning: Even though all three sell orders had the same price, the matching engine respected the order they arrived (timestamp order).

Example 4: Market Order Execution

Market orders don't specify a price, so they match at whatever prices are available in the book.

Initial Order Book State:

Side	Price	Quantity	Order ID	Timestamp
SELL	$150.50	200	order-010	10:00:00.000
SELL	$150.52	300	order-011	10:00:05.000

Side	Price	Quantity	Order ID	Timestamp
SELL	$150.55	400	order-012	10:00:10.000

New Market Order Arrives:

```
{
  "side": "BUY",
  "type": "MARKET",
  "quantity": 600
}
```

Matching Process:

Market orders match at the best available prices, walking through the book:

Step 1: Match at best ask ($150.50)

- Match 200 shares at $150.50

- Remaining: 600 - 200 = 400 shares

Step 2: Match at next ask ($150.52)

- Match 300 shares at $150.52

- Remaining: 400 - 300 = 100 shares

Step 3: Match at next ask ($150.55)

- Match 100 shares at $150.55

- Remaining: 100 - 100 = 0 shares

- Fully filled!

Result:

- Trade 1: 200 shares at $150.50 (cost: $30,100)

- Trade 2: 300 shares at $150.52 (cost: $45,156)

- Trade 3: 100 shares at $150.55 (cost: $15,055)

- Total cost: $90,311 (average price: $150.52 per share)

Key Learning: Market orders accept whatever prices are available. The buyer here paid different prices for different portions of their order. This is called "slippage" in real trading.

Example 5: Insufficient Liquidity (Market Order Rejection)

Initial Order Book State:

Side	Price	Quantity	Order ID	Timestamp
SELL	$150.50	100	order-013	10:00:00.000
BUY	$150.45	500	order-014	10:01:00.000

New Market Order Arrives:

```
{
  "side": "BUY",
  "type": "MARKET",
  "quantity": 500
}
```

Matching Process:

- Available to buy (sell side): Only 100 shares at $150.50

- Requested: 500 shares

- Insufficient liquidity!

Result:

- **Reject the order** with error: "Insufficient liquidity"

- Return HTTP 400 Bad Request

- No trades executed

Key Learning: Market orders must execute immediately and completely, or they're rejected. They cannot partially fill and remain in the book like limit orders can.

3.4 Summary Table: When Orders Match

Buy Order Type	Buy Price	Sell Order Type	Sell Price	Match?	Execution Price
LIMIT	$150.50	LIMIT	$150.50	✓ YES	$150.50 (seller's price)
LIMIT	$150.55	LIMIT	$150.50	✓ YES	$150.50 (seller's price - better for buyer)
LIMIT	$150.45	LIMIT	$150.50	✗ NO	No match

Buy Order Type	Buy Price	Sell Order Type	Sell Price	Match?	Execution Price
MARKET	N/A	LIMIT	$150.50	✓ YES	$150.50 (best available)
LIMIT	$150.50	MARKET	N/A	✓ YES	$150.50 (buyer's price)
MARKET	N/A	MARKET	N/A	N/A	Both would execute against existing limit orders

Important Rule: When prices cross (buy price ≥ sell price), execution happens at the **price of the order that was already in the book** (the "resting order"). The incoming order is the "aggressor" and accepts the resting order's price.

3.5 Your Implementation Requirements

Your matching engine must:

1. **Maintain the order book** with proper sorting (buys by price descending, sells by price ascending)
2. **Check for matches** whenever a new order arrives
3. **Execute trades** by matching against the best available prices
4. **Handle partial fills** by continuing to walk the book
5. **Respect time priority** at each price level (FIFO)
6. **Generate trade records** for each execution
7. **Update the order book** by removing filled orders and updating partially filled orders
8. **Reject market orders** if insufficient liquidity exists

You do NOT need to:

- Know anything about real stock prices
- Implement complex trading strategies
- Handle financial regulations or compliance
- Process actual money
- Connect to real exchanges

This is purely a data structures and algorithms challenge with concurrent access requirements.

4. API Requirements

Implement a REST API with the following endpoints:

4.1 Submit Order

```
POST /api/v1/orders
Content-Type: application/json

Request Body:
{
  "symbol": "AAPL",
  "side": "BUY",
  "type": "LIMIT",
  "price": 15050,        // Required for LIMIT, omit for MARKET
  "quantity": 100
}

Response (201 Created - Accepted but not filled):
{
  "order_id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "ACCEPTED",
  "message": "Order added to book"
}

Response (202 Accepted - Partial Fill):
{
  "order_id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "PARTIAL_FILL",
  "filled_quantity": 60,
  "remaining_quantity": 40,
  "trades": [
    {
      "trade_id": "trade-123",
      "price": 15050,
      "quantity": 60,
      "timestamp": 1698567890123
    }
  ]
}

Response (200 OK - Fully Filled):
{
  "order_id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "FILLED",
  "filled_quantity": 100,
  "trades": [
    {
      "trade_id": "trade-123",
      "price": 15050,
      "quantity": 100,
      "timestamp": 1698567890123
    }
  ]
}

Response (400 Bad Request):
{
  "error": "Invalid order: quantity must be positive"
}

Response (400 Bad Request - Insufficient Liquidity for Market Order):
{
  "error": "Insufficient liquidity: only 50 shares available, requested 100"
}
```

4.2 Cancel Order

```
DELETE /api/v1/orders/{order_id}

Response (200 OK):
{
  "order_id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "CANCELLED"
}

Response (404 Not Found):
{
  "error": "Order not found"
}

Response (400 Bad Request):
{
  "error": "Cannot cancel: order already filled"
}
```

4.3 Get Order Book

```
GET /api/v1/orderbook/{symbol}?depth=10

Response (200 OK):
{
  "symbol": "AAPL",
  "timestamp": 1698567890123,
  "bids": [
    {"price": 10045, "quantity": 500},    // Highest buy price first
    {"price": 10040, "quantity": 1000},
    {"price": 10035, "quantity": 800}
  ],
  "asks": [
    {"price": 10050, "quantity": 800},    // Lowest sell price first
    {"price": 10055, "quantity": 600},
    {"price": 10060, "quantity": 1200}
  ]
}

Notes:
- Bids sorted by price descending (highest first)
- Asks sorted by price ascending (lowest first)
- Depth parameter controls how many price levels to return
- Quantities at each price level are aggregated
```

4.4 Get Order Status

```
GET /api/v1/orders/{order_id}

Response (200 OK):
{
  "order_id": "550e8400-e29b-41d4-a716-446655440000",
  "symbol": "AAPL",
  "side": "BUY",
  "type": "LIMIT",
  "price": 10050,
  "quantity": 100,
  "filled_quantity": 60,
  "status": "PARTIAL_FILL",    // ACCEPTED | PARTIAL_FILL | FILLED | CANCELLED
  "timestamp": 1698567890123
}

Response (404 Not Found):
{
  "error": "Order not found"
}
```

4.5 Health Check

```
GET /health

Response (200 OK):
{
  "status": "healthy",
  "uptime_seconds": 3600,
  "orders_processed": 150000
}
```

4.6 Metrics Endpoint

```
GET /metrics

Response (200 OK):
{
  "orders_received": 150000,
  "orders_matched": 120000,
  "orders_cancelled": 5000,
  "orders_in_book": 25000,
  "trades_executed": 95000,
  "latency_p50_ms": 2.5,
  "latency_p99_ms": 8.2,
  "latency_p999_ms": 15.7,
  "throughput_orders_per_sec": 52000
}
```

Data Structure Requirements

You must implement an efficient order book that supports:

Required Operations:

- Insert order: Should be efficient (O(log n) or better preferred)
- Remove order: O(1) or O(log n)
- Get best bid/ask: O(1)
- Match orders: Efficient iteration through price levels
- Cancel by order ID: Fast lookup

Suggested Approaches:

- HashMap for O(1) order lookup by ID
- Sorted structure (skip list, B-tree, heap) for price levels
- Queue/list at each price level for FIFO ordering

Concurrency Requirements

Your system must handle concurrent requests safely:

Requirements:

- Multiple clients submitting orders simultaneously
- No race conditions
- No deadlocks
- Maintain correctness under load
- Handle at least 100 concurrent connections

Acceptable Approaches:

- Single-threaded event loop (simple, no locks needed)
- Thread-per-symbol partitioning
- Lock-based synchronization (RWMutex/Mutex)
- Lock-free data structures (advanced)

You must document your concurrency strategy in your README.

Error Handling

Handle and return appropriate errors for:

- Invalid order data (negative price/quantity, invalid symbol)
- Duplicate order IDs
- Cancel non-existent order
- Market order with insufficient liquidity
- Malformed JSON
- Server errors

Use appropriate HTTP status codes:

- 200: Success
- 201: Created
- 400: Bad Request
- 404: Not Found
- 500: Internal Server Error
- 503: Service Unavailable

5. Performance Requirements

Performance is a critical evaluation criterion for this assignment. Your implementation must meet specific mandatory targets to pass the performance portion of the evaluation. These targets are designed to be challenging but achievable within the eight-hour time constraint given thoughtful design choices and appropriate optimization.

Mandatory Performance Targets

Your system must sustain a minimum throughput of 30,000 orders per second measured over a sixty-second load test with concurrent clients. The median latency (p50) from order submission to response must not exceed 10 milliseconds. More critically, the 99th percentile latency (p99) must remain under 50 milliseconds, and the 99.9th percentile latency (p999) must stay below 100 milliseconds. Your matching logic must maintain 100% correctness with no invalid trades, race conditions, or data corruption. Additionally, the system must successfully handle at least 100 concurrent client connections without degradation.

Metric	Requirement	How Measured
Throughput	≥ 30,000 orders/second sustained	60-second load test with concurrent clients
Latency (p50)	≤ 10 ms	Order submission to response time
Latency (p99)	≤ 50 ms	99th percentile of all requests
Latency (p999)	≤ 100 ms	99.9th percentile of all requests
Correctness	100%	All trades must be valid, no race conditions
Concurrent Connections	≥ 100	Simultaneous client connections

Stretch Goals

If you exceed the mandatory requirements and achieve 100,000 or more orders per second throughput, p99 latency of 10 milliseconds or less, or p999 latency of 20 milliseconds or less, you will receive bonus points that reflect exceptional performance optimization.

Metric	Stretch Goal
Throughput	≥ 100,000 orders/second
Latency (p99)	≤ 10 ms
Latency (p999)	≤ 20 ms

Performance Measurement Methodology

Latency measurements represent the complete round-trip time from when the server receives an HTTP request to when it sends the complete response. Throughput must be sustained over time rather than representing burst capacity, and all performance tests will run on standard development hardware with a four-core CPU and 16GB of RAM. Your solution should not require exotic hardware or unusual system configurations to meet performance targets.

6. Deliverables

Source Code

Submit well-organized source code with clear structure. Include all source files, dependency management files (go.mod or Cargo.toml), and any build scripts that make it easy to run your code.

Suggested Structure:

```
order-matching-engine/
■■■ README.md                 # Setup instructions and overview
■■■ go.mod / Cargo.toml      # Dependencies
■■■ main.go / main.rs        # Entry point
■■■ src/                     # Your source code
■■■ tests/                   # Test files
■■■ scripts/                 # Optional: run/test scripts
```

Documentation (README.md)

Keep documentation simple and focused. Your README should include:

Essential:

- How to build and run your application
- How to run tests
- Brief explanation of your approach (1-2 paragraphs)
- Performance results you measured (throughput, latency)

Nice to have:

- Key design decisions you made
- Any assumptions or limitations
- What you would improve with more time

Testing

Include basic tests that demonstrate your solution works:

Required:

- Unit tests for core matching logic
- A few integration tests showing end-to-end order flow
- Basic performance measurements

You don't need comprehensive test coverage, just enough to show your logic is correct and meet our criteria.

7. Evaluation Criteria

We will evaluate your submission based on the following areas. Our goal is to understand your technical capabilities and engineering judgment under time constraints.

Correctness

Your matching engine must work correctly. This is the most important criterion because incorrect matching logic disqualifies a trading system regardless of performance.

What we expect:

- Orders match correctly according to price-time priority
- No race conditions or data corruption under concurrent load
- Partial fills handled correctly
- Market orders execute properly
- Cancel operations work correctly
- No incorrect trades (wrong prices, wrong quantities, etc.)
- Proper error handling and input validation
- Edge cases handled (empty order book, insufficient liquidity, etc.)

We will test this using automated test suites with various scenarios, concurrent stress tests, and validation of all trade outputs.

Performance

Your system must meet specific performance requirements. We will measure these under realistic load conditions.

Mandatory Requirements:

- **Throughput:** Minimum 30,000 orders/second sustained over 60 seconds
- **Latency (p50):** ≤ 10 milliseconds
- **Latency (p99):** ≤ 50 milliseconds
- **Latency (p999):** ≤ 100 milliseconds

- **Concurrent Connections:** Must handle at least 100 simultaneous connections

- **Correctness:** 100% - all trades must be valid

Outstanding Performance (bonus consideration):

- Throughput ≥ 100,000 orders/second

- p99 latency ≤ 10 milliseconds

- p999 latency ≤ 20 milliseconds

Latency is measured as complete round-trip time from when the server receives the HTTP request to when it sends the complete response. Tests run on standard development hardware (4-core CPU, 16GB RAM).

Code Quality

We look for clean, maintainable code that demonstrates good engineering practices.

What we expect:

- Clean separation of concerns and logical organization

- Readable, idiomatic Go or Rust code

- Appropriate naming and minimal but helpful comments

- Proper error handling and graceful degradation

- No obvious code duplication

- Clear data flow through the system

System Design

We want to understand your design thinking and technical decision-making.

What we expect:

- Clear concurrency strategy with justification for your approach

- Appropriate data structures for the problem (efficient insert, delete, match operations)

- Understanding of trade-offs in your design decisions

- Consideration of scalability

- Memory-efficient implementation

Testing & Documentation

Basic testing and documentation are important to demonstrate your solution works and help us understand your approach.

What we expect:

- Unit tests for core matching logic
- Some integration tests showing end-to-end flow
- Basic benchmarks showing performance measurements
- Clear README with setup instructions
- Brief explanation of your approach and key decisions

8. Testing Methodology

Our Evaluation Process

After you submit your assignment, we follow a structured evaluation process. We begin with a five-minute build and run phase where we follow your README instructions exactly to build and start your application. Your code must compile without errors, start without crashes, and respond successfully to health check requests.

Next, we conduct fifteen minutes of automated correctness testing, running our comprehensive test suite containing more than 50 test scenarios. These tests include concurrent access validation, edge case verification, and race condition detection using language-specific tools.

Performance benchmarking takes approximately twenty minutes and includes three testing profiles. The standard profile generates 30,000 orders per second for sixty seconds to verify you meet minimum requirements. The stress profile pushes to 100,000 orders per second for sixty seconds to test system limits. The spike profile generates sudden bursts up to 200,000 orders per second to evaluate system behavior under extreme conditions.

We then spend thirty minutes conducting detailed code review, assessing your architecture, evaluating code quality, examining design decisions, and reviewing your documentation. Finally, we schedule a thirty-minute follow-up discussion where you walk us through your solution, explain key decisions, discuss trade-offs you considered, and explore potential scale-up scenarios.

What Causes Failure

We will reject submissions that:

- Don't build or compile

- Crash under load

- Produce incorrect trades or matching logic

- Have data races or concurrency issues

- Fail to meet mandatory performance requirements

Significant issues that hurt your evaluation:

- Missing core functionality (e.g., no order cancellation)

- Poor code organization that's hard to understand

- No tests or basic documentation
- Memory leaks or deadlocks

9. Submission Guidelines

What to Submit

Submit your complete code via GitHub, GitLab, or as a zip file.

Required:

- All source code
- README.md with setup instructions and brief explanation of your approach
- Basic tests demonstrating your solution works
- Performance results (throughput, latency measurements)

Optional but appreciated:

- Your reasoning for key design decisions
- Any profiling or optimization insights
- Load testing scripts if you built them

How to Submit

Email to: naman@repello.ai

Subject: Technical Assignment - [Your Name] - Order Matching Engine

Include in email:

- Your name
- Link to repository (or attach zip file)
- Language used (Go or Rust)
- Brief summary (2-3 sentences) of your approach
- Performance results you achieved:
- Throughput: X orders/second
- p99 Latency: X ms
- p999 Latency: X ms

Before You Submit

Quick checklist:

- [ ] Code compiles and runs

- [ ] Tests pass

- [ ] README has clear setup instructions

- [ ] Performance numbers included

- [ ] No sensitive information (API keys, etc.) in code

Test it yourself: Clone your repo in a fresh directory and follow your own README to make sure someone else can run your code.

10. Bonus Features

If you complete the mandatory requirements early and have time remaining, you may consider implementing bonus features. However, you should only work on these features if your core requirements are solid and well-tested. We strongly prefer excellent core functionality over mediocre core implementation with many buggy bonus features. Each bonus feature must be fully implemented and tested to receive credit, as partially implemented features receive no points.

Available Bonus Features

1. WebSocket Streaming API (5 points)

Implement real-time order book updates and trade notifications through WebSocket connections. Clients should be able to subscribe to specific symbols and receive immediate notifications of trades and order book changes as they occur, enabling real-time market data feeds.

2. Advanced Order Types (3-5 points each)

Implement stop loss orders that trigger market orders when price reaches a specified threshold, providing automated risk management. Fill-or-Kill (FOK) orders must execute entirely and immediately or be cancelled with no partial fills allowed. Immediate-or-Cancel (IOC) orders execute whatever quantity is immediately available and cancel the remainder, allowing partial fills but requiring immediate execution.

3. Persistence and Recovery (8 points)

Implement a Write-Ahead Log recording all operations, create periodic snapshot mechanisms to checkpoint system state, and build crash recovery capability to restore complete order book state upon restart. Benchmark your system with persistence enabled to demonstrate the performance impact.

4. Multiple Symbol Optimization (5 points)

Support efficient trading across 100 or more symbols simultaneously through per-symbol sharding techniques. Implement symbol-level metrics and benchmark performance with 100 active symbols to demonstrate scalability.

5. Advanced Matching Modes (5 points)

Implement pro-rata matching that allocates fills proportionally rather than pure FIFO, support minimum quantity requirements on orders, prevent self-trading where a single participant's buy and sell orders match, or implement time-in-force options such as Day orders, Good-Till-Cancel, or Good-Till-Date.

6. Comprehensive Observability (5 points)

Export metrics in Prometheus format for monitoring system integration, implement distributed tracing using OpenTelemetry standards, use structured JSON logging throughout the application, and optionally create a real-time dashboard for visualization.

7. Market Data Features (3 points)

Calculate OHLCV data (Open, High, Low, Close, Volume) for time windows, maintain trade history by symbol, aggregate order book depth across price levels, and provide market statistics through dedicated API endpoints.

8. Rate Limiting and Back-pressure (4 points)

Implement per-client rate limiting to prevent abuse, add circuit breaker patterns for overload protection, design graceful degradation strategies under extreme load, and monitor queue depth to detect back-pressure conditions.

9. Advanced Testing (3 points)

Implement property-based testing that verifies system invariants hold across random inputs, add fuzz testing to discover edge cases, conduct chaos engineering tests that inject failures, or build sophisticated load testing harnesses that simulate realistic trading patterns.

10. Production Readiness (5 points)

Create Docker setup with docker-compose for easy deployment, provide Kubernetes manifests for container orchestration, implement graceful shutdown handling, add health checks with separate readiness and liveness probes, and implement comprehensive configuration management.

Remember: Only work on bonus features if core functionality is excellent. Quality over quantity!

11. Frequently Asked Questions

General Questions

Q: Can I use external libraries?

A: Yes, but exercise good judgment. Standard libraries and common utilities are appropriate. Avoid using pre-built order matching engine libraries as that defeats the assignment's purpose. You should implement the core matching logic yourself. For Go, libraries like net/http, encoding/json, sync primitives, and testing libraries are acceptable, as are third-party routing and logging libraries. For Rust, tokio, serde, axum or actix-web, and testing crates are fine. Focus on your implementation rather than showcasing framework knowledge.

Q: How strictly should I follow the API spec?

A: The API specification is a contract that you must follow exactly. Your endpoints should match the specification precisely, though response formats may include additional fields beyond those required.

Q: What if I can't meet all performance requirements?

A: Submit what you have with documentation explaining what you achieved, identifying the bottleneck, and describing what you would do to improve performance. Partial credit is awarded for demonstrating understanding and sound approach.

Q: Can I use a database?

A: While you can use a database, doing so will likely hurt performance. Most high-performance matching engines maintain all state in memory for speed. If you choose to use a database, you must still meet performance requirements.

Q: Should this run in production?

A: No, this is a simplified version. Real matching engines include many features we are not asking for such as clearing, settlement, and regulatory compliance. Focus on the core technical challenges we have outlined.

Technical Questions

Q: How should I handle price precision?

A: Handle price precision using integer arithmetic exclusively. Store prices as the smallest meaningful unit such as cents for dollar-denominated assets or satoshis for bitcoin. Multiply by 100 or 10000 depending on required precision. Never use floating-point arithmetic for monetary calculations as it

introduces precision errors.

Example:

```
// Good
type Price int64  // Price in cents
price := Price(10050)  // $100.50

// Bad
type Price float64
price := 100.50  // Precision issues!
```

Q: What about self-trading?

A: For this assignment, self-trading is allowed, meaning the same client can match their own orders. Preventing self-trading earns bonus points but is not required.

Q: How do I handle order IDs?

A: Generate order IDs server-side rather than accepting them from clients. Use UUIDs or any guaranteed-unique scheme.

Q: Should cancellation be synchronous?

A: Yes. Cancellation should be synchronous, meaning when the cancel endpoint returns, the order must be cancelled and unable to match future incoming orders.

Q: What if order book is empty for market order?

A: If the order book is empty when a market order arrives, reject the order with a 400 Bad Request status and an appropriate error message such as 'Insufficient liquidity'.

Q: Partial fills - does the order stay in the book?

A: Yes. When limit orders are partially filled, the remaining quantity stays in the order book at the original price and timestamp. The partially filled order maintains its time priority at that price level.

Q: What symbols should I support?

A: Your system should support any string as a trading symbol. We will test with common stock tickers such as AAPL, GOOGL, MSFT, TSLA, and AMZN, but you should design to handle arbitrary symbol strings.

Performance Questions

Q: What hardware will you test on?

A: We test on standard development machines with four-core Intel or AMD CPUs, 16GB of RAM, and SSD storage. We will note your machine specifications in your performance report and normalize for hardware differences when comparing submissions.

Q: Can I use all CPU cores?

A: Yes, you may use as many CPU cores as you want since real-world systems exploit multiple cores. Document your approach to parallelism.

Q: What about memory usage?

A: There is no hard memory limit, but usage should be reasonable, typically under 2GB for normal load. Measure and document your actual memory consumption.

Q: How do you measure latency?

A: We measure latency as the complete round-trip time from sending the HTTP request to receiving the full response, measured client-side with high-precision timers. Since tests run on localhost, network latency is minimal at typically less than 1 millisecond, meaning your server processing time dominates the measurement.

Q: Does latency include network time?

A: Yes, but tests run on localhost so network latency is minimal (<1ms). Your server processing time dominates.

Q: What if performance varies widely?

A: If performance varies widely between runs, we focus on tail latencies at the p99 and p999 percentiles. A few slow requests are acceptable, but consistent performance matters. Document any variance you observe and explain potential causes such as garbage collection pauses or OS scheduling effects.

Scope Questions

Q: 8 hours seems short for all this!

A: Eight hours may seem short, but you do not need to implement everything perfectly. We expect a working core matching engine with correct logic, basic API functionality, some performance optimization, and basic tests and documentation. Avoid over-engineering. We are testing fundamentals rather than completeness.

Q: What should I prioritize?

A: Prioritize correctness first since wrong answers disqualify your submission. Next, focus on core functionality including limit and market orders, matching logic, and basic API operations. Achieve

reasonable performance that meets minimum requirements. Maintain code quality with clean, understandable code. Provide good documentation explaining your decisions. Only if time permits should you add bonus features.

Q: Can I exceed 8 hours?

A: You may technically exceed eight hours, but we evaluate what you can accomplish in a single working day. If you spend sixteen hours, we will judge accordingly. Quality matters more than quantity.

Q: Is this a trick question?

A: No. This is not a trick question. This is a genuinely hard problem requiring solid engineering. We want to see how you approach complex technical challenges under realistic time constraints.

Submission Questions

Q: Can I submit early?

A: Yes! You may submit early if you finish in six hours with excellent results, which would be impressive.

Q: What if I need clarification during the assignment?

A: If you need clarification during the assignment, email us with your questions and we will respond within two hours during business hours.

Q: Can I ask for deadline extension?

A: In exceptional circumstances requiring deadline extensions, email us explaining your situation and we will evaluate case-by-case.

Q: Will I get feedback if I don't pass?

A: Yes. We provide feedback to all candidates who submit, regardless of outcome.

Q: What happens after submission?

A: After submission, we review your work within two to three days. If you pass our evaluation, we schedule a technical interview lasting thirty to sixty minutes to discuss your solution. The interview covers your design decisions, trade-offs you considered, how you would scale the system, and what you would change given more time or different constraints.

12. Resources

Recommended Reading

To understand order matching concepts, research how trading engines work, study FIFO matching algorithms, and understand price-time priority rules. For performance engineering, consider reading 'Designing Data-Intensive Applications' Chapter 1 and 'Systems Performance' by Brendan Gregg, along with Go performance tips or the Rust performance book depending on your language choice. For concurrency, Go developers should read 'Concurrency in Go' by Katherine Cox-Buday, while Rust developers should study 'Rust Atomics and Locks' by Mara Bos.

Recommended Tools

Go Developers:

- `pprof` for CPU/memory profiling
- `-race` flag for race detection
- `benchstat` for benchmark comparison

Rust Developers:

- `cargo flamegraph` for profiling
- `cargo bench` for benchmarking
- `miri` for unsafe code checking (if using unsafe)

Load Testing:

- `wrk` - HTTP benchmarking tool
- `vegeta` - HTTP load testing
- `k6` - Modern load testing tool

Monitoring:

- `htop` / `top` for resource monitoring
- `perf` for Linux profiling
- Time/profiling built into your code

Example Commands

Go profiling:

```
go test -bench=. -cpuprofile=cpu.prof
go tool pprof cpu.prof
```

Go race detection:

```
go test -race ./...
go run -race main.go
```

Rust profiling:

```
cargo install flamegraph
cargo flamegraph
```

Rust benchmarking:

```
cargo bench
```

Load testing:

```
wrk -t12 -c400 -d30s http://localhost:8080/api/v1/orders
```

Final Notes

This assignment is intentionally challenging. We do not expect perfection. We are looking for problem-solving ability that demonstrates how you approach difficult problems, engineering judgment reflected in the trade-offs you make, code quality showing you can write clean and maintainable code under pressure, performance thinking demonstrating you understand what makes systems fast, and communication skills shown through your ability to explain your decisions clearly.

Tips for Success

Start simple by getting basic functionality working first before adding complexity. Test early rather than waiting until the end, as this helps catch issues when they are easier to fix. Measure rather than guess when optimizing, using profiling to identify actual bottlenecks rather than optimizing based on intuition. Document as you go rather than leaving all documentation for the end when you may be rushed. Manage your time carefully, reserving adequate time for testing and documentation. If something is unclear, ask questions rather than making assumptions.

Suggested Time Allocation

Consider spending thirty minutes on planning and setup to think through your approach and set up your development environment. Allocate approximately four hours to core implementation of matching logic, order book, and basic API. Spend two hours on performance optimization once basic functionality works. Reserve one hour for testing including writing tests and benchmarks. Finally, allocate thirty minutes for documentation including README, ARCHITECTURE, and PERFORMANCE files. Adjust these allocations based on your strengths, but ensure you leave time for all phases.

Remember that we are rooting for you. This is your opportunity to demonstrate your capabilities. Good luck, and we look forward to seeing what you build!

Questions? Email naman@repello.ai

Submission Deadline: 24 hrs

Good luck! We're excited to see what you build.