High-Frequency Trading (HFT) Engine Simulation

A Project Report by
GAJENDER
Email: mandiwalgajender0001@gmail.com

October 1, 2025

1 Introduction

High-Frequency Trading (HFT) involves executing large volumes of trades at extremely high speeds (microseconds). This requires specialized engines with:

- Ultra-low latency
- High throughput (millions of orders/trades per second)
- Efficient memory and CPU usage
- Deterministic behavior for consistency

This project simulates a lightweight HFT engine in **C++17**, focusing on optimized order matching, high-volume handling, and key engine features.

2 Problem Statement

The goal is to develop a C++ simulation that:

- Supports **Limit**, **Market**, **IOC**, and **GFD** orders
- Matches buy and sell orders with minimal latency
- Handles high-volume order events
- Maintains order lifecycle (**add**, **cancel**, **replace**)
- Provides throughput and trade statistics

Note: This is a simulation; it does not connect to a live exchange.

3 Architecture and Workflow

3.1 Components

- **Order Book:** Stores bids and asks in **tick-indexed price levels** using **ring buffers** for constant-time insert/remove.
- **Order Pool:** Preallocated memory pool for O(1) allocation and cancellation.
- **Matching Engine:** Core logic executes trades when buy $\geq$ sell. Supports market sweeps.
- **Time-In-Force (TIF):**

  - **GFD:** Good-for-day orders
  - **IOC:** Immediate-Or-Cancel
- **Trade Logger:** Records trades with timestamp, price, and quantity.
- **Workload Generator:** Simulates market activity to stress-test the engine.

3.2 Workflow

1. **Order Arrival:** Orders submitted, validated, timestamped.
2. **Order Matching:** Market orders sweep the book; limit orders match at acceptable prices.
3. **Trade Execution:** Each match generates a trade event logged in-memory.
4. **Order Lifecycle:** Supports cancel/replace efficiently using preallocated pool.
5. **Performance Monitoring:** Measures throughput, latency, and trade stats.

4 Real-World Relevance

- Production HFT engines run on bare-metal servers with kernel bypass for low-latency I/O.
- Symbol sharding and risk checks are standard.
- Trade logging at nanosecond resolution ensures compliance.

This simulation models **core matching logic** and can be extended to study system performance.

5 Example Use Cases

- **Quant Research:** Benchmark new strategies against synthetic order flows.
- **Systems Research:** Test data structures (ring buffers vs. linked lists) for latency.
- **Education/Training:** Demonstrate engine mechanics to developers/students.
- **Stress Testing:** Simulate high-volume market conditions.

6 Key Challenges and Solutions

Challenge	**Simulation Solution**
High allocation latency	Preallocated OrderPool, no dynamic allocation in hot path
Cancel/Replace efficiency	Direct clientID $\rightarrow$ engineID mapping
Scalability	Symbol-level sharding (future extension)
Trade logging overhead	Lightweight in-memory logging
Book overflow	Fixed-size ring buffers per price level

7 Extensions and Future Work

- **O(1) Cancels:** Use position indices for instant removal.
- **Advanced Orders:** FOK, hidden, iceberg orders.
- **Concurrency:** Multi-threaded ingestion with lock-free queues.
- **Network Integration:** Simulate exchange connections via UDP/FIX.
- **Persistence and Replay:** Binary logs for deterministic backtesting.
- **Latency Profiling:** High-resolution measurements (p99/p999).

8 Conclusion

This simulation provides a **core HFT matching engine** in C++ with low-latency design, preallocation, efficient data structures, and support for common order types. It serves as a foundation for strategy testing, system research, and learning real-world HFT principles.