# **Project Requirements Report**

**Project Title:** *Latency-Aware Limit Order Book Simulator with Execution Strategy Benchmarking***Timeline:** 2 Weeks  
**Developer:** Salih Mert Uzun

## **1. Project Overview**

This project aims to design and implement a **high-performance limit order book (LOB) simulator in C++**, with a **Python interface for strategy research**, and a **latency modeling module** to evaluate how execution delays affect profitability.

The simulator will:

* Replay historical or synthetic market data.
* Allow strategy agents to submit/cancel orders in real time.
* Benchmark trading strategies under varying latency conditions.
* Produce performance metrics (PnL, Sharpe ratio, fill ratio, risk statistics).

This project demonstrates **quantitative trading system engineering** skills, **hybrid Python/C++ development**, and **market microstructure knowledge**, making it highly relevant for quant internships.

## **2. Functional Requirements**

### **2.1 Core Limit Order Book**

* Maintain two sides: **bids** (buy) and **asks** (sell).
* Data structure:  
  + std::map<price, std::list<Order>> for each side.
  + Each Order stores:  
    - id (unique identifier)
    - price
    - size
    - side (buy/sell)
    - timestamp
* Support operations:  
  + add\_limit\_order(order)
  + add\_market\_order(order)
  + cancel\_order(order\_id)
  + modify\_order(order\_id, new\_size)
* Matching rules:  
  + FIFO at each price level.
  + Market orders consume from best price until filled or book empty.
  + Limit orders that cross the spread execute immediately.

### **2.2 Latency Modeling**

* Orders and cancels experience configurable delay (0µs, 10µs, 100µs, 1ms).
* Simulator maintains an **event queue** where pending actions are stored until their latency expires.
* Measure:  
  + Fill probability at different latencies.
  + PnL degradation vs latency.

### **2.3 Strategy Layer (Python via C++ bindings)**

* Use **pybind11** for Python interface.
* Strategies implemented in Python:  
  1. **Market Making**: place bid/ask spread around mid-price.
  2. **Momentum Sniping**: buy/sell aggressively on price jumps.
  3. **TWAP**: split large order evenly over time.

Python API example:  
  
 book.add\_limit\_order("BUY", price=100.5, size=10)

book.step() # advance one event

### **2.4 Market Data Replay**

* Input formats:  
  + **Historical LOBSTER dataset** (academic, Level 2 order flow).
  + **Binance API snapshot + incremental updates** (crypto, free).
  + **Synthetic generator** (randomized arrivals, for testing).
* Simulator replays events in **event-driven loop**.

### **2.5 Metrics & Reporting**

* Compute per-strategy:  
  + **PnL** (profit & loss)
  + **Sharpe ratio** (risk-adjusted return)
  + **Fill ratio** (% of orders executed)
  + **Max drawdown**
  + **Inventory exposure**
* Generate plots:  
  + Latency vs PnL
  + Latency vs Fill Ratio
  + Distribution of trade outcomes

## **3. Non-Functional Requirements**

* **Performance:** Must handle **≥ 1 million order book events/sec** on a modern laptop.
* **Extensibility:** Easy to add new strategies.
* **Portability:** Buildable via CMake, bindings installable via pip.
* **Testing:** Unit tests for order matching, cancels, latency.

## **4. Deliverables**

* **Codebase:**
  + cpp/OrderBook.cpp, .h (engine)
  + cpp/LatencyModule.cpp
  + bindings/pybind.cpp
  + python/strategies.py
  + python/run\_experiment.py
* **Documentation:**
  + README with build/run instructions
  + API usage examples
* **Report (PDF or blog):**
  + Methods, experiments, graphs, conclusions

## **5. Timeline (2 Weeks)**

**Week 1**

* Day 1–2: Implement Order struct, book data structures, matching logic.
* Day 3–4: Add support for cancels, modifies, market orders.
* Day 5–7: Benchmark replay with synthetic data.

**Week 2**

* Day 8–9: Add latency module (event queue).
* Day 10–11: Bind to Python (pybind11).
* Day 12: Implement strategies in Python.
* Day 13: Run latency experiments, collect metrics.
* Day 14: Prepare report & graphs.

## **6. Future Extensions (Stretch Goals)**

* **Adversarial agents**: competing bots that front-run or cancel aggressively.
* **Reinforcement learning strategy**: RL agent adapts spreads dynamically.
* **Parallelization**: multi-threaded matching for higher throughput.

## **7. Success Criteria**

* Replay at least **5M events** in under 10 seconds.
* Demonstrate measurable **PnL degradation at higher latency**.
* Deliver a professional repo + report suitable for CV/portfolio.