Latency Benchmarking Report

Abhiraj Kumar

May 18, 2025

1. Introduction

This report documents the benchmarking results of latency metrics collected from the real-time trade simulator implemented for the GoQuant internship assignment. These metrics were collected during simulation runs and help evaluate the system's real-time performance under typical operating conditions.

2. Measured Latency Metrics

The following latency benchmarks were recorded:

- **Data Processing Latency:** 1.861 ms
- **UI Update Latency:** 2710.648 ms
- **End-to-End Simulation Loop Latency:** 2710.894 ms

3. Methodology for Measuring Latency

To measure the three latency metrics, we embedded timing code in the main application script `main.py`. The timings were captured using Python's `time.perf_counter()` function. The code snippet below shows the approach used:

```
import time

start_time = time.perf_counter()
# Step 1: Fetch and process orderbook data
process_start = time.perf_counter()
processed_data = process_orderbook(orderbook_data)
process_end = time.perf_counter()

# Step 2: Update UI
ui_start = time.perf_counter()
update_ui(processed_data)
ui_end = time.perf_counter()
```

```
end_time = time.perf_counter()

print(f"[Latency] Data Processing Latency: {(process_end - process_start)*1000:.3f} ms
print(f"[Latency] UI Update Latency: {(ui_end - ui_start)*1000:.3f} ms")
print(f"[Latency] End-to-End Loop Latency: {(end_time - start_time)*1000:.3f} ms")
```

4. Analysis of Benchmarking Results

4.1 Data Processing Latency

The data processing latency is the time taken to parse and prepare L2 orderbook data, compute metrics such as spread, depth, and imbalance, and apply feature engineering for slippage and market impact estimation.

This latency is relatively low (1.861 ms), indicating that the orderbook data parsing and feature extraction operations are efficient. The use of optimized data structures such as NumPy arrays and vectorized operations helped reduce this overhead significantly.

4.2 UI Update Latency

The UI update latency is significantly higher (2710.648 ms), and this is primarily due to the overhead introduced by Streamlit when rendering complex visual components and updating plots or interactive widgets in real time.

Streamlit operates on a synchronous model and may re-render multiple components on each interaction or data refresh, which introduces notable latency.

4.3 End-to-End Simulation Loop Latency

The end-to-end loop latency (2710.894 ms) includes the full duration from receiving data, processing it, updating the UI, and performing all intermediate calculations like slippage and market impact.

4.4 Root Cause: Market Impact Function Complexity

The core reason for the UI and end-to-end latency being higher is the complexity of the market impact model implementation. The Almgren–Chriss model, combined with live orderbook-based slippage estimation and maker/taker classification, involves matrix operations, regression inference, and volatility computations, all of which are computationally intensive.

In addition, these analytics are run in the same thread as the UI, which amplifies the latency perceived at the interface layer.

5. Recommendations

To further reduce UI and loop latency, the following improvements are proposed:

- Move computationally heavy tasks like market impact calculation to a background thread or asynchronous worker.
- Limit Streamlit's UI update frequency using caching or conditional re-renders.
- Simplify visual components or reduce the refresh rate for non-critical metrics.

6. Conclusion

The system achieves low latency for data processing, but UI rendering remains a bottleneck due to synchronous updates and compute-bound model evaluation. Future optimizations can target asynchronous execution and more efficient UI rendering to improve end-to-end performance.