A Proposed Backtesting Framework for SOR

Hongyi (Harry) Wang

January 9, 2025

Part 1: Methodology and Framework

1 Introduction

Smart Order Routing (SOR) optimizes trade execution by breaking and distributing orders into multiple parts and multiple trading venues to achieve optimal execution results. A robust backtesting framework is crucial for evaluating SOR strategies before live deployment. This document proposes a backtesting methodology that covers the following:

- Data generation & handling,
- SOR execution strategies,
- Performance metrics,
- Simulation logic,
- System extensibility.

2 Data Pipeline

2.1 Data Sourcing

1. **Market Data:** Historical limit order book (LOB) data, including time-stamped bids, asks, and trades from multiple venues. For simplicity, I'm modeling prices using a random walk and volume with the Poisson Distribution. Realistically, data sources may include:
    - Direct exchange feeds (e.g. NASDAQ TotalView, NYSE OpenBook),
    - Consolidated feeds (e.g. consolidated tape for equities),
    - Third-party data vendors offering aggregated or raw LOB snapshots.
2. **Order Flow Data:** Historical order placement and execution details. This is crucial for simulating realistic behavior and measuring slippage or partial fills. For simplicity, I assume in my implementation that each level of the order book has the same volume, and that the volume is spread 50-50 between bids and asks.

2.2 Data Processing

1. **Resampling & Synchronization:**
    - Align bid/ask updates and trade ticks across multiple venues on a common timeline. For simplicity, I'm ignoring this feature in my implementation.
    - Handle different data frequencies (e.g. top-of-book vs. full depth-of-book, 1-second bars vs. tick-by-tick).
2. **Data Cleaning:**

    - Use forward-fill or backward-fill methods for missing bid/ask prices to maintain consistency in market snapshots.
    - For large gaps, use interpolation techniques or exclude affected intervals to avoid introducing bias.
    - Verify data integrity (e.g. check for out-of-order timestamps and reorder data where necessary to preserve temporal accuracy; validate that bids are always lower than asks).

3. **Data Storage & Access:**
    - Use a time-series database or columnar storage for efficient slicing of relevant time windows. I'm using a Pandas dataframe in my implementation, again for simplicity.
    - Preferably, maintain an interface for on-demand retrieval of LOB snapshots at any given timestamp.

2.3 Data Handling Challenges

- **Missing Data:** Implement interpolation or carry-forward last known best bid/ask if updates are missing.
- **Latency & Out-of-Sync Feeds:** Use event-time or exchange timestamps to re-sequence data. For instance, consider FIFO ordering for same-timestamp events.
- **Complexity:** Multi-venue data can grow large; efficient I/O and caching are essential for scalable backtesting. Again such subtleties are ignored in this sample implementation.

3 Execution Strategies

3.1 TWAP (Time-Weighted Average Price)

- **Definition:** Divides the total order into equal parts, executed evenly over a specified time window.
- **Mechanics:**
    - Split order size into chunks.
    - Schedule orders at fixed time intervals that are integral multiples of a timestep (e.g. every minute).
    - Optionally place limit or market orders depending on aggressiveness.
- **Sensitivity to Market Conditions:**
    - In highly volatile markets, static time-slicing may lead to higher slippage.
    - Rapid price movement can render fixed-interval orders less optimal.
    - The empirical results are discussed in detail in the report.

3.2 VWAP (Volume-Weighted Average Price)

- **Definition:** Executes in proportion to observed trading volume.
- **Mechanics:**
    - Estimate volume distribution for the day (historical or real-time on-the-fly).
    - Allocate slices of the total order relative to each time interval's forecasted volume.
- **Sensitivity to Market Conditions:**
    - Low liquidity periods may lead to partial fills or higher spreads.
    - Sudden spikes in volume can cause overshooting or undershooting the target (well reflected in the results).

3.3 Other Considerations

- **Adaptive SOR:** Dynamically adjust slice size or venue preference based on real-time market signals (liquidity, spreads, volume).
- **Market vs Limit Orders:** Balance immediate execution (market) against better pricing (limit).

4 Performance Metrics

1. **Execution Cost:** Difference between executed prices and a price benchmark (e.g. VWAP, mid-price at order submission).
2. **Slippage:** Difference between expected and actual execution prices, computed as a volume weighted average.
3. **Fill Rate / Fill Probability:** Ratio of shares/contracts filled vs. total order size.
4. **Implementation Shortfall (IS):** Total cost of completing a trade relative to decision time price.
5. **Latency / Execution Time:** Time from order release to fill confirmation.

5 Simulation Logic

In my implementation, the backtesting loop is orchestrated by a dedicated `Backtester` class. It uses the following steps on each simulation timestep:

- **Market Environment State:** A `MarketEnvironment` class generates time-series data (bid/ask prices, volumes) via a synthetic random walk. On any timestep, we can retrieve the current bid/ask snapshot and volumes.
- **Strategy Interaction:** The `Strategy` interface (and its TWAP/VWAP variants) determines how many shares to trade. Each step, the strategy's `generate_orders` method is called with current market data.
- **Partial-Fill Logic:** The `Backtester` then simulates actual fills. For a `BUY`, we calculate a naive partial-fill price impact by consuming as much volume as possible at the current ask, then moving to the next price level if still unfilled. Conversely, for a `SELL`, we deplete the bid volumes and move down if more shares remain. This is handled in private helper methods `_simulate_buy()` and `_simulate_sell()`.
- **Record-Keeping:** Each executed portion of the order is stored (price, volume, timestamp) to compute metrics later. The environment is then advanced by one timestep.
- **Metrics:** After all time steps are exhausted, the `Backtester` aggregates metrics (slippage, average fill price, etc.) and plots execution activity against the synthetic price series.

This framework is flexible for plugging in different strategies and can be extended to handle more sophisticated partial-fill models or multi-venue logic.

6 Extensibility and Scalability

In my codebase, the `Strategy` abstraction is designed so that new execution methods can be quickly implemented, while the `Backtester` remains unchanged. Likewise, the `MarketEnvironment` can be swapped out for real historical data or a more detailed simulation engine. Some additional points include:

- **Multi-Leg Strategies:** The approach can be adapted to multi-leg derivatives. We can add additional classes or modify the order fill logic if required for correlated instruments.
- **Parallelization:** Running many Monte Carlo simulations is trivial by distributing multiple Backtester instances across CPU threads or compute nodes.
- **Plugin-Based Architecture:** By registering strategies or loading them from external modules, it is easy to integrate new SOR strategies without disrupting the core backtesting loop.