Methodology and Framework

Namit Shrivastava
January 6, 2025

Introduction

Smart Order Router (SOR) is a system that automatically routes orders to various trading venues to achieve the best possible execution price and speed. This involves considering factors such as price, liquidity, and transaction costs. So, backtesting is important for validating and refining an SOR before using it in live trading.

Data Pipelining

So firstly, for getting the real data, I believe sourcing the market data can be done through direct exchange feeds, APIs, or snapshots of order books. After that, any missing data or out-of-order timestamps needs to be carefully handled.
For controlled tests I think randomly generated time series of bid/ask quotes and volumes can help in stress-test edge cases and market regimes, if say, synthetic data was to be used. So the components can be:

- ❖ **Price Data:** This will have the simulated price movements, including timestamps, to model market changes.
- ❖ **Volume Data:** Simply put, this shows the synthetic volumes to mimic trading activity and liquidity at different price levels.
- ❖ **Timestamps:** The code will show accurate timestamps to provide the time element that is critical for order execution simulation.

Preprocessing

Now after collecting the raw data, I must point out some of the challenges that include missing or delayed quotations, incomplete fills and cancellations, and data volume (tick-level data may have a high volume). As a result, I would have to interpolate or eliminate missing records based on their severity, then align all data streams (including quotations, trades, and reference rates) with a set timeline, and annotate each time point with market microstructure characteristics (like the best bid, best ask, and last trade price).

Execution Strategies

Now my trading system implements several key execution strategies, with TWAP (Time-Weighted Average Price) serving as the foundation. So I thought of TWAP as like breaking down a large grocery shopping list into smaller trips throughout the day to avoid emptying the store's shelves all at once. For instance, instead of buying 10,000 shares in one go, TWAP would split this into 1,000 share purchases every hour over 10 hours. Hence, the system also handles multi-leg trades, similar to coordinating multiple shopping trips at different stores simultaneously. For example, a person might want to sell tech stocks while buying healthcare stocks as part of a sector rotation strategy. While researchers like myself start with simple market orders, the framework is built to handle more sophisticated order types like limit orders in the future.

Simulation Logic

I was quite intrigued with this framework's concept as it operates like a sophisticated delivery service, but instead of packages, it is now handling stock orders. I will try to imagine my approach to the simulation logic in very simple terms so that anyone can understand.
Just as a delivery company might choose between overnight air shipping (which is fast but expensive) or ground transportation (is slower but cheaper), sameway, the system intelligently routes stock orders across different exchanges. For example, when placing a 5,000 share order for Tesla stock, the system might notice that NYSE offers better prices for the first 3,000 shares while NASDAQ has better rates for the remaining 2,000 shares. Like a smart GPS system, it automatically splits and routes these orders to get the best overall execution price.

So, the simulation brings this virtual trading floor to life by carefully tracking how each order performs across these different venues. In layman's perspective, think of it like having a control room with multiple screens monitoring different delivery routes, meaning each shows real-time traffic (market liquidity), toll costs (exchange fees), and delivery times (execution speed). When a person places an order, the system considers all these factors simultaneously. For instance, if one needs to buy $1 million worth of Microsoft shares by the end of the day, the system might break this into smaller orders of $100,000 each, spacing them out every hour to avoid causing a sudden price spike.
Basically similar to how one would space out large deliveries to avoid overwhelming a single warehouse. This careful orchestration helps traders test their strategies safely before risking real money, much like how delivery companies run simulations to optimize their routes during peak seasons.

Performance Metrics

So based on the logic I stated before, the system actually tracks how well each trade performs by measuring two key metrics, mainly:

- **Slippage**, meaning how much the price moved between when one decides to trade and when the trade actually happened.
- **Execution costs**, signifying the total cost of making the trade, including fees and price impact. For example, if a person wanted to buy a stock at $100, but by the time that individual's order went through the price was $100.50, his/her slippage would be $0.50 per share.

Extensibility and Scalability

So again, I will be metaphorically explaining how this backtesting framework is built like a flexible Lego set for trading strategies, where each piece can be easily swapped or modified. At the very foundational level, it handles different types of trades much like a master chef managing multiple dishes in a kitchen. Some orders might be simple (like a market order to buy 100 shares), while others might be complex combinations (like simultaneously buying stocks and selling options). For instance, when executing a pairs trading strategy between Apple and Microsoft stocks, the system can coordinate both trades to happen together, just like ensuring both the main course and side dish arrive at the table simultaneously. The framework is smart enough to route these orders to different exchanges based on where they'll get the best execution, similar to a chef choosing different suppliers for different ingredients.

The system is also designed to work with various types of financial instruments, much like a universal remote or a phone with an IR blaster that can control different brands of TVs, sound systems, and streaming devices. Whether one is trading stocks, cryptocurrencies, or options, the framework adapts its behavior to match each market's unique characteristics.
For instance, it knows that cryptocurrency markets operate 24/7 with different volatility patterns compared to stock markets that have specific trading hours. The system adjusts its execution strategy accordingly, like how one might use different cooking techniques for different types of food. To handle all this efficiently, the framework processes multiple trades in parallel, similar to how a modern processor handles multiple tasks simultaneously, ensuring quick and efficient execution of even the most complex trading strategies!