GoQuant Assignment

Objective

Create a high-performance trade simulator leveraging real-time market data to estimate transaction costs and market impact.
This system will connect to provided WebSocket endpoints that stream full L2 orderbook data for cryptocurrency exchanges.

Important: You will need to use a VPN(using proton vpn) to access OKX for this assignment. Since the market data you need is public, there is no requirement to create an account.

Initial Setup

1. Review the API documentation for OKX SPOT exchange
2. Set up a development environment for either Python or C++ (your choice)

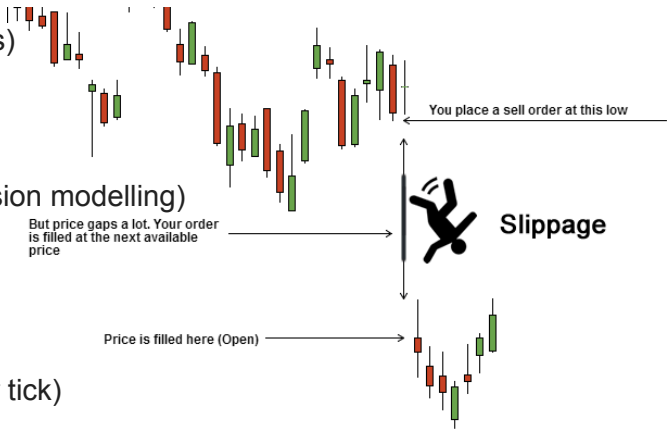
Core Requirements:

UI Components

1. Implement a user interface with:
   - Left panel: Input parameters section
   - Right panel: Processed output values section

Input Parameters

1. Exchange (OKX)
2. Spot Asset (Any available on the selected exchange)
3. Order Type (market)
4. Quantity (~100 USD equivalent)
5. Volatility (market parameter -- see exchanges' docs)
7. Fee Tier (based on exchange documentation)



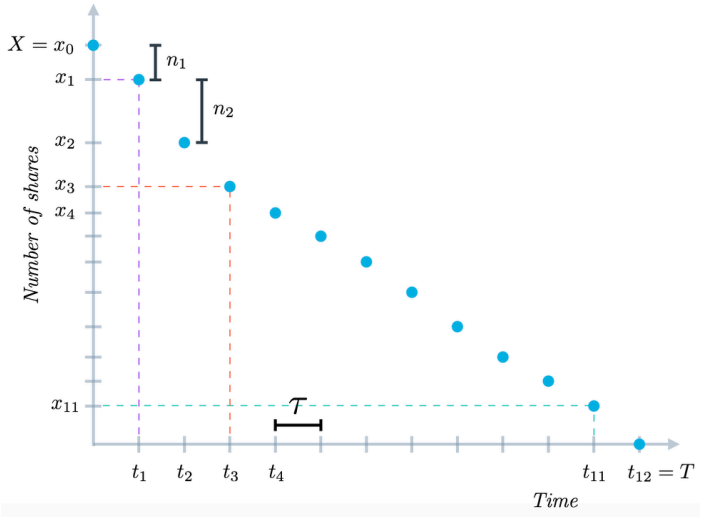
Output Parameters

1. Expected Slippage (using linear or quantile regression modelling)
2. Expected Fees (rule-based fee model)
3. Expected Market Impact (Almgren-Chriss model)
4. Net Cost (Slippage + Fees + Market Impact)
5. Maker/Taker proportion (logistic regression)
6. Internal Latency (measured as processing time per tick)

WebSocket Implementation

1. Connect to provided WebSocket endpoint:
2. Sample response format:

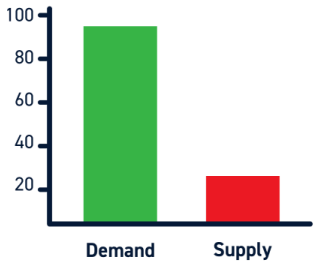
```
{
  "timestamp": "2025-05-04T10:39:13Z",
  "exchange": "OKX",
  "symbol": "BTC-USDT-SWAP",
  "asks": [
    ["95445.5", "9.06"],
    ["95448", "2.05"],
```



```
    // ... more ask levels ...
  ],
  "bids": [
    ["95445.4", "1104.23"],
    ["95445.3", "0.02"],
    // ... more bid levels ...
  ]
}
```

LEVEL 2 DATA					
EDGA	175.26	1	NASD	175.38	2
EDGX	175.25	1	BOST	175.38	1
NYSE	175.22	2	BATS	175.4	2
ARCA	175.20	1	NYSE	175.4	1
NASD	175.15	26	EDGX	175.45	2
BATS	175.15	5	ARCA	175.45	2
NSXX	175.1	45	PHLX	175.59	6
BYX	175.09	3	EDGA	175.61	3
PHLX	174.99	6	BYX	175.62	4

Demand Supply



3. Process L2 orderbook data in real-time
4. Update output parameters with each new tick

Technical Requirements

1. Implementation in either Python or C++ (your choice - I am doing it as web application)
2. System must process data faster than the stream is received
3. Include proper error handling and logging
4. Implement clean, maintainable code architecture
5. Provide documentation for models and algorithms used

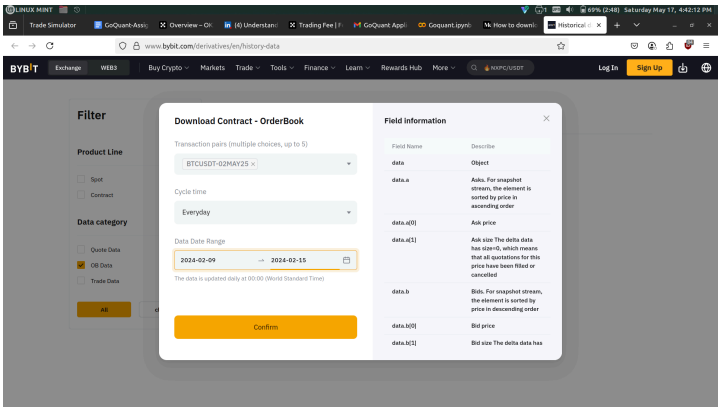
Bonus Section (recommended): Performance Analysis and Optimization

Latency Benchmarking

1. Measure and document the following metrics:
   - Data processing latency
   - UI update latency
   - End-to-end simulation loop latency

Optimization Requirements

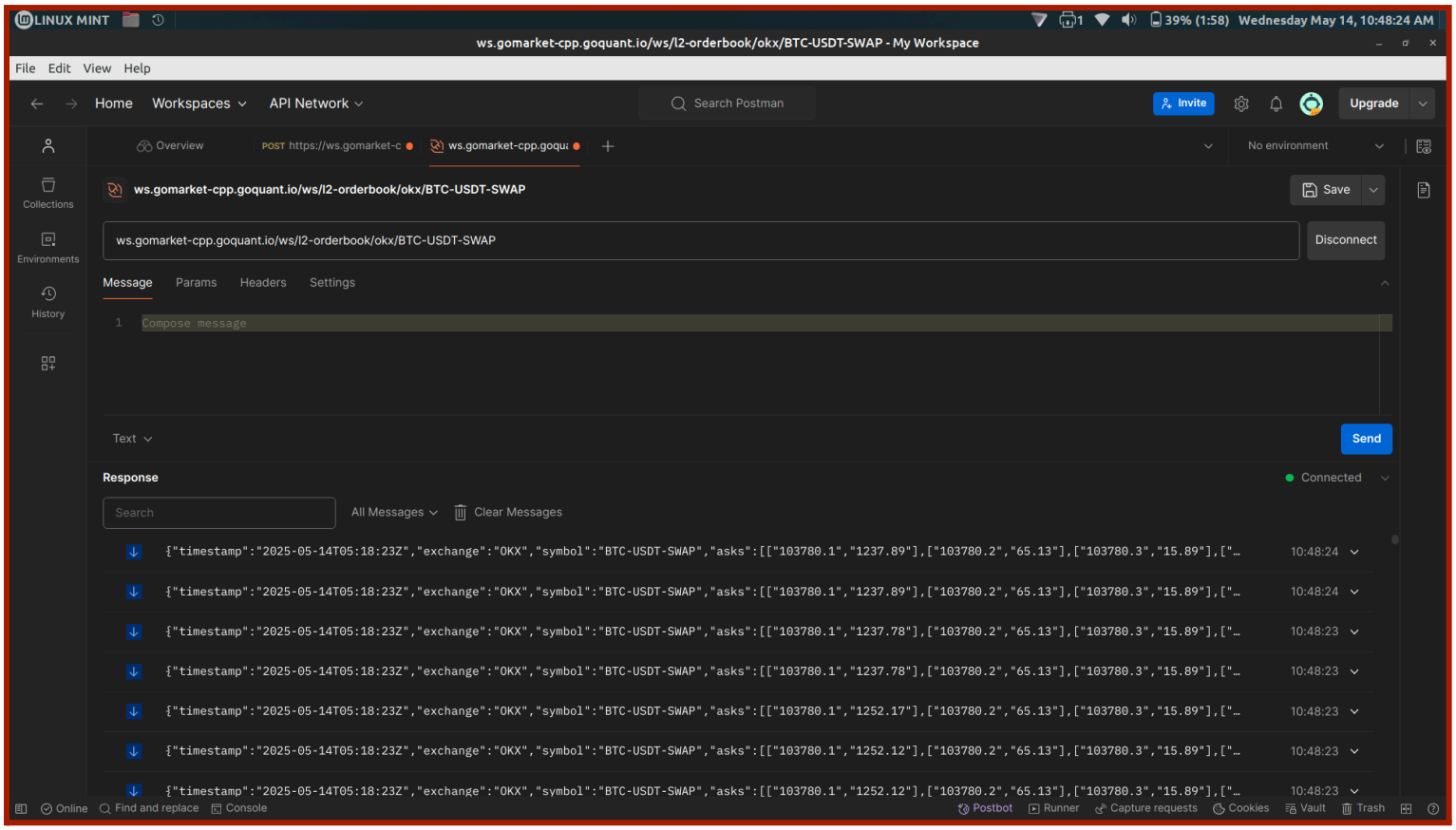
1. Implement and justify optimization techniques for:
   - Memory management
   - Network communication
   - Data structure selection
   - Thread management
   - Regression model efficiency



Model Implementation

1. Detailed implementation of:
   - Almgren-Chriss market impact model
   - Regression models for slippage estimation
   - Maker/Taker proportion prediction

+ I am using the ByBit L2 orderbook dataset. I have trained the model for this dataset but it can't be used here as it doesn't match the assignment requirement.



- OKX's V5 WebSocket channels batch and push updates at roughly **50 milliseconds** intervals before delivery.
- In addition, OKX has reported reducing its inter-cloud (data-center to data-center) network latency down to about **2 milliseconds** using optimized cloud networking.
- Thus if you run my application you will see that the processing time is even less than 2 ms
- Initial connections to the end point of websocket have some latency. But that is my laptop problem. As it is very far from OKX central exchange.
- OKX caps the number of simultaneous WebSocket connections *per channel* per **sub-account** at **30**.
- Each connection is tagged with a unique `connId`, and the server will notify you of your current usage and reject any subscription that would exceed the limit.

Full snapshot (initial state)	**Incremental update** (diffs)

{
"timestamp":"…",
"exchange":"OKX",
"symbol":"BTC-USDT-SWAP",
"snapshot": true,
"bids":[[price, size],…],
"asks":[[price, size],…]
}
	{
	"timestamp":"…",
	"exchange":"OKX",
	"symbol":"BTC-USDT-SWAP",
	"bids":[[price, newSize],…], ← only changed levels
	"asks":[[price, newSize],…]
	}

You'll need to keep a data structure (e.g. two maps or sorted containers) in your code to:

On snapshot	On each update
- Clear your local book - Insert every `[price → size]` for bids and asks	For each `[price, size]` in `bids` (and `asks`): - If `size == 0`, remove that price level - Else, set/update that level to the new size

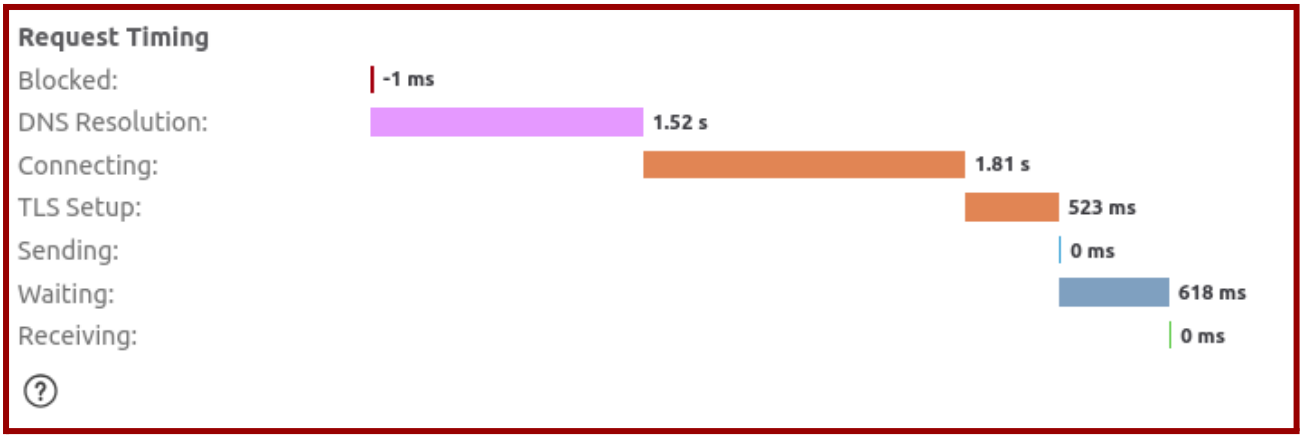
Next steps

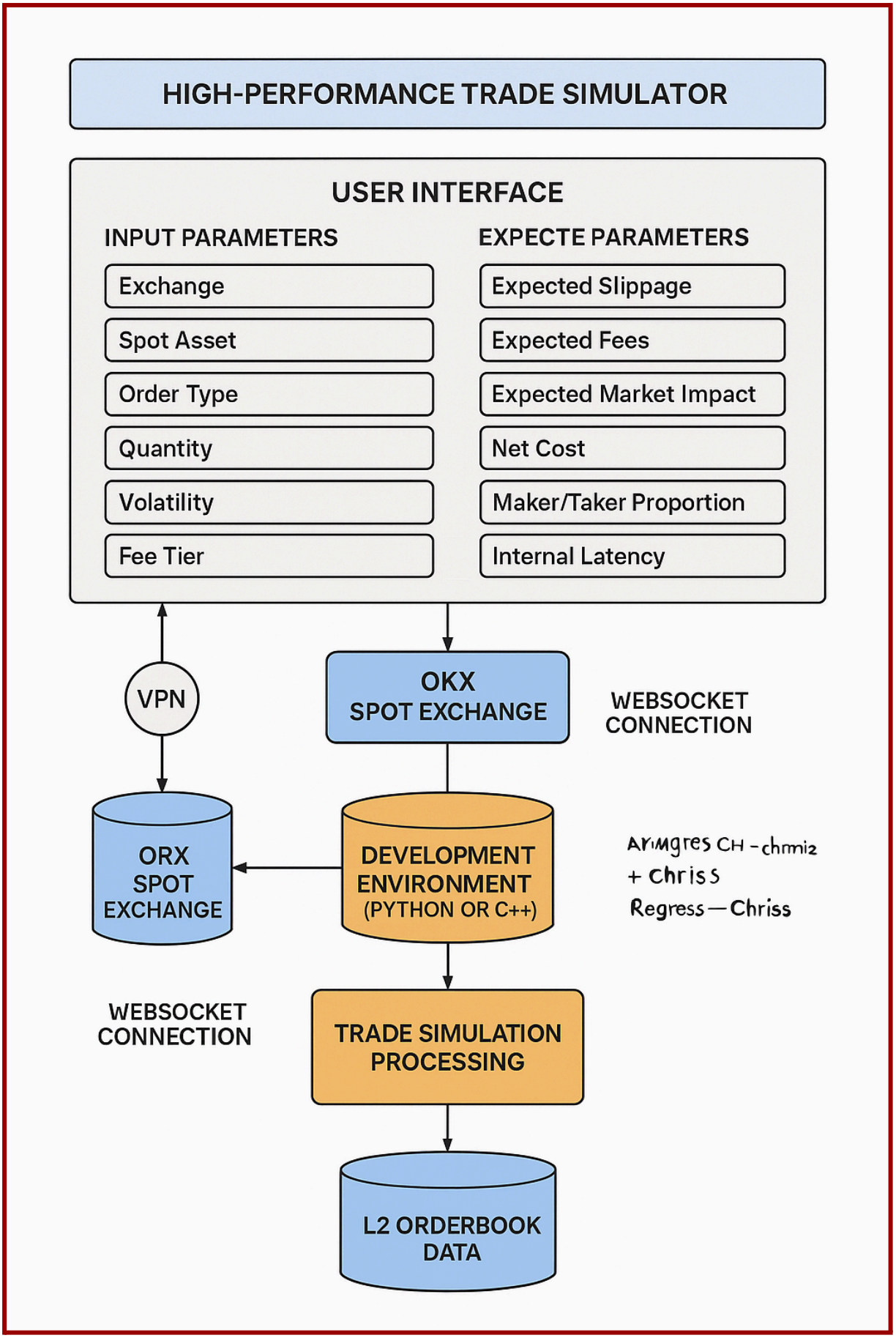
- **Validate your reconstructed book by occasionally comparing it to a REST snapshot.**
- **Compute metrics: mid-price, spread, depth at top levels, VWAP over N levels, etc.**
- **Feed this data into your trading strategy or charting UI.**

🔍 Model Summary & Training Plan

Model	Purpose	Input Features	Target / Label	Suggested Model Type
Slippage Estimation	Predict how far execution price drifts from mid-price	Order size, volatility, depth imbalance, spread	Slippage in basis points	Linear regression or Quantile regression
Market Impact (Almgren-Chriss)	Predict permanent/ temporary impact of trade	Order size, liquidity, volatility, trade horizon	Market impact cost	Analytical model (parameter estimation from historical data)
Maker/Taker Proportion	Estimate how much of the order gets filled passively	Spread, order size, depth, volatility	% filled as maker	Logistic regression
Latency Profiling (optional)	Estimate or profile internal processing time	Tick size, order size, system load	Processing time	Logging only (no ML)

Request for OKX L2 Order Book




HIGH-PERFORMANCE TRADE SIMULATOR
USER INTERFACE
INPUT PARAMETERS
EXPECTE PARAMETERS
Exchange
Spot Asset
Order Type
Quantity
Volatility
Fee Tier
Expected Slippage
Expected Fees
Expected Market Impact
Net Cost
Maker/Taker Proportion
Internal Latency
VPN
OKX
SPOT EXCHANGE
WEBSOCKET
CONNECTION
ORX
SPOT
EXCHANGE
DEVELOPMENT
ENVIRONMENT
(PYTHON OR C++)
Armgres CH - chmiz
+ Chriss
Regress — Chriss
WEBSOCKET
CONNECTION
TRADE SIMULATION
PROCESSING
L2 ORDERBOOK
DATA

1. Model Selection and Parameters

The simulator employs two primary regression models to estimate various trading costs and probabilities. These models are trained dynamically using features derived from the live order book data.

1.1 Slippage Prediction Model

- **Model Type:** sklearn.linear_model.LinearRegression (Linear Regression)
- **Purpose:** To predict the expected slippage a trade might incur. Slippage refers to the difference between the expected price of a trade and the price at which the trade is actually executed.
- **Features for Training:**
  The model is trained using historical data points, where features for each point i are:
  1. Spread: The difference between the best ask and best bid price at time i (spread_history[i]).
  2. Ask Volume: Total quantity available in the top 10 ask levels at time i (volume_history[i][0]).
  3. Bid Volume: Total quantity available in the top 10 bid levels at time i (volume_history[i][1]).
  4. Short-Term Volatility (Proxy): Calculated as the standard deviation of the mid-prices over the last 20 data points (or fewer if less data is available), divided by the mean mid-price over the same period. This serves as a localized, historical volatility measure.
     np.std(price_history[max(0, i-20):i+1]) / np.mean(price_history[max(0, i-20):i+1])
- **Target Variable for Training (y_slippage):**
  The target variable is a proxy for realized slippage. For each historical point i, it's calculated as the absolute relative difference between the mid-price at time i and the mid-price at the next time step i+1:
  + abs((price_history[i+1] - price_history[i]) / price_history[i])
  This simulates the slippage if a small market order were executed at price_history[i] and filled at price_history[i+1].
- **Features for Prediction (Live Calculation):**
  When calculating metrics for the UI, the model uses:
  1. Current Spread: The latest observed spread.
  2. Current Ask Volume: Sum of quantities in the current top 10 ask levels.
  3. Current Bid Volume: Sum of quantities in the current top 10 bid levels.
  4. User-Input Volatility: The volatility value entered by the user in the GUI. **Note:** This differs from the volatility proxy used during training.
- **Training:**
  The model is initially trained after 100 order book updates have been processed. Subsequently, it is retrained periodically (e.g., every 100 new updates) to adapt to changing market conditions.
- **Fallback:** If the model is not yet trained, a default slippage value of 0.001 (0.1%) is used.

1.2 Maker/Taker Probability Model

- **Model Type:** sklearn.linear_model.LogisticRegression (Logistic Regression)
- **Purpose:** To predict the probability of an order (or a portion of it) being executed as a "maker" order (adding liquidity) versus a "taker" order (removing liquidity). This distinction is important for fee calculation.
- **Features for Training:**
  The features are identical to those used for training the Slippage Prediction Model:
  1. Spread at time i.
  2. Ask Volume (top 10 levels) at time i.
  3. Bid Volume (top 10 levels) at time i.
  4. Short-Term Volatility (Proxy) at time i.
- **Target Variable for Training (y_maker_taker):**
  The target variable is a binary indicator (0 or 1). It's a heuristic based on order book imbalance:
  1. 1 (Maker): If Bid Volume at time i > Ask Volume at time i. The assumption is that a deeper bid side relative to the ask side might offer a better chance for a buy limit order to be filled (or a sell limit order to rest without immediate execution).
  2. 0 (Taker): Otherwise.
- **Features for Prediction (Live Calculation):**
  Similar to the slippage model, when calculating metrics live, it uses:
  1. Current Spread.
  2. Current Ask Volume (top 10 levels).
  3. Current Bid Volume (top 10 levels).
  4. User-Input Volatility.
- **Training:**
  Trained concurrently with the slippage model under the same conditions (initial training after 100 updates, periodic retraining).
- **Fallback:** If the model is not yet trained, a default maker probability of 0.5 (and thus taker probability of 0.5) is assumed.

2. Regression Techniques Chosen

The choice of regression techniques is guided by the nature of the prediction task and the need for relatively quick, dynamic retraining.

- **Linear Regression (for Slippage Prediction):**
  - **Rationale:** Linear Regression is chosen for its simplicity and interpretability. It aims to model a linear relationship between the input features (spread, volumes, volatility) and a continuous target variable (slippage). It serves as a good baseline for predicting a quantitative outcome.
  - **Suitability:** While actual slippage can be complex and non-linear, linear regression provides a first-order approximation that can be effective with carefully chosen features, especially in a real-time simulation context where complex model training might be too slow.

- **Logistic Regression (for Maker/Taker Probability):**
    - **Rationale:** Logistic Regression is a standard and effective algorithm for binary classification problems. It models the probability that an instance belongs to a particular class (in this case, the probability of an order being classified as "maker").
    - **Suitability:** It outputs probabilities (between 0 and 1), which is directly applicable for estimating the proportion of an order that might be filled as maker vs. taker, and subsequently for calculating blended fees.
- **General Considerations:**
    - Both models are computationally inexpensive to train and predict, making them suitable for an environment where they need to be periodically updated with new market data.
    - The "training data" is generated heuristically from observed order book dynamics rather than from actual trade execution data. This is a simplification inherent in the simulator's design.

3. Market Impact Calculation Methodology

Market impact refers to the effect an order has on the market price due to its size. The simulator uses a simplified version of the Almgren-Chriss model to estimate this.

- **Model:** Almgren-Chriss (Simplified)
- **Purpose:** To estimate the adverse price movement caused by the act of executing a trade. Larger orders consume more liquidity and are expected to have a greater market impact.
- **Formula Used in _calculate_almgren_chriss_impact:**
  Market Impact = volatility * sqrt(tau) * (quantity / market_volume) * gamma
- **Parameters:**
    - volatility: This is the **user-inputted volatility** from the GUI. It represents the expected annualized standard deviation of returns for the asset.
    - tau (Time Horizon): This parameter typically represents the duration over which the order is executed. In this implementation, tau is **fixed at 1.0** for simplicity. This implies an assumption of a standardized, short execution period.
    - quantity: The size of the order. In the context of the formula, this is treated as the quantity in base asset units that would exert pressure on the market. The GUI input is in USD, but it's used directly here, implicitly assuming market_volume is also in comparable units or that the ratio is meaningful.
    - market_volume: An estimate of the available liquidity. In the code, this is calculated as the **sum of quantities (sizes) in the top 10 levels of the current bid book and the top 10 levels of the current ask book**.
      market_volume = sum(size for _, size in self.current_asks[:10]) + sum(size for _, size in self.current_bids[:10])
      If market_volume is zero, it's set to 1 to avoid division by zero.
    - gamma (Market Impact Factor): This is an empirical constant that calibrates the model's sensitivity. It is **fixed at 0.5** in the code. In practice, gamma would be estimated from historical trade data.
- **Implementation Notes:**

- The market impact is calculated based on the current snapshot of the order book (for market_volume) and user-defined parameters (quantity, volatility).
- The result is a fractional value representing the expected price impact relative to the current price (e.g., 0.0005 means 0.05% impact).

4. Performance Optimization Approaches

Several strategies are employed to ensure the application remains responsive and handles real-time data efficiently:

1. **Asynchronous WebSocket Handling:**
    - **asyncio and websockets:** The application uses the asyncio library along with the websockets client for handling the WebSocket connection non-blockingly. This allows the program to receive and process data efficiently without halting other operations.
    - **Dedicated QThread:** The WebSocket communication and initial data processing logic are encapsulated within WebSocketThread, a custom class inheriting from QThread. This moves all network I/O and intensive processing off the main GUI thread, preventing the user interface from freezing.
    - **Robust Connection Management:**
        - **Automatic Reconnection:** If the WebSocket connection is lost (due to network issues, server-side disconnects, or timeouts), the WebSocketThread attempts to reconnect automatically.
        - **Exponential Backoff:** Reconnection attempts use an exponential backoff strategy (reconnect_delay starts at 1s and doubles, capped at max_reconnect_delay), preventing server overload from rapid retries.
        - **Maximum Retries:** A limit (max_reconnect_attempts) is set on reconnection cycles to avoid indefinite attempts.
        - **Receive Timeout:** asyncio.wait_for(websocket.recv(), timeout=5.0) is used. If no message is received within 5 seconds, it's treated as a timeout, prompting a reconnection. This helps detect stale or unresponsive connections more proactively than relying solely on TCP keepalives or WebSocket pings (which are disabled by ping_interval=None to allow this custom timeout logic).
2. **Efficient Data Processing and Management:**
    - **Limited History Buffers:**
        - OrderbookProcessor maintains historical lists for prices, spreads, and volumes (price_history, spread_history, volume_history, orderbook_history). These lists are capped at a maximum size (e.g., 1000 entries for orderbook_history) by removing the oldest entries when new ones are added. This prevents unbounded memory consumption.
        - The (currently unused) OrderBook class also has a MAX_HISTORY_SIZE constant for its history arrays.
    - **Order Book Depth Pruning:**
        - While the WebSocket might provide deep order book data, the OrderBook class (if it were actively used in the main flow) processes only up to MAX_ORDERBOOK_LEVELS (50) for asks and bids.

        - More critically, for feature generation in OrderbookProcessor (used by the models), calculations like volume often only consider the top 10 levels of the asks and bids. This reduces the amount of data to iterate over for each update.
    - **Periodic Model Retraining:** The regression models (slippage_model, maker_taker_model) are not retrained on every single order book update. Training occurs initially after a sufficient number of data points (100) are collected and then periodically (e.g., every 100 new updates). This significantly reduces the computational load compared to continuous retraining.
3. **Responsive Graphical User Interface (GUI):**
    - **Decoupled UI Updates:** A QTimer (update_timer in MainWindow) triggers UI updates (fetching and displaying metrics) at a fixed interval (UPDATE_INTERVAL_MS = 100ms).

      This decouples the GUI refresh rate from the potentially much higher frequency of incoming WebSocket messages, ensuring smoother UI performance and preventing overload from too-frequent redraws.

    - **Thread-Safe Communication:** PyQt's signals and slots mechanism (data_received, processing_complete, connection_status pyqtSignals in WebSocketThread) is used for all communication between the background WebSocketThread and the main GUI thread.

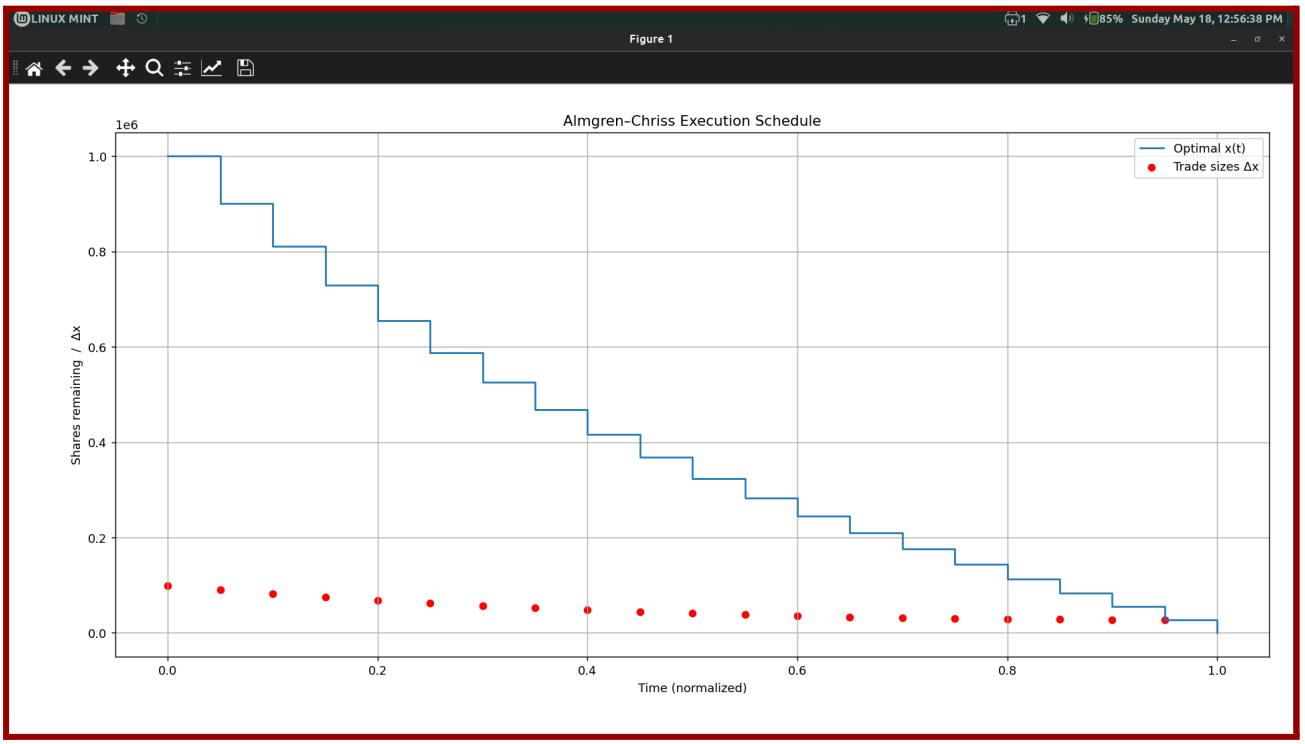
      This is the standard Qt way to ensure thread safety when updating UI elements or passing data across threads.

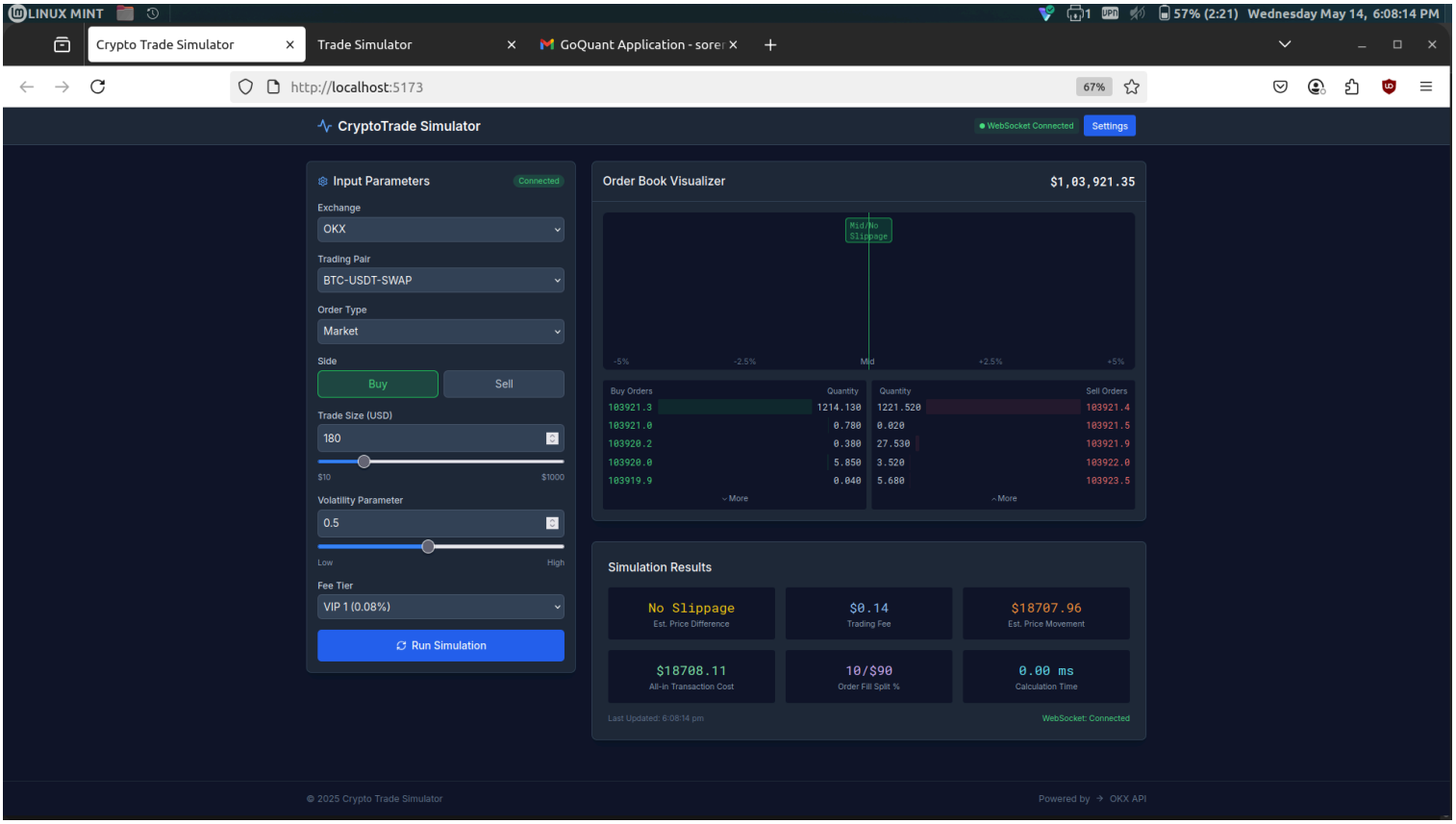
4. **Numerical Computations:**
    - numpy is utilized for certain numerical calculations (e.g., np.std, np.mean for the volatility proxy in model training, np.array for preparing feature arrays for model input).

      numpy operations are generally more efficient for array-based numerical tasks than standard Python loops and lists.

5. **Logging:**
    - The application uses Python's logging module to record events, errors, and informational messages to both a file (simulator.log) and the console.

      This is more performant and manageable than using print() statements for debugging and monitoring, especially in a multi-threaded application.





I do have an advanced React Application for this purpose under development but for Now we will use a simple demonstration for this assignment.

Author: Ramesh Chandra Soren

References have been linked in these documents as text links. Do refer to them. Along with that I have collected this data for 1hr and trained the model. If you want to see the data you can visit my Kaggle acc. To get the data.

Thank You