GOQUANT

Crypto Exchange Simulator

CONTENTS

VANSH AGARWAL

1 INTRODUCTION

The **Crypto Exchange Trading Simulator** is an advanced real-time market simulation tool developed in **C++17 & CMake build ecosystem** that enables traders and researchers to analyze trading strategies and market impact in cryptocurrency spot exchange markets. This high-performance application integrates real-time market data processing with sophisticated financial models to provide accurate trading cost estimates and market impact analysis.

1.1 Key Features

- Real-Time Order Book Aggregator
  - Price Level Aggregation
  - Maintains bid-ask spread statistics
- Real-Time Output Calculation
  - Volume Weighted Slippage calculation
  - Fee Calculation Engine
  - Maker/Taker with Fee Tier Analysis
  - Market Impact Estimation
  - Internal Latency Reporting
- Quant Models
  - Linear/Quantile Regression
  - Almgren Chriss Model
  - Logistic Regression
- SSL Websocket Data Streaming with Thread-Safe Lock-Free Message Broker.
- JSON Configuration Management & runtime input parameter updates.
- GUI Visualization
- Error Handling & Logging
- Test Driven Development

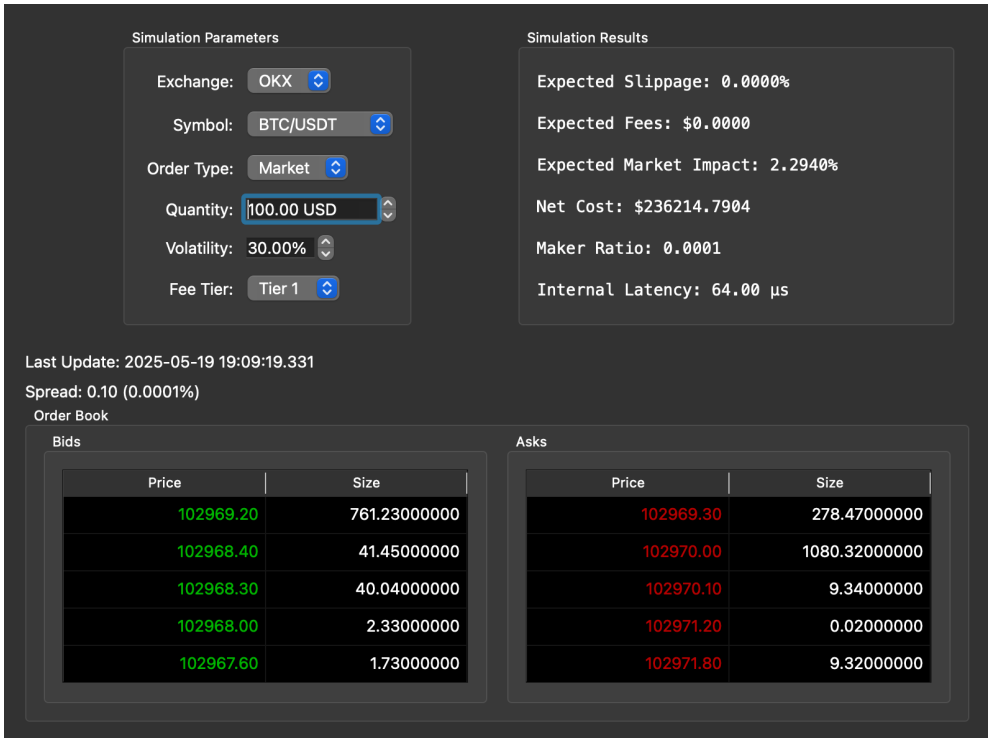

Figure 1

2 PROJECT STRUCTURE

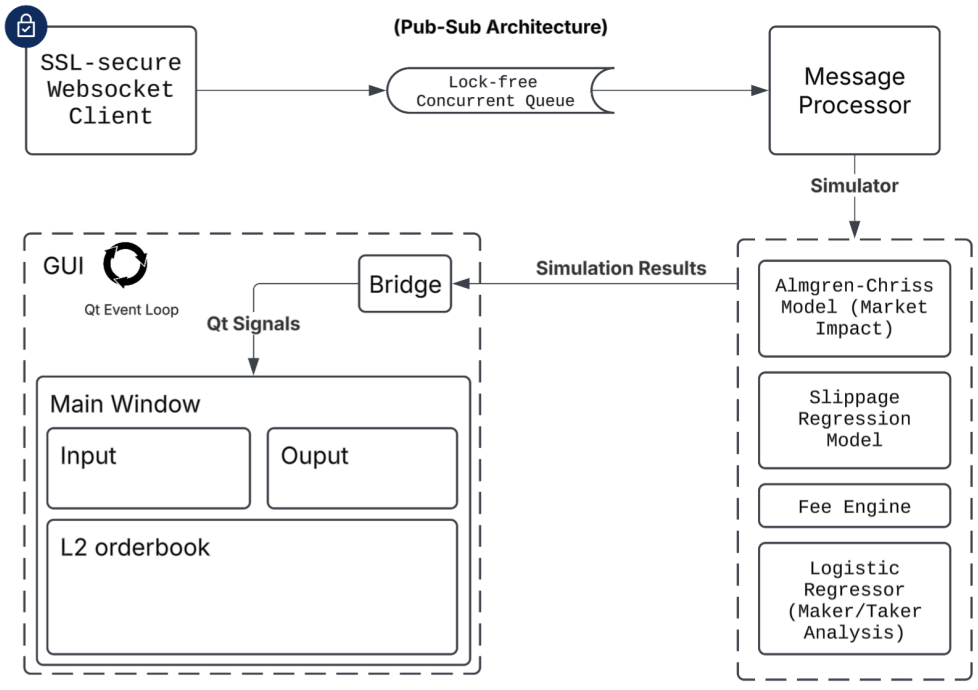

Figure 2: Data Architecture

2.1 Tech Stack

- `C++17`: Core application logic
- `Qt6`: GUI framework
- `MoodyCamel`: Concurrent Queue
- `WebSockets`: Real-time data streaming
- `OpenSSL`: Secure communication
- `Boost`: C++ libraries for networking and utilities
- `NlohmannJson`: JSON parsing and handling
- `spdlog`: Logging system
- `GTest`: Unit testing framework
- `CMake`: Build Ecosystem

3 CLEAN CODE PRACTICES

- **Single Responsibility Principle**: Each class has a single, well defined responsibility.
- **Observer Pattern**: Consumer subscription for event handling/ UI updates.
- **Thread Safe Singleton pattern**: Ensures a class has only one instance. Used in Logging & configuration.
- **RAII Resource Acquisition Is Initialization**: Resource acquisition and release handled by constructors/destructors.
- **Code Organization**: Usage of namesapces (core, models, ui etc) for ambiguity resolution.
- **Thread Safety**: Usage of `atomic` & lock-free queues.
- **Test Driven Development**: Unit Tests using GTests
- **const Correctness**: prevents from inadvertently modifying read-only members.
- **Smart Pointers**: Automatic memory management, preventing memory leaks.
- **Secure Streaming**: Secure SSL Websocket with OpenSSL

4 QUANT MODELS

4.1 Expected Slippage

Slippage refers to the difference between the expected price of a trade and the price at which the trade is actually executed. This is primarily due to market depth and liquidity constraints.

We use **Quantile Regression with Linear Components** to estimate expected slippage based on order size and order book characteristics. Quantile regression is used to model conditional quantiles of the slippage distribution, allowing more robust estimation in the presence of outliers.

$$\text{Quantile}_\tau(S) = \beta_0 + \beta_1 \cdot \text{Volume} + \beta_2 \cdot \text{Spread} + \beta_3 \cdot \text{Volatility} + \epsilon \quad (1)$$

Default regression coefficients: $\beta_0 = 0.0001$ (Base Slippage), $\beta_1 = 0.0002$ (Volume Impact), $\beta_2 = 0.5$ (Spread Impact), $\beta_3 = 0.0003$ (volatility impact). Quantile Levels = $[0.1, 0.25, 0.5, 0.75, 0.9]$

4.2 Expected Fees

Fees are calculated using a rule-based model derived from the exchange's published fee schedule. The fees vary based on account tier, order type (maker vs. taker), and volume.

$$F = r_f \cdot Q \cdot P \tag{2}$$

where:

- F is the total fee
- r_f is the fee rate (e.g., 0.001 = 0.1%)
- Q is the order quantity in units of asset
- P is the average fill price
- Taker fee rate: $r_f = 0.001$
- Maker fee rate: $r_f = 0.0008$

4.3 Expected Market Impact: Almgren-Chriss Model

The Almgren-Chriss model quantifies both temporary and permanent price impact from executing large trades over a fixed time horizon. This model is commonly used in optimal execution strategies.

The expected cost C for executing a trade of quantity X is:

$$\text{Cost} = \sum_{t=1}^{N} (\gamma x_t^2 + \eta X_t) \; + \; \lambda \sigma^2 T \sum_{t=1}^{N} (\frac{x_t}{T})^2 \tag{3}$$

where: $\gamma = 2.5 \times 10^{-6}$: permanent impact coefficient, $\eta = 1.0 \times 10^{-4}$: temporary impact coefficient, V: daily traded volume, X: trade size (units of asset), $\lambda = 0.5$ (Risk Aversion Parameter), $T = 1 \; day$ (Time Horizon)

4.4 Net Cost

Net cost represents the total expected cost of execution.

$$\text{Net Cost} = \text{Expected Slippage} + \text{Expected Fee} + \text{Expected Market Impact Cost} \tag{4}$$

4.5 Maker/Taker Proportion

Used logistic regression model to estimate the probability that a trade will be executed as a maker or a taker, based on current order book spread and order aggressiveness.

$$P(\text{maker}) = \frac{1}{1 + e^{-(\alpha + \beta_1 \cdot \text{spread} + \beta_2 \cdot \text{limit\_offset})}} \tag{5}$$

- $\alpha = -1.0$
- $\beta_1 = -2.5$ (higher spread $\Rightarrow$ less likely to be a maker)
- $\beta_2 = 5.0$ (larger limit offset $\Rightarrow$ more likely to be a maker)

5 PERFORMANCE OPTIMIZATION

Internal latency as less as $50\mu s$ **is achieved** with the following optimizations:

- **Minimal Copies & Data Locality**: Use of move semantics (std::move) in processing pipeline.
- **Model-Specific Optimizations**: Slippage estimation uses pre-aggregated depth levels, not full L2 book.
- **Almgren-Chriss model** simplified to closed-form expressions.
- **UI Throttling** Output signals to Qt UI are rate-limited using a timer mechanism.
- **Lock-Free & Bounded Queues**: Uses `MoodyCamel::ConcurrentQueue` for larger volumes of data & ensures bound to prevent memory overflow.
- **Boost.Beast + Asio WebSocket Client**: High-performance asynchronous I/O with dedicated IO threads, isolated from GUI or model thrreads.
- **Threading and Concurrency**: Mainly 3 threads:
  - **1. Websocket Client** Parses incoming JSON and pushes data to lock- free concurrent message queues.
  - **2. Simulator & Bridge** Pops from queue, runs slippage + fees + market impact models. Posts result to GUI via Qt signals/slots.
  - **3. Qt Main/UI Thread** Handles visualization
- **Thread-safety** Guaranteed by `MoodyCamel::ConcurrentQueue`
- **Qt Signal-Slot Mechanism** Used to emit data from model threads to GUI thread in a thread-safe, latency-aware way.
- **RAII Resource Acquisition Is Initialization**: Resource acquisition and release handled by constructors/destructors. **Smart Pointers** are also used, ensuring automatic memory management & avoiding memory leaks