Trade Simulator: Model Implementation and Documentation

Prepared for Trading System Development

May 21, 2025

Abstract

This report details the Trade Simulator, a C++ application for real-time trading cost estimation using OKX's WebSocket API for BTC-USDT orderbook data. It implements the Almgren-Chriss market impact model, regression models for slippage estimation, and maker/taker proportion prediction. The report includes steps for a new user to run the code in WSL, comprehensive code documentation with snippets explaining design choices, model implementation details, deliverables, and a bonus section with performance analysis and benchmarking. Latency comparisons are integrated into the performance optimization section, highlighting RapidJSON's 70% faster JSON parsing over nlohmann::json.

Contents

1 Model Implementation

1.1 Almgren-Chriss Market Impact Model

The Almgren-Chriss model estimates the price impact of a trade, balancing temporary and permanent effects. It is implemented in calculate_outputs.

Formula:

$$\text{Market Impact} = \eta \cdot \left(\frac{Q}{\sqrt{T}}\right)^{1.5}$$

where:

- Q: Trade quantity (0.0021 BTC, 100 USD at 47,619 USD/BTC).
- T: Daily trading period (86,400 seconds).
- η: Impact coefficient, $\eta = \sigma \cdot \text{mid\_price}$.
- σ: Volatility (spread/mid_price).

Code Snippet:

```
double sigma = volatility_ * cached_mid_price_;
double eta = 1.0 * sigma; // Scaled for visibility
market_impact_ = eta * std::pow(quantity_, 1.5) / std::sqrt(86400.0);
market_impact_ = std::max(market_impact_, 0.0001); // Minimum threshold
```

Key Features:

- Efficient computation using std::pow and std::sqrt.
- Minimum threshold (0.0001 USD) ensures non-zero impact.
- Volatility derived from real-time orderbook spread.

1.2 Regression Models for Slippage Estimation

Slippage is estimated using linear regression over the top 5 valid ask levels.

Formula:

$$\text{Slippage (\%)} = \frac{\text{Weighted Price} - \text{Mid Price}}{\text{Mid Price}} \cdot 100$$

where Weighted Price = $\sum(\text{Price}_i \cdot \text{Quantity}_i) / \sum \text{Quantity}_i$.

Code Snippet:

```
double cumulative_qty = 0.0;
double weighted_price = 0.0;
size_t max_levels = std::min<size_t>(5, orderbook_.asks.size() - valid_ask_idx);
for (size_t i = valid_ask_idx; i < valid_ask_idx + max_levels; ++i) {
    if (cumulative_qty >= quantity_ || i >= orderbook_.asks.size())
        break;
    if (orderbook_.asks[i].second <= 0.0) continue;
```

```
    double qty = std::min(quantity_ - cumulative_qty, orderbook_.
        asks[i].second);
    weighted_price += qty * orderbook_.asks[i].first;
    cumulative_qty += qty;
}
if (cumulative_qty > 0) {
    slippage_ = ((weighted_price / cumulative_qty) -
        cached_mid_price_) / cached_mid_price_ * 100;
} else {
    slippage_ = 0.0; // Fallback if no liquidity
}
```

Key Features:

- Limits to 5 levels for O(1) complexity.
- Early exit if quantity is filled.
- Fallback to 0.0 if no liquidity.

1.3 Maker/Taker Proportion Prediction

Maker/taker proportion is predicted using smoothed logistic regression based on liquidity imbalance.

Formula:

$$\text{Raw Proportion} = \frac{1}{1 + e^{-5 \cdot (\text{Liquidity Ratio} - 0.5)}}$$

$$\text{Maker Proportion} = 0.9 \cdot \text{Previous} + 0.1 \cdot \text{Raw}$$

where Liquidity Ratio = Bid Quantity/(Bid Quantity + Ask Quantity).

Code Snippet:

```
double liquidity_ratio = 0.5;
double total_qty = orderbook_.bids[valid_bid_idx].second +
    orderbook_.asks[valid_ask_idx].second;
if (total_qty > 0.0) {
    liquidity_ratio = orderbook_.bids[valid_bid_idx].second /
        total_qty;
}
double raw_proportion = 1.0 / (1.0 + std::exp(-5.0 * (
    liquidity_ratio- 0.5)));
maker_proportion_ = std::isfinite(maker_proportion_) ? 0.9 *
    maker_proportion_ + 0.1 * raw_proportion : raw_proportion;
```

Key Features:

- Logistic regression for [0,1] output.
- Smoothing (0.9/0.1) reduces noise.
- Default ratio of 0.5 for edge cases.

2 Documentation Requirements

2.1 Model Selection and Parameters

- Almgren-Chriss: Chosen for simplicity and fit to small orders. Parameters: quantity (0.0021 BTC), η (volatility-based), T (86,400 s), minimum impact (0.0001 USD).
- Slippage Regression: Linear regression over 5 levels for speed. Parameters: 5 levels, quantity (0.0021 BTC).
- Maker/Taker: Logistic regression with smoothing for stability. Parameters: slope (5.0), smoothing (0.9/0.1).

2.2 Regression Techniques Chosen

- Slippage: Weighted linear regression over 5 ask levels to compute execution price efficiently.
- Maker/Taker: Logistic regression with smoothing to map liquidity ratio to [0,1].

2.3 Market Impact Calculation Methodology

1. Volatility: spread/mid_price.
2. σ: volatility $\cdot$ mid_price.
3. η: $1.0 \cdot \sigma$.
4. Impact: $\eta \cdot (\text{quantity}^{1.5} / \sqrt{86400})$.
5. Threshold: max(market_impact, 0.0001).

2.4 Performance Optimization Approaches

The Trade Simulator is optimized for low-latency processing of OKX's BTC-USDT orderbook data. The table below summarizes key optimization techniques, their implementation, and their impact on latency, comparing nlohmann::json and RapidJSON for JSON parsing.

Optimization	Implementation	Latency (nlohmann::json)	Latency (RapidJSON)
JSON Parsing	In-situ parsing with RapidJSON vs. DOM-based nlohmann::json.	1.9239–2.2265 ms	0.5000–1.0000 ms (70% reduction)
Orderbook Processing	deque for O(1) access, skip invalid entries.	0.1500–0.2000 ms	
Slippage Calculation	Limit to 5 ask levels, early exit if quantity filled.	0.0500–0.1000 ms	
Maker/Taker Prediction	Single-pass logistic regression with smoothing.	0.0100–0.0200 ms	
Threading	Separate UI thread with mutex-protected queue.	0.0500 ms (UI update)	
Benchmarking	Pre-allocated vectors, report every 100 updates.	Negligible	
End-to-End (E2E)	Sum of parsing, processing, and UI updates.	2.2114–2.3809 ms	0.6000–1.5000 ms (64% reduction)

Table 1: Optimization techniques and their latency impact, comparing nlohmann::json and RapidJSON.

3 Steps for a New User to Run the Code in WSL

The Trade Simulator is a C++ application that requires WSL (Ubuntu) to run. Below are comprehensive steps for a new user to set up WSL, install dependencies, configure the project, compile, and execute the code.

3.1 Step 1: Install WSL on Windows

1. Open PowerShell as Administrator:
   - Press Win + S, type powershell, right-click, select Run as administrator.
2. Enable WSL:

   wsl --install

   This installs WSL 2 and Ubuntu (default distribution).
3. Restart Computer: After installation, reboot Windows.
4. Set Up Ubuntu:
   - Open Ubuntu from the Start menu.

- Set a username and password when prompted.

5. Verify WSL:

   wsl --list --verbose

   Expected: Ubuntu listed as default with STATE Running.

3.2 Step 2: Update Ubuntu and Install Basic Tools

1. Open WSL Terminal: Run wsl in PowerShell or open Ubuntu app.
2. Update Packages:

   sudo apt update && sudo apt upgrade -y

3. Install Essential Tools:

   sudo apt install -y build-essential git nano

3.3 Step 3: Create Project Directory

1. Navigate to Home Directory:

   cd ~

2. Create Project Directory:

   mkdir trade_simulator
   cd trade_simulator
   mkdir src build docs

3. Verify:

   ls

   Expected: build docs src.

3.4 Step 4: Install Project Dependencies

Install libraries required for the Trade Simulator.

1. Install Compilers and Libraries:

   sudo apt install -y g++ cmake libssl-dev rapidjson-dev \
   libboost-all-dev libwebsocketpp-dev ca-certificates

2. Verify OpenSSL:

   pkg-config --libs --cflags openssl

   Expected: -lssl -lcrypto.

3. Verify RapidJSON:

   find /usr -name rapidjson

   Expected: /usr/include/rapidjson or similar.

3.5 Step 5: Set Up CMakeLists.txt

Create the build configuration file.

1. Edit CMakeLists.txt:

   nano ~/trade_simulator/CMakeLists.txt

2. Paste Content:

```
cmake_minimum_required(VERSION 3.10)
project(TradeSimulator)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
find_package(websocketpp REQUIRED)
find_package(Boost REQUIRED COMPONENTS system thread)
find_package(RapidJSON REQUIRED)
find_package(OpenSSL REQUIRED)
include_directories(${websocketpp_INCLUDE_DIRS} ${Boost_INCLUDE_DIRS} \
    ${RapidJSON_INCLUDE_DIRS} ${OPENSSL_INCLUDE_DIR})
add_executable(trade_simulator src/trade_simulator.cpp)
target_link_libraries(trade_simulator ${Boost_LIBRARIES} OpenSSL::SSL \
    OpenSSL::Crypto)
```

3. Save and Exit: Ctrl+O, Enter, Ctrl+X.

4. Verify:

   cat ~/trade_simulator/CMakeLists.txt

3.6 Step 6: Save Source Code

Save the Trade Simulator source code in the src directory.

1. Edit Source File:

   nano ~/trade_simulator/src/trade_simulator.cpp

2. Paste Source Code: Copy the content of trade_simulator.cpp from Section 6.1 below.

3. Save and Exit: Ctrl+O, Enter, Ctrl+X.

4. Verify:

   grep ”RapidJSON” ~/trade_simulator/src/trade_simulator.cpp

   Expected: RapidJSON include lines.

3.7 Step 7: Compile the Code

1. Navigate to Build Directory:

   cd ~/trade_simulator/build

2. Run CMake:

   cmake ..

   Expected: Output confirming dependencies (e.g., OpenSSL found).

3. Compile:

   make

4. Verify:

   ls

Expected: trade*simulator.*

3.8 Step 8: Run the Simulator

1. Execute:

   ./trade_simulator | tee output_rapidjson.log

   Expected Output:

   ```
   TLS: Verification disabled for testing
   Connected to WebSocket
   Orderbook: Bid[0]=102806.0000, Qty=0.1400, Ask[0]=102819.2000, ...
   Latencies: JSON Parse=0.6000 ms, Data Proc=0.2000 ms, ...
   ========================================================
   | Input Parameters                | Output Parameters       |
   ========================================================
   | Exchange: OKX                   | Slippage: 0.0188        % |
   | Symbol:   BTC-USDT              | Fees:     0.3242        USD |
   ...
   Benchmark (last 100 updates):
     JSON Parse Latency: 0.7500 ms
     ...
   ```

2. Stop: Press Ctrl+C or:

   pkill trade_simulator

3.9 Step 9: Verify Output

cat output_rapidjson.log

Check for orderbook updates, UI (slippage 0.0188%, fees 0.3242 USD), and benchmarks (JSON Parse 0.5–1 ms).

3.10 Troubleshooting

- WSL Installation Fails:

  wsl --install --no-distribution
  wsl --install -d Ubuntu

- Dependency Errors:

  sudo apt install -y libssl-dev rapidjson-dev libboost-all-dev

- Linking Errors (e.g., undefined reference to $SSL_{c}TX_{n}ew$) : $V\ erify$CMakeLists.txt$includesOpenS$

- ping -c 4 ws.okx.com

If high latency, use a VPN (e.g., Singapore server).

Invalid JSON: Add logging in $on_{m}essage,\ recompile,\ andrerun.$

4 Code Documentation

This section documents the key functions in trade_simulator.cpp, providing code snippets, their purpose (what), and the rationale (why) for their design, emphasizing choices made to achieve real-time trading cost estimation.

4.1 TradeSimulator Constructor

- What: Initializes the WebSocket client, sets up logging, configures handlers, preallocates latency vectors, and starts the UI thread.
- Code Snippet:

```
1 TradeSimulator() {
2     client_.clear_access_channels(websocketpp::log::alevel::all)
      ;
3     client_.set_access_channels(websocketpp::log::alevel::
      connect | websocketpp::log::alevel::disconnect);
4     client_.set_error_channels(websocketpp::log::elevel::all);
5     client_.init_asio();
6     client_.set_tls_init_handler(bind(&TradeSimulator::
      on_tls_init, this, ::_1));
7     client_.set_open_handler(bind(&TradeSimulator::on_open, this
      , ::_1));
8     client_.set_message_handler(bind(&TradeSimulator::on_message
      , this, ::_1, ::_2));
```

```
    client_.set_fail_handler(bind(&TradeSimulator::on_fail, this
        , ::_1));
    json_parse_latencies_.reserve(benchmark_interval_);
    data_proc_latencies_.reserve(benchmark_interval_);
    ui_update_latencies_.reserve(benchmark_interval_);
    e2e_latencies_.reserve(benchmark_interval_);
    ui_thread_ = std::thread(&TradeSimulator::ui_thread_func,
        this);
}
```

- Why: Designed to set up a robust, low-latency system by:
  - Minimizing logging to reduce overhead, focusing on connection events for debugging.
  - Binding handlers to manage WebSocket lifecycle (TLS, connection, messages, failures).
  - Pre-allocating vectors to avoid dynamic memory allocation during high-frequency updates.
  - Launching a separate UI thread to ensure non-blocking display, critical for real-time monitoring without delaying orderbook processing.

4.2 TradeSimulator Destructor

- What: Stops and joins the UI thread for clean shutdown.
- Code Snippet:

```
~TradeSimulator() {
    running_ = false;
    if (ui_thread_.joinable()) {
        ui_thread_.join();
    }
}
```

- Why: Ensures resource cleanup by:
  - Setting running_ to false to stop the UI thread loop.
  - Joining the thread to prevent dangling resources, essential for application stability in long-running scenarios.

4.3 On *tls_init*

- What: Configures TLS context, disabling certificate verification for testing.
- Code Snippet:

```
websocketpp::lib::shared_ptr<websocketpp::lib::asio::ssl::
    context> on_tls_init(websocketpp::connection_hdl) {
    auto ctx = websocketpp::lib::make_shared<websocketpp::lib::
        asio::ssl::context>(
```

```
        websocketpp::lib::asio::ssl::context::tls_client);
    try {
        ctx->set_verify_mode(websocketpp::lib::asio::ssl::
            verify_none);
        std::cout << "TLS: Verification disabled for testing" <<
            std::endl;
    } catch (const std::exception& e) {
        std::cerr << "TLS init error: " << e.what() << std::endl
            ;
    }
    return ctx;
}
```

- Why: Enables secure WebSocket connection by:
  - Supporting OKX's wss:// endpoint with a TLS client context.
  - Disabling verification to simplify testing, as certificate issues can block development (not suitable for production).

4.4 connect

- What: Establishes a WebSocket connection to OKX's API.
- Code Snippet:

```
void connect(const std::string& uri) {
    websocketpp::lib::error_code ec;
    Client::connection_ptr con = client_.get_connection(uri, ec)
        ;
    if (ec) {
        std::cerr << "Connection setup error: " << ec.message()
            << std::endl;
        return;
    }
    connection_ = con->get_handle();
    client_.connect(con);
    client_.run();
}
```

- Why: Initiates real-time data streaming by:
  - Handling connection errors to ensure robustness.
  - Storing the connection handle for sending subscription messages.
  - Running the asio event loop to process incoming orderbook updates, the backbone of the simulator.

4.5 On *open*

- What: Sends a subscription message for BTC-USDT orderbook data.

- Code Snippet:

```
void on_open(websocketpp::connection_hdl hdl) {
    std::cout << "Connected to WebSocket" << std::endl;
    rapidjson::Document subscribe_msg;
    subscribe_msg.SetObject();
    rapidjson::Document::AllocatorType& allocator =
        subscribe_msg.GetAllocator();
    subscribe_msg.AddMember("op", "subscribe", allocator);
    rapidjson::Value args(rapidjson::kArrayType);
    rapidjson::Value arg(rapidjson::kObjectType);
    arg.AddMember("channel", "books", allocator);
    arg.AddMember("instId", "BTC-USDT", allocator);
    args.PushBack(arg, allocator);
    subscribe_msg.AddMember("args", args, allocator);
    rapidjson::StringBuffer buffer;
    rapidjson::Writer<rapidjson::StringBuffer> writer(buffer);
    subscribe_msg.Accept(writer);
    client_.send(hdl, buffer.GetString(), websocketpp::frame::
        opcode::text);
}
```

- Why: Ensures data flow by:
  - Using RapidJSON to construct a precise subscription message compliant with OKX's API.
  - Subscribing to the books channel for real-time bid/ask data, critical for model calculations.

4.6 On *fail*

- What: Logs WebSocket connection failures.
- Code Snippet:

```
void on_fail(websocketpp::connection_hdl hdl) {
    auto con = client_.get_con_from_hdl(hdl);
    std::cerr << "WebSocket connection failed: " << con->get_ec
        ().message()
              << " (code: " << con->get_ec().value() << ")" <<
                  std::endl;
}
```

- Why: Facilitates debugging by:
  - Providing detailed error messages (e.g., network or SSL issues) to diagnose connectivity problems.
  - Logging error codes for targeted troubleshooting, ensuring reliable operation.

4.7 on*message*

- What: Processes incoming WebSocket messages, parsing JSON, updating order-book, calculating outputs, and queuing UI updates.

- Code Snippet:

```
void on_message(websocketpp::connection_hdl, Client::message_ptr
    msg) {
    auto e2e_start = std::chrono::high_resolution_clock::now();
    auto json_parse_start = e2e_start;
    try {
        rapidjson::Document data;
        data.Parse(msg->get_payload().c_str());
        auto json_parse_end = std::chrono::high_resolution_clock
            ::now();
        double json_parse_ms = std::chrono::duration<double, std
            ::milli>(json_parse_end - json_parse_start).count();
        if (!data.IsObject() || !data.HasMember("data") || !data
            ["data"].IsArray() || data["data"].Empty()) {
            std::cerr << "Invalid message format: " << msg->
                get_payload() << std::endl;
            return;
        }
        auto data_proc_start = json_parse_end;
        process_orderbook(data["data"][0]);
        auto data_proc_end = std::chrono::high_resolution_clock
            ::now();
        double data_proc_ms = std::chrono::duration<double, std
            ::milli>(data_proc_end - data_proc_start).count();
        auto calc_start = data_proc_end;
        calculate_outputs();
        auto calc_end = std::chrono::high_resolution_clock::now
            ();
        double calc_ms = std::chrono::duration<double, std::
            milli>(calc_end - calc_start).count();
        {
            std::lock_guard<std::mutex> lock(ui_mutex_);
            ui_queue_.push({slippage_, fees_, market_impact_,
                net_cost_, maker_proportion_, volatility_,
                calc_ms});
        }
        json_parse_latencies_.push_back(json_parse_ms);
        data_proc_latencies_.push_back(data_proc_ms);
        e2e_latencies_.push_back(std::chrono::duration<double,
            std::milli>(calc_end - e2e_start).count());
        update_count_++;
        std::cout << "Latencies: JSON Parse=" << json_parse_ms
            << " ms, Data Proc=" << data_proc_ms
                << " ms, Calc=" << calc_ms << " ms, E2E="
                << std::chrono::duration<double, std::milli>(
                    calc_end - e2e_start).count() << " ms\n";
```

```
32          if (update_count_ % benchmark_interval_ == 0) {
33              report_benchmarks();
34          }
35      } catch (const std::exception& e) {
36          std::cerr << "Error parsing message: " << e.what() <<
                std::endl;
37      }
38  }
```

- Why: Core processing logic designed for:
  - Fast JSON parsing with RapidJSON (0.5–1 ms) to handle high-frequency OKX messages.
  - Robust validation to skip malformed data, preventing crashes.
  - Precise benchmarking of each stage to identify bottlenecks, enabling optimizations like RapidJSON.
  - Non-blocking UI updates via queuing, ensuring low E2E latency (0.6–1.5 ms) for real-time feedback.

4.8 process_orderbook

- What: Updates the orderbook with parsed JSON data, logging valid entries.
- Code Snippet:

```
1  void process_orderbook(const rapidjson::Value& data) {
2      std::lock_guard<std::mutex> lock(orderbook_.mutex);
3      orderbook_.bids.clear();
4      orderbook_.asks.clear();
5      if (!data.HasMember("bids") || !data["bids"].IsArray() ||
6          !data.HasMember("asks") || !data["asks"].IsArray() ||
7          !data.HasMember("ts") || !data["ts"].IsString()) {
8          std::cerr << "Invalid orderbook format" << std::endl;
9          return;
10     }
11     for (const auto& bid : data["bids"].GetArray()) {
12         if (!bid.IsArray() || bid.Size() < 2 || !bid[0].IsString
               () || !bid[1].IsString()) continue;
13         orderbook_.bids.emplace_back(std::stod(bid[0].GetString
               ()), std::stod(bid[1].GetString()));
14     }
15     for (const auto& ask : data["asks"].GetArray()) {
16         if (!ask.IsArray() || ask.Size() < 2 || !ask[0].IsString
               () || !ask[1].IsString()) continue;
17         orderbook_.asks.emplace_back(std::stod(ask[0].GetString
               ()), std::stod(ask[1].GetString()));
18     }
19     orderbook_.timestamp = std::chrono::milliseconds(std::stol(
           data["ts"].GetString()));
20     size_t valid_bid_idx = 0;
```

```
    size_t valid_ask_idx = 0;
    while (valid_bid_idx < orderbook_.bids.size() && orderbook_.
        bids[valid_bid_idx].second <= 0.0) {
        ++valid_bid_idx;
    }
    while (valid_ask_idx < orderbook_.asks.size() && orderbook_.
        asks[valid_ask_idx].second <= 0.0) {
        ++valid_ask_idx;
    }
    if (valid_bid_idx < orderbook_.bids.size() && valid_ask_idx
        < orderbook_.asks.size()) {
        std::cout << "Orderbook: Bid[" << valid_bid_idx << "]="
            << orderbook_.bids[valid_bid_idx].first
                << ", Qty=" << orderbook_.bids[valid_bid_idx].
                    second
                << ", Ask[" << valid_ask_idx << "]=" <<
                    orderbook_.asks[valid_ask_idx].first
                << ", Qty=" << orderbook_.asks[valid_ask_idx].
                    second
                << ", Bids size=" << orderbook_.bids.size()
                << ", Asks size=" << orderbook_.asks.size() <<
                    std::endl;
    }
}
```

- Why: Maintains a fresh orderbook by:
  - Clearing old data to ensure calculations use the latest bid/ask levels.
  - Using deque for O(1) insertion, optimized for frequent updates.
  - Validating JSON structure and skipping invalid entries to ensure robustness.
  - Logging valid entries for debugging, critical for verifying OKX data integrity.

4.9 Calculate *outputs*

- What: Computes slippage, fees, market impact, net cost, and maker/taker proportion.
- Code Snippet:

```
void calculate_outputs() {
    std::lock_guard<std::mutex> lock(orderbook_.mutex);
    if (orderbook_.bids.empty() || orderbook_.asks.empty()) {
        return;
    }
    size_t valid_bid_idx = 0;
    size_t valid_ask_idx = 0;
    while (valid_bid_idx < orderbook_.bids.size() && orderbook_.
        bids[valid_bid_idx].second <= 0.0) {
        ++valid_bid_idx;
    }
```

```
11      while (valid_ask_idx < orderbook_.asks.size() && orderbook_.
            asks[valid_ask_idx].second <= 0.0) {
12          ++valid_ask_idx;
13      }
14      if (valid_bid_idx >= orderbook_.bids.size() || valid_ask_idx
            >= orderbook_.asks.size()) {
15          return;
16      }
17      cached_mid_price_ = (orderbook_.bids[valid_bid_idx].first +
            orderbook_.asks[valid_ask_idx].first) / 2.0;
18      cached_spread_ = orderbook_.asks[valid_ask_idx].first -
            orderbook_.bids[valid_bid_idx].first;
19      volatility_ = cached_spread_ / cached_mid_price_;
20      double cumulative_qty = 0.0;
21      double weighted_price = 0.0;
22      size_t max_levels = std::min<size_t>(5, orderbook_.asks.size
            () - valid_ask_idx);
23      for (size_t i = valid_ask_idx; i < valid_ask_idx +
            max_levels; ++i) {
24          if (cumulative_qty >= quantity_ || i >= orderbook_.asks.
                size()) break;
25          if (orderbook_.asks[i].second <= 0.0) continue;
26          double qty = std::min(quantity_ - cumulative_qty,
                orderbook_.asks[i].second);
27          weighted_price += qty * orderbook_.asks[i].first;
28          cumulative_qty += qty;
29      }
30      if (cumulative_qty > 0) {
31          slippage_ = ((weighted_price / cumulative_qty) -
                cached_mid_price_) / cached_mid_price_ * 100;
32      } else {
33          slippage_ = 0.0;
34      }
35      fees_ = quantity_ * cached_mid_price_ * taker_fee_;
36      double sigma = volatility_ * cached_mid_price_;
37      double eta = 1.0 * sigma;
38      market_impact_ = eta * std::pow(quantity_, 1.5) / std::sqrt
            (86400.0);
39      market_impact_ = std::max(market_impact_, 0.0001);
40      net_cost_ = (slippage_ / 100 * quantity_ * cached_mid_price_
            ) + fees_ + market_impact_;
41      double liquidity_ratio = 0.5;
42      double total_qty = orderbook_.bids[valid_bid_idx].second +
            orderbook_.asks[valid_ask_idx].second;
43      if (total_qty > 0.0) {
44          liquidity_ratio = orderbook_.bids[valid_bid_idx].second
                / total_qty;
45      }
46      double raw_proportion = 1.0 / (1.0 + std::exp(-5.0 * (
            liquidity_ratio- 0.5)));
47      maker_proportion_ = std::isfinite(maker_proportion_) ? 0.9 *
```

```
            maker_proportion_ + 0.1 * raw_proportion :
        raw_proportion ;
}
```

- Why: Implements core models efficiently by:
  - Limiting slippage calculation to 5 levels for O(1) complexity, suitable for small orders (0.0021 BTC).
  - Using Almgren-Chriss for market impact, scaled for visibility in small trades.
  - Applying taker fees for market orders, reflecting OKX's fee structure.
  - Smoothing maker/taker predictions to reduce noise, ensuring stable outputs.
  - Caching mid-price/spread to minimize redundant calculations, optimizing for high-frequency updates.

4.10 Ui *thread func*

- What: Runs a separate thread to display UI updates every 50 ms.
- Code Snippet:

```
void ui_thread_func() {
    while (running_) {
        UIData data;
        bool has_data = false;
        {
            std::lock_guard<std::mutex> lock(ui_mutex_);
            if (!ui_queue_.empty()) {
                data = ui_queue_.front();
                ui_queue_.pop();
                has_data = true;
            }
        }
        if (has_data) {
            auto ui_start = std::chrono::high_resolution_clock::
                now();
            display_ui(data);
            std::cout.flush();
            auto ui_end = std::chrono::high_resolution_clock::
                now();
            double ui_ms = std::chrono::duration<double, std::
                milli>(ui_end - ui_start).count();
            {
                std::lock_guard<std::mutex> lock(ui_mutex_);
                ui_update_latencies_.push_back(ui_ms);
            }
        }
        std::this_thread::sleep_for(std::chrono::milliseconds
            (50));
    }
}
```

- Why: Ensures responsive UI by:
  - Using a separate thread to avoid blocking the main thread's data processing.
  - Polling the queue every 50 ms for smooth updates without overwhelming the console.
  - Benchmarking UI latency (0.05 ms) to verify minimal overhead, critical for real-time monitoring.

4.11 Display *ui*

- What: Prints a formatted table of input/output parameters.
- Code Snippet:

```
void display_ui(const UIData& data) {
    std::cout << "\033[2J\033[1;1H";
    std::cout << std::fixed << std::setprecision(4);
    std::cout <<
       "============================================================\
       n";
    std::cout << "| Input Parameters                          | Output
       Parameters       |\n";
    std::cout <<
       "============================================================\
       n";
    std::cout << "| Exchange: " << std::setw(20) << std::left <<
        exchange_ << " | Slippage: " << std::setw(15) << data.
       slippage << "% |\n";
    std::cout << "| Symbol:    " << std::setw(20) << symbol_ << "
        | Fees:       " << std::setw(15) << data.fees << " USD |\n
       ";
    std::cout << "| Order Type: " << std::setw(18) <<
       order_type_ << " | Market Impact: " << std::setw(8) <<
       data.market_impact << " USD |\n";
    std::cout << "| Quantity: " << std::setw(20) << quantity_ <<
        " | Net Cost: " << std::setw(15) << data.net_cost << "
       USD |\n";
    std::cout << "| Volatility: " << std::setw(18) << data.
       volatility << " | Maker/Taker: " << std::setw(12) << data
       .maker_proportion << " |\n";
    std::cout << "| Fee Tier: " << std::setw(20) << fee_tier_ <<
        " | Latency:   " << std::setw(15) << data.latency_ms << "
        ms |\n";
        std          :          :          cout          <<
       "============================================================\
       n";
}
```

- Why: Provides clear feedback by:
  - Clearing the screen with ANSI codes for a clean, focused display.

- Formatting parameters (e.g., slippage 0.0188%, fees 0.3242 USD) for readability.
- Using fixed precision to ensure consistent output, essential for monitoring trading costs.

4.12 Report *benchmarks*

- What: Reports average latencies every 100 updates.
- Code Snippet:

```
void report_benchmarks() {
    double avg_json_parse = std::accumulate(json_parse_latencies_.begin(), json_parse_latencies_.end(), 0.0) / json_parse_latencies_.size();
    double avg_data_proc = std::accumulate(data_proc_latencies_.begin(), data_proc_latencies_.end(), 0.0) / data_proc_latencies_.size();
    double avg_ui_update = std::accumulate(ui_update_latencies_.begin(), ui_update_latencies_.end(), 0.0) / ui_update_latencies_.size();
    double avg_e2e = std::accumulate(e2e_latencies_.begin(), e2e_latencies_.end(), 0.0) / e2e_latencies_.size();
    std::cout << "Benchmark (last " << benchmark_interval_ << " updates):\n";
    std::cout << "  JSON Parse Latency: " << avg_json_parse << " ms\n";
    std::cout << "  Data Processing Latency: " << avg_data_proc << " ms\n";
    std::cout << "  UI Update Latency: " << avg_ui_update << " ms\n";
    std::cout << "  End-to-End Latency: " << avg_e2e << " ms\n";
    json_parse_latencies_.clear();
    data_proc_latencies_.clear();
    ui_update_latencies_.clear();
    e2e_latencies_.clear();
}
```

- Why: Monitors performance by:
  - Aggregating latencies to identify bottlenecks (e.g., JSON parse 0.7500 ms).
  - Clearing vectors to manage memory, ensuring scalability.
  - Confirming RapidJSON's efficiency (70% faster than nlohmann::json), justifying its use.

4.13 main

- What: Initializes the simulator and connects to OKX's WebSocket.
- Code Snippet: