Documentation

DEX Order Execution Engine

Eterna Labs
Backend Assignment

Rajneesh Kumar
220121047
rajneesh.kumar@iitg.ac.in

December 14, 2025

Contents

1 Introduction

Decentralized exchanges (DEXs) enable users to trade cryptocurrencies without relying on a central authority. However, trading across multiple DEX platforms presents several challenges: each platform may offer different prices, liquidity, and transaction fees at any given time. Selecting the optimal execution path requires a system that can dynamically compare DEXs and route orders efficiently.

In addition, trading platforms often need to handle multiple orders arriving simultaneously while providing clear, real-time feedback to users. Without proper design, users may experience delayed execution, failed orders, or limited visibility into the status of their trades.

This project presents an **Order Execution Engine** that addresses these challenges by providing reliable, observable, and concurrent order processing with real-time status updates.

1.1 Problem Statement

The primary challenge is designing a backend system capable of executing trading orders reliably, while managing concurrency, DEX routing decisions, and execution status tracking.

Specifically, the system must:

- Accept user orders through a REST API endpoint (`/api/orders/execute`) and respond immediately with a unique order ID.
- Process orders asynchronously to prevent blocking incoming requests, ensuring high throughput.
- Query multiple DEXs (Raydium and Meteora) for price quotes and select the DEX offering the best execution in terms of price and liquidity.
- Provide continuous, real-time feedback on the order lifecycle via WebSocket updates, including statuses such as `PENDING`, `ROUTING`, `BUILDING`, `SUBMITTED`, `CONFIRMED`, and `FAILED`.
- Implement retry logic with exponential backoff to handle temporary failures, while logging errors for post-mortem analysis.
- Ensure safe execution including slippage handling and simulated transaction finalization.

Without such a system, users have no assurance of optimal execution and limited visibility into the status of their orders, which can lead to failed trades or suboptimal outcomes.

Why Choose Market Orders?

Question: Why did we choose to implement market orders first?
Answer: Market orders are the simplest and most common type of orders. They execute immediately at the current market price, which makes it easier to demonstrate and test the core functionalities of the engine such as:

- Fetching and comparing DEX quotes
- Routing orders to the best execution venue
- Processing orders asynchronously
- Sending real-time updates via WebSocket

By starting with market orders, we can focus on building the architecture, queue management, and live updates. The same engine design can later support limit orders (execute at a target price) and sniper orders (trigger on token launches) with minimal changes.

1.2 Key Features

- Asynchronous market order execution with queue-based management
- Dynamic price comparison and routing between Raydium and Meteora
- Real-time order status notifications using WebSocket
- Retry mechanism with exponential backoff for failed executions
- Slippage protection during simulated execution
- Persistent order history in PostgreSQL for reporting and analysis

1.3 Technology Stack

- **Backend:** Spring Boot 3.2 with Java 17 for REST APIs, business logic, and asynchronous order processing
- **Frontend:** Thymeleaf for server-rendered dashboards and monitoring pages
- **Database:** PostgreSQL for persistent order storage
- **Queue Management:** In-memory `LinkedBlockingQueue` for concurrent order execution
- **WebSocket Communication:** Spring WebSocket with STOMP for real-time status updates

2 Setup and Execution

2.1 Prerequisites

Before running the Order Execution Engine, ensure the following software is installed and configured on your system:

- **Java 17** – Required to run the Spring Boot backend.
- **Maven** – For building and managing project dependencies.
- **PostgreSQL** – Used to store order data and execution history.

2.2 Application Configuration

The application settings are defined in the `application.properties` file:

Application Configuration (application.properties)

```
spring.application.name=Order Execution Engine

# Server configuration
server.port=5000
server.address=0.0.0.0

# Database configuration
spring.datasource.url=jdbc:postgresql://${PGHOST}/${PGDATABASE}?sslmode=
    disable
spring.datasource.username=${PGUSER}
spring.datasource.password=${PGPASSWORD}
spring.datasource.driver-class-name=org.postgresql.Driver

# Hibernate/JPA configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.
    PostgreSQLDialect

# Thymeleaf template configuration
spring.thymeleaf.cache=false
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html

# Async task pool configuration
spring.task.execution.pool.core-size=10
spring.task.execution.pool.max-size=20
spring.task.execution.pool.queue-capacity=100
```

2.3 Running the Application Locally

To start the application locally, navigate to the project root and run:

```
mvn spring-boot:run
```

Once the application is running, the dashboard can be accessed via:

`http://localhost:5000`

2.4 Environment Variables

The application uses environment variables to configure database connectivity. Set the following variables before running the application:

- `PGHOST` – PostgreSQL host (e.g., localhost)
- `PGDATABASE` – Database name (e.g., order_engine)
- `PGUSER` – Database username
- `PGPASSWORD` – Database password

2.5 Additional Notes

- Order processing and DEX routing are handled using asynchronous thread pools, ensuring concurrent order execution.
- WebSocket endpoint: `/ws` is used for real-time updates.
- Orders updates are published to:
  - `/topic/orders` – Broadcast updates for all orders.
  - `/topic/orders/{orderId}` – Updates for a specific order.
- The engine includes retry logic with exponential backoff for temporary failures.

3 Frontend Overview

The frontend dashboard provides the following features:

- **Order Form:** Allows users to submit token pairs, amount, and slippage.
- **Live Orders:** Displays real-time order status, executed price, and transaction hash.
- **Queue Statistics:** Shows the number of active and queued orders.
- **Activity Log:** Displays WebSocket connection status and order-related events.

DEX Order Execution Demo

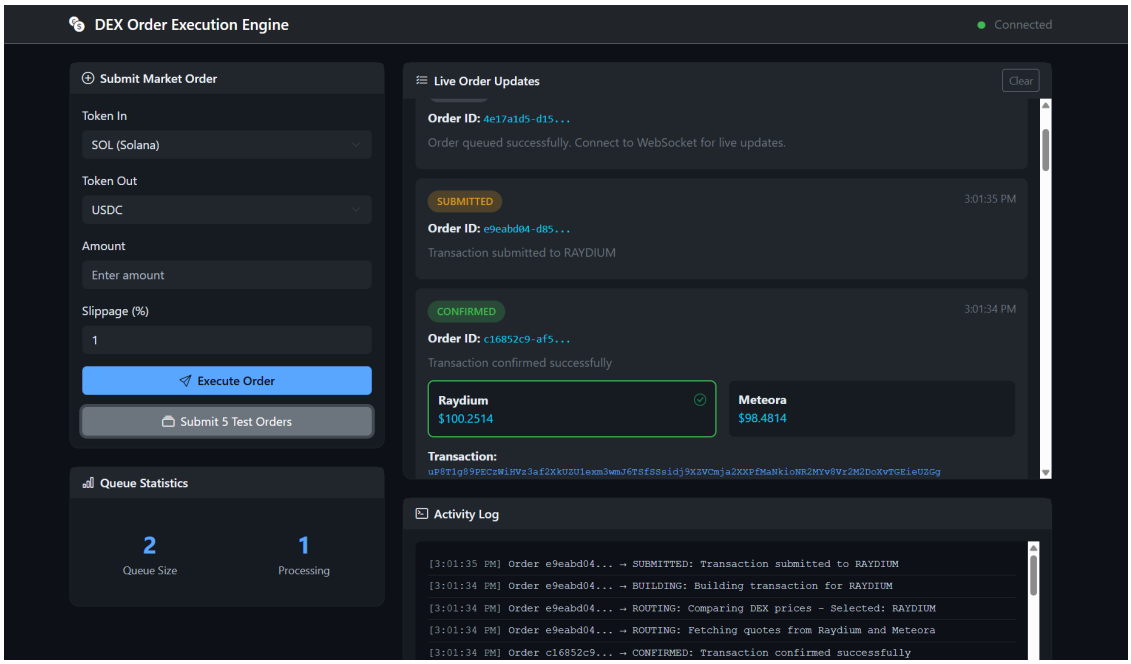

Figure 1: DEX Order Execution Engine in action: submitting and executing market orders.

Order Execution Flow Diagram

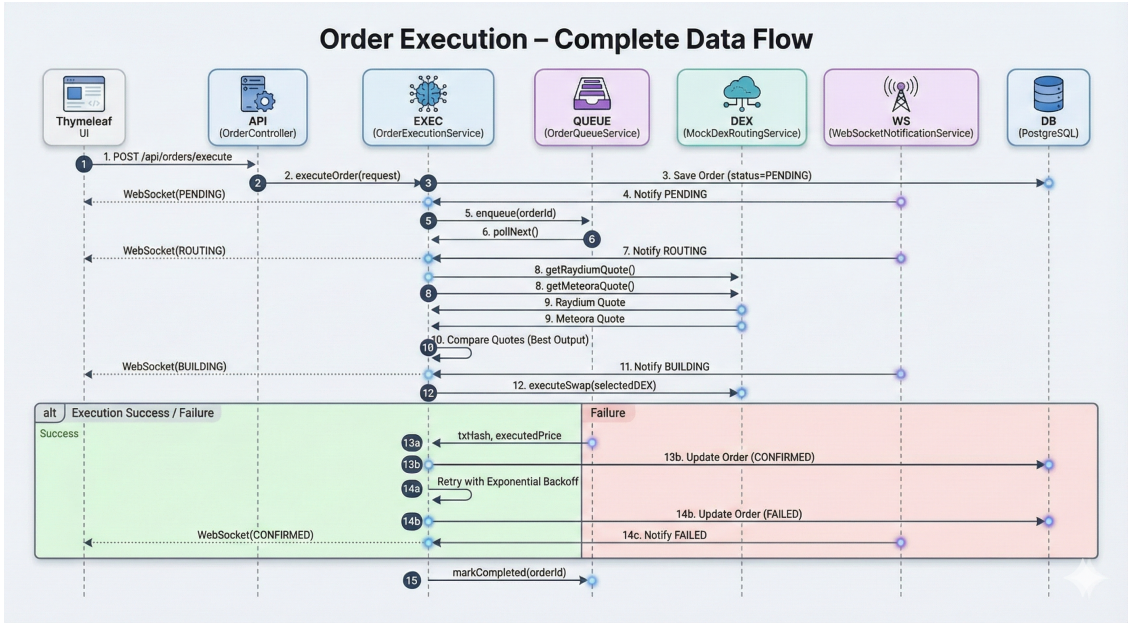

Figure 2: Complete order execution flow: shows how an order progresses from submission, through queue management and DEX routing, to final execution with real-time status updates.

4 Project Structure

The project follows a layered architecture that separates responsibilities across configuration, controller, service, data access, and model layers. This structure improves maintainability, scalability, and clarity of execution flow.

4.1 Project Directory Structure

The project is organized in a layered architecture, as shown below:

Project Directory Structure

```
src/main/java/com/dex/orderengine/
  OrderExecutionEngineApplication.java
 config/
    AsyncConfig.java
    WebSocketConfig.java
 controller/
    GlobalExceptionHandler.java
    OrderController.java
    WebController.java
 dto/
    DexQuote.java
    ExecutionResult.java
    OrderRequest.java
    OrderResponse.java
    WebSocketMessage.java
 model/
    DexType.java
    Order.java
    OrderStatus.java
    OrderType.java
 repository/
    OrderRepository.java
 service/
     MockDexRoutingService.java
     OrderExecutionService.java
     OrderQueueService.java
     WebSocketNotificationService.java
```

This structure clearly separates responsibilities across configuration, controllers, services, DTOs, models, and repositories.

4.2 Root Package

`com.dex.orderengine` is the base package containing the main application entry point and all submodules.

- `OrderExecutionEngineApplication.java`: Bootstraps the Spring Boot application and initializes all components.

4.3 Configuration Layer

The `config` package contains application-wide configuration classes.

- `AsyncConfig.java`: Configures thread pools for asynchronous order execution.
- `WebSocketConfig.java`: Defines WebSocket endpoints and message brokers for real-time updates.

4.4 Controller Layer

The `controller` package exposes REST and WebSocket endpoints to external clients.

- `OrderController.java`: Handles order submission and query APIs.
- `WebController.java`: Serves the dashboard and UI-related endpoints.
- `GlobalExceptionHandler.java`: Provides centralized exception handling and consistent error responses.

4.5 Data Transfer Objects (DTO)

The `dto` package defines objects used to transfer data between layers without exposing internal entities.

- `OrderRequest`: Represents incoming order parameters.
- `OrderResponse`: Represents order status and execution details.
- `DexQuote`: Encapsulates DEX price quote information.
- `ExecutionResult`: Represents the outcome of a swap execution.
- `WebSocketMessage`: Defines real-time notification payloads.

4.6 Model Layer

The `model` package defines persistent domain entities and enumerations.

- `Order`: Core entity representing an order and its lifecycle.
- `DexType`: Supported decentralized exchanges.
- `OrderStatus`: Execution states of an order.
- `OrderType`: Different order execution strategies.

4.7 Repository Layer

The `repository` package abstracts database access using Spring Data JPA.

- `OrderRepository`: Provides CRUD operations and query methods for orders.

4.8 Service Layer

The `service` package contains the core business logic of the system.

- `OrderExecutionService`: Manages the full order lifecycle.
- `OrderQueueService`: Controls queuing and concurrency limits.
- `MockDexRoutingService`: Simulates DEX routing and execution.
- `WebSocketNotificationService`: Sends real-time execution updates to clients.

5 Model and Entity Layer

The model layer defines the core domain objects of the order execution engine. These entities represent the persistent state of orders, execution metadata, and system-level enumerations. This layer acts as the foundation for the service, queue, and controller layers.

5.1 Order Entity

The `Order` entity represents a single trading instruction submitted by a user. Each order captures both input parameters and execution results throughout its lifecycle.

Identity and Tokens

- `id`: A unique identifier generated using UUID to ensure global uniqueness.
- `tokenIn`: The input token being sold.
- `tokenOut`: The output token being purchased.

These fields uniquely define the trading pair involved in the order.

Order Parameters

- `amount`: Quantity of the input token to be traded.
- `slippage`: Maximum acceptable price deviation during execution.

Order Classification

- `orderType`: Defines the execution logic of the order.
- `status`: Represents the current state in the order lifecycle.

These fields enable state-based processing and extensibility.

DEX Routing Information

- `selectedDex`: The decentralized exchange chosen for execution.
- `raydiumQuote`: Price quoted by Raydium.
- `meteorQuote`: Price quoted by Meteora.

Quote comparison follows a simple optimization rule:

$$SelectedDEX = \arg \max_{d \in \{Raydium, Meteora\}} Q_d \tag{1}$$

where Q_d is the expected output value from DEX d.

Execution Results

- `executedPrice`: Final execution price after slippage.
- `txHash`: Transaction identifier representing on-chain execution.

The transaction hash uniquely links the order to a blockchain transaction.

Failure Handling

- `errorMessage`: Reason for execution failure.
- `retryCount`: Number of retry attempts performed.

Retry logic allows controlled re-execution while preventing infinite loops.

Timestamps

- `createdAt`: Time of order creation.
- `updatedAt`: Last modification timestamp.
- `completedAt`: Time when execution finalized.

These timestamps enable performance analysis and auditing.

Lifecycle Hooks

- `@PrePersist`: Initializes default values and timestamps.
- `@PreUpdate`: Automatically updates modification time.

These hooks ensure data consistency without manual intervention.

5.2 DEX Type Enumeration

The `DexType` enumeration defines the supported decentralized exchanges:

- `RAYDIUM`
- `METEORA`

This abstraction allows routing logic to remain independent of specific DEX implementations.

5.3 Order Status Enumeration

The `OrderStatus` enumeration defines discrete execution states:

- `PENDING`: Order accepted and queued
- `ROUTING`: Fetching DEX quotes
- `BUILDING`: Transaction construction
- `SUBMITTED`: Sent to DEX
- `CONFIRMED`: Successfully executed
- `FAILED`: Execution terminated unsuccessfully

This forms a finite state machine governing order progression.

5.4 Order Type Enumeration

The `OrderType` enumeration defines execution strategies:

- `MARKET`: Immediate execution at best available price
- `LIMIT`: Execution when a target price is reached
- `SNIPER`: Execution triggered by specific on-chain events

This design allows new order types to be added without modifying core logic.

6 System Configuration

The engine uses two main configuration components: `WebSocketConfig` for real-time notifications and `AsyncConfig` for asynchronous order processing.

6.1 WebSocket Configuration

The `WebSocketConfig` class configures real-time WebSocket messaging using the STOMP protocol. This enables the server to push live updates about orders to clients without requiring them to poll the server repeatedly.

6.1.1 Enabling WebSocket Message Broker

- The annotation `@EnableWebSocketMessageBroker` activates WebSocket support and sets up a message broker.
- The broker allows:
  1. Broadcasting messages to multiple subscribers (`/topic`).
  2. Sending messages to specific users (`/queue`).
- This provides a structured and scalable way to deliver real-time updates for order events such as submission, routing, execution, and confirmation.

6.1.2 Message Broker Configuration

- `enableSimpleBroker("/topic", "/queue")` configures a simple in-memory broker for:
  - `/topic`: Broadcast messages to all subscribers (e.g., all clients monitoring orders).
  - `/queue`: Private messages sent to a single client (e.g., notifications for a specific order).
- `setApplicationDestinationPrefixes("/app")` defines the prefix used for messages sent from clients to server endpoints.
- `setUserDestinationPrefix("/user")` allows sending messages to individual users, enabling personalized notifications.

6.1.3 STOMP Endpoints Configuration

- The WebSocket endpoint `/ws` is exposed for client connections.
- `withSockJS()` provides a fallback mechanism for browsers that do not support native WebSocket, ensuring compatibility.
- `setAllowedOriginPatterns("*")` allows cross-origin WebSocket connections, which is important for clients hosted on different domains.

6.1.4 Purpose and Benefits

- Provides **low-latency, bidirectional communication** between the server and clients.
- Supports real-time notifications for:
  - Order status updates (e.g., PENDING, ROUTING, CONFIRMED, FAILED)
  - Quote comparisons and selected DEX
  - Execution results and transaction hashes
- Reduces the need for repeated HTTP polling, improving system efficiency and client experience.
- Ensures scalability by separating broadcast channels (`/topic`) from private messages (`/queue`).

6.1.5 Conceptual Representation

Mathematically, the WebSocket system can be represented as a message mapping function:

$$M : E \to C \tag{2}$$

where E represents server-side events (order updates, execution results) and C represents client subscribers. Each event $e \in E$ is mapped to one or more clients $c \in C$ based on the subscription type (broadcast or private), ensuring timely and relevant delivery of notifications.

6.2 Asynchronous Task Configuration

The `AsyncConfig` class in the order execution engine is responsible for defining and managing thread pools that handle tasks asynchronously. Asynchronous execution allows the system to process multiple orders concurrently without blocking the main application thread, which is crucial for maintaining high throughput and responsiveness in a decentralized exchange environment.

6.2.1 Enabling Asynchronous Execution

- The annotation `@EnableAsync` enables Spring's asynchronous method execution capability.
- Methods annotated with `@Async` run in separate threads provided by a configured `Executor`.
- This allows long-running tasks, such as fetching quotes or submitting transactions, to run concurrently without delaying other operations.

6.2.2 Order Executor Thread Pool

Order Executor Thread Pool

Bean: `orderExecutor`

Core Pool Size: 10 threads (always alive to handle incoming tasks)
Max Pool Size: 20 threads (scales for bursts of order execution tasks)
Queue Capacity: 100 tasks (orders waiting if all threads are busy)
Thread Name Prefix: `OrderExecutor-` (for logging and debugging)
Purpose: Handles all major order execution tasks:

[leftmargin=*]

- Fetching price quotes from multiple DEXs
- Selecting the best quote
- Building and submitting transactions

6.2.3 DEX Routing Executor Thread Pool

DEX Routing Executor Thread Pool

Bean: `dexRoutingExecutor`

Core Pool Size: 5 threads (always alive for routing tasks)
Max Pool Size: 10 threads (scales during peak routing operations)
Queue Capacity: 50 tasks (pending routing tasks)
Thread Name Prefix: `DexRouter-` (distinguishes routing tasks in logs)
Purpose: Dedicated for quote fetching and comparison between DEXs (Raydium, Meteora) without blocking order execution threads.

6.2.4 Concepts and Rationale

- **Separation of Executors:** Having distinct executors for order execution and DEX routing ensures that heavy, time-consuming order operations do not block lighter tasks like quote fetching and routing.
- **Concurrency and Throughput:** The effective concurrency of the system can be represented as:
$$C_{eff} = \min(ActiveThreads, MaxThreads) \tag{3}$$
where `Active Threads` represents threads currently executing tasks and `Max Threads` is the configured maximum for the pool. This ensures system resources are efficiently utilized.
- **Queueing and Backpressure:** Tasks exceeding the pool's active threads are stored in a bounded queue. If the queue fills up, additional tasks are rejected or delayed, preventing resource exhaustion.
- **Latency and Responsiveness:** By processing tasks asynchronously, the main application thread remains free to handle new client requests, maintaining low response times.

The `AsyncConfig` setup ensures that:

1. Orders are executed concurrently without blocking other tasks.
2. DEX quote retrieval and comparison occur in parallel, improving system efficiency.
3. Resource usage is predictable and controlled via thread pool sizing and queue limits.
4. High throughput is maintained while keeping response times low, crucial for real-time trading environments.

7 API Layer and Exception Handling

This section describes the external interfaces of the order execution engine and the mechanism used to handle errors consistently across the system. The API layer exposes REST and web endpoints for order submission, tracking, and monitoring, while ensuring robustness through centralized exception handling.

7.1 Global Exception Handling

The system uses a centralized exception handler to manage errors occurring during request processing. This ensures that all error responses follow a consistent structure and improves debuggability.

Validation Errors When an incoming request fails validation constraints, a validation exception is raised. Each invalid field and its corresponding error message are extracted and returned to the client.

- Invalid input fields are mapped to descriptive error messages.
- The response clearly indicates that validation has failed.
- HTTP status code `400 (Bad Request)` is returned.

This approach allows clients to correct input errors without ambiguity.

Generic Errors Any unexpected runtime exception is handled by a generic exception handler.

- The error is logged for diagnostic purposes.
- A standardized error response is returned to the client.
- HTTP status code `500 (Internal Server Error)` is used.

Centralized error handling prevents internal exceptions from leaking into the API layer.

7.2 Order REST Controller

The order controller exposes REST endpoints for creating and querying orders. It acts as the primary interface between external clients and the execution engine.

Order Execution Endpoint Clients submit orders using the following endpoint:

```
POST /api/orders/execute
```

- The request body is validated before processing.
- A new order is created and submitted to the execution pipeline.
- An immediate response is returned containing the order identifier and initial status.

This design decouples request handling from asynchronous execution.

Order Query Endpoint Clients can retrieve the current status of an order using:

```
GET /api/orders/{orderId}
```

- If the order exists, its latest state is returned.
- If not found, a `404 (Not Found)` response is sent.

Recent Orders Endpoint The system also provides an endpoint to retrieve recently processed orders:

```
GET /api/orders
```

This is useful for dashboards, analytics, and monitoring execution history.

Queue Statistics Endpoint Queue-level statistics are exposed via:

`GET /api/orders/queue/stats`

The response includes:

- Current queue size
- Number of active executions
- Maximum queue capacity
- Maximum concurrent processing limit

These metrics provide real-time visibility into system load.

7.3 Web Interface Controller

In addition to REST APIs, the system provides a simple web interface for monitoring purposes.

- The root endpoint renders a dashboard view.
- Recent orders and queue statistics are displayed.
- Data is fetched from the same service layer used by the REST API.

This controller demonstrates separation of concerns by keeping presentation logic independent from execution logic.

7.4 Design Considerations

- Controllers are thin and delegate all business logic to services.
- Exception handling is centralized and consistent.
- REST and web interfaces share the same execution core.
- The API layer remains stateless and scalable.

8 Service Layer

The service layer contains the core business logic of the DEX Order Execution Engine. It handles order submission, queue processing, DEX price routing, execution flow, retry handling, and real-time notifications. Each service has a clear responsibility and communicates with others to complete the full order lifecycle.

8.1 MockDexRoutingService

The `MockDexRoutingService` simulates the behavior of decentralized exchanges (DEXs) without connecting to a real blockchain network. It is used to generate price quotes, compare different DEXs, and simulate swap execution with realistic delays and failures.

8.1.1 DEX Price Quote Generation

This service generates price quotes from two decentralized exchanges, namely Raydium and Meteora. Both DEXs follow the same underlying mathematical model for quote generation, with minor variations in price range, trading fee, and response latency. A fixed base SOL price of **100.00** is used as the reference value for all calculations, while random variations are applied to simulate real market behavior.

Raydium and Meteora Quote Generation: Mathematical Model The objective of the quote generation logic is to estimate the amount of output tokens a user would receive when swapping a given input amount on a DEX. Since the implementation is a mock system, blockchain interaction is not performed. Instead, the behavior of real-world DEXs is approximated using probabilistic price movement, fee deduction, and latency modeling.

Input Parameters Let the input parameters be defined as:

- A : Amount of input tokens provided by the user
- T_{in} : Input token symbol
- T_{out} : Output token symbol

Only the input amount A directly affects the numerical computation. The token symbols are used for identification, logging, and client-side display.

Base Price Assumption A constant base price is assumed for the token pair:

$$P_0 = 100 \tag{4}$$

This value represents a reference market price (for example, 1 SOL = 100 units of another token). In a production system, this price would be obtained dynamically from on-chain liquidity pools or oracle services.

Market Price Fluctuation Modeling To simulate market volatility, a random price multiplier is applied. Let r be a random variable uniformly distributed in the interval $[0, 1)$.

For Raydium, the multiplier is defined as:

$$m_R = 0.98 + 0.04r \tag{5}$$

For Meteora, a slightly wider range is used:

$$m_M = 0.97 + 0.05r \tag{6}$$

The quoted prices are then computed as:

$$P_R = P_0 \times m_R \tag{7}$$

$$P_M = P_0 \times m_M \tag{8}$$

These variations represent short-term price changes caused by liquidity depth and trading activity on each DEX.

Trading Fee Modeling Each DEX applies a different trading fee:

- Raydium fee:

$$f_R = 0.003 \quad (0.3\%) \tag{9}$$

- Meteora fee:

$$f_M = 0.002 \quad (0.2\%) \tag{10}$$

The effective value received by the user after fees is given by $(1 - f)$.

Output Amount Computation The output amount O is calculated using the same formula for both DEXs:

$$O = A \times P \times (1 - f) \tag{11}$$

Substituting the corresponding price expressions:

$$O_R = A \times P_0 \times (0.98 + 0.04r) \times 0.997 \tag{12}$$

$$O_M = A \times P_0 \times (0.97 + 0.05r) \times 0.998 \tag{13}$$

These equations form the core mathematical model for DEX quote generation.

Numerical Precision All monetary values are rounded to six decimal places using half-up rounding:

$$scale = 6 \tag{14}$$

This ensures consistency with common precision standards used in cryptocurrency exchanges and avoids floating-point rounding errors.

Latency and Response Time Modeling To simulate network and API delays, each DEX introduces a random response time:

- Raydium latency: 150–250 ms
- Meteora latency: 180–300 ms

The response time t_r is computed as:

$$t_r = t_{end} - t_{start} \tag{15}$$

This value is included in the quote for performance analysis and routing decisions.

Role in DEX Routing Once quotes are received from both Raydium and Meteora, the execution engine compares their output amounts. The DEX that provides the higher output value is selected for order execution. This approach ensures that the user receives the maximum possible return for the submitted order.

Mathematical Summary The generalized quote generation model used by both DEXs can be expressed as:

$$O(A) = A \times P_0 \times m \times (1 - f) \tag{16}$$

where the multiplier m and fee f depend on the selected DEX. This unified formulation captures market volatility, fee structures, and execution realism while avoiding redundant logic across different DEX implementations.

8.1.2 Best DEX Selection and Swap Execution

After receiving price quotes from multiple DEXs, the execution engine performs routing and swap execution in two stages: (i) selecting the optimal DEX based on quote comparison and (ii) simulating the execution of the swap on the selected DEX.

Best Quote Selection Logic Let O_R and O_M denote the output amounts obtained from Raydium and Meteora respectively. The routing decision is based on a simple comparison of these values.

The selection rule is defined as:

$$SelectedDEX = \{\ Raydium, if O_R > O_M Meteora, otherwise \tag{17}$$

This strategy ensures that the DEX offering the maximum output amount is chosen for execution. Since both quotes are computed using a consistent mathematical model, comparing output values provides a reliable and fair routing decision.

Swap Execution Simulation Once the optimal DEX is selected, the system simulates the execution of the swap. In real decentralized exchanges, transaction execution involves network propagation, validator confirmation, and block finalization. These factors are modeled in the mock implementation through execution delay, probabilistic failure, and price slippage.

Execution Delay Modeling To simulate blockchain confirmation time, a random execution delay is introduced:

$$ExecutionDelay \in [2000, 3000]ms \tag{18}$$

This delay represents transaction submission, processing, and confirmation on the network.

Execution Failure Modeling Blockchain transactions are not guaranteed to succeed due to network congestion or timeout errors. To reflect this uncertainty, a failure probability is introduced:

$$P(Failure) = 0.05 \tag{19}$$

If a failure occurs, the execution is marked unsuccessful and an appropriate error message is returned. This helps test retry logic and failure handling in the order execution pipeline.

Slippage Modeling Quoted prices may differ from executed prices due to market movement during transaction confirmation. To simulate this effect, a slippage factor is applied.

Let s be a random variable in the range $[0, 0.01)$. The slippage factor is defined as:

$$\lambda = 1 - s \tag{20}$$

The executed price P_e is then calculated as:

$$P_e = P_q \times \lambda \tag{21}$$

where P_q is the quoted price. This models price deterioration of up to 1% during execution.

Transaction Hash Generation Upon successful swap execution, a mock transaction hash is generated to emulate a real on-chain transaction identifier. This hash serves as a unique reference for tracking execution status, logging events, and presenting confirmation details to the user.

The transaction hash is constructed as a fixed-length string of 88 characters, where each character is selected uniformly at random from the Base58 character set. Base58 encoding is commonly used in Solana to avoid visually ambiguous characters and improve human readability.

Let Σ denote the Base58 character set. The generated transaction hash $TxHash$ is defined as:

$$TxHash \in \Sigma^{88} \tag{22}$$

Each character in the hash is independently sampled, resulting in a very low probability of collision. Although no real blockchain transaction is performed, this approach produces realistic transaction identifiers that closely resemble actual Solana transaction hashes, making the mock execution flow suitable for testing and demonstration purposes.

Execution Result Representation The outcome of the swap execution is encapsulated in an execution result object containing:

- Execution status (success or failure)
- Selected DEX
- Executed price
- Transaction hash (if successful)
- Error message (if failed)

This structured result allows the system to update order status, notify users in real time, and maintain execution history.

Significance of Execution Modeling By simulating routing, execution delay, failure probability, and slippage, this module closely approximates real-world decentralized exchange behavior. It enables thorough testing of order lifecycle management, WebSocket notifications, and failure recovery logic without requiring actual blockchain interaction.

8.2 OrderExecutionService: Mock Order Lifecycle Management

The `OrderExecutionService` is the central component responsible for managing the complete lifecycle of a market order. It coordinates order submission, queue-based execution, DEX routing, retry handling, and final order settlement. This service integrates persistence, asynchronous execution, and real-time notifications to simulate a realistic decentralized trading environment.

8.2.1 Order Submission

submitOrder() The `submitOrder()` method is responsible for accepting a new market order request from the user and initializing the execution workflow. Upon receiving an order request, a new order entity is created with an initial status of `PENDING`.

Key actions performed by this method include:

- Persisting the order in the database
- Assigning an initial retry count of zero
- Enqueuing the order for asynchronous execution

If the execution queue is full, the order is immediately marked as `FAILED`. Otherwise, the order is successfully queued and a WebSocket notification is sent to inform the user that execution will begin shortly. This separation ensures that order submission remains fast and non-blocking.

8.2.2 Queue Processing

processQueue() The `processQueue()` method runs periodically using a scheduled task. Its responsibility is to poll the next order identifier from the execution queue and initiate asynchronous processing.

This design introduces controlled execution throughput and prevents system overload by ensuring that orders are processed sequentially or within defined concurrency limits.

executeOrderAsync() Once an order identifier is retrieved from the queue, execution is delegated to an asynchronous worker thread. This prevents long-running operations, such as network delays or retries, from blocking the main scheduling thread.

If the order cannot be found in persistent storage, execution is safely aborted and the order is marked as failed in the queue.

8.2.3 Order Routing and Execution

executeOrder() The `executeOrder()` method contains the core execution logic. The order is first moved into the `ROUTING` state, indicating that price discovery is in progress.

Price quotes are requested concurrently from both Raydium and Meteora. Let:

- Q_R denote the Raydium quote
- Q_M denote the Meteora quote

Once both quotes are available, the service selects the optimal DEX by comparing their output amounts:

$$\max(Q_R, Q_M)$$

The selected DEX is stored in the order and communicated to the client via WebSocket.

Transaction Building and Submission After routing, the order transitions through the following states:

- `BUILDING`: Transaction preparation
- `SUBMITTED`: Transaction submission to the selected DEX

A short delay is introduced during transaction building to simulate real-world transaction preparation overhead.

Swap Execution The actual swap execution is delegated to the routing service. Execution success is modeled probabilistically, with a fixed failure probability to simulate network issues and timeouts.

If execution succeeds, the order is marked as `CONFIRMED`. The executed price and transaction hash are recorded, and the order completion timestamp is set.

8.2.4 Retry and Failure Handling

handleRetry() If execution fails, the system applies a retry mechanism with exponential backoff. Let:

- n be the current retry attempt
- D_0 be the initial delay

The retry delay is computed as:

$$D_n = D_0 \times 2^{(n-1)}$$

This strategy reduces system pressure during transient failures while allowing recovery from temporary network issues.

If the retry count exceeds the maximum allowed attempts, the order is marked as `FAILED` and no further execution attempts are made.

handleExecutionFailure() This method serves as a safety mechanism to ensure that unexpected execution errors still trigger the retry logic. It guarantees that all execution failures are handled consistently.

8.2.5 Order Querying

getOrder() and getRecentOrders() These methods provide read-only access to order data. They allow users or dashboards to retrieve the status and execution details of a specific order or view recent orders for monitoring purposes.

8.2.6 Response Mapping

mapToResponse() The `mapToResponse()` method converts the internal order entity into a response-friendly format. It includes execution details such as:

- Selected DEX
- Quoted and executed prices

- Transaction hash
- Error messages (if any)

This ensures a clean separation between internal persistence models and external API responses.

8.2.7 Design Rationale

The mock-based design of the `OrderExecutionService` allows:

- End-to-end testing of order execution flow
- Validation of routing and retry strategies
- Realistic simulation without blockchain dependency

This makes the service suitable for academic analysis and backend system evaluation while keeping implementation complexity manageable.

8.3 OrderQueueService: Controlled Order Scheduling and Concurrency Management

The `OrderQueueService` is responsible for managing the scheduling and controlled execution of orders within the system. Its primary role is to regulate how many orders can be queued and processed concurrently, thereby preventing system overload and ensuring stable execution behavior. This service acts as an intermediate layer between order submission and order execution.

8.3.1 Design Objectives

The queue service is designed with the following objectives:

- Limit the total number of pending orders
- Restrict the number of concurrently executing orders
- Maintain thread-safe access to shared order state
- Provide visibility into queue and execution statistics

These constraints closely resemble real-world exchange engines, where execution capacity is finite and must be carefully managed.

8.3.2 Queue and Concurrency Constraints

Two system-wide limits are enforced:

- Maximum queue size: $Q_{\max} = 100$
- Maximum concurrent executions: $C_{\max} = 10$

The queue size limit ensures that the system does not accept more orders than it can reasonably handle, while the concurrency limit caps the number of orders being executed at the same time.

8.3.3 Queue Data Structures

Orders are managed using the following data structures:

- A bounded FIFO queue for pending orders
- A concurrent map for actively executing orders
- An atomic counter for tracking concurrent execution count

The bounded queue enforces backpressure, while the concurrent map allows fast lookup of active orders in a thread-safe manner.

8.3.4 Order Enqueuing

enqueue() When a new order is submitted, it is added to the queue using the `enqueue()` method. Let Q denote the current queue size. An order is accepted if and only if:

$$Q < Q_{\max}$$

If this condition is violated, the order is rejected and marked as failed. Upon successful insertion, the order identifier is stored in the queue, and the order object is tracked in the active orders map.

This approach ensures that memory usage remains bounded and that excessive order submission does not degrade system performance.

8.3.5 Order Dequeuing and Execution Control

pollNext() The `pollNext()` method retrieves the next order for execution. Execution is permitted only if the number of currently processing orders C satisfies:

$$C < C_{\max}$$

If this condition holds, the next order identifier is removed from the queue and the processing counter is incremented atomically:

$$C \leftarrow C + 1$$

This mechanism guarantees that at no point does the system execute more than $C_{\max}$ orders concurrently.

8.3.6 Order Completion and Failure Handling

markCompleted() When an order finishes execution successfully, it is removed from the active orders map and the processing counter is decremented:

$$C \leftarrow C - 1$$

This signals that system capacity has been freed and allows queued orders to proceed.

markFailed() In case of execution failure, the same decrement operation is performed. Treating failures and completions uniformly ensures that system capacity is always accurately tracked.

8.3.7 Active Order Tracking

getActiveOrder() The service maintains a real-time view of active orders through a concurrent map. This allows other components, such as monitoring or retry logic, to inspect the state of an executing order without blocking execution threads.

8.3.8 Queue Metrics and Monitoring

Queue Statistics The service exposes runtime metrics including:

- Current queue size
- Number of active executions
- Total active orders
- Configured concurrency limits

These metrics enable system monitoring, debugging, and performance evaluation, and they can be easily extended for dashboard visualization.

8.3.9 Thread Safety and Concurrency Guarantees

Thread safety is ensured through:

- Use of a thread-safe blocking queue
- Concurrent hash maps for shared state
- Atomic counters for concurrency tracking

These choices eliminate race conditions and ensure consistent behavior under concurrent access.

8.3.10 Why This Queue Design Works

This queue-based execution model provides:

- Predictable system load
- Controlled parallelism
- Natural backpressure during peak demand
- Realistic simulation of exchange order processing

By separating order scheduling from order execution, the system achieves modularity, scalability, and clarity, making it suitable for both academic study and backend system evaluation.

8.4 WebSocketNotificationService: Real-Time Order Status Communication

The `WebSocketNotificationService` is responsible for delivering real-time updates about order execution to connected clients. Instead of relying on repeated REST API calls, this service pushes updates automatically whenever the state of an order changes. This design improves responsiveness and reduces unnecessary network traffic.

8.4.1 Purpose of WebSocket-Based Communication

Order execution is a multi-stage and time-consuming process that includes routing, transaction building, submission, and confirmation. Let an order pass through a sequence of states:

$$S = \{PENDING, ROUTING, BUILDING, SUBMITTED, CONFIRMED, FAILED\}$$

WebSocket communication ensures that every transition between these states is immediately delivered to the client as an event, rather than requiring the client to poll for updates.

8.4.2 Message Abstraction

All updates are encapsulated using a common data structure called `WebSocketMessage`. Each message represents a snapshot of the order state at a specific time.

Formally, a WebSocket message can be represented as:

$$M = (o, s, m, t, d)$$

where:

- o is the order identifier
- s is the current order status
- m is a human-readable message
- t is the timestamp of the event
- d represents optional execution details

This unified message structure keeps communication consistent across different order events.

8.4.3 Order Status Notifications

notifyOrderStatus() This method is used for general order lifecycle updates such as order submission, routing start, transaction building, and submission. When the order transitions to a new status, a corresponding WebSocket message is created and broadcast.

Mathematically, this represents a state transition:

$$S_i \rightarrow S_{i+1}$$

Each transition triggers the emission of a message to subscribed clients.

8.4.4 DEX Routing Notifications

notifyRouting() During the routing phase, price quotes from Raydium and Meteora are compared. This method sends a message containing:

- Raydium quoted price
- Meteora quoted price
- Selected DEX

This provides transparency into the routing decision and allows users to observe how the best execution venue was chosen.

8.4.5 Order Confirmation Notifications

notifyConfirmed() When a swap is executed successfully, a confirmation message is sent. This message includes execution-specific data such as:

- Selected DEX
- Executed price
- Transaction hash
- Original price quotes

This message marks the terminal success state of the order:

$$S_{final} = CONFIRMED$$

8.4.6 Failure Notifications

notifyFailed() If an order fails after all retry attempts, a failure message is sent containing an error description. This represents the terminal failure state:

$$S_{final} = FAILED$$

Explicit failure notifications ensure that users are informed immediately and clearly.

8.4.7 Message Broadcasting Strategy

Each WebSocket message is delivered to two destinations:

- An order-specific channel: `/topic/orders/{orderId}`
- A global monitoring channel: `/topic/orders`

This dual-channel approach supports both individual order tracking and system-wide monitoring dashboards.

8.4.8 Timestamping and Event Ordering

Each message includes a timestamp $t \in R^+$ representing the event time. Clients can use these timestamps to order events chronologically and reconstruct the full order lifecycle.

8.4.9 Advantages of This Design

This WebSocket-based notification system provides:

- Real-time feedback to users
- Clear visibility into order progress
- Reduced API polling overhead
- Clean separation between execution logic and communication logic

By decoupling execution from notification, the system remains scalable, responsive, and easy to extend.

9 Order Lifecycle

The order lifecycle describes the complete sequence of stages that an order passes through, starting from client submission and ending with final execution or failure. This section provides a high-level view of the system behavior by connecting the previously discussed components into a single execution flow.

9.1 Order Submission

- The client submits a market order through the REST API endpoint `/api/orders/execute`.
- The `OrderExecutionService` validates the request and creates a new order record in the database.
- The order is initialized with status `PENDING` and added to the execution queue.
- A WebSocket notification is immediately sent to confirm successful order submission.

9.2 Queue Management and Scheduling

- The `OrderQueueService` manages a bounded queue with a fixed maximum capacity.
- Orders are dequeued only when execution slots are available, ensuring controlled concurrency.
- If the queue is full, new orders are rejected to prevent system overload.

9.3 Order Routing and Quote Evaluation

- The system requests price quotes from multiple decentralized exchanges, namely Raydium and Meteora.
- Quote retrieval occurs asynchronously to minimize latency.
- Each quote contains pricing, fee, expected output, and response time information.
- The routing logic selects the DEX that offers the highest expected output amount.
- Clients are notified in real time about the routing decision.

9.4 Transaction Preparation

- After routing, the order status transitions to `BUILDING`.
- The system simulates transaction construction using the selected DEX parameters.
- This step ensures that execution conditions such as slippage are finalized.

9.5 Transaction Submission and Execution

- The transaction is submitted to the selected DEX and the order status is updated to `SUBMITTED`.
- Execution is simulated asynchronously with realistic network delays.
- Slippage is applied to model real-world price movement during execution.

9.6 Confirmation and Failure Handling

- If execution succeeds:
  - The order status becomes `CONFIRMED`.
  - Executed price and transaction hash are recorded.
  - A final WebSocket notification is sent to the client.
- If execution fails:
  - The system retries execution using exponential backoff.
  - A maximum of three retry attempts are allowed.
  - After all retries fail, the order is marked as `FAILED`.
  - Failure details are communicated to the client via WebSocket.

9.7 Post-Execution Processing

- Completed or failed orders are removed from active execution tracking.
- Order data remains stored for future queries and analysis.
- Clients can retrieve order details using REST APIs or WebSocket subscriptions.

9.8 Summary of Status Transitions

Status	**Description**
PENDING	Order received and queued
ROUTING	Fetching and comparing DEX quotes
BUILDING	Transaction under construction
SUBMITTED	Transaction submitted for execution
CONFIRMED	Order executed successfully
FAILED	Execution failed after retries

10 Extending the Order Execution Engine

The current implementation of the order execution engine focuses on market orders to demonstrate the complete execution pipeline, including queuing, routing, execution, and real-time notifications. However, the modular design of the system allows it to be easily extended to support additional order types and advanced trading strategies.

10.1 Limit Orders

Limit orders allow users to specify a target price at which the order should be executed. Unlike market orders, these orders are not executed immediately.

- The system continuously monitors market prices obtained from supported DEXs.
- A limit order is triggered only when the observed market price satisfies the user-defined condition.
- Once the condition is met, the order enters the standard execution pipeline, including routing, transaction building, and execution.
- This approach ensures precise price control while reusing existing execution logic.

10.2 Sniper Orders

Sniper orders are designed to execute immediately when a specific market event occurs, such as the launch of a new token or the creation of a new liquidity pool.

- The engine listens for on-chain or off-chain events related to token launches or liquidity pool creation.
- Upon detecting the target event, the sniper order is triggered with minimal latency.
- The order bypasses extended price monitoring and is executed as a high-priority market order.
- This order type is suitable for time-sensitive trading strategies where execution speed is critical.

10.3 Design Benefits

- The queue-based execution model naturally supports multiple order types.
- Routing, execution, and notification components remain unchanged.
- New order strategies can be introduced without modifying core system behavior.