An Architect's Guide to Equity Matching Engines: From First Principles to High-Performance Implementation

Section 1: The Central Limit Order Book (CLOB) - The Foundation of Modern Markets

An equity matching engine does not exist in a vacuum; it is the heart of a specific market model designed to achieve fairness, transparency, and efficiency. To understand how to build a matching engine, one must first understand the environment it serves. For the vast majority of modern electronic equity exchanges, this environment is the Central Limit Order Book, or CLOB.[1] The CLOB is not merely a data structure but a comprehensive trading method that acts as the central nervous system for price discovery. Its design principles dictate the fundamental requirements and purpose of the matching engine itself.

1.1. Defining the CLOB

A Central Limit Order Book is a system used by most global exchanges to execute trades. It functions as a centralized repository where all buy and sell orders for a given financial instrument are collected and displayed.[2] At its core, the CLOB consists of two key components: the order book itself and a matching engine.[1] The order book is the transparent, real-time ledger of all outstanding intentions to trade. The matching engine is the automated process that consumes these intentions and executes trades based on a strict, predetermined set of rules.

This model allows for a diverse range of market participants—including institutional investors, hedge funds, market makers, and retail traders—to interact directly or indirectly within a single, unified marketplace.[2] An order submitted to a CLOB can be a

market order, which seeks immediate execution at the best available price, or a *limit order*, which specifies a maximum price a trader is willing to pay (for a buy) or a minimum price they are willing to accept (for a sell).[2] It is the collection of these unexecuted limit orders that constitutes the "book."

1.2. The Pillars of the CLOB: Transparency, Liquidity, and Fairness

The dominance of the CLOB model stems from three core advantages it provides to the market. These are not just desirable features; they are the philosophical underpinnings that the matching engine is built to enforce.

- **Transparency:** The most significant advantage of a CLOB is the unparalleled level of transparency it offers. All market participants have access to the same critical information in real-time.[2] This includes the prices and volumes of all visible buy orders (bids) and sell orders (offers or asks). This visible list of orders, ranked by price, is known as the "market depth" or the "stack".[3] By eliminating information asymmetry, the CLOB ensures that prices are determined purely by the forces of supply and demand, fostering trust and encouraging broader participation.[2] The highest bid and the lowest offer at any given moment constitute the "best market" or "the touch," representing the tightest possible spread for a trade.[3]
- **Liquidity:** By aggregating order flow from a vast and diverse pool of participants, a CLOB creates deep liquidity.[2] Liquidity refers to the ease with which an asset can be bought or sold without causing a significant change in its price. In a liquid market, there are many buyers and sellers, which typically leads to tighter bid-ask spreads (the difference between the best bid and the best offer). This enhanced liquidity allows traders to execute orders more quickly and at more favorable prices, which is particularly crucial for institutional investors and high-frequency trading firms that deal in large volumes.[2]
- **Fairness:** The CLOB operates on a deterministic and publicly known set of rules, ensuring a level playing field for all participants. The most common rule set is **'price-time priority'**.[1] This principle dictates that orders are matched first based on the best price, and then, for orders at the same price, based on who submitted their order first. This simple, unwavering logic rewards traders for offering better prices and for being early, promoting active price discovery and competition.[5] The matching engine's primary role is to be the impartial and unerring enforcer of this rule, guaranteeing that no participant receives preferential treatment outside of

these established principles.

1.3. How the CLOB Functions in Practice

The operational flow of a CLOB is straightforward. When a trader submits an order, it enters the system and is handled by the matching engine.[6]

1. **Order Submission:** A participant sends an order to the exchange, for example, a limit order to "Buy 100 shares of XYZ at $50.10 or lower."
2. **Matching Attempt:** The matching engine immediately checks the opposite side of the order book. In this case, it looks at the sell orders (the "ask" side). If there is one or more sell orders with a price at or below $50.10, a trade is executed. The engine will match against the best-priced sell order first (e.g., one at $50.09) before moving to a worse price (e.g., $50.10), following price priority.
3. **Resting in the Book:** If no matching sell order is available (e.g., the best available sell price is $50.11), the buy order is not executed. Instead, it is placed in the buy side of the order book (the "bid" side) to "rest," where it waits for a new sell order to arrive that is willing to meet its price.[2] This resting order now contributes to the market's visible liquidity.
4. **Execution:** The order remains in the book until it is either matched by an incoming sell order, canceled by the trader, or expires based on its time-in-force conditions.

This continuous process of orders arriving, matching, and resting is what drives price movement and facilitates trade in a CLOB-based market.

1.4. Contextualizing the CLOB - Comparison with Alternative Models

Understanding the CLOB is sharpened by contrasting it with other market structures. Its design represents a deliberate departure from older, more opaque models.

The primary alternative is the **Request For Quote (RFQ)** model.[3] In an RFQ system, a customer who wants to trade does not post their order to a public book. Instead, they query a finite, private group of dealers or market makers, who then respond with their individual bid and offer prices. The customer can then choose to trade with the dealer

offering the best price.

The differences are profound. The RFQ model is asymmetric and dealer-centric. Customers can only trade with dealers; they cannot trade directly with other customers, nor can they become price-makers themselves by posting a public limit order.[3] This preserves the privileged position of the dealer as an intermediary. In contrast, the CLOB is a more democratic model. By allowing any participant to post an order on the public book, it enables customer-to-customer, dealer-to-dealer, and customer-to-dealer trading on equal footing.[3] Any participant who places a limit order is, in effect, acting as a market maker for that quantity.

This distinction is fundamental. While other asset classes like foreign exchange (FX) and fixed income may use a bifurcated structure with CLOBs for inter-dealer (D2D) markets and RFQ systems for dealer-to-customer (D2C) markets [7], the equity market is dominated by the CLOB philosophy. Therefore, when setting out to build an equity matching engine, the developer is not just creating a piece of software; they are building the core enforcement mechanism for a market structure founded on the principles of open access, universal transparency, and impartial, rule-based execution.

Section 2: The Language of Trading - A Comprehensive Taxonomy of Order Types and Attributes

A matching engine is, at its heart, an instruction processor. Its sophistication and utility are defined by the variety and complexity of the instructions—the "order types"—it can understand and execute. For a developer, a deep and precise understanding of this "language of trading" is a non-negotiable prerequisite for designing the engine's core logic. The evolution from simple commands to buy and sell into a rich vocabulary of conditional and attribute-based orders reflects the growing needs of traders for granular control over risk, cost, and execution strategy.

2.1. Foundational Order Types

These are the two most fundamental instructions that form the basis of all trading activity.

- **Market Order:** This is an instruction to buy or sell a security immediately at the best available price in the market.[8] A market buy order will execute against the lowest-priced sell orders currently in the book, while a market sell order will execute against the highest-priced buy orders. The primary advantage of a market order is that it guarantees execution; the trader can be certain their order will be filled (assuming sufficient liquidity exists).[8] The disadvantage is the lack of price certainty. In a fast-moving or volatile market, the price at which the order executes may be significantly different from the last-traded price the trader observed, a phenomenon known as "slippage".[9] For the engine, processing a market order involves "walking the book"—consuming liquidity from one price level and moving to the next worse price level until the order's quantity is fully satisfied.[11]
- **Limit Order:** This is an instruction to buy or sell a security at a specific price or better.[8] A buy limit order can only be executed at the specified limit price or lower, while a sell limit order can only be executed at the limit price or higher.[8] For example, a buy limit order for 100 shares at $10.00 will only execute if a seller is available at or below $10.00. This order type provides price control but sacrifices execution certainty. If the market price never reaches the limit price, the order will not be filled.[9] Limit orders are the lifeblood of a CLOB; the unexecuted, "resting" limit orders are what form the visible order book that provides liquidity to the market.[12]

2.2. Conditional and Triggered Orders

These orders are more complex, as they are not immediately active upon submission. Instead, they lie dormant and are activated by the matching engine only when specific market conditions are met. This requires the engine to constantly monitor the state of the market (e.g., the last traded price) to trigger these orders.

- **Stop Order (or Stop-Loss Order):** This is an order to buy or sell a security that is triggered when the market price reaches a specified "stop price".[8] Once triggered, a stop order becomes a **market order** and is executed immediately at the best available price.[9]
  - A *sell stop order* is placed below the current market price and is typically used to limit a loss or protect a profit on a stock that is owned.[8]

    - A *buy stop order* is placed above the current market price and is often used to limit losses on a short position.[8]

      The key feature is that, like a market order, execution is guaranteed once triggered, but the exact execution price is not.10
- **Stop-Limit Order:** This is a hybrid that combines the features of a stop order and a limit order.[10] The trader specifies two prices: a stop price and a limit price. When the market reaches the stop price, the order is triggered, but instead of becoming a market order, it becomes a **limit order** at the specified limit price.[9] This gives the trader protection against poor execution prices in a volatile market but introduces the risk that the order may not be filled at all if the market moves quickly past the limit price after being triggered.[13] For example, a trader might place a buy stop-limit order with a stop at $110 and a limit at $115. If the stock price rises to $110, a limit order to buy at $115 or better is activated.[10]
- **Market/Limit If Touched (MIT/LIT):** These are similar to stop orders but are typically used to enter the market when a certain price level is reached, rather than to exit a position. They are submitted with a trigger price, and when the market reaches or passes that price, a market (MIT) or limit (LIT) order is submitted.[11] A key difference from stop orders is their placement relative to the market: Buy MIT/LIT orders are placed *below* the current market price, and Sell MIT/LIT orders are placed *above* the current market price.[11]

2.3. Time-in-Force (TIF) Qualifiers

TIF attributes are attached to orders to define how long they should remain active before being expired by the matching engine.

- **Day Order:** This is the default for many systems. The order is active only for the current trading session and is automatically canceled by the engine if it is not executed by the market close.[9]
- **Good 'Til Canceled (GTC):** The order remains active across multiple trading sessions until it is either fully executed or explicitly canceled by the trader.[9] To prevent orders from remaining in the system indefinitely, most exchanges impose a maximum duration on GTC orders, such as 180 calendar days.[13]
- **Immediate or Cancel (IOC):** This instruction requires the engine to execute as

much of the order as possible immediately upon receipt. Any portion of the order that cannot be filled instantly against resting orders in the book is canceled.[10] Partial fills are allowed.[10] This is used by traders who want to capture whatever liquidity is available at a specific moment without leaving a resting order on the book.

- **Fill or Kill (FOK):** This is a stricter version of IOC. It requires the order to be executed **immediately and in its entirety**.[11] If the full quantity of the order cannot be filled against resting orders at that moment, the entire order is canceled. No partial fills are permitted.[10]
- **Book or Cancel (BOC) / Post-Only:** This is a limit order with a special instruction: the order should be canceled if it would execute immediately upon entry (i.e., if it would "take" liquidity).[11] The purpose of this order type is to ensure that the trader's order only "adds" liquidity to the book by becoming a resting order. This is strategically important in markets where exchanges offer lower fees (or even rebates) to "makers" of liquidity compared to the "takers" of liquidity. A post-only order guarantees the trader will be classified as a maker.[11]

2.4. Advanced and Synthetic Order Types

As trading strategies have become more complex, a range of advanced order types have been developed, placing further demands on the matching engine's logic.

- **Iceberg Order:** This is a mechanism for executing a large limit order without revealing the total size to the market, thus minimizing price impact.[11] The trader specifies a total quantity and a smaller "display" quantity. The matching engine only shows the display quantity on the public order book. When this visible "slice" is fully executed, the engine automatically reveals the next slice until the total quantity is filled.[11] The engine must internally track both the visible working quantity and the total remaining undisclosed quantity.
- **All-or-None (AON):** Similar to FOK in that it requires the entire order quantity to be filled, but it does not require immediate execution.[10] An AON order can rest in the book until sufficient liquidity becomes available to fill the entire order in a single transaction.
- **Market on Open (MOO) / Market on Close (MOC):** These are market orders specifically designed to be executed during an exchange's opening or closing auction period, rather than during the continuous trading session.[11] This implies that a sophisticated matching engine must support different market states or

phases (e.g., Pre-Open, Continuous Trading, Closing Auction), each with potentially different matching rules.

- **One-Cancels-Other (OCO):** This is not a single order but a pair of linked orders. When one of the orders is executed, the matching engine must automatically cancel the other.[11] A common use case is to place a limit order to take profits and a stop order to limit losses on the same position. The OCO instruction automates the management of this bracket.

The breadth of these order types demonstrates that a matching engine cannot be a simple processor of buy and sell commands. It must be an intricate state machine, capable of managing orders with complex lifecycles, conditional triggers, interdependencies, and attributes that change their behavior based on market events and fee structures. An architect designing such a system must plan for this complexity from the outset, creating a flexible framework that can be extended to support new order types as trading strategies and market structures inevitably continue to evolve.

The following table provides a consolidated reference for the developer, summarizing the key characteristics of the order types the engine must handle.

Order Type	Mnemonic	Key Parameters	Execution Logic	Allows Partial Fill?	Typical Use Case
Market	MKT	Side, Quantity	Executes immediately at best available price(s).	Yes	Certainty of execution, speed is priority.
Limit	LMT	Side, Quantity, Price	Executes at specified price or better. Rests in book if not immediately fillable.	Yes	Price control is priority. Adding liquidity.
Stop / Stop-Loss	STP	Side, Quantity, Stop Price	Becomes a Market order when stop price is reached.	Yes	Limiting losses or protecting profits on an existing position.

Stop-Limit	STP LMT	Side, Quantity, Stop Price, Limit Price	Becomes a Limit order when stop price is reached.	Yes	Limiting losses with control over execution price, avoiding slippage.
Immediate or Cancel	IOC	Side, Quantity, Price	Executes immediately against available liquidity; any unfilled portion is canceled.	Yes	Capturing available liquidity without leaving a resting order.
Fill or Kill	FOK	Side, Quantity, Price	Must execute immediately and in its entirety; otherwise, the entire order is canceled.	No	Ensuring a full-sized execution at a specific moment.
Book or Cancel / Post-Only	BOC	Side, Quantity, Price	A limit order that is canceled if it would execute on entry. Only adds liquidity.	No (on entry)	Guaranteeing "maker" status to receive fee rebates or avoid taker fees.
Iceberg	-	Side, Quantity, Price, Display Quantity	A large limit order where only the smaller "display quantity" is visible at a time.	Yes	Hiding total order size to minimize market impact.

Section 3: The Rules of Engagement - A Deep Dive into Matching Algorithms

The matching algorithm is the definitive set of rules that governs how buy and sell orders are prioritized and paired. It is the core logic of the matching engine, and its choice is one of the most fundamental design decisions an exchange makes. This algorithm is not merely a technical detail; it is a powerful policy tool that shapes the behavior of market participants, defines the nature of competition, and ultimately determines the economic character of the marketplace. For the developer, understanding these algorithms and their implications is crucial for building a system that is not only technically sound but also aligned with specific market objectives.

3.1. The Standard: Price-Time Priority (FIFO)

The Price-Time Priority algorithm, often referred to as First-In, First-Out (FIFO), is the most prevalent matching model used in equity markets and by many electronic communication networks (ECNs) worldwide.[5] Its logic is transparent, fair, and easy to understand, which contributes to its widespread adoption.[5] The algorithm prioritizes orders based on two sequential criteria:

1. **Price Priority:** The best price always takes precedence.[5] For buy orders (bids), the highest price is considered the best because it shows the greatest willingness to pay. For sell orders (asks), the lowest price is the best because it represents the most competitive offer.[17] An incoming order will always attempt to match with the best-priced order(s) on the opposite side of the book first.
2. **Time Priority:** When multiple orders exist at the same price level, the order that was received by the exchange first gets priority.[5] This "first-in, first-out" principle rewards traders who place their orders earlier at a given price, creating a clear and unambiguous queue.[5]

This model directly encourages competition among traders, leading to improved price discovery and narrower bid-ask spreads, as participants are incentivized to offer

better prices to gain priority in the execution queue.[6]

To illustrate this process, consider the following step-by-step example of a Price-Time Priority matching engine in action:

Initial State: The order book for stock "XYZ" is empty.

- **Step 1: Order A Arrives (10:00:01.000)**
  - **Order:** BUY 100 shares @ $10.01
  - **Action:** The engine checks the sell side of the book. It is empty. Order A cannot be matched, so it is placed on the buy side of the book to rest.
  - **Book State:**
    - **Bids:** 100 @ $10.01 (from 10:00:01)
    - **Asks:** (empty)
- **Step 2: Order B Arrives (10:00:02.000)**
  - **Order:** BUY 200 shares @ $10.02
  - **Action:** The engine again checks the empty sell side. No match. Order B has a higher price than Order A ($10.02 > $10.01), so it receives **price priority**. It is placed at the top of the bid side.
  - **Book State:**
    - **Bids:** 200 @ $10.02 (from 10:00:02), 100 @ $10.01 (from 10:00:01)
    - **Asks:** (empty)
- **Step 3: Order C Arrives (10:00:03.000)**
  - **Order:** BUY 50 shares @ $10.02
  - **Action:** No match. Order C's price ($10.02) is the same as Order B's price. Now, **time priority** applies. Since Order B arrived earlier (10:00:02 vs. 10:00:03), Order C is placed in the queue *behind* Order B at the $10.02 price level.
  - **Book State:**
    - **Bids:** 200 @ $10.02 (from 10:00:02), 50 @ $10.02 (from 10:00:03), 100 @ $10.01 (from 10:00:01)
    - **Asks:** (empty)
- **Step 4: Order D Arrives (10:00:04.000)**
  - **Order:** SELL 220 shares @ $10.02
  - **Action:** This is an "aggressing" order that can match against the resting bids. The engine checks for compatible bids (buy orders with a price ≥ $10.02). It finds two price levels: $10.02 and $10.01. Following price priority, it will only interact with the $10.02 level.
    - **Match 1:** The engine matches Order D against the highest priority order at $10.02, which is Order B (due to time priority). 200 shares are traded.

Order B is fully filled and removed from the book. Order D now has 20 shares remaining (220 - 200).
        - **Match 2:** The engine continues to sweep the $10.02 price level. The next order in the queue is Order C. The remaining 20 shares of Order D are matched against Order C. 20 shares are traded. Order D is now fully filled and removed. Order C is partially filled and its remaining quantity is updated to 30 shares (50 - 20).
    - **Final Book State:**
        - **Bids:** 30 @ $10.02 (from 10:00:03), 100 @ $10.01 (from 10:00:01)
        - **Asks:** (empty)

This detailed flow, based on logic described in sources [5], shows how the deterministic rules of price and time create a predictable and fair execution environment.

3.2. Alternative Models and Their Incentives

While FIFO is the standard for equities, other algorithms exist, particularly in different asset classes like futures, each designed to incentivize different types of trading behavior.

- **Pro-Rata Algorithm:** This algorithm also gives priority to the best price. However, at a given price level, it allocates trades proportionally based on order size, not time of arrival.[6] For example, if there are two resting buy orders at $10.00, one for 300 shares and one for 100 shares, and a sell order for 80 shares arrives, the Pro-Rata algorithm would allocate 75% of the trade (60 shares) to the 300-share order and 25% (20 shares) to the 100-share order.[15] This model directly rewards traders for posting large orders, as it increases their chance of being matched. The primary incentive is to provide liquidity in size, rather than being the fastest.[6]
- **Hybrid Models:** Recognizing the benefits and drawbacks of pure FIFO and Pro-Rata systems, some exchanges have developed hybrid models. A common approach is a **Split FIFO/Pro-Rata** algorithm, where a certain percentage of an incoming order is allocated based on time priority (FIFO), and the remaining percentage is allocated based on size (Pro-Rata).[18] This attempts to strike a balance, rewarding both speed and size to attract a more diverse ecosystem of participants.
- **NYSE Parity/Priority Model:** The New York Stock Exchange (NYSE) employs a unique hybrid model. While it still rewards the participant who first sets the best

price, it then allocates subsequent trades at that price among all other orders at that level, rather than strictly following the time-based queue.[21] This is a deliberate policy choice designed to promote broader participation and improve fill rates for large institutional investors, who may not be the absolute fastest participants but contribute significant liquidity to the market.[21] It explicitly aims to reduce the competitive advantage of pure speed, which is a hallmark of high-frequency trading.

3.3. Implications of Algorithm Choice

The existence of these different models reveals a critical concept for the system architect: the matching algorithm is a powerful lever for market design. The choice is not technical but strategic.

- A **Price-Time (FIFO)** algorithm creates an environment where speed is paramount. It incentivizes investment in low-latency technology, colocation, and fast algorithms, which is attractive to high-frequency trading firms.[6]
- A **Pro-Rata** algorithm creates an environment that favors large, passive liquidity providers. It encourages participants to post large resting orders to get a higher allocation, which can lead to deeper, more liquid markets.[6]
- A **Hybrid** model, like the NYSE's, is an attempt to engineer a specific market dynamic—in their case, to ensure that large institutional orders can interact with liquidity without being consistently disadvantaged by faster, smaller-volume players.[21]

This has a profound architectural implication. A matching engine should not have its core matching logic hardcoded. The most robust and valuable design would treat the matching algorithm as a **pluggable strategy or module**. The core engine would be responsible for maintaining the order book data structure, while the specific rules for matching (FIFO, Pro-Rata, Hybrid) could be implemented as interchangeable components. This architectural separation of concerns would allow the engine to be adapted for different asset classes or to implement new, innovative market models in the future without requiring a fundamental rewrite of the entire system.

Section 4: Architectural Blueprint of a High-Performance

Matching Engine

Building a matching engine is an exercise in high-performance computing and distributed systems design. It is not a single, monolithic application but rather a collection of specialized, interconnected services, each optimized for its specific role.[22] The entire system is architected around the relentless pursuit of low latency, high throughput, and absolute determinism. This section provides a blueprint for such a system, deconstructing it into its logical components, diving deep into the critical data structures, and outlining the physical infrastructure required for competitive performance.

4.1. Core Logical Components: A System Deconstructed

The journey of an order from a trader's screen to a trade confirmation passes through several distinct logical components within the exchange's architecture. Understanding this flow is key to designing the system.

- **Order Gateway:** This is the front door to the exchange. It is a server application responsible for managing client connections, typically over TCP/IP for reliability.[23] The gateway's primary functions are to authenticate clients, manage sessions, and receive incoming order messages. These messages are often formatted in a standard industry protocol like the Financial Information eXchange (FIX) or a more performant proprietary binary protocol.[23] The gateway performs initial, basic validation on the order (e.g., checking for required fields) before passing it on for sequencing. To handle a large number of clients and provide redundancy, exchanges run multiple instances of gateway applications, which act as load balancers.[23]
- **Sequencer:** The sequencer is the architectural lynchpin that guarantees order and consistency across the entire trading system. Every single input that can alter the state of the market—a new order, a cancellation request, a modification—is funneled through this central component.[24] The sequencer's sole job is to receive these unordered messages from the various gateways and assign each one a unique, strictly monotonic, and globally consistent sequence number. This creates a single, ordered stream of events.[24] All downstream components, especially the matching engine itself, process these events

only in this prescribed sequence. This elegant design completely eliminates race conditions and makes the system's behavior deterministic. The state of the market at any given sequence number is always the same, which is critical for replication, fault tolerance, and auditing.[25]

- **Core Matching Logic (The Engine Proper):** This is the component that most people think of as the "matching engine." It subscribes to the ordered stream of messages produced by the sequencer. For each instrument it manages (e.g., a specific stock), it maintains an in-memory order book.[27] As it processes each message from the sequenced stream, it applies the action to its order book—adding a new order, canceling an existing one, or modifying an order. After any action that could result in a trade, it applies the exchange's matching algorithm (e.g., Price-Time Priority) to check for matches.[12] To eliminate the overhead and non-determinism of multithreaded locking, the core matching logic for a given instrument or market segment is almost always a **single-threaded process**.[28] This ensures that events are processed serially at maximum speed without contention.
- **Market Data Publisher:** After the core engine processes an event, it generates one or more outbound messages describing the outcome: an acknowledgment of a new order, a trade confirmation, a cancellation confirmation, or a change in the public order book. The Market Data Publisher is the component responsible for disseminating this information.[22] It typically produces two main feeds: a private feed of trade confirmations sent back to the specific clients involved, and a public, anonymized feed of all trades and changes to the order book (market data) that is broadcast to all subscribers.[27]
- **Post-Trade and Risk Management:** When a trade occurs, the engine generates a trade report. This report is sent to downstream systems for clearing and settlement, which are the processes that handle the final exchange of money and securities between the buyer and seller.[12] Additionally, the system must integrate with risk management components. This can happen pre-trade, where an order might be checked against a client's credit limit before being accepted by the engine.[29]

4.2. The Heart of the Machine: Implementing a High-Performance Order Book

The performance of the entire system hinges on the efficiency of the order book data structure. This is where microseconds are won or lost. The goal is to implement the

three primary operations—add, cancel, and execute—with the lowest possible latency, ideally in constant O(1) or logarithmic O(log N) time.[31] While standard computer science data structures provide a good starting point, the extreme performance requirements of financial markets have led to highly specialized implementations.

- **A Baseline Approach:** A common initial thought is to use a balanced binary search tree (like a Red-Black Tree, often the basis for std::map in C++) to store the price levels, with each node in the tree pointing to a queue or list of orders at that price.[32] This approach is functionally correct and provides O(log M) complexity for operations, where M is the number of distinct price levels. However, in the world of high-frequency trading, this is often considered too slow. The main culprit is poor CPU cache performance; traversing a tree involves following pointers, which often leads to cache misses as the nodes can be scattered throughout memory.[34]
- **The High-Performance Hybrid Design:** A far more performant approach, widely used in real-world HFT systems, recognizes that software performance is dictated by hardware realities ("mechanical sympathy"). This design combines several data structures, each chosen for its specific strengths [31]:
    1. **Direct Order Lookup:** A hash map (e.g., std::unordered_map in C++ or an open-addressing hash table) is used to store all active orders, keyed by their unique Order ID. The value stored in the map is a pointer or handle to the full order object. This provides **O(1) average-case lookup** for handling cancellation or modification requests, which are extremely common.[31]
    2. **Price Level Storage:** Instead of a tree, the price levels for the bid and ask sides are stored in separate **contiguous arrays or vectors, sorted by price**.[17] Because most trading activity (new orders, executions) occurs at or near the best price (the "top of the book"), which corresponds to the end of these sorted arrays, operations are incredibly fast. A linear search from the inside of the book for a few levels is often faster than a binary search on a tree due to superior CPU cache locality and better branch prediction.[34]
    3. **Order Queue at Each Price Level:** Within each price level, orders must be maintained in a strict time-based queue. A **doubly linked list** is the perfect data structure for this. It allows for O(1) insertion at the tail (for new orders) and O(1) removal from the head (for executions). Each order object would contain pointers to the next and previous orders in the queue at its price level.[31]
- **A Concrete High-Performance Pattern:** A practical implementation based on these principles would involve a Book object that contains two Side objects (one for bids, one for asks). Each Side object would contain a sorted array of

PriceLevel objects. Each PriceLevel object would contain a doubly linked list of Order objects. A global hash map would map Order IDs directly to their Order object pointers, allowing for O(1) cancellation. An even more optimized design, described in [17], uses fixed-size circular arrays for both the price levels and the order queues within them, combined with a map from price to array index, to minimize memory allocation and further improve cache performance.

The following table compares these data structure strategies, highlighting the critical trade-offs for a developer.

Data Structure Strategy	Add Order Complexity	Cancel Order Complexity	Execute Order Complexity	Cache Performance	Key Advantage	Key Disadvantage
Red-Black Tree of Linked Lists	O(log M)	O(log M) + O(1)	O(log M) + O(1)	Poor	Simple to reason about; guaranteed logarithmic performance.	High rate of cache misses due to pointer chasing; generally slower in practice.
Sorted Array of Linked Lists + Hash Map	O(M) worst-case, O(1) typical	O(1)	O(1)	Excellent	Extreme speed for top-of-book operations; O(1) cancellations.	Insertion of a new price level far from the top is slow due to array shifting.
Contiguous Circular Arrays + Hash Map	O(1)	O(1)	O(1)	Optimal	Minimal memory allocation; excellent cache locality; extremely fast.	More complex to implement correctly; requires careful management of array

						indices.

Note: M is the number of price levels.

4.3. Physical and Network Architecture for Ultra-Low Latency

The pursuit of low latency extends beyond software design into the physical realm of hardware and networking.

- **Colocation:** The single most effective way to reduce latency is to eliminate physical distance. Trading firms pay significant fees for **colocation**, which is the practice of placing their own servers in the same physical data center as the exchange's matching engine.[22] This reduces network travel time from milliseconds (across a city or country) to microseconds (across a data center floor).
- **High-Performance Hardware:** The servers themselves are highly specialized. They use CPUs with the highest clock speeds and largest L3 caches, and high-speed, low-latency RAM.[36] The most critical component is often the **Network Interface Card (NIC)**. Standard NICs are not sufficient; HFT systems use specialized ultra-low-latency NICs that often incorporate **Field-Programmable Gate Arrays (FPGAs)**. These FPGAs can be programmed to perform tasks like network packet processing and even basic risk checks directly in hardware, bypassing the server's CPU and operating system entirely for ultimate speed.[36]
- **Network Protocols and Techniques:**
  - **TCP and UDP:** As mentioned, order entry gateways use reliable, connection-oriented protocols like TCP/IP.[23] Market data, which needs to be broadcast to many subscribers simultaneously, is often sent over UDP multicast. UDP is connectionless and less reliable, but its low overhead makes it ideal for one-to-many data dissemination where performance is paramount.[22]
  - **Kernel Bypass:** A standard operating system's network stack introduces significant latency as it processes packets. **Kernel bypass** technologies like Solarflare's OpenOnload or DPDK allow an application to communicate directly with the NIC's hardware buffers, completely avoiding the OS kernel's involvement and saving tens of microseconds per message.[36]
  - **Internal Fabric:** The internal network connecting the exchange's own

components (e.g., gateways to matching engines) is not standard Ethernet. It is often a specialized, high-bandwidth, low-latency fabric like **InfiniBand**.[23]

In summary, the architecture of a modern matching engine is a holistic system where software algorithms are designed in concert with the hardware they run on. The developer must shift their thinking from abstract performance to concrete, physical reality, where every CPU cycle, every cache miss, and every network hop is a critical resource to be optimized.

Section 5: Engineering for Extremes - System Qualities and Non-Functional Requirements (NFRs)

A functional matching engine is only the first step. A production-grade system, capable of operating at the heart of financial markets, must be engineered to meet a set of extreme non-functional requirements (NFRs). These system qualities—performance, availability, scalability, and security—are not features to be added later; they are the fundamental forces that dictate the engine's architecture from its inception.[37] For a developer, understanding and designing for these NFRs is what separates a proof-of-concept from a viable, robust trading platform.

5.1. Performance: Latency and Throughput

Performance is the most defining characteristic of a modern matching engine. It is typically measured along two axes:

- **Latency:** This is the time delay in processing a message. In the context of trading, "tick-to-trade" latency—the time from when a market data event is received to when a corresponding order is sent—is a critical benchmark. While retail systems might operate in hundreds of milliseconds, high-frequency trading (HFT) systems aim for latency **under 100 microseconds**, with core engine operations often measured in **single-digit microseconds or even nanoseconds**.[38] Achieving these speeds requires the aggressive optimization strategies discussed previously: low-level programming languages like C++ or Rust [36], cache-aware data structures that minimize memory access time [34], and hardware-level

techniques like kernel bypass and FPGAs.[36]

- **Throughput:** This is the number of messages the system can process per second. During periods of high market volatility, exchanges can experience message rates in the **millions per second**.[27] The system must be ableto handle these peak loads without a significant degradation in latency. This requires not only a fast core engine but also highly scalable gateway and data dissemination components.[12]

5.2. Availability and Reliability

A financial exchange is critical infrastructure; downtime can result in massive financial losses and a severe loss of confidence in the market. Consequently, matching engines are designed for extreme levels of availability and reliability.

- **Requirement:** The target uptime for a major exchange is often "five nines" (**99.999% availability**) or higher, which translates to less than 5.26 minutes of unplanned downtime per year.[37] Reliability means the system must function correctly and without failures or data corruption.[37]
- **Architectural Patterns:** Achieving this level of uptime is impossible with a single server. It requires a distributed architecture built on redundancy and fault tolerance.
    - **Component Redundancy:** Every component in the system is run with multiple instances. There are multiple gateways, multiple market data publishers, and, most importantly, multiple matching engine instances.[12] If one component fails, traffic can be seamlessly rerouted to a standby instance.
    - **State Machine Replication and Determinism:** Simple primary/backup failover is insufficient for a stateful system like a matching engine, as it risks data loss or a "split-brain" scenario where both nodes believe they are the primary. The industry-standard solution is **state machine replication**. By using the **Sequencer** architecture described in Section 4, the system's behavior is made fully deterministic.[24] The ordered log of input messages becomes the single source of truth. A backup engine can achieve a state identical to the primary engine simply by processing the exact same log of messages in the exact same order.[25]
    - **Consensus Protocols:** To manage which node is the primary and to ensure that the sequenced log is reliably replicated across all nodes in the cluster, a **consensus protocol** like **Raft** is often used.[26] This provides a mathematically

proven mechanism for building a fault-tolerant, highly available cluster that can survive the failure of individual nodes without service interruption or data loss.

5.3. Scalability

As markets grow, the system must be able to handle an increasing number of instruments, users, and overall message volume.

- **Requirement:** The architecture must be ableto scale to accommodate growth without requiring a complete redesign.[12]
- **Strategies:** The key to scalability is **horizontal scaling**—adding more machines to the system—rather than vertical scaling (making a single machine more powerful). For a matching engine, this is typically achieved through **partitioning**. The universe of tradable instruments is divided among multiple, independent matching engine instances.[30] For example, one engine instance might handle all stocks with symbols A-M, while another handles N-Z. This allows the system's total capacity to be increased linearly by simply adding more engine instances and assigning them a new slice of the market.[41]

5.4. Security and Integrity

Given the vast sums of money being transacted, security and data integrity are paramount.

- **Requirement:** The system must be protected from unauthorized access and must have internal controls to prevent both accidental errors and malicious activity from disrupting the market.[37]
- **Implementation:**
    - **Access Control:** Gateways are responsible for strong authentication of clients and authorization to ensure they can only perform actions on their own orders.
    - **Risk Controls:** The engine itself must have a suite of built-in, pre-trade risk checks. These are not optional features; they are often mandated by regulators.[29] Examples include:

- **"Fat Finger" Protection:** Rejecting orders with clearly erroneous prices or quantities (e.g., an order to buy a $50 stock for $5,000, or an order for a quantity that exceeds a predefined maximum).[30]
- **Self-Match Prevention (SMP):** Preventing a participant from accidentally trading with themselves, which can have undesirable regulatory and tax consequences.[14]
- **Message Throttling:** Limiting the number of messages a single participant can send in a given time period to prevent them from intentionally or unintentionally overloading the system.[30]

The following table maps these critical NFRs to the architectural solutions required to achieve them, providing a clear guide for the developer.

Non-Functional Requirement	Target Metric	Key Architectural Patterns/Technologies	Relevant Regulatory Mandate
Performance (Latency)	< 100 µs tick-to-trade	Kernel Bypass (DPDK), Colocation, FPGAs, Cache-Aware Data Structures (Arrays), C++/Rust	MiFID II (implied by HFT definition)
Performance (Throughput)	> 1M messages/sec	Horizontal Scaling (Partitioning), UDP Multicast for Market Data, Efficient Gateways	MiFID II Art. 17 (Sufficient Capacity)
Availability	99.999% Uptime	State Machine Replication (Raft), Sequencer for Deterministic Replay, Redundant Components	MiFID II Art. 17 (Business Continuity)
Reliability	Zero Data Loss	Consensus Protocols (Raft), Sequenced Event Logging	MiFID II Art. 17 (Record Keeping)
Scalability	Linear scaling with load	Horizontal Partitioning by	-

		Instrument, Load-Balanced Gateways	
Security & Integrity	No unauthorized access or erroneous trades	FIX/TLS Authentication, Pre-Trade Risk Controls (Price Collars, Max Qty), Self-Match Prevention	MiFID II Art. 17 (Prevent Erroneous Orders)

Ultimately, these NFRs are not a simple checklist. They are the fundamental constraints that give the matching engine its unique and often counter-intuitive architecture. The decision to use a single-threaded core logic, for instance, is a direct consequence of prioritizing deterministic low latency above the conventional wisdom of multithreaded design. Similarly, the complexity of a sequencer and a consensus protocol is the price that must be paid to achieve the required level of availability and data integrity. A developer must embrace these NFRs as the primary drivers of their design, as they cannot be effectively retrofitted onto a system not built for them from the ground up.

Section 6: The Regulatory Mandate - Compliance and Market Fairness

A modern equity matching engine is not built in a regulatory vacuum. It is a highly regulated piece of financial infrastructure, and its design must be deeply informed by a complex web of rules aimed at ensuring market stability, fairness, and transparency. For a developer, treating regulatory requirements as a primary source of architectural specifications is not optional; it is essential for building a system that is legally operable in any major jurisdiction. Regulations like the Markets in Financial Instruments Directive II (MiFID II) in Europe provide a clear blueprint of the controls and capabilities that must be built into the engine's core.[42]

6.1. Overview of the Regulatory Landscape

In the wake of market events driven by the speed and complexity of automated trading, regulators worldwide have implemented rules specifically targeting algorithmic and high-frequency trading (HFT). The primary motivation is to mitigate systemic risks.[42] Regulators are acutely concerned about the potential for automated systems to cause rapid and significant market distortion, including overloading exchange systems with excessive messages, creating destabilizing volatility ("flash crashes"), or gaining unfair advantages over other participants.[42] MiFID II is a cornerstone of this regulatory effort, and its technical requirements have a direct impact on the design of a matching engine.

6.2. Systems and Risk Controls (MiFID II, Article 17)

Article 17 of MiFID II is particularly relevant, as it mandates a suite of effective systems and risk controls for any firm engaging in algorithmic trading, and for the trading venues that host them.[29] A matching engine must be designed to implement or facilitate these controls.

- **Mandatory Pre-Trade Controls:** The engine cannot simply accept any order. It must be part of a pipeline that validates orders against a set of risk limits before they are allowed to enter the order book. These controls include:
    - **Trading Thresholds and Limits:** The system must be capable of enforcing price collars, which automatically reject orders that are priced too far away from the current market, and limits on the maximum value or volume of a single order.[29] This is a crucial defense against "fat finger" errors.
    - **Order Throttling:** To prevent system overload, the engine or its gateways must be ableto enforce limits on the rate of messages (orders, cancels, modifies) that a single participant can send.[29]
    - **Sufficient Capacity:** The regulation explicitly requires that trading systems be resilient and have sufficient capacity to handle peak message volumes, which reinforces the NFRs for performance and scalability.[29]
- **Emergency Controls:** The system must have mechanisms to intervene in disorderly markets.
    - **Kill Switch:** This is a mandatory function that allows a trader or an exchange administrator to immediately and automatically cancel all outstanding orders

for a specific participant or algorithm.[30] This functionality must be built into the order management and cancellation logic of the system.

- **Testing and Monitoring:** Venues are required to provide facilities for their clients to test their algorithms in a non-live environment before they are deployed.[29] This means the overall exchange architecture must include a robust conformance testing or certification environment that accurately simulates the production matching engine's behavior. Furthermore, all systems must be properly monitored in production to ensure they comply with the rules.[29]
- **Business Continuity:** The mandate for effective business continuity arrangements to deal with system failures directly aligns with the NFR for high availability, reinforcing the need for architectural patterns like state machine replication.[29]

6.3. Transparency and Record-Keeping

A key goal of regulation is to give authorities the ability to reconstruct market events and investigate potential abuse. This places significant data and reporting requirements on the matching engine.

- **Algorithmic Trading Identification:** MiFID II requires that orders generated by an algorithm be clearly flagged as such. The order message itself must identify the specific algorithm used and the person responsible for the investment decision.[43] This has a direct impact on the design of the engine's API or order entry protocol (e.g., the FIX message format), which must include dedicated fields for these flags. The engine must be ableto persist and process this metadata.
- **High-Frequency Trading Records:** Firms engaging in HFT must store extremely detailed, time-sequenced records of all placed orders, executed orders, and cancellations for at least five years.[29] The matching engine is the source of this data. It must be designed to output a comprehensive and accurate audit trail of every single event with high-precision timestamps, sufficient for regulators to perform their oversight functions.

6.4. Fair and Non-Discriminatory Access

To ensure a level playing field, regulators impose strict rules on how exchanges provide access to their systems, particularly for latency-sensitive services.

- **Colocation Services:** The rules governing colocation services—where participants place their servers in the exchange's data center—must be transparent, fair, and non-discriminatory.[44] This means the exchange must offer these services on equivalent commercial terms to all participants. Some exchanges go further, implementing **latency equalization**, where they use precisely measured lengths of fiber optic cable to ensure that the physical network path from every participant's rack to the matching engine is identical, thus neutralizing any small distance advantages within the data center itself.[22]
- **Fee Structures:** The fees charged by an exchange (e.g., for trading, connectivity, or market data) must also be transparent and non-discriminatory. Crucially, they cannot be structured in a way that creates incentives for disorderly trading or market abuse.[42]

The overarching lesson for the developer is that regulatory compliance is a foundational design pillar, not a feature to be added at the end. The requirements for pre-trade risk checks, order flagging, audit trails, and fair access must be considered from the very beginning of the design process. They directly influence the order entry API, the internal data models, the validation logic in the critical path, and the data output capabilities of the system. Building an engine without this "compliant by design" mindset would result in a system that is fundamentally unfit for purpose in any modern, regulated financial market.

Section 7: From Theory to Practice - A Survey of Open-Source Implementations

The architectural principles and complex requirements discussed thus far can seem abstract. To bridge the gap between theory and practice, studying existing open-source matching engine projects is an invaluable exercise for any developer embarking on this journey. These projects provide tangible examples of data structures, API design, and system organization, offering a powerful learning tool and, in some cases, a practical foundation upon which to build.[45]

7.1. The Value of Open-Source Engines

Building a matching engine from scratch is a monumental undertaking.[17] Open-source projects serve several critical functions for a developer:

- **Learning and Reference:** They provide a working implementation of complex concepts like price-time priority matching and order book management. Reading the code can clarify theoretical understanding in a way that documentation alone cannot.
- **Architectural Patterns:** They showcase different architectural choices. Some are full-fledged (but simplified) exchange systems, while others are designed as modular component libraries, illustrating different approaches to system decomposition.
- **Practical Scaffolding:** Leveraging a proven open-source core can dramatically accelerate development. Instead of reinventing the highly optimized order book, a developer can integrate a library and focus their efforts on the surrounding components like gateways, risk controls, and user interfaces.

7.2. Deep Dive: Liquibook (C++)

Among the available open-source projects, **Liquibook** stands out as a particularly instructive and well-regarded example. It is a high-performance, open-source library written in modern C++ that provides the low-level components of an order matching engine.[46]

- **Overview and Philosophy:** Liquibook is intentionally not a standalone exchange server. It is designed as a **component library** that can be integrated into a larger application.[41] Its philosophy is to solve the hardest, most performance-critical part of the problem—the maintenance and matching of the limit order book—while leaving the surrounding concerns like network connectivity, user management, and post-trade processing to the host application. This modular design makes it extremely flexible.
- **Key Design Features:**
    - **Rich Order Support:** Liquibook has built-in awareness of a comprehensive set of order types and attributes, including Market, Limit, Stop, Immediate or

Cancel (IOC), and All or None (AON).[46] It also understands how to combine flags to create common behaviors like Fill or Kill (IOC + AON).[46]
    - **Event-Driven Notification System:** Liquibook does not handle networking or trade settlement itself. Instead, it communicates with the host application through a **callback interface**. When a significant event occurs—an order is accepted, rejected, filled (partially or fully), or a trade happens—Liquibook invokes a corresponding method on a listener object provided by the application.[46] This allows the application to take the necessary actions, such as sending a FIX message to a client, publishing a market data update, or forwarding a trade report to a clearing system.[48]
    - **High Performance:** The library is engineered for speed, with benchmark tests demonstrating sustained rates of **2.0 to 2.5 million order inserts per second** on appropriate hardware.[46] This level of performance makes it suitable for demanding applications.
    - **Flexible Integration:** A key design principle is that Liquibook adapts to the application's existing data models, not the other way around. It does not impose its own Order class. Instead, an application can use its own order objects by implementing a trivial interface wrapper, preserving existing code and identifiers for securities, accounts, and orders.[46]
- **Architectural Significance:** The power of Liquibook's component-based design is demonstrated by projects like **DistributedATS**.[41] This project builds a complete, distributed alternative trading system by integrating several best-of-breed open-source libraries. It uses Liquibook as the core matching engine for its Central Limit Order Book (CLOB). It then integrates **QuickFIX**, a popular FIX protocol engine, to build its client-facing gateways. For internal communication between the gateways and the multiple matching engine instances, it uses a high-performance messaging middleware (DDS). This project is a perfect real-world example of the modular architecture discussed in Section 4, showing how a developer can assemble a sophisticated system by combining specialized components.

7.3. Other Notable Implementations

While Liquibook is a prime example of a C++ component library, other projects showcase different approaches and language choices:

- **Rust-based Engine with Raft:** One notable project is an implementation of a

matching engine in Rust that uses the Raft consensus protocol to build a replicated, highly-available system.[39] This project is an excellent case study for understanding how to implement the fault tolerance and high availability NFRs discussed in Section 5.

- **C++ Engine with Red-Black Trees:** Another open-source example built in C++ uses Red-Black Trees as the primary data structure for the order book.[33] Studying this code would be a practical way to understand the baseline approach to order book design and contrast it with the more cache-optimized array-based methods.
- **Simplified C/C++ Engine:** A more straightforward project available on GitHub focuses on processing a specific set of commands (New, Amend, Cancel, Match) from an input stream.[49] While less architecturally complex, it provides a clear, contained example of the basic command processing loop.

The key lesson from surveying these projects is that while the core principles of matching are universal, the implementation details can vary significantly based on the project's goals. For the developer aiming to build their own engine, the architectural pattern of encapsulating the high-performance order book logic into a core library, as exemplified by Liquibook, is a powerful and proven strategy. It de-risks the project by allowing the developer to leverage an optimized core for the most difficult part of the system, freeing them to focus on the equally important, but less latency-critical, task of systems integration.

Section 8: Conclusion and Implementation Roadmap

This report has provided an exhaustive deconstruction of the modern equity matching engine, journeying from the foundational market theory of the Central Limit Order Book to the nanosecond-level details of high-performance software and hardware architecture. We have explored the rich language of order types, dissected the economic incentives of matching algorithms, and navigated the critical constraints imposed by non-functional requirements and regulatory mandates. For the developer tasked with building such a system, this knowledge forms the basis of a strategic and achievable implementation plan.

8.1. Summary of Critical Design Trade-offs

The design of a matching engine is a series of critical trade-offs. The optimal choice is always context-dependent, but the most important decisions revolve around three key tensions:

1. **Performance vs. Complexity:** The most performant data structures for the order book (e.g., cache-friendly contiguous arrays) are often the most complex to implement correctly compared to theoretically elegant but practically slower structures like balanced binary trees. The developer must choose a point on this spectrum that balances their performance needs with their implementation capacity.
2. **Fairness vs. Incentives:** The choice of matching algorithm is not a neutral technical decision. A Price-Time (FIFO) algorithm creates a fair-for-all race for speed, while a Pro-Rata algorithm incentivizes large liquidity providers. The choice fundamentally shapes the market's character and the type of participants it will attract. A flexible, pluggable algorithm design is the most robust architectural choice.
3. **Determinism vs. Simplicity:** Implementing a Sequencer to create a deterministic, replicable system adds significant architectural complexity. However, it dramatically simplifies the core matching logic (allowing for a single-threaded design) and is the bedrock of achieving true high availability and fault tolerance. For any production-grade system, the benefits of determinism far outweigh the initial implementation cost.

8.2. A Phased Implementation Roadmap

Building a complete, production-grade matching engine is a marathon, not a sprint. A phased, iterative approach is essential for managing complexity and ensuring a successful outcome.

- **Phase 1: The Core In-Memory Engine (Proof of Concept)**
    - **Goal:** Create a single, standalone, in-memory program that correctly implements the core matching logic.
    - **Steps:**
        1. **Implement the Order Book:** Start with a functionally correct data

structure. A balanced binary tree (like std::map or SortedDict) is a good initial choice.
    2. **Implement the Matching Algorithm:** Code the Price-Time Priority (FIFO) algorithm.
    3. **Support Basic Orders:** Handle only the most fundamental order types: Limit and Market orders, with a simple Day (or GTC) time-in-force.
    4. **Create a Test Harness:** Build a simple interface to feed orders into the engine from a text file or command-line input and print the resulting trades and book state.
  - **Acceleration:** To speed up this phase and build on a proven foundation, consider using an open-source library like **Liquibook** [46] as the core engine from the start.
- **Phase 2: Networking and Expanded Order Types**
  - **Goal:** Make the engine accessible over a network and expand its vocabulary of supported instructions.
  - **Steps:**
    1. **Build a Gateway:** Implement a simple TCP server that accepts connections and parses order messages.
    2. **Expand Order Support:** Add the logic to handle conditional orders (Stop, Stop-Limit) and common TIF qualifiers (IOC, FOK). This will require adding market state tracking to trigger the conditional orders.
    3. **Basic Market Data:** Create a simple market data publisher that broadcasts executed trades to connected clients.
- **Phase 3: Production-Grade Features**
  - **Goal:** Incorporate the features required for a robust and compliant system.
  - **Steps:**
    1. **Implement the Sequencer:** Refactor the architecture to funnel all incoming commands through a sequencer to ensure deterministic processing.
    2. **Add Regulatory Risk Controls:** Build in the mandatory pre-trade risk checks, such as price collars and maximum quantity limits. Implement a "kill switch" mechanism.
    3. **Develop a Full Market Data Feed:** Design and implement a proper market data feed that publishes changes to the order book depth in addition to trades.
- **Phase 4: High Availability and Scalability**
  - **Goal:** Transform the single-instance engine into a fault-tolerant, scalable, distributed system.
  - **Steps:**

1. **Implement State Machine Replication:** Use a consensus protocol like Raft to replicate the state of the matching engine and sequencer across a cluster of servers, providing high availability.
2. **Design for Horizontal Scalability:** Architect the system to support partitioning, where different sets of instruments can be assigned to different engine instances, allowing overall system capacity to grow by adding more hardware.

8.3. Final Recommendations

The journey to building an equity matching engine is challenging but achievable with a disciplined, iterative approach. The key recommendations are:

- **Start Simple, Focus on Correctness:** Begin with the simplest possible version in Phase 1. Ensure the matching logic is perfectly correct before adding complexity or optimizing for performance.
- **Embrace Modularity:** Design the system as a set of communicating components (Gateway, Sequencer, Engine, Publisher). This separation of concerns is critical for managing complexity.
- **Study the Masters:** The most valuable resource is existing code. A deep and thorough study of the **Liquibook** source code is strongly recommended. It provides a masterclass in high-performance, component-based design for an order matching core.

By following this roadmap and internalizing the architectural principles outlined in this guide, a dedicated developer can successfully navigate the complexities of financial technology and build a powerful, robust, and performant equity matching engine.

Works cited

1. en.wikipedia.org, accessed on July 24, 2025, https://en.wikipedia.org/wiki/Central_limit_order_book#:~:text=A%20central%20limit%20order%20book,'price%20time%20priority'%20basis.
2. What Is A Central Limit Order Book (CLOB)? | Financial Glossary ..., accessed on July 24, 2025, https://equalsmoney.com/financial-glossary/central-limit-order-book
3. Central limit order book - Wikipedia, accessed on July 24, 2025, https://en.wikipedia.org/wiki/Central_limit_order_book
4. Central limit order book (Clob) definition - Risk.net, accessed on July 24, 2025,

https://www.risk.net/definition/central-limit-order-book-clob
5. Price-time allocation model - MarketsWiki, A Commonwealth of ..., accessed on July 24, 2025, https://www.marketswiki.com/wiki/Price-time_allocation_model
6. Order matching system: Explained - TIOmarkets, accessed on July 24, 2025, https://tiomarkets.com/en/article/order-matching-system-guide
7. Look up the concept of a "Central Limit Order Book" (CLOB) -- this is the fundam... | Hacker News, accessed on July 24, 2025, https://news.ycombinator.com/item?id=34785384
8. Types of Orders | Investor.gov, accessed on July 24, 2025, https://www.investor.gov/introduction-investing/investing-basics/how-stock-markets-work/types-orders
9. Order Types | FINRA.org, accessed on July 24, 2025, https://www.finra.org/investors/investing/investment-products/stocks/order-types
10. Guide to the Different Stock Market Order Types | SoFi, accessed on July 24, 2025, https://www.sofi.com/learn/content/stock-order-types/
11. Order Types | Order Ticket Help and Tutorials - TT Help Library, accessed on July 24, 2025, https://library.tradingtechnologies.com/trade/ot-order-types.html
12. Matching Engine Architecture - FinchTrade, accessed on July 24, 2025, https://finchtrade.com/glossary/matching-engine-architecture
13. 3 Order Types: Market, Limit, and Stop Orders - Charles Schwab, accessed on July 24, 2025, https://www.schwab.com/learn/story/3-order-types-market-limit-and-stop-orders
14. Chronicle Matching Engine, accessed on July 24, 2025, https://chronicle.software/wp-content/uploads/2022/11/Matching_Engine_TR.pdf
15. Matching Engine for a Stock Trading Application - GeekyAnts, accessed on July 24, 2025, https://geekyants.com/blog/matching-engine-for-a-stock-trading-application
16. How Matching Engine Software Works and Helps Execute Trades | B2PRIME, accessed on July 24, 2025, https://b2prime.com/news/what-role-does-matching-engine-software-play
17. Designing Low Latency High Performance Order Matching Engine | by Amitava Biswas, accessed on July 24, 2025, https://medium.com/@amitava.webwork/designing-low-latency-high-performance-order-matching-engine-a07bd58594f4
18. Matching Algorithms - Edge Clear, accessed on July 24, 2025, https://support.edgeclear.com/portal/en/kb/articles/fifo-vs-lifo
19. Order matching system - Wikipedia, accessed on July 24, 2025, https://en.wikipedia.org/wiki/Order_matching_system
20. Matching Orders - Overview, Process, and Algorithms - Corporate Finance Institute, accessed on July 24, 2025, https://corporatefinanceinstitute.com/resources/career-map/sell-side/capital-markets/matching-orders/
21. NYSE parity | Why trading on the NYSE is different than other markets, accessed

on July 24, 2025, https://www.nyse.com/article/parity-priority-explainer
22. An introduction to matching engines: A guide by Databento, accessed on July 24, 2025, https://medium.databento.com/an-introduction-to-matching-engines-a-guide-by-databento-d055a125a6f6
23. Technology FAQ and Best Practices: Equities - NYSE, accessed on July 24, 2025, https://www.nyse.com/publicdocs/nyse/markets/nyse/NYSE_Group_Equities_Technology_FAQ.pdf
24. Proof Engineering: The Message Bus | by Prerak Sanghvi | Proof Reading | Medium, accessed on July 24, 2025, https://medium.com/prooftrading/proof-engineering-the-message-bus-a7cc84e1104b
25. The Sequencer Architecture and Its Features (and why financial ..., accessed on July 24, 2025, https://www.youtube.com/watch?v=DyktSiBTCdk
26. Understanding Aeron Sequencer Architectures for Capital Markets, accessed on July 24, 2025, https://aeron.io/not-listed/aeron-sequencer-architectures/
27. How to Build an Exchange - Jane Street, accessed on July 24, 2025, https://www.janestreet.com/tech-talks/building-an-exchange/
28. Multicast and the Markets - Signals and Threads, accessed on July 24, 2025, https://signalsandthreads.com/multicast-and-the-markets/
29. Article 17 Algorithmic trading | European Securities and Markets ..., accessed on July 24, 2025, https://www.esma.europa.eu/publications-and-data/interactive-single-rulebook/mifid-ii/article-17-algorithmic-trading
30. Order Matching Engine: Everything You Need to Know - Devexperts, accessed on July 24, 2025, https://devexperts.com/blog/order-matching-engine-everything-you-need-to-know/
31. How to Build a Fast Limit Order Book · GitHub, accessed on July 24, 2025, https://gist.github.com/halfelf/db1ae032dc34278968f8bf31ee999a25
32. What are some good data structures to store a large orderbook? - Stack Overflow, accessed on July 24, 2025, https://stackoverflow.com/questions/21222129/what-are-some-good-data-structures-to-store-a-large-orderbook
33. How I created an Order Matching Engine | by Nishant Sharma | May, 2025 | Medium, accessed on July 24, 2025, https://medium.com/@nishant19072003/how-i-created-an-order-matching-engine-33e0669be77a
34. market microstructure - What is an efficient data structure to model ..., accessed on July 24, 2025, https://quant.stackexchange.com/questions/3783/what-is-an-efficient-data-structure-to-model-order-book
35. Exploring the Potential of Reconfigurable Platforms for Order Book Update - Imperial College London, accessed on July 24, 2025, https://www.doc.ic.ac.uk/~wl/papers/17/fpl17ch.pdf

36. How to Build a Low-Latency Trading Infrastructure (in 6 Steps), accessed on July 24, 2025, https://www.forexvps.net/resources/low-latency-trading-infrastructure/
37. Non-functional requirements basics - Technical Program Management, accessed on July 24, 2025, https://technicalprogrammanager.com/non-functional-requirements-basics/
38. Latency Standards in Trading Systems - LuxAlgo, accessed on July 24, 2025, https://www.luxalgo.com/blog/latency-standards-in-trading-systems/
39. pgellert/matching-engine: Matching Engine implementation in Rust - GitHub, accessed on July 24, 2025, https://github.com/pgellert/matching-engine
40. Nonfunctional Requirements Explained: Examples, Types, Tools, accessed on July 24, 2025, https://www.modernrequirements.com/blogs/what-are-non-functional-requirements/
41. DistributedATS is a FIX Protocol based multi matching engine exchange(CLOB) that integrates QuickFIX and LiquiBook over DDS - GitHub, accessed on July 24, 2025, https://github.com/mkipnis/DistributedATS
42. MiFID II | frequency and algorithmic trading obligations | Global law ..., accessed on July 24, 2025, https://www.nortonrosefulbright.com/en/knowledge/publications/6d7b8497/mifid-ii-mifir-series
43. MiFID II/MiFIR - Eurex, accessed on July 24, 2025, https://www.eurex.com/ex-en/rules-regs/regulations/mifid-mifir
44. MiFID II Review Report - | European Securities and Markets Authority, accessed on July 24, 2025, https://www.esma.europa.eu/sites/default/files/library/esma70-156-4572_mifid_ii_final_report_on_algorithmic_trading.pdf
45. Open Source Order Matching Engine : r/algotrading - Reddit, accessed on July 24, 2025, https://www.reddit.com/r/algotrading/comments/127z0g5/open_source_order_matching_engine/
46. enewhuis/liquibook: Modern C++ order matching engine - GitHub, accessed on July 24, 2025, https://github.com/enewhuis/liquibook
47. Liquibook Implementation of Order Book with the CMake build system - GitHub, accessed on July 24, 2025, https://github.com/dendisuhubdy/liquibook
48. Building a Market Data Feed with Liquibook | Object Computing, Inc., accessed on July 24, 2025, https://objectcomputing.com/resources/publications/sett/august-2013-building-a-market-data-feed-with-liquibook
49. souradipp76/Equity_Order_Matcher: Implementation of a ... - GitHub, accessed on July 24, 2025, https://github.com/souradipp76/Equity_Order_Matcher