Crystal Whitepaper

February 2026

legionevm
legionevm@crystal.exchange

bhealthyfences
bh@crystal.exchange

Abstract

Crystal is a fully onchain and non-custodial limit order book exchange built on the Ethereum Virtual Machine (EVM). The protocol's mission is to become the primary venue for onchain price discovery by enabling capital efficient and performant markets without compromising on security or composability. Through constant-time order placements, optimal storage packing, and an atomic matching engine, Crystal allows active liquidity providers to quote more precisely and update quotes more frequently than on traditional onchain markets. Additional functionality, including atomic cross-market trades and an integrated constant-product liquidity curve, position the protocol as liquidity infrastructure within a decentralized financial ecosystem rather than solely as a standalone spot exchange. Without any centralized intermediaries or privileged parties, on-chain financial markets allow the price discovery process to be globally accessible and transparently enforceable, removing the trusted middleman layer. Accordingly, Crystal is intended to be deployed as an autonomous, standalone protocol operating entirely without the need for intervention or maintenance by Crystal Labs or any other party. This paper seeks to summarize the implemented and planned architectures for the protocol.

1 Introduction

Decentralized finance (DeFi) was established to promote self-custodial, transparent, and accessible financial infrastructure, yet centralized venues are still preferred by most users today due to their superior latency, operational reliability, and liquidity depth.[1] Automated market makers (AMMs) currently are established as the leading architecture for on-chain asset exchange,[2,3] but their uninformed design and fixed pricing logic limits their suitability as primary discovery engines and constrains capital efficiency in many cases.[4,5] More recent designs such as request-for-quote (RFQ) systems delegate pricing logic to off-chain solvers; however, this reintroduces dependence on opaque infrastructure and discretionary operators, subsequently constraining transparency and composability.[6,7] In addition, these architectures cannot function independently as primary trading venues the same way an order book or AMM can.[8] Recent advancements in virtual-machine (VM) performance and parallelized execution relax these historical constraints, enabling fully auditable exchange designs that can approach centralized performance along the dimensions most critical to price discovery: latency, throughput, determinism, and expressivity.[9–11]

Crystal is designed to address these structural constraints by utilizing a fully on-chain central limit order book that bridges the gap between the transparency, security, and permissionless nature of decentralized exchanges (DEXs) and the speed, capital efficiency, and lower fees of traditional centralized exchanges (CEXs). On a high-performance EVM blockchain, we facilitate the matching and settlement of orderflow entirely on-chain while preserving serial semantics to ensure deterministic price-time priority.[12] The system adopts a singleton architecture that coordinates per-market

order books and, where beneficial, composes with a constant-product curve as a liquidity backstop to stabilize long-tail markets. This shared execution layer further enables atomic cross-market transactions while providing standardized and permissionless interfaces for external integrations such as vaults and aggregators.

2 Architecture

Crystal's on-chain architecture follows a singleton pattern: a single smart contract ("Exchange") serves as the protocol's unified settlement layer, with execution delegated to individual Market contracts via `delegatecall`. Each market is between two ERC-20 tokens and the Exchange has built-in support for native token wrapping and unwrapping. Internally, user balances are held in a per-user, per-token ledger split between available and locked portions that are fungible across all markets. Each market maintains both a central limit order book and an integrated constant-product AMM fallback for one base/quote pair, with bitmap-indexed price levels, tick sizes that scale with price magnitude to preserve relative precision across wide ranges, client order IDs for in-flight tracking and cancellation, configurable taker fees, maker rebates, and builder code revenue sharing. The core protocol is entirely permissionless and composable, allowing any contract to deposit, place orders, and settle atomically within a single transaction. The Vault framework demonstrates this advantage: vaults custody user deposits, delegate order placement to a designated operator, issue shares with proportional redemption, and enforce withdrawal lockups and order caps, all without any protocol-level privilege. All market and order state exists entirely on-chain, and emitted events provide a canonical record for off-chain indexers.

2.1 Exchange Contract

The Exchange (`Crystal.sol`) serves as the unified settlement layer. It holds all user balances, all order state, and all market configurations for the protocol. This contrasts with traditional factory-based architectures where each trading pair is deployed as an independent stateful contract.

The singleton architecture yields several advantages. Users can grant a token approval just once to the Exchange, and that approval persists across all current and future markets. Integrators interact with one canonical contract address, reducing exposure to potential attacks that may arise if arbitrary parties deploy lookalike pools. Indexing efficiency is greatly simplified because the standard RPC interface filters events via an address set, and a singleton architecture emits all protocol events from one address. Atomic composability is also simplified: a user can deposit once, execute across multiple markets in a single transaction, and net settlement across positions before any external transfer occurs.

For each token balance per user, the Exchange maintains a single storage slot encoding two separate values. This balance ledger is defined as

$$B = B_{\text{locked}} \cdot 2^{128} + B_{\text{available}}$$

The available balance may be withdrawn or committed to new orders. The locked balance represents the locked collateral on any outstanding limit orders. When an order is placed, balances update according to

$$B'_{\text{available}} = B_{\text{available}} - \delta, \quad B'_{\text{locked}} = B_{\text{locked}} + \delta$$

When an order is filled, the locked amount is subtracted and the equivalent value is converted to the corresponding asset and added to the available balance. If an order is canceled, the locked

amount is simply released back into the available balance. This separation ensures that all resting orders are fully collateralized, eliminating the need for balance checks during the matching process.

The Exchange does not implement any matching logic directly. Instead, it invokes a `delegatecall` targeting the relevant market, executing the Market's matching logic within the Exchange's storage context. Markets therefore act as stateless templates parameterized at deployment, with all persistent state stored in the Exchange.

In addition to the default parameters, users may attach a 10-bit client order identifier in the range from 1 to 1023 to any order. This identifier provides a stable reference for order cancellation or replacement without requiring the tracking of internal protocol order identifiers through event logs. A cross-reference mapping resolves each client order identifier to its corresponding market, price, and index, enabling retrieval and modification using this stable identifier alone. This is especially useful for the cancellation in-flight orders, as a user does not have to wait for execution to assign an identifier to each order.

2.2 Market

A Market (`CrystalMarket.sol`) is a smart contract consisting of a matching engine defined over a single base and quote asset pair. It specifies the admissible price domain, the queuing and execution semantics of resting orders, and the fee rules governing matched trades. Importantly, each Market itself is stateless and should never be interacted with directly, as their sole purpose is to provide parameters for the central exchange.

Each Market instance is parameterized at deployment by an immutable tuple

$$\mathcal{M} = (Q,\, B,\, \tau,\, s,\, p_{\max},\, q_{\min},\, f_t,\, r_m)$$

where Q and B denote the quote and base asset addresses, τ is the minimum tick size, s is a scale factor used for decimal normalization, $p_{\max}$ is the maximum representable price, $q_{\min}$ is the minimum admissible order size, f_t is the taker fee rate, and r_m is the maker rebate rate. All parameters are immutable after deployment with the exception of fee rates, which may be adjusted through governance.

Valid order prices lie on a discrete lattice induced by the tick size τ. Tick spacing follows a graduated scheme in which effective tick width increases with price magnitude so that relative precision remains approximately constant across the full representable domain. This construction preserves basis point level granularity near the prevailing price while bounding the total number of valid price levels.

The matching engine follows price-time priority. At each valid price level, the Market maintains an ordered queue of resting limit orders; orders within a price level are matched according to arrival time with the earliest order filled first. Across price levels, matching starts with the most competitive order and continues in order until there is no remaining liquidity within the slippage limit. An incoming order executes against resting liquidity until it is fully filled or until execution constraints terminate matching. Any residual quantity may rest on the book, be canceled immediately, or be rejected depending on order type. A formal definition of the matching transition is provided in Appendix A.

If a market has the AMM backstop enabled, the Market additionally exposes constant product reserves (R_Q, R_B) satisfying the invariant inspired by Uniswap V2

$$R_Q \cdot R_B = k$$

If resting liquidity cannot fully satisfy an order within its limit price, execution may continue against

the automated market maker at the prevailing marginal rate subject to a fee γ. Execution across the order book and the pool selects the lowest cost liquidity source at each step while preserving atomic settlement. The AMM execution and hybrid execution rules are formalized in Appendix A.

Liquidity provider positions are represented by ERC-20 tokens encoding pro rata claims on reserves. Given initial deposits A_Q and A_B, the initial liquidity supply is defined as

$$L_0 = \sqrt{A_Q \cdot A_B}$$

Subsequent minting and burning preserve proportional ownership of reserves as fees accrue.

For each matched trade with notional value N, the taker pays a fee equal to $f_t N$, while the maker receives a rebate equal to $r_m N$. A configurable fraction of taker fees may optionally be routed to an on-chain builder identifier associated with an external integration, referral, or market creator.

2.3 Composability

The Exchange can be interacted with through an expansive set of permissionless endpoints. Any contract may deposit assets, place orders, cancel orders, and withdraw funds within a single atomic transaction, without any privileged role. This means anyone capable of transacting with the Exchange may trade under the same rules and constraints as any other externally owned address or smart contract.

This property is critical because decentralized finance systems are inherently compositional; for example, aggregators route across multiple venues. For an exchange to participate as a first-class component in this ecosystem, it must eliminate friction in enabling atomic programmatic access.

The Vault framework, implemented in `CrystalVault.sol` and `CrystalVaultFactory.sol`, serves as a concrete example of this composability. A vault is a standard smart contract with no protocol level privileges that constructs a managed trading product on top of the Exchange. Liquidity providers deposit base and quote assets into the vault, which then deposits those assets into the Exchange under its own address and receives balances indistinguishable from those of any other participant. The vault issues non transferable shares to liquidity providers representing proportional claims on its holdings. A designated operator submits orders using the vault's Exchange balance, and liquidity providers redeem shares directly against the vault for their pro-rata share of assets.

The vault interacts with the Exchange exclusively through the public interface. Deposits, order placement, cancellation, and withdrawals use the same functions, incur the same fees, and obey the same constraints as any other caller. The Exchange does not distinguish vaults from externally owned accounts or other contracts.

Vault share accounting follows deterministic rules. On the first deposit, the initial share supply is defined as

$$S_0 = \sqrt{A_Q \cdot A_B}$$

where A_Q and A_B denote the deposited amounts of quote and base assets respectively. For subsequent deposits, shares are minted proportionally according to

$$S = \min\left(\frac{A_Q \cdot S_{\text{total}}}{V_Q}, \frac{A_B \cdot S_{\text{total}}}{V_B}\right)$$

where S_{total} is the total outstanding share supply and V_Q, V_B are the vault's current holdings of quote and base assets. Shares are non transferable, so liquidity providers cannot transfer or sell

shares and may only redeem them directly against the vault.

Vaults may enforce configurable operational constraints. These include a minimum lockup period between deposit and withdrawal, a maximum number of concurrently open orders, and a requirement that the designated operator maintain a minimum ownership stake in the vault. These controls bound risk without introducing protocol level permissions.

The VaultFactory contract handles vault deployment, routes deposits and withdrawals with automatic native asset wrapping where applicable, and enforces global bounds on minimum deposit sizes and parameter ranges to mitigate denial of service vectors.

The vaults represent only a single example of a possible external protocol built on top of the Exchange. The same permissionless interface can support margin products, aggregators, liquidations for lending protocols, and any other smart contract logic that requires the exchange of any two assets.

3 Enabling Performant Markets

On-chain order books face an inherent limitation. Centralized matching engines run thousands of operations a second, an approach that seems prohibitively expensive on the EVM where every storage write costs thousands of gas and every operation must be deterministically reproducible.

Crystal's design addresses this challenge through a set of optimizations, which allow the protocol to support trading at gas costs competitive with AMM swaps while maintaining the expressivity of an order book.

3.1 Constant-Time Placement

Placing a limit order requires inserting a new data structure into the book at a specific price level. A naive implementation locates the insertion point by iterating over existing orders and price levels, resulting in placement cost proportional to the book's order count. Instead, Crystal achieves constant time order placement by indexing the price grid using a hierarchical bitmap structure.

Let T denote the maximum tick index supported by a market. The discrete tick space $\{0, 1, \ldots, T\}$ is partitioned into contiguous slots of 255 consecutive ticks (the highest bit of each 256-bit word is reserved as an initialization marker to distinguish touched slots from uninitialized storage). For a given tick index t, the slot index $\sigma(t)$ and intra-slot offset $\omega(t)$ are defined as

$$\sigma(t) = \left\lfloor \frac{t}{255} \right\rfloor, \quad \omega(t) = t \bmod 255$$

The protocol maintains two bitmap mappings. The primary bitmap $\mathcal{A}_1$ maps each slot index to a 256-bit word. Within this word, bit position ω is set if and only if the tick at index $\sigma \cdot 256 + \omega$ contains at least one resting order. The secondary bitmap $\mathcal{A}_2$ aggregates slot occupancy. In this bitmap, bit position σ is set if and only if the corresponding slot contains at least one non-empty price level.

To place an order at price p, the protocol computes the tick index $t = t(p)$ according to the tick mapping. It then retrieves the bitmap word $\mathcal{A}_1[\sigma(t)]$ and appends the order to the queue at price level t. The corresponding bit is set according to

$$\mathcal{A}_1[\sigma(t)] \leftarrow \mathcal{A}_1[\sigma(t)] \vee 2^{\omega(t)}$$

If the slot was previously empty, the secondary bitmap is updated as

$$\mathcal{A}_2 \leftarrow \mathcal{A}_2 \vee 2^{\sigma(t)}$$

This procedure requires at most two storage reads and two storage writes, independent of the number of populated price levels in the book.

Order cancellation follows the symmetric process. Removing the final order at a given price level clears the corresponding bit in the primary bitmap. If this removal causes the primary bitmap to become empty, the associated bit in the secondary bitmap is also cleared. The operation remains near-constant in computation, although the final cancellation incurs a marginally higher cost than normal as it must update the highest bid or lowest ask pointer.

This bitmap indexing scheme optimizes insertion and deletion operations at the expense of traversal complexity. Identifying the next executable price level requires scanning storage slots across the bitmap. Section 3.3 describes how the matching engine performs this traversal efficiently.

3.2 Batching

Each EVM transaction incurs a fixed base cost in addition to calldata costs proportional to message size.[13] For users submitting many orders per block, these fixed costs dominate execution cost. Crystal supports order batching in order to amortize transaction overhead across multiple operations executed atomically.

An action is defined as a single order operation, such as placement, cancellation, replacement, or trade (market order). The protocol defines an encoding for sequences of actions. Let a_i denote the i-th action in a batch, encoded as the tuple

$$a_i = (\texttt{op}_i, \texttt{params}_i)$$

where $\texttt{op}_i \in \{\texttt{LIMIT}, \texttt{CANCEL}, \texttt{REPLACE}, \texttt{MARKET}\}$ specifies the operation type and $\texttt{params}_i$ contains the operation specific parameters.

Actions are packed into a byte sequence with the operation type encoded in the leading bits of each action word. For a batch consisting of n actions, calldata is laid out as

$$\texttt{calldata} = \texttt{selector} \parallel \texttt{market} \parallel a_1 \parallel a_2 \parallel \cdots \parallel a_n$$

where $\parallel$ denotes byte concatenation. The selector identifies the batch entry point, the market address specifies the target order book, and actions follow sequentially.

During execution, the contract iterates through the action sequence and dispatches each operation to the appropriate handler. All actions execute within a single `delegatecall` context and share the same cached reads for market parameters and user balances.

For maximally compact submission, the Exchange exposes a fallback entry point that accepts raw packed calldata without a function selector. In this mode, the leading bytes encode the target market and operation mode, and the remaining bytes are decoded as an action sequence. This mechanism reduces calldata overhead by eliminating the four byte selector and enabling tighter packing.

If any action within a batch fails, the entire transaction reverts. Callers that require partial execution behavior may submit multiple transactions or deploy wrapper contracts that selectively catch and handle individual failures.

3.3 Matching Engine

Matching an incoming order against resting liquidity requires traversing price levels adhering to the sequence of price-time priority. Pointer-based data structures such as linked lists or heaps enable efficient traversal in conventional systems but impose prohibitive overhead on the EVM, where each pointer reference corresponds to a storage read with an associated high gas cost. Crystal's matching engine instead performs a linear scan guided by bitmap indexing when traversing price levels and accumulates execution results in memory before committing state changes to storage.

Consider an incoming buy order with limit price p_ℓ. The engine must locate all resting ask price levels in the interval $[p_{\text{best\_ask}}, p_\ell]$ in ascending order. Rather than maintaining explicit pointers between price levels, the engine queries the bitmap structure defined in Section 3.1. Starting from the best ask, it extracts the lowest set bit from the active primary bitmap word, processes the corresponding price level, clears the bit in a working copy, and repeats until either the bitmap is exhausted or the limit price constraint is violated. When a slot becomes empty, the engine consults the secondary bitmap to skip directly to the next non empty region of the price space.

Let k denote the number of distinct price levels traversed during matching. The traversal hence performs $O(k)$ bitmap reads with constant time bit extraction per level, independent of the total depth of the order book.

During matching, the engine maintains execution state in memory. Data is recorded in a contiguous memory buffer defined as

$$\mathcal{E} = [(p_1, q_1, u_1), (p_2, q_2, u_2), \ldots, (p_m, q_m, u_m)]$$

where each tuple records a fill at price p_i for quantity q_i against a resting order owned by user u_i. Memory writes incur a constant and minimal gas cost per word compared to storage writes, yielding substantial savings when orders match across many resting positions.

Additionally, order book updates must emit events for off-chain observability; for this, the engine accumulates event data in a dedicated memory buffer. Let μ_0 denote a fixed memory offset at which event data begins. For the i-th matched order, the engine writes

$$\texttt{mem}[\mu_0 + 32i] \leftarrow \texttt{encode}\big(\texttt{market}, p_i, \texttt{id}_i, q_i^{\text{filled}}, q_i^{\text{remaining}}\big)$$

A length counter stored at offset $\mu_0 - 32$ tracks the number of accumulated entries. After traversal completes, the engine emits a single event containing the entire encoded buffer.

Balance updates for matched counterparties are similarly deferred. The engine aggregates balance deltas in memory and performs a single storage write per distinct user. For a user u receiving fills across multiple price levels, the aggregate credit is computed as

$$\Delta_u = \sum_{i\,:\,u_i = u} q_i \cdot p_i$$

and applied in a single update to the user's balance slot. This batching reduces the number of storage writes from $O(m)$ to $O(d)$, where m is the number of individual fills and d is the number of distinct counterparties.

For an incoming order that traverses k price levels, consumes m individual resting orders, and settles against d distinct counterparties, the matching engine performs $O(k)$ bitmap reads, $O(m)$ memory writes for execution accumulation, and $O(d)$ storage writes for balance settlement. Since storage access dominates gas cost, total execution cost scales with the number of distinct counterparties.

3.4 Static Storage Keys

The EVM distinguishes between cold and warm storage access. The first read or write to a storage slot within a transaction incurs a cold access cost, while subsequent accesses to the same slot are charged at a lower warm access rate. We may instead use access lists,[14–15] which allow transactions to pre-declare storage slots that will be accessed during execution. Declared slots are treated as warm from the beginning of execution, reducing first access cost in exchange for additional calldata overhead.

For access lists to be effective, storage keys must be computable prior to execution. Crystal's storage layout is designed to ensure that all relevant keys are derived deterministically from known parameters, enabling off-chain construction of complete access lists.

All order book state is stored in mappings keyed by composite identifiers constructed from market and price parameters. For a market identified by M, a price level p, and an order index i, the storage key for an individual order is defined as

$$\kappa_{\text{order}}(M, p, i) = M \cdot 2^{128} + p \cdot 2^{48} + i$$

Price level aggregates are stored under keys defined as

$$\kappa_{\text{level}}(M, p) = M \cdot 2^{128} + p$$

Bitmap slots are stored under keys defined as

$$\kappa_{\text{bitmap}}(M, \sigma) = M \cdot 2^{128} + \sigma$$

where σ denotes the bitmap slot index derived from the tick. Given the market identifier, the target price, and the expected order indices, all relevant keys can be computed off-chain prior to transaction submission.

A user placing a limit order at price p in market M can therefore precompute the storage keys corresponding to the target price level, the relevant bitmap slots, and their own balance entries. For order cancellation, the storage key is known from the client order identifier cross reference. The access list is constructed as

$$\mathcal{L} = \big((\texttt{Exchange}, \kappa_1), (\texttt{Exchange}, \kappa_2), \ldots, (\texttt{Exchange}, \kappa_n)\big)$$

where each κ_i denotes a precomputed storage slot associated with the Exchange contract.

Let n denote the number of distinct storage slots accessed during execution. Without an access list, total storage access cost scales linearly with n at the cold access rate. With a complete access list, execution incurs additional calldata overhead per slot and all accesses are charged at the warm rate. The resulting net gas cost is reduced by a constant amount per accessed slot. For transactions that touch many storage slots, such as batch submissions or matches spanning multiple price levels, this reduction can become significant.

Matching operations present an additional constraint because the set of resting orders consumed depends on book state at execution time. However, the set of price levels that may be traversed is bounded by the order's limit price and the prevailing best bid or ask, both of which are observable off-chain. Submitters may construct conservative access lists that include all storage slots corresponding to price levels within this executable range. Although some declared slots may remain unused in rare cases, the cost of unused declarations from access lists is typically lower than the additional cost of cold access that would otherwise be incurred.

3.5 Indexing

Off-chain systems require efficient access to order book state; user interfaces must display resting orders, and indexing services must reconstruct book state from on-chain data. The standard mechanism for reading contract state without submitting a transaction is `eth_call`, which executes view functions against the current state and returns the result to the caller.

Although `eth_call` does not consume real gas, RPC providers impose execution limits to bound computational cost. A naive order book query that iterates through all price levels or all resting orders can exceed these limits in active markets, causing the call to fail.

Crystal's architecture simplifies indexing by consolidating all state behind a single contract address. Queries for any market, any user's orders, or any price level are routed through the single Exchange contract. Indexers maintain a single subscription filter, and off-chain systems do not need to track per-market contract addresses.

To ensure bounded execution, view functions that expose order book state accept explicit range parameters. The price level query interface accepts a starting price, a traversal direction, a maximum price distance, a bucket interval, and an upper bound on the number of returned entries. Formally, the interface is

$$\texttt{getPriceLevels}(M, \texttt{ascending}, p_{\text{start}}, \Delta_{\max}, \texttt{interval}, n_{\max})$$

The function traverses at most $n_{\max}$ buckets, each aggregating liquidity over a fixed price interval, and returns encoded results. Callers can paginate through the full order book by issuing successive queries with updated starting prices and bounds.

User order queries are similarly bounded. The interface

$$\texttt{getAllOrdersByCloid}(u, n_{\max})$$

returns at most $n_{\max}$ orders for a given user u, indexed by client order identifier. Bounding iteration in this manner ensures that query execution remains within read operation gas limits regardless of the total number of open orders associated with the user.

Query results are returned as packed byte arrays. Each price level or order is encoded into a fixed width word, and responses take the form

$$\texttt{response} = [w_1][w_2]\cdots[w_n]$$

where each word w_i encodes price, quantity, and associated metadata within a single 256-bit value.

For historical queries and state reconstruction, indexers should rely on event streams emitted by the Exchange. Because all order operations emit events from the single contract address, indexers can maintain a single event subscription. Events encode sufficient information to reconstruct order book state incrementally. For example, an order update event takes the form

$$\texttt{OrdersUpdated}(\texttt{market}, [(\texttt{price}_1, \texttt{id}_1, \texttt{size}_1), \ldots])$$

Each event includes the market identifier and a batch of order state changes, allowing off-chain systems to update local book representations without repeated polling of view functions.

4 Security and Transparency

Crystal is designed to align with the normative principles of decentralized finance. The protocol operates without privileged operators, executes deterministically according to published rules, and exposes all state transitions to public verification.

The Exchange defines a minimal governance surface. Governance may adjust fee parameters within bounded ranges, deploy new markets, and update canonical market designations. No governance action can access funds or modify resting orders. The matching engine, balance accounting, and settlement logic are immutable after deployment. Users interacting with the Exchange do not rely on ongoing operator discretion because execution rules cannot change after deployment.

All participants interact with the Exchange through the same public interface. Every order enters the same matching process and executes under identical rules regardless of the identity of the submitter. This uniform treatment eliminates advantages derived from privileged infrastructure access and ensures that execution outcomes depend solely on publicly observable state.

Every state transition occurs on-chain and is publicly verifiable. Deposits, withdrawals, order placements, cancellations, and fills emit events from the Exchange address. Any observer can reconstruct the full state of any market or account from public chain data alone. The order book state at any block height is fully determined by the sequence of transactions included up to that block.

The fully on-chain execution model provides an immutable audit trail for all trades. Funds move directly between user controlled balances through deterministic contract logic. Each fill can be traced from taker to maker with exact prices, quantities, and timestamps. This transparency supports operational oversight for participants subject to compliance requirements and counters the association between decentralized trading systems and opaque execution.

Orderbook-based execution substantially mitigates sandwich attacks relative to automated market makers. Contrary to AMM liquidity, a limit order is one-way and specifies a fixed execution price. If the market is moved away from that price in the case of attempted price manipulation, the opposing bids will remain in place rather than move along with the market as in the case of an AMM. Taker orders remain exposed to adverse price movement and still must set a reasonable slippage limit, but the risk and impact of sandwich attacks are greatly reduced. On high throughput chains with sub second finality and unique transaction broadcast mechanisms, the window for observing and front running pending transactions compresses further, reducing the practical attack surface.

References

[1] Budish, E., Cramton, P. and Shim, J. (2015) 'The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response'. *Quarterly Journal of Economics*, 130(4), pp. 1547–1621.

[2] Adams, H., Zinsmeister, N. and Robinson, D. (2020) *Uniswap v2 Core*. Uniswap Labs. Available at: `https://uniswap.org/whitepaper.pdf`

[3] Adams, H. et al. (2021) *Uniswap v3 Core: Concentrated Liquidity*. Uniswap Labs. Available at: `https://uniswap.org/whitepaper-v3.pdf`

[4] Milionis, J., Moallemi, C.C., Roughgarden, T. and Zhang, A.L. (2024) 'Automated Market Making and Loss-Versus-Rebalancing'. *Operations Research*. Available at: `https://arxiv.org/abs/2208.06046`

[5] Angeris, G. and Chitra, T. (2020) 'Improved Price Oracles: Constant Function Market Makers'. arXiv preprint arXiv:2003.10001. Available at: https://arxiv.org/abs/2003.10001

[6] Chitra, T., Kulkarni, K., Pai, M. and Diamandis, T. (2024) 'An Analysis of Intent-Based Markets'. arXiv preprint arXiv:2403.02525. Available at: https://arxiv.org/abs/2403.02525

[7] 0x Project (2021) 'RFQ Liquidity'. Available at: https://0x.org/docs/introduction/0x-rfq

[8] Bachu, B., Wan, X. and Moallemi, C.C. (2024) 'Quantifying Price Improvement in Order Flow Auctions'. arXiv preprint arXiv:2405.00537. Available at: https://arxiv.org/abs/2405.00537

[9] Lin, H. et al. (2025) 'Operation-Level Concurrent Transaction Execution for EVM Blockchains'. *Proceedings of EuroSys 2025*. Available at: https://yajin.org/papers/EuroSys_2025_ParallelEVM.pdf

[10] Monad Labs (2025) 'Parallel Execution'. Monad Developer Documentation. Available at: https://docs.monad.xyz/monad-arch/execution/parallel-execution

[11] Buterin, V. (2021) 'Endgame'. Ethereum Research. Available at: https://vitalik.ca/general/2021/12/06/endgame.html

[12] dYdX Trading Inc. (2023) *The dYdX v4 Protocol*. Available at: https://dydx.exchange/v4-whitepaper.pdf

[13] Ethereum Foundation (2019) 'EIP-2028: Transaction Data Gas Cost Reduction'. Available at: https://eips.ethereum.org/EIPS/eip-2028

[14] Ethereum Foundation (2020) 'EIP-2929: Gas Cost Increases for State Access Opcodes'. Available at: https://eips.ethereum.org/EIPS/eip-2929

[15] Ethereum Foundation (2020) 'EIP-2930: Optional Access Lists'. Available at: https://eips.ethereum.org/EIPS/eip-2930

Appendix A. Formal Execution Semantics

This appendix provides a formal definition of matching and hybrid execution semantics referenced in Section 2.2.

A.1 Matching Transition

Let $q_{\text{in}}^{(i)}$ denote the remaining incoming quantity prior to matching against the i-th resting order, and let $q_i^{(i)}$ denote the available quantity of that resting order. The matching process induces the state transition

$$(q_{\text{in}}^{(i+1)}, q_i^{(i+1)}) = \left(q_{\text{in}}^{(i)} - \min(q_{\text{in}}^{(i)}, q_i^{(i)}),\ q_i^{(i)} - \min(q_{\text{in}}^{(i)}, q_i^{(i)})\right)$$

Execution proceeds iteratively until

$$q_{\text{in}} = 0, \quad p \notin \mathcal{P}_\ell, \quad \text{or} \quad \text{execution constraints terminate matching}$$

where $\mathcal{P}_\ell$ denotes the admissible price set induced by the order's limit price and any additional constraint flags.

A.2 Automated Market Maker Execution

When automated market maker execution is enabled, reserves (R_Q, R_B) satisfy the constant product invariant

$$R_Q \cdot R_B = k$$

A trade exchanging ΔB units of base for quote updates reserves according to

$$(R_Q - \Delta Q)(R_B + \Delta B) = k$$

with executed quote amount

$$\Delta Q = (1 - \gamma)\frac{R_Q \cdot \Delta B}{R_B + \Delta B}$$

where $\gamma \in [0, 1)$ denotes the automated market maker fee.

A.3 Hybrid Execution Rule

Let $p_{\text{book}}(x)$ denote the marginal execution price for an incremental quantity x against resting limit orders, and let

$$p_{\text{amm}}(x) = \frac{R_Q}{R_B + x}$$

denote the marginal automated market maker price prior to fees. Hybrid execution selects liquidity sources by minimizing effective marginal price subject to the order's limit constraint. For a buy order, execution selects

$$\min\left(p_{\text{book}}(x), \frac{p_{\text{amm}}(x)}{1 - \gamma}\right)$$

at each marginal increment until termination. Sell-side execution follows an opposite symmetric rule. This construction ensures deterministic selection of the lowest cost liquidity source while preserving atomic execution across mechanisms.

Disclaimer

This paper is for general information purposes only. It does not constitute investment advice or a recommendation or solicitation to buy or sell any investment and should not be used in the evaluation of the merits of making any investment decision. It should not be relied upon for accounting, legal or tax advice or investment recommendations. The views expressed herein are solely those of the authors and are not made on behalf of Crystal Labs or any of its affiliates. These views do not necessarily reflect the opinions of Crystal Labs, its investors, affiliates, or any individuals associated with them, and are subject to change without notice.