Sei: The Sector Specific Layer 1

Sei Labs

1 Abstract

We propose a sector-specific layer 1 blockchain that is specialized for trading. At a protocol level, we introduce novel approaches for transaction ordering, block processing, and parallelization. Additionally, we offer a heavily optimized order placement and matching engine that is built into the chain itself.

2 Introduction

Decentralized exchanges (DEXes) are one of the killer applications of crypto. They are everywhere, spanning across Automated Market Makers, Orderbooks, NFT marketplaces, and in-game exchanges. DEXes command the largest network effect, and major ecosystems get built around them.

Ironically, decentralized exchanges are also the most underserved application in crypto. They demand a unique level of requirements for reliability, scalability, and speed that no other apps need. If a large exchange goes down for a few moments it is catastrophic, but the same downtime is far more tolerable for most other application types.

As a result, we propose Sei, which is a sector-specific L1 blockchain that is specialized for trading, and will bridge the performance gap between centralized and decentralized exchanges. At a protocol level, Sei makes use of Twin-Turbo consensus and multiple degrees of parallelization to help reduce latency and improve throughput. Sei also has a native order matching engine, which fills trades using frequent batching auctions, helping with price fairness and frontrunning prevention. This paper will also cover some of the user experience customizations that Sei has, including native price oracles and transaction order bundling.

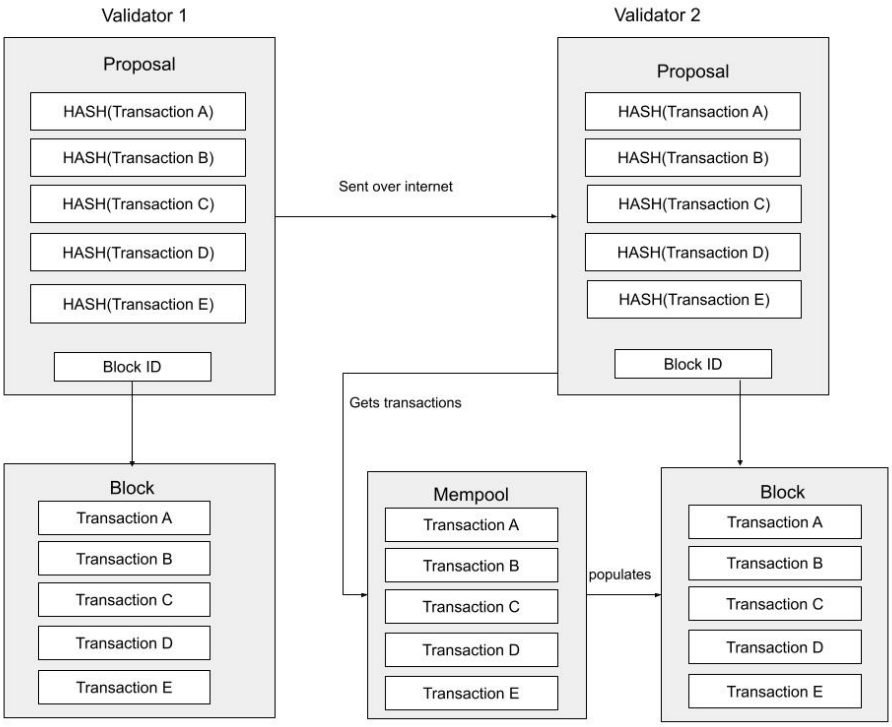


Fig.1. Block proposals with transaction identifiers

3 Protocol Improvements

Sei is built using the Cosmos SDK and Tendermint Core. At the time of writing, Sei has forked both Cosmos SDK and Tendermint Core and has added significant specialized functionality and optimizations.

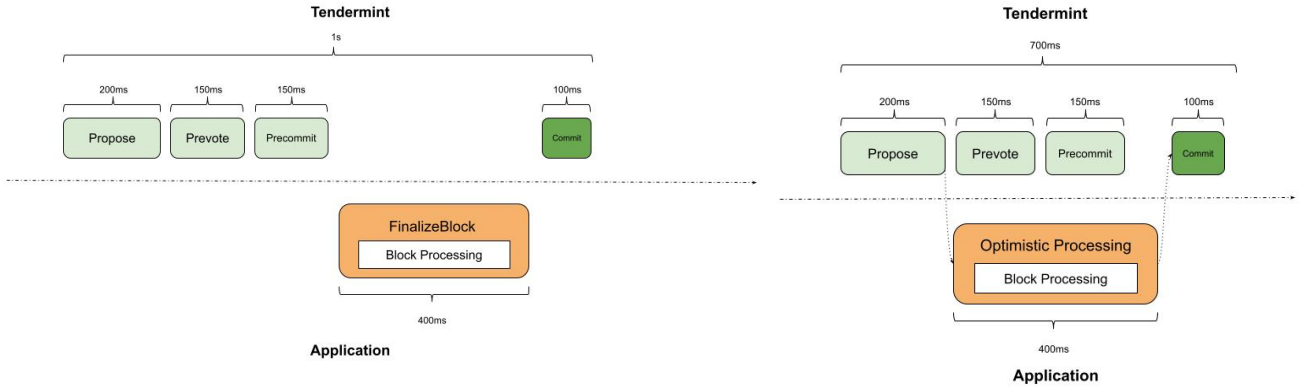
3.1 Twin-Turbo Consensus

3.1.1 Intelligent Block Propagation

Once a full node receives a transaction from a user, it must broadcast that to other nodes in the network. Full nodes will randomly gossip this transaction to other nodes in the network. Once a transaction is received by a validator, it verifies the validity of the transaction, and adds that transaction to that validator's local mempool.

Block proposers will look at the current state of their mempool and propose a block to be committed. Since most, if not all, transactions will already have been received by validators through the transaction dissemination approach discussed above, proposers will include unique transaction identifiers in the block proposal, along with a reference to the full block. Proposers will first disseminate the proposal to other validators in the network, followed by the entire block (containing the full contents of each transaction). The proposal will get sent as one message, whereas the entire block will get broken up in parts and gossiped to the network. If a validator has all of the transactions from the proposal in its local mempool, it will reconstruct the entire block from its mempool rather than waiting for all block parts to arrive. If it doesn't have all transactions, it will wait to receive all of the block parts from the network, and will construct the block with all of its transactions.

This process significantly decreases the overall amount of time that a validator waits to receive a block. Once validators have all of the transactions as part of



(a) (b)

Fig.2. Block processing with example times (a) Block processing after precommit (b) Optimistic block processing

the block proposal, they will follow the Tendermint BFT consensus to agree on the transaction ordering. In particular, there will be a prevote step, a precommit step, and a commit step before the block and the associated state changes have been committed to the blockchain.

3.1.2 Optimistic Block Processing

As part of Tendermint consensus, validators will receive a block proposal, verify the validity of the block, and then proceed to the prevote steps.

Rather than waiting till after the precommit step to begin transaction processing (figure 2a), validators will start a process concurrently to optimistically process the first block proposal they receive for any height (figure 2b). The optimistic block processing will write the candidate state to a cache.

If that block gets accepted by the network, then the data from the cache will get committed. If the network rejects the block, then the data from the cache will get discarded, and future rounds for that height will not use optimistic block processing.

The theoretical improvement in latency due to optimistic block processing is

$$min(T_{prevote} + T_{precommit}, N * T)$$

where $T_{prevote}$ is the prevote latency, $T_{precommit}$ is the precommit latency, N is the number of transactions and T is the average latency of a single transaction.

3.2 Parallelization

As part of the Cosmos SDK, when validators receive a block and start processing it to update the state of the network, they will initially run BeginBlock logic, followed by DeliverTx logic, followed by EndBlock logic. Each of these are completely configurable, and Sei has configured DeliverTx and EndBlock to parallelize transaction processing, as shown in figure 3.

Sei first processes all transactions in a block during the DeliverTx phase. This results in state changes for most types of transactions (sending tokens, governance proposals, smart contract invocations, etc.). However, central limit order book (CLOB) related transactions only go through basic processing during the DeliverTx

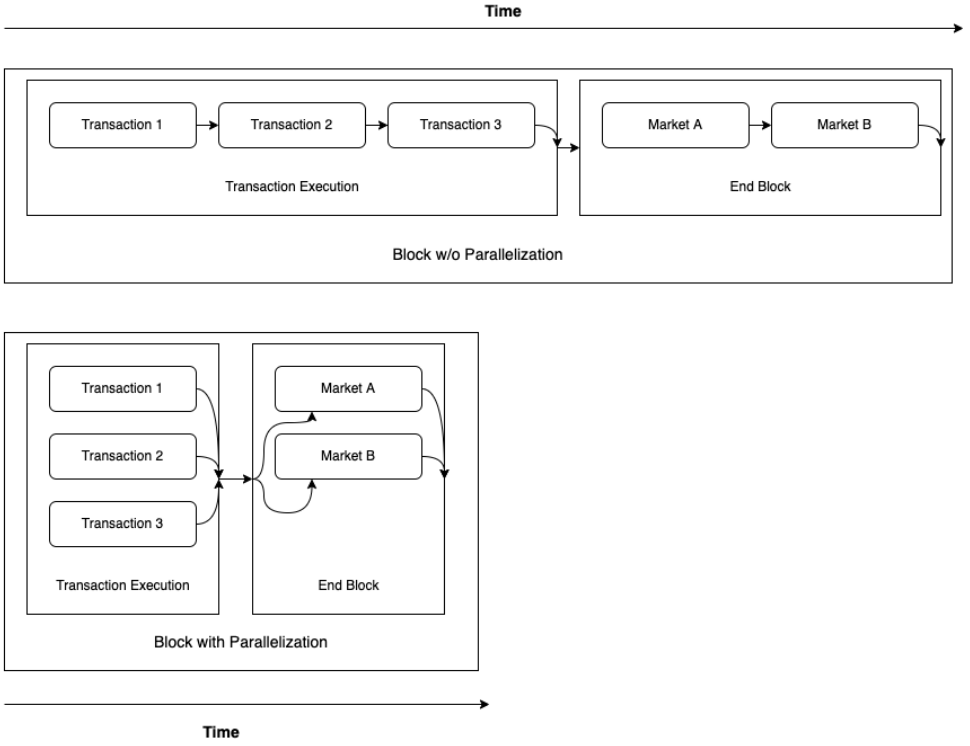


Fig.3. Block processing with and without parallelization

phase, and have most of their state changes get applied during the EndBlock logic. This is done to support frequent batch auctions, where orders are aggregated and a uniform clearing price is calculated at which to execute orders (see section 4.1.3 for more information).

Sei has added in parallelization to both DeliverTx and EndBlock to get optimal performance.

3.2.1 DeliverTx Transaction Parallelization

Rather than processing transactions sequentially during DeliverTx, Sei processes transactions in parallel (see figure 4). This allows multiple transactions to be processed simultaneously, which leads to improved performance. Data for Sei is persisted in a key-value store. To prevent race conditions and nondeterminism, Sei needs to ensure that multiple parallel processes are not updating the same key.

This is achieved by maintaining a mapping of transaction message types to the keys they need to access (dependency mappings). Messages that are updating different keys can be run in parallel, but messages updating the same key will need to be run sequentially and in a deterministic order (the ordering is determined by the ordering of transactions in the block).

Prior to executing transactions for a block, any dependencies between transactions are identified by constructing a directed acyclic graph (DAG) of dependencies between the different resources that each message in each transaction needs to use.

An example of a basic dependency mapping is for messages related to an example X module. All messages to this module update the same key ABC, so all of these messages will need to be run sequentially in the same branch of the access DAG.

In many cases the contents of the message are used to give further parallelism. For example, transfers of tokens from (1) account A to B and (2) account C to D can be run in parallel since they update different keys.

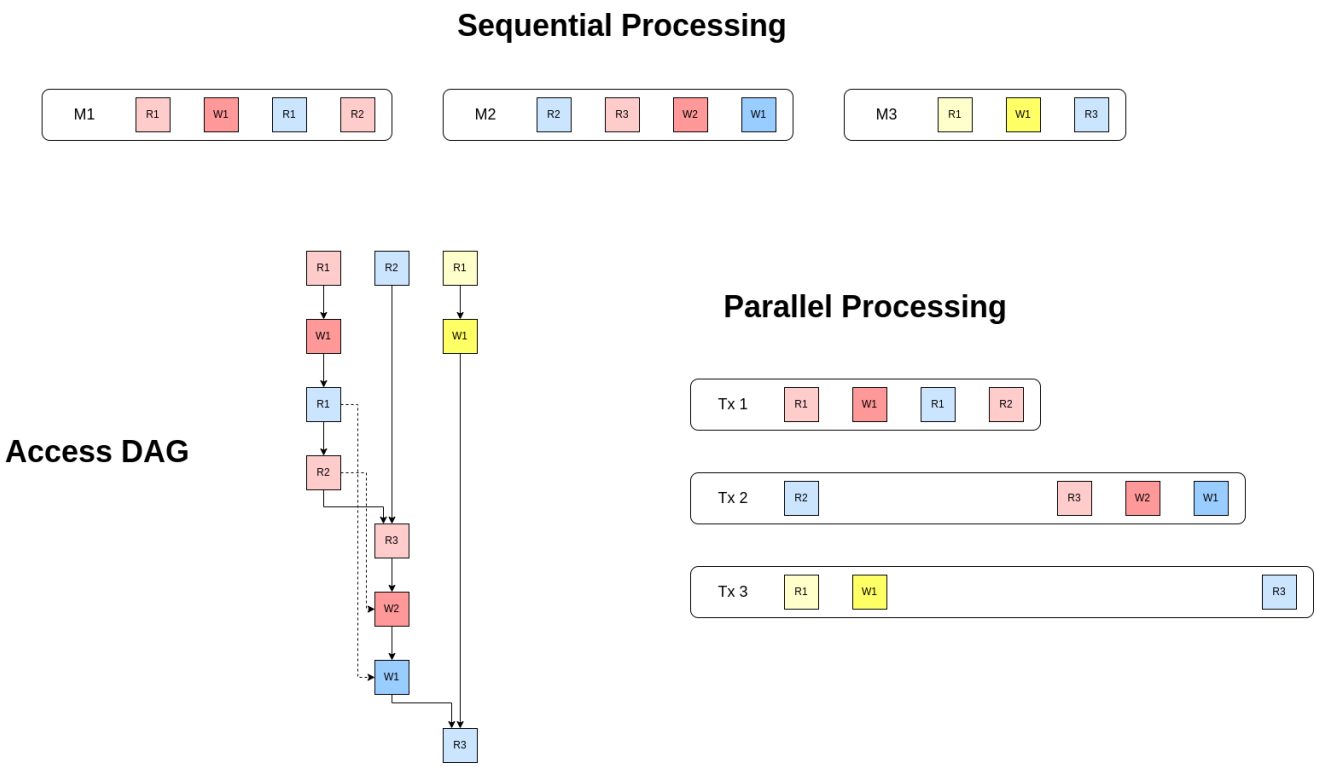


Fig.4. Access DAG for parallel processing

However, only defining the mapping based on the message type (and ignoring message contents) will result in these two transfers running sequentially.

To give flexibility around this type of parallelism, dependency mappings can be defined as templates, which will get filled with more granular resources at runtime. In this token transfer example, the sender and receiver accounts will be passed into the template to yield more granular parallelism.

For message types that are defined by the chain (staking, oracle updates, bank sends, etc.), mappings are set at blockchain genesis, and can be updated via a governance proposal. There is one special case for the message type related to gas fee collections, which affects every transaction. This is handled by writing data to an in-memory datastore that is flushed at the end of the DeliverTx logic.

For message types that are set by developers building on Sei, smart contracts will need to define their own resource dependencies. These will be set at contract initialization and can be updated by the smart contract admin through update transactions. If the dependencies are properly written, then smart contracts will benefit from parallelism and pay cheaper gas fees. If no dependencies are defined, then smart contracts will run sequentially and block other transactions from running. Since they are blocking the rest of the network, transactions to those smart contracts will need to pay greater gas fees. If the dependencies for a specific smart contract are incorrectly defined, then messages for that particular smart contract will fail and greater gas fees will be charged, but the network overall will be unaffected and other messages will succeed.

3.2.2 Market Based Parallelization

At the end of the block, all CLOB related orders will be processed by the native order matching engine. Rather than processing orders sequentially, Sei will pro-

cess independent CLOB related orders in parallel at the end of the block. Two orders are independent if they do not affect the same market in the same block. By default, the chain will assume all orderbook transactions touching different markets are independent, unless developers explicitly define dependencies between different markets. These dependencies will be defined when a smart contract is deployed. If these dependencies are defined incorrectly, then transactions to the dependent smart contract will fail.

3.3 Native Price Oracles

Sei has a native price oracle to support asset exchange rate pricing. Validators are required to participate as oracles in order to ensure the most reliable and accurate pricing for assets. In order to maintain freshness of oracle pricing, voting windows can be configured to be as small as 1 block long, resulting in rapid price updates and fresh asset pricing.

In the vote step for a voting window, the validator provides their proposed exchange rates for that window. At the end of the voting period, all of the exchange rate votes are accumulated and a weighted median is computed (weighted by validator voting power) to determine the true exchange rate for each asset.

There are penalties for non-participation and participation with bad data. Validators have a miss count that tracks the number of voting windows in which a validator has either not provided data or provided data that deviated too much from the weighted median. In a given number of voting periods, if a validators miss count is too high, they are slashed as a penalty for misbehaving over an extended period of time.

4 Native Order Matching Engine

Sei offers the functionality of a general purpose blockchain (i.e. allowing users to transfer assets and deploy smart contracts). In addition to that, Sei has created an order placement and matching engine (referred to as the "matching engine" for the remainder of the paper) that can be used by any exchanges building on top of Sei.

4.1 Deploy CLOBs with Sei

The matching engine allows decentralized exchanges that are building on top of Sei to deploy their own CLOBs. The matching engine maintains their respective order books at a chain level, and provides functionality to create markets and allow users to trade.

4.1.1 Creating Markets

Creating a new orderbook (equivalent to creating a new market) can be done by a two-transaction process:

1. Deploying a smart contract onto Sei
2. Submitting a transaction to add a new order book to the registered smart contract. A new order book proposal needs to include the asset, the base denomination of pricing, and the minimum price interval

4.1.2 Order Types

The matching engine supports the following order types

- Limit orders: This is an order to buy/sell an asset at a specified price or better. When a limit order is submitted, it is generally (see section 4.1.3 for more information) added directly to the order book and will be matched against market orders that come in.
- Market orders: This is an order to buy or sell an asset at the best available price. Market orders will get executed immediately if there is any liquidity in the order book (i.e. there are any limit

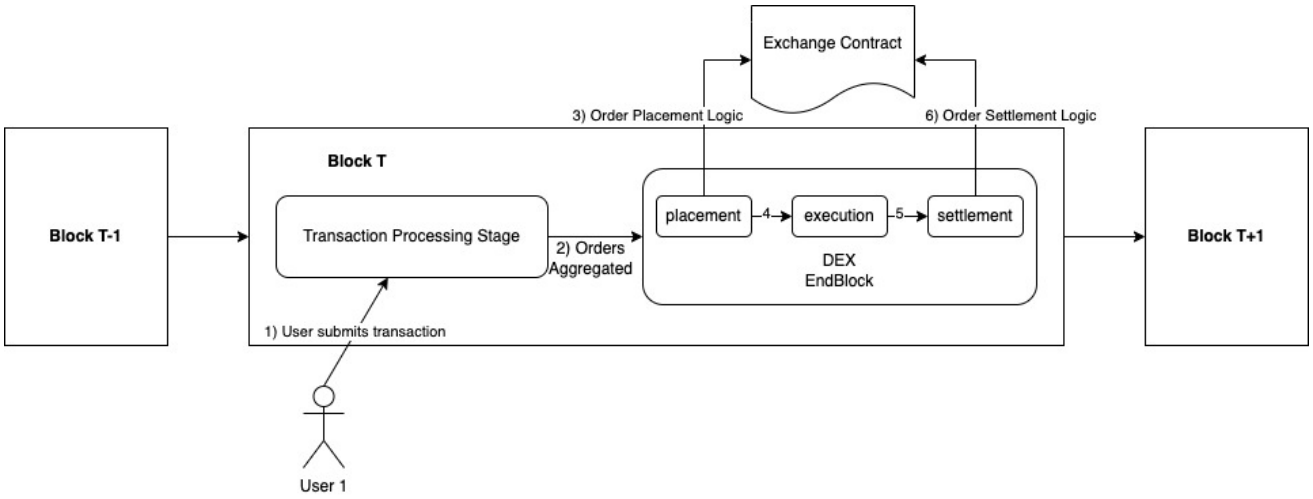


Fig.5. Lifecycle of a transaction

orders to match the market order that is submitted). To prevent orders getting filled at prices that are wildly different from what users expect, users placing trades can also submit a max slippage parameter.

- Fill-or-kill order: This is a special market order type in which either the entire order gets executed immediately if there is enough liquidity in the order book, or the order gets canceled. There is no partial execution with fill-or-kill orders.

- Stop-loss order: This is an order to close out a position by buying or selling a security at the market price when it reaches a certain price known as the stop price. The stop prices of these orders will be visible on chain.

- Cancel order: This will remove an order from the order book.

Partial executions, where only part of the overall order is executed, are possible for limit, market, and stop-loss orders if there is not enough liquidity to fill the entire order.

4.1.3 Lifecycle Of An Order

All CLOB related transactions will be executed atomically in the scope of a block (as opposed to other architectures, such as Serum, that require 3 separate transactions to handle order matching/execution).

Transactions related to the matching engine will be sent to the dex module, as shown in figure 5.

One transaction may be composed of one or more orders (see section 4.4.1 for more information). Upon submission, the transaction handler processes the transaction by adding the orders included from each transaction into the dex module‘‘s internal MemState (Figure 5.1).

While processing each block, the dex module has an EndBlocker hook that processes orders recorded in the MemState in bulk (Figure 5.2). Specifically, when dex module EndBlocker hook is invoked, orders across transactions will be aggregated by market (i.e. all orders for a BTC perpetual), and combined into one smart contract call for that particular market (see section 4.4.2 for more information about chain level bundling).

The chain will then call the smart contract associated with that market (i.e. calling a perpetual ex-

change smart contract), which has all of the logic defined for how to interact with the matching engine (Figure 5.3). The smart contract will implement its own custom logic, and then call the matching engine (Figure 5.4).

The matching engine will first process all order cancellations. This will remove the associated limit orders from the order book store.

Then all limit orders will be added to the orderbook. This ensures that orders are getting filled with maximal liquidity.

Then, the matching engine will process market orders. A uniform clearing price will be calculated (see section 4.3 for more information) and all market orders will be filled at that price. If there is not enough liquidity to fill all market orders, then the orders that accept greater slippage will be prioritized.

Then, matching limit orders will get processed. If any limit orders can be filled, they will be filled at the best price. For example, assume the order book store has sell orders for P_1 with quantity Q and P_2 with quantity Q, where $P_1 < P_2$. A buy limit order exists for $P_3 > P_2$ for quantity 1.5 * Q. In this case, the buy limit order will get filled by purchasing Q shares at price P_1 and 0.5 * Q share at price P_2.

Finally, any unfilled market orders will expire. At the conclusion of the matching engine logic, it will call the relevant smart contract to handle asset settlement (Figure 5.6).

4.1.4 Hook Support

Sei allows contracts to register hooks with the network. The registered hooks will be invoked every block and allow operations like flashloan payback to happen in the same block as any associated trade settlements. Specifically, a contract can define two hooks. The first one is called at the beginning of a block to give contracts an opportunity to prepare for any potential trade that may happen in the same block. The second is called at the very end of a block, after order matching and settlement, allowing contracts to perform any post-trade logic if needed.

4.2 Asset Agnostic Order Book

The matching engine does not require tokens to be traded, and instead offers a flexible interface that lets decentralized exchanges decide how to represent assets. For example, instead of tokenizing positions, decentralized exchanges can track positions as a list in their smart contract state.

4.3 Frequent Batch Auctioning

Executing orders in a sequential order encourages validators to order transactions in ways that can be profitable for themselves. For example, when they see an incoming market order, they can include their own market order to buy that asset, and their own limit order to sell that asset at a higher price before the incoming transaction is processed. To discourage this form of MEV (maximal extractable value) Sei aggregates all market orders and executes them at the same uniform clearing price.

For example, if the order book has two asks (sell orders) orders for prices P_1 and P_2 and there are two incoming bids (buy orders) B_1 and B_2, then both B_1 and B_2 will get executed at the uniform clearing price

$$\frac{P_1 + P_2}{2}$$

rather than having B_1 getting executed at P_1 and B_2 getting executed at P_2. This results in the existing limit orders getting filled at their intended price (P_1 and P_2) while the incoming market orders get a fairer price.

4.4 Transaction Order Bundling

Sei offers multiple layers of order bundling to improve user experience and performance, outlined in the sections below.

4.4.1 Client Order Bundling

Sei transactions can be composed of orders going to multiple trading markets (even those spanning smart contracts, i.e. orders to both a BTC/USDC spot pair and a BTC perpetual exchange). During block processing, Sei will correctly route all orders to their respective smart contracts. This will help market makers cut down on gas costs associated with updating their positions.

4.4.2 Chain Level Order Bundling

Each orderbook related transaction will require instantiating the virtual machine (VM). Rather than having multiple VM instantiations, Sei bundles all orders across all transactions (per market) and only performs one VM instantiation. This reduces latency by roughly 1ms per order, which is substantial in periods of high throughput.

4.5 Trading Fees

The matching engine will not charge any trading fees at the chain level at launch. Governance can choose to start applying trading fees in the future. Decentralized exchanges that are building on top of Sei can add in their own trading fees depending on the experience they want to offer their users. This would be defined at the smart contract level, and will be easily configurable for developers.

4.6 Middleware

The matching engine is an extremely flexible and composable system, so it's anticipated that projects will build decentralized exchange middleware on top of the matching engine. This would be middleware that allows certain new types of DEXes to be created. For example, it's anticipated that middleware will be created to allow market makers to post offers that are not fully provisioned. This would involve writing middleware smart contracts that are built on top of the matching engine.