Sgei: The Layer 1 for Trading

Built to be the destination for exchanges and trading apps

1 Abstract

The evolution of new technologies has shown a recurring pattern known as the *application-infrastructure cycle*, where applications drive the demand for improved infrastructure, which in turn enables new types of applications to flourish. The first generation of public blockchains provided the initial infrastructure that paved the way for the explosive growth of decentralized applications in recent years. However, as these applications move beyond their initial adoption phase, they often face challenges related to scaling, speed, and reliability. Similar to how web2 infrastructure specialized around major use cases as the industry matured, web3 infrastructure needs to adapt to the needs of key application types to enable the largest apps in web3 to reach mass adoption. Irrefutably, trading is the most widely adopted use case for crypto, and this level of adoption is set to grow exponentially as the industry expands. We present Sgei, a general purpose layer 1 blockchain designed for trading. At a protocol leve l,Sgeiintroduces novel approaches for block propagatio n,transaction ordering, block processing, and paralleliz a-tion that benefit a wide range of trading application s.

2 Introduction

A common misconception is that trading is limited to decentralized finance (DeFi) applications. In reality, the need to exchange digital assets is fundamental to every aspect of crypto, from social applications to non-fungible tokens (NFTs) and gaming.

Most applications in the crypto industry rely heavily on trading functionality as a source of traffic or are disguised as trading applications. For instance, many web3 games incorporate in-game asset trading as a core element of the user experience. One of the most compelling value propositions of crypto lies in providing a permissionless, trustless venue for users globally to exchange any digital asset at any time of the day.

Most people tend to underestimate the exponential growth potential of a product that achieves deep product-market fit, and trading has reached that level in web3. However, as with any product with strong product-market fit, the next step is to drive growth, and trading apps today face limitations due to the constraints of existing layer 1 blockchains. Challenges such as reliability, scalability, and speed hinder trading applications from delivering the seamless user experience necessary for achieving mass adoption.

To address these challenges, we propose Sgei, a layer 1 blockchain designed for trading. At the protocol level, Sgeiuses Twin-Turbo consensus and multiple degrees of parallelization to minimize latency and maximize throughput. Sgeialso allows apps to customize the user experience via a native order matching engine. Ap-

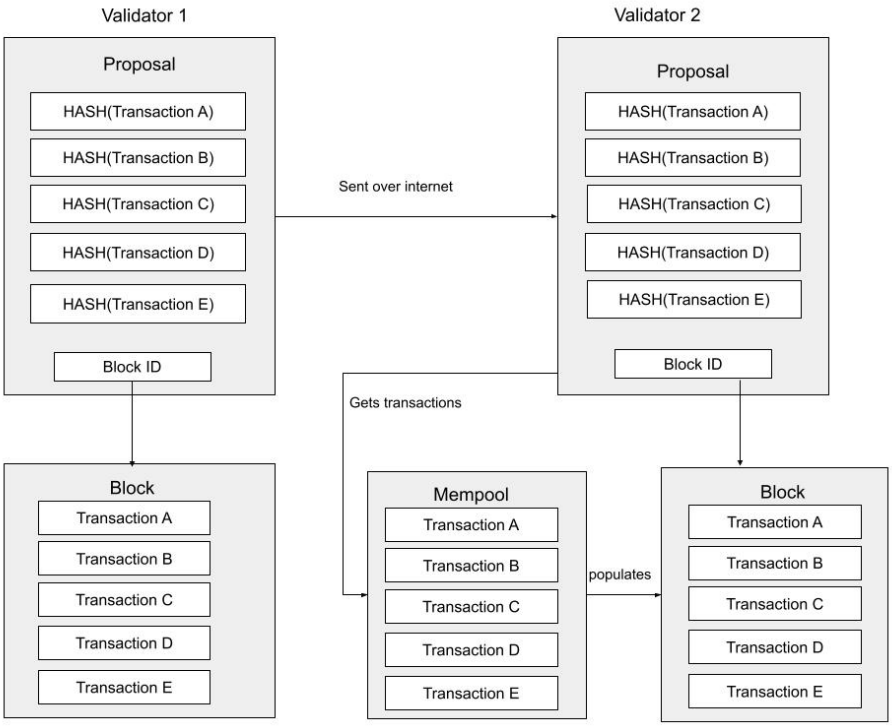


Fig.1. Block proposals with transaction identifiers

plications can thrive with infrastructure that has been tailor-built to meet the unique demands of the trading user experience. By addressing the limitations of previous layer 1 blockchains, Sgei's robust architecture offer s a foundation for applications to scale and reach mas s adoption.

3 Protocol Improvements

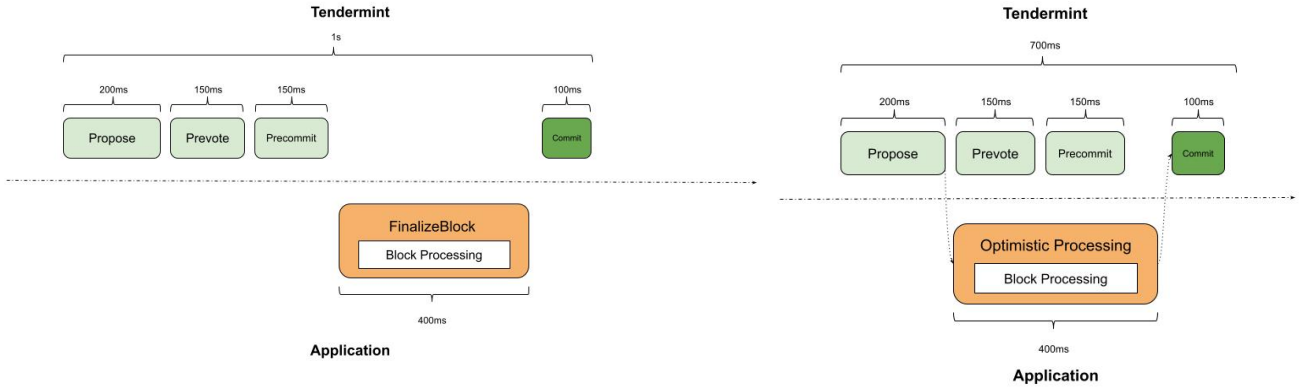
3.1 Twin-Turbo Consensus

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.



(a) (b)

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

Once validators have all of the transactions as part of 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 until 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

Sgeiuses Cosmos SDK as the base for the application logic. As part of this logic, 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 Sgeihas configured DeliverTx and EndBlock to parallelize transaction processing, as shown in figure 3.

Sgeifirst 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.).

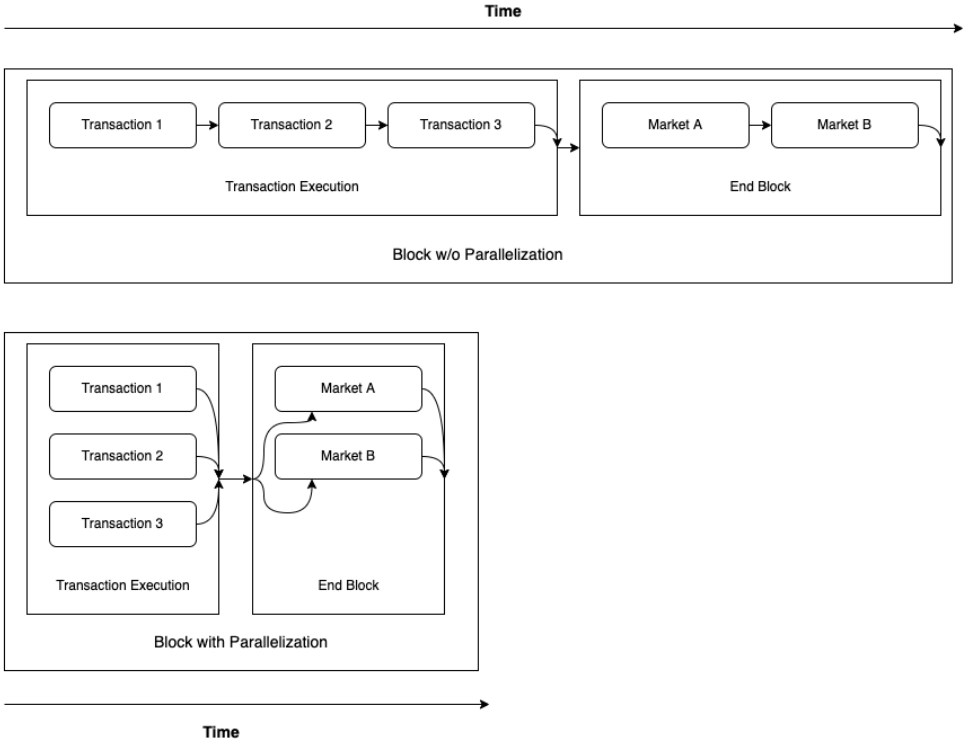


Fig.3. Block processing with and without parallelization

However, order matching engine related transactions only go through basic processing during the DeliverTx 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 4.1.1 for more information).

Sgeihas 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, Sgeiprocesses transactions in parallel (see figure 4). This allows multiple transactions to be processed simultaneously, which leads to improved performance. Data for Sgeiis persisted in a key-value store. To prevent race conditions and nondeterminism, Sgeineeds to ensure that multiple parallel processes ar e 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 needed to give further parallelism. For example, trans-

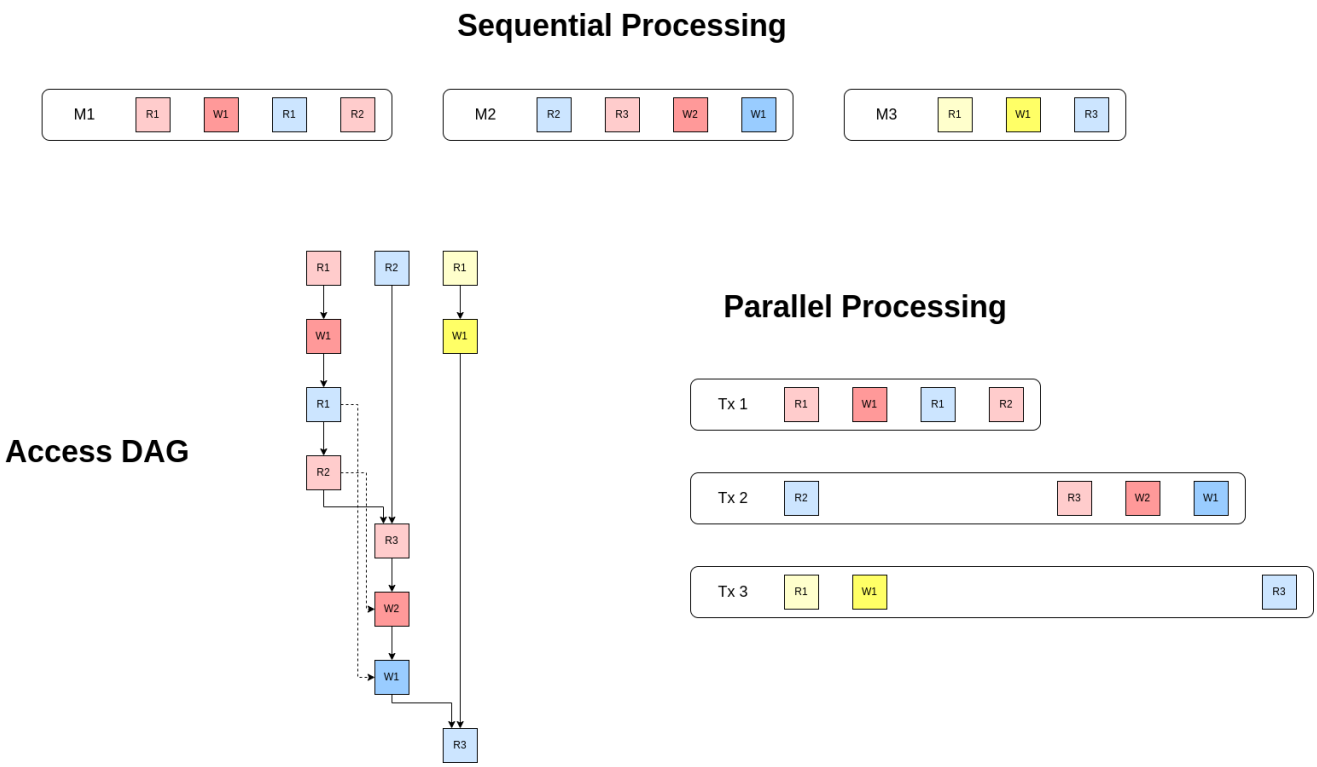


Fig.4. Access DAG for parallel processing

fers of tokens from account A to B and account C to D can be run in parallel since they update different keys. 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 edge 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 Sgei, smart contracts will need to define thei r own resource dependencies. These will be set at co n-tract initialization and can be updated by the sma rt contract admin through update transactions. If the d e-pendencies are properly written, then smart contrac ts will benefit from parallelism and pay cheaper gas fe es.If no dependencies are defined, then smart contra cts will run sequentially and block other transactions fr om running. Since they are blocking the rest of the net-work, transactions to those smart contracts will nee d to pay greater gas fees. If the dependencies for a spe cific smart contract are incorrectly defined, then mess ages for that particular smart contract will fail and gr eater 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 matching engine related orders will be processed by the native matching engine. Rather than processing orders sequentially, Sgei will process independent matching engine related orders in parallel at the end of the block. Two orders are independent if they do not affect the same market in t he same block. By default, the chain will assume all or ders touching different markets are independent, un-les s developers explicitly define dependencies between diffe rent markets. These dependencies will be defined when a smart contract is deployed. If these dependen-cies ar e defined incorrectly, then transactions to the depende nt smart contract will fail.

3.3 Native Price Oracles

Sgeihas 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 g e i
ther 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

Sgeioffers the functionality of a general purpose blockchain (i.e. allowing users to transfer assets and deploy smart contracts). In addition to that, Sgeihas 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 Sgei.

4.1 Deploy decentralized exchanges with Sgei

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

4.1.1 Lifecycle Of An Order

All matching engine related transactions will be executed atomically in the scope of a block.

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 action 1).

While processing each block, the dex module has an EndBlocker hook that processes orders recorded in the MemState in bulk (Figure 5 action 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 associ-

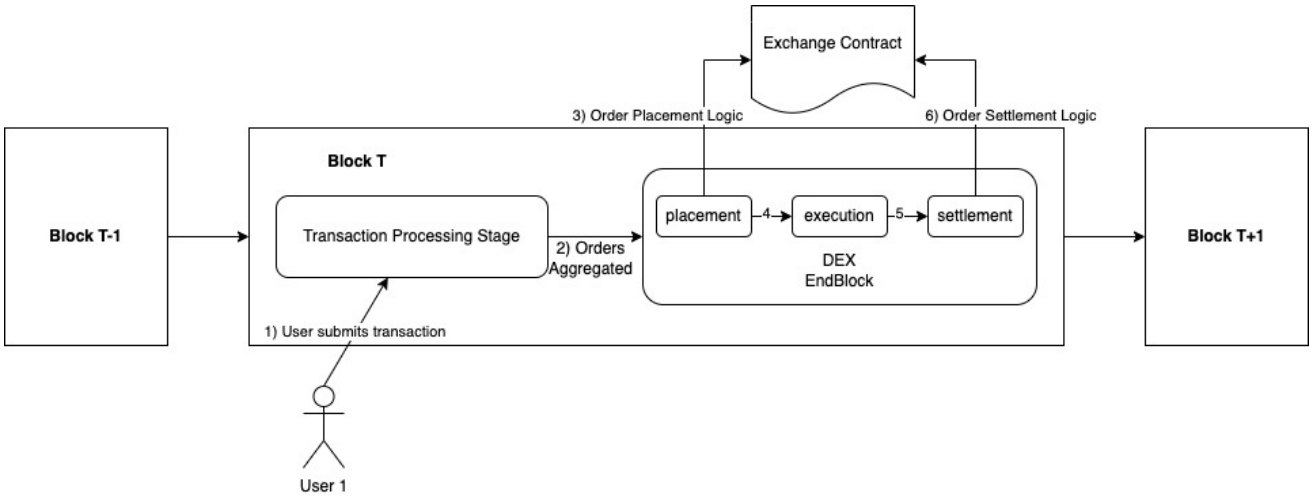


Fig.5. Lifecycle of a transaction

ated with that market (i.e. calling a perpetual exchange smart contract), which has all of the logic defined for how to interact with the matching engine (Figure 5 action 3). The smart contract will implement its own custom logic, and then call the matching engine (Figure 5 action 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 action 6).

4.1.2 Hook Support

Sgeiallows 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 set-tlement, allowing contracts to perform any post-trade logic if needed.

4.2 Asset Agnostic Matching Engine

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, decentral-

ized exchanges can track positions as a list in their smart contract state.

4.3 Frequent Batch Auctioning

Executing orders in a sequential manner encourages validators to arrange 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) Sgei‘‘s matching e n-gine 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

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

4.4.1 Client Order Bundling

Sgeitransactions 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 process-ing, Sgeiwill 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 matching engine related transaction will require instantiating the virtual machine (VM). Rather than having multiple VM instantiations, Sgeibundles 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 Sgei
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.