The Backbone of Modern Finance: Design and Implementation of a Simplified Financial Exchange System Prototype

Peter Torpis

June 2025

1 Abstract

In this paper, I explore the technical background of digital exchanges, then I show how I built a prototype (called MiniExchange) in Python. Emphasis was put on correctness of behavior, alongside mindful considerations of system performance.

In real exchanges, these systems are subdivided into smaller services, and the functionality is kept specific for each service. This modularity was complemented by the use of well-defined protocols and communication standards, allowing future feature integration with minimal friction.

While not intended to be a comprehensive guide in designing real-world, production ready, high-performance digital exchanges, this paper is an exploration into how such a system could be built from the ground up.

This paper represents the first phase of a broader effort to dive into systems design, performance, and application of exchange systems, with future work planned in C++ to re-implement a similar design, focused more on latency and performance.

Contents

Listings

List of Tables

List of Figures

2 Introduction

When a user clicks ”buy” or ”sell” on an online trading platform, a complex system is activated behind the scenes that may seem simple on the surface, but behind that screen sits decades of engineering, high-throughput, low-latency distributed systems, making sure that things are kept in sync, everyone is getting treated as fairly as possible and clients’ orders are getting processed as fast as possible.

Speed is not just for speed’s sake: at a busy time of day, large exchanges could face millions of requests a second. To deal with that kind of load, time-in-flight - the duration between order submission and processing - is kept as low as possible. There are many techniques developed to achieve this kind of performance, some of them I’m going to touch on in later sections.

2.1 Definitions, and Purpose of a Limit Order Book

Limit order books are the central data structure that hold information about the current state of the market. As described by Gould et al., 2013, the LOB (Limit Order Book) ”[...] contains, at any given point in time, on a given market, the list of all the transactions that one could possibly perform on this market.” It is divided into two parts, the bids (orders to buy the asset) and asks (orders to sell the asset). The bid and ask price is the best offer from either side, the mid price is the average between the two. The bid-ask spread is the difference between the ask price and the bid price.

Table 1: Simplified Limit Order Book Snapshot

Price	**Ask Quantity**	**Bid Quantity**
101.5	200	
101.0	300	
100.5	100	
100.0	150	
99.5		120
99.0		250
98.5		180
98.0		300

Question for the reader: Based on the snapshot in Table 1, what are the bid and ask prices of this imaginary market?[1]

2.2 Quote Driven vs. Order Driven Markets

There is a fundamental difference between quote driven markets and order driven markets. In a quote driven market there is a central entity called the market maker, whose job is to ”quote” or offer a price for buying and selling an asset, and provide the opposite side liquidity to match incoming orders. Both the bid and ask price is set by the market maker, the bid ask spread is partly there to cover their operational and inventory costs. On the other hand, order driven markets rely on limit orders to supply liquidity, and the available prices are determined by the open limit orders Roşu, 2009.

2.3 Trader Behavior and Bid-Ask Dynamics

Market participants who wish to get their orders filled immediately, can use so called ”market orders”, which immediately execute trades matching with the best offers. Roşu, 2009 In contrast, traders placing limit orders prefer to get a better price over immediate execution. Limit and market orders can be thought of as providers and takers of liquidity, respectively.

When the order book is full and traders wish to still try to get a better price, they place new orders between the bid and ask price, bringing the 2 sides of the book closer, reducing the bid ask spread. Roşu, 2009

[1] **Answer:** Bid price (highest bid): 99.5, Ask price (lowest ask): 100.0

2.4 Benefits of Limit Order Books

Tripathi and Dixit, 2020 in their systematic review outline five major advantages to using limit order books:

Low trading costs: the competitive nature of order driven markets tends to shrink spreads, market makers do not have to face inventory costs and fees are usually lower.

Remote access: decentralization and remote access allows broader market participation through an API or FIX gateway.

High Transparency (pre- and post-trade): the state of the market is visible through the LOB, executed trades are published in a (anonymized) market feed.

Efficient price discovery: the LOB aggregates information quickly and in real time.

Less information asymmetry: while LOBs close the information gap between informed and uninformed traders, there are still strategies that can grant advantages to informed traders Bloomfield et al., 2015

2.5 Industry Communication Standards

The Financial Information eXchange (FIX) protocol is widely used as the medium for transmitting orders, market queries, and other types of trading-related messages between exchanges and market participants. Originally developed in the 1990s, FIX provides a standardized and extensible messaging format that facilitates low-latency and interoperable communication across a wide range of financial institutions. In modern deployments, particularly where performance is critical, FIX messages may also be encoded in binary (e.g., FIX Binary Encoding or Simple Binary Encoding) to reduce bandwidth usage and parsing overhead. Community, n.d.

Many large-scale venues - including NYSE, NASDAQ, and other institutional trading platforms - support FIX as a primary or fallback interface for order entry and market data access. While my prototype does not implement FIX messaging directly, its design is similar in spirit to FIX, in being a key-value pair protocol. For now, the system uses a simplified internal message format for handling orders, but the structure of the dispatch and session layers mirrors the kind of separation and extensibility that FIX-based systems often rely on.

Example: New Order Submission in FIX vs Prototype API

Listing 1: FIX Message Format

```
8=FIX.4.2|9=176|35=D|34=215|
49=CLIENT12|56=BROKER1|
52=20240617-12:30:00.123|
11=123456|21=1|40=2|54=1|38=100|
55=AAPL|44=150.25|59=0|10=072|
```

Listing 2: Custom API Request

```
{
    "type": "order",
    "payload": {
      "token": "abc12345",
      "side": "buy",
      "price": 100.0,
      "qty": 10,
      "order_type": "limit"
    }
}
```

This paper presents the design of MiniExchange.[2]

[2]Source code available at https://github.com/ptorpis/MiniExchange

3 System Design

The system design and matching engine implementation were influenced by educational materials, such as Jane Street trading firm's video ”How to Build an Exchange” Jane Street, 2017 and other articles such as by Collins, 2023.

Python 3.12 was used to develop the prototype. To keep external dependencies to a minimum, the only external library required to run the core functionality of the program is `sortedcontainers` (Jenks, n.d.), which provides data structures that serve critical roles in the matching engine in keeping orders sorted.

For testing other 3rd party libraries, such as `psutil` (Rodola, n.d.), `objgraph` (Gedminas, n.d.), and `py-spy` (Frederickson, n.d.) were used, but they are not part of the core modules.

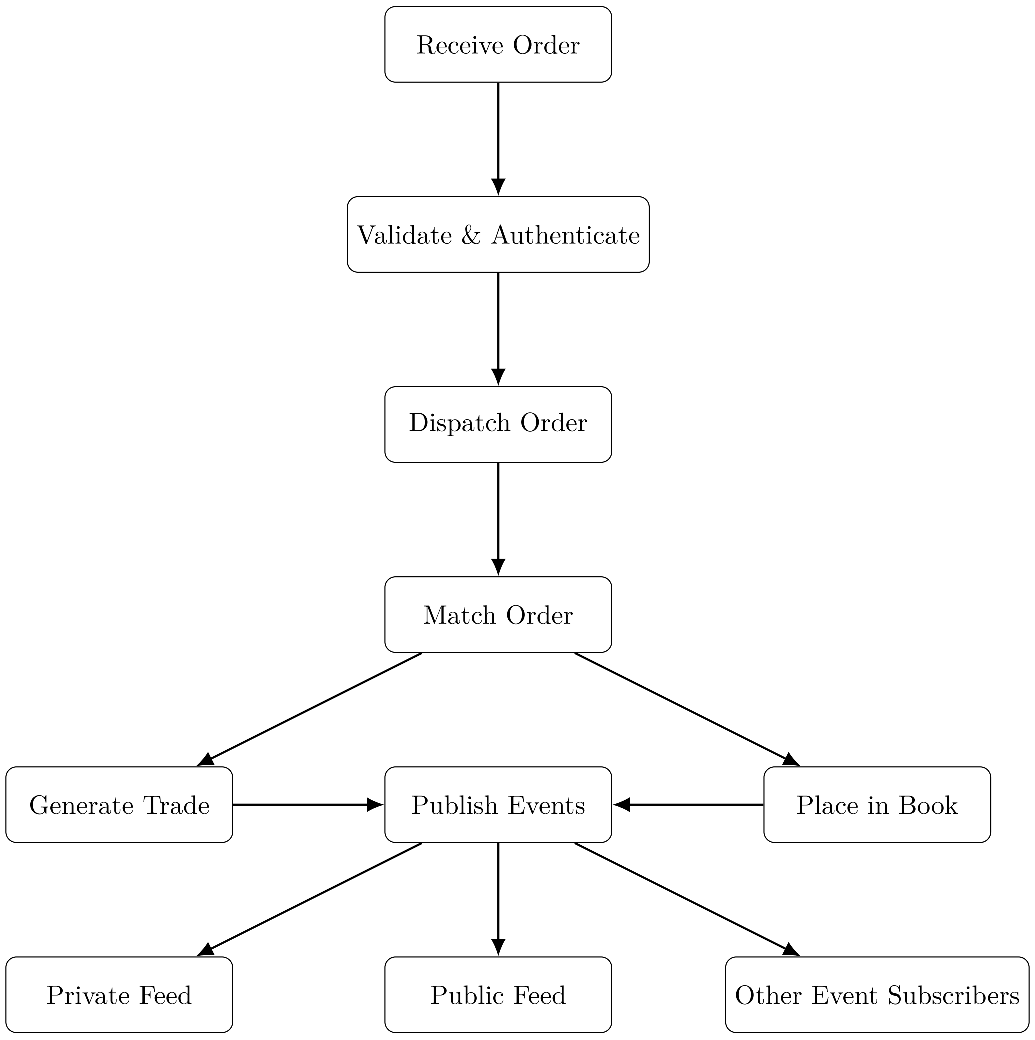

Figure 1: Logical Flow of an Incoming Order

3.1 System Overview

The order book policy is FIFO (First In - First Out). In FIFO, the orders are matched based on price level, and in each price level a time priority queue is established, where the orders are matched in exact order that they were placed. There are other matching policies out there, these are beyond the scope of this prototype.

The main entry point is the API, it accepts JSON-like requests (full specification can be read on the project's GitHub repository site). The API validates the requests, making sure that the fields are not missing and are not malformed.

Then the order gets passed down to the dispatcher layer, which does some further checking, this time to make sure that the orders are valid, then it routes the order to the correct spot into the matching engine.

The matching engine (order book) receives the orders and tries to find the best match for it. It matches orders, places them in the order book, handles cancellations, keeps price-time priority.

Whenever an event occurs that we would like to record, the order book creates an event object and passes it to the event bus, which then connects the publisher (the order book) to the subscribers (market data feeds, loggers), keeping the order book internal logic simple, but still being able to follow events.

The events are put in a queue when created; the event bus creates a (or multiple) separate threads and the handlers (the subscribers) deal with elements from this queue one by one. For example an events subscriber might be a data feed, which then prints, logs, or otherwise records the event and then moves onto the next.

3.2 The Matching Engine

3.2.1 Role and Responsibilities

The order book is the state of the market, and the matching engine is the algorithm that performs operations on the order book.

The matching engine, is responsible for the complete life-cycle of orders in the system. This includes handling new order submissions, managing partial and full fills, tracking open orders, and processing cancellations. It also ensures the internal state of the order book remains consistent and free from orphaned or stale orders, which is critical in preventing inconsistencies during matching or reporting.

Furthermore, it enforces the matching policy - in this case, price-time priority - ensuring deterministic and fair trade execution. This means that for a given set of input orders, the same output (trades and order states) is guaranteed, which is essential for reproducibility, auditability, and backtesting.

3.2.2 Data Structures for Efficiency

Efficiency is crucial for all components of the system, but that is especially true for the matching engine, because it must be able to handle high-frequency order flow under tight time constraints. This system uses `SortedDicts` from `sortedcontainers`. Within each price level, orders are stored in a queue to maintain time-priority and enable constant-time access to the next eligible order.

3.2.3 Order Types and Matching Logic

`Dataclass`-es are used to create `LimitOrder`s and `MarketOrder`s, containing order side (sometimes called ”sign”) for ”buy” and ”sell”, quantity, a token to verify user identity, an order ID, a timestamp of when the order was placed and a status field.

The status field in this system can be:

- `new`
- `filled`
- `partially_filled`
- `cancelled`

The only difference between the 2 classes is that the `LimitOrder`s contain a price field, while the `MarketOrder`s do not, for this reason they both inherit from a base class called `Order`.

When an order comes in, it is matched with a resting order, first based on the best available price, then at the price level, the first order in the time priority queue. There can be any combination of partial and full order fills, based on the current scenario, and the incoming order is matched until either the order is fully filled, or the order book has no more available orders (for that price limit set by the order). As per the policy of this exchange, if a market order is placed and the book is empty, the order is cancelled.

3.2.4 Order Cancellation

Each order is assigned a unique identifier upon creation and stored in an internal mapping, enabling efficient cancellation. This allows traders to modify their intentions as market conditions evolve. When a cancel request is received, the engine verifies the order's existence and state, and then removes it from the order book. Orders that are already filled or cancelled are ignored to maintain integrity.

This design choice ensures the order book remains loosely coupled with downstream consumers, enhancing modularity and testability.

3.2.5 Event Publishing

This prototype uses a pub-sub (publisher-subscriber) model to deal with events. These events are: an order getting added to the book, an order filled (fully or partially), an order cancelled, a trade executed.

Since the order book does not have to be concerned with what happens to the event, it just needs to let the rest of the system know that something has just occurred, a pub-sub model is perfect for this case.

The order book is the publisher, it creates different types of events based on what had just happened and sends it off to the event bus (more on the event bus later). This allows multiple parts of the system to listen to these events without them having to be tightly coupled with the order book.

3.3 API Design

The `MiniExchangeAPI` is a stateless, JSON-based interface designed to emulate the core functionality of a financial exchange. It supports operations like authentication, order entry and cancellation and market data queries. It was designed to be simple and to accept human-readable input to make testing and interacting with it easy.

3.3.1 Request and Response format

Each API request follows the structure:

Listing 3: General request format

```
{
  "type": "request_type",
  "payload": {
    "key1": "value1"
  }
}
```

Responses include a `success` field and either a `result` or `error` field:

Listing 4: General response format

```
{
  "success": true,
  "result": { ... }
}
```

```
{
  "success": false,
  "error": "Description of the error"
}
```

3.4 Command Line Interface

For testing, debugging, and playing around, a command line interface (CLI) is provided, with the same capabilities as the API, just with some quality of life changes, like the use of the authenticated user's username instead of having to type out the token. Here is an example of using the interface:

Listing 5: Example CLI interaction with MiniExchange

```
MiniExchange > order alice sell 100.0 limit 101

Order:
    ID: eadba0aa-e071-4693-82dc-796b99b52eb9
    Side: sell
    Status: NEW
    Original Quantity: 100.0
    Remaining Quantity: 100.0
    Filled Quantity: 0

  No Trades.

MiniExchange > order bob buy 100.0 limit 99

Order:
    ID: f78911b1-57d7-4b72-98f3-23bcc27fdbba
    Side: buy
    Status: NEW
    Original Quantity: 100.0
    Remaining Quantity: 100.0
    Filled Quantity: 0

  No Trades.

MiniExchange > book

Order Book:
Bids:
  99.00: [id: f789, qty: 100.0, status: new]
Asks:
  101.00: [id: eadb, qty: 100.0, status: new]

MiniExchange >
```

3.5 Authentication and Session Management

Like in any system where clients interact with a service, this system has a way to mimic session management and authentication capabilities. Authentication and session management are currently implemented in a simplified, non-production grade manner. The operations supported are logging in - to establish a connection between the client and the exchange - and logging off. After logging in, the client receives a UUID prefix that they have to include in their order management requests to identify themselves.

User credentials are stored in an in-memory dictionary, as text, and the usernames and passwords are hard-coded by hand. All of these practices are things that should never happen in real deployment. In a real system, there are a multitude of more sophisticated authentication methods and practices that could and should be used, but this part is not a major concern for this prototype at the current state.

3.6 Events and Market Data Feeds

The events and data feeds are managed by an event bus, which uses multithreading and thread safe queues (`queue.Queue`) for asynchronous event processing. The publisher (the order book) submits events to the bus and through the bus, the subscribers to that event can handle it. The feeds employ a shared `print_lock` to prevent output interleaving when printing from multiple threads.

There are 2 different market data feeds, a private and a public one, with the public one obfuscating client identities, much like in real systems. The private feed can be used internally for testing and auditing, while the public feed can be broadcast to the users, keeping all participants' identities hidden.

There is a test mode, which disables the event queue, which was used when measuring the raw throughput of the order book, because as it turns out, since there could be multiple events that are created from a single order, the event handlers could not keep up, and the events just pile up, leading to memory usage by the application to slowly go up. This starts being an issue when there are millions of orders being processed at the maximum speed, more on this later in the performance and testing section.

For example an order comes in that fills a resting order and partially fills another, that is 5 events, just from one request (Order filled, Order filled, Order Partially Filled, Trade, Trade). This is part of the reason why the feeds cannot keep up with the order book when processing orders at the maximum speed, and this is a problem that I will try to solve in future versions.

4 Performance and Testing

4.1 Unit and Fuzz Testing

To ensure correct behavior of the order book and the system as a whole. The core matching engine an the API is covered by a suite of 30 unit test cases, to verify that edge cases are handled correctly. Things tested are for example to make sure that partial fills fill at the correct price, at the correct quantity and that large market orders can fill on multiple price levels. If the reader is interested to see the test cases covered, they are included in the appendix.

Another part of testing was to see how well does the system hold up under heavy load. For this a fuzz data generator was made, along a CSV writer and decoder that can make and decode an arbitrarily large CSV file and process requests from it. This can be used to act as a market replay tool, because it always yields the same results given the same input, or a performance benchmark.

I've tested different file sizes, from 10 thousand lines all the way up to 10 million lines to test whether the theoretical $O(1)$ order lookup is there even after a large number of orders have been processed and that the total time complexity given an input of length n is $O(n)$.

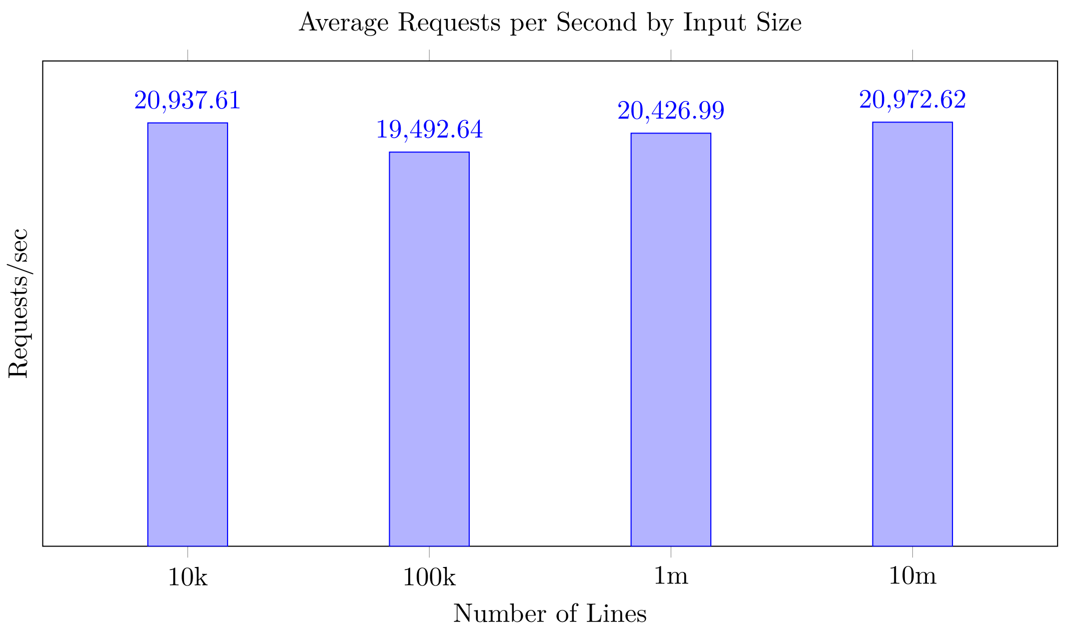

Figure 2: Benchmark Results: Requests/sec over Input Size

As seen in the chart, requests processed per second stays stable when scaling up the number of orders.

Note that during the benchmark, of which the results are shown in the bar chart, there was a print statement in the loop that updated the counter on the screen to let me know how far along the test was, removing the print statement made the request per second figure to go up to around 30 thousand a second, which goes to show that printing to the screen at this rate becomes an expensive operation.

4.2 Event Handling Performance Issue

In heavy testing, I found that the event processing was slower than the order book itself, this lead to a pile up of event objects, causing memory usage to linearly increase as more requests were processed. When turning off events, the memory usage issue went away, indicating that the issue was because of the events.

I couldn't find a good solution for this, probably a separate process would need to be created to have enough processing bandwidth to take care of all the events.

5 Discussion

The results from the benchmarks show that the design is scalable, as there is no significant performance drop-off beyond a certain number of requests. This highlights the use of efficient data structures when building an order book, since lookup operations can become expensive quickly, and if there is not a quick way to get to the desired price level and in there, the desired order, the order traversal become the bottleneck.

This prototype can handle up to almost 30,000 requests per second when unnecessary I/O is removed (such as logging or print statements). On better hardware or with further optimization, this number could be higher. Even at lower rates (around 20–22k during instrumented tests), the system exceeds typical REST API performance. While this system operates on standard hardware, in real financial systems, ultra low latency network cards and multicasts are used to distribute information, that can handle very large number of requests and have latency on the order of microseconds or even less.

In this system, the limiting factor ended up being the event handling, indicating that a separate process would have to be created to process, broadcast, or record these events.

A limitation to the testing is that it is synthetic data that was generated through a random generator. Multiple ratios of limit to market orders were tested to see how much the amount of resting orders in the system would affect performance, and it could still handle orders at the similar or the same rates.

The testing was not done on realistic data, however, where there are multiple market players, each trying to execute a strategy they deem reasonable.

There are some features missing, like slippage, prevention of self trading, order modification and order replacement. Adding these would greatly improve realism, and I may consider the implementations of these features in future versions.

6 Summary and Conclusion

This project set out to design and implement a performant and scalable order book matching engine, with a focus on correctness, speed and modularity. The goal was to simulate the components of a real trading system in Python.

Main achievements

- A working matching engine with support for maker and limit orders.
- Fast order matching using efficient data structures.
- Throughput of almost 30 thousand requests per second under synthetic load.
- Robust unit testing and fuzz testing frameworks.
- Interactive command line interface.

Limitations

- Synthetic testing only.
- Event processing became a bottleneck under heavy load.
- Missing key features like modifying, replacing orders, self-trade prevention.
- No slippage in transactions.

Future work will focus on implementing more realistic trading features, implementing in other languages for higher performance, integrating real-world market data, and addressing performance bottlenecks in the event system through multiprocessing or dedicated services.

A Appendix

Table 2: CPU Time Breakdown from `py-spy top`

%Own	%Total	Own Time	Total Time	Function
8.00	21.00	5.37s	10.08s	`_match_market_order (order_book.py)`
12.00	22.00	3.33s	7.94s	`_match_limit_order (order_book.py)`
8.00	17.00	3.23s	6.31s	`uuid4 (uuid.py)`
9.00	9.00	3.08s	3.08s	`__init__ (uuid.py)`
4.00	5.00	2.44s	2.66s	`__next__ (csv.py)`
7.00	7.00	2.13s	2.13s	`__str__ (uuid.py)`
5.00	5.00	1.61s	2.08s	`_add_order (order_book.py)`
4.00	28.00	1.48s	7.48s	`create (order.py)`
2.00	79.00	1.21s	28.65s	`dispatch (dispatcher.py)`
3.00	85.00	1.08s	30.84s	`handle_request (api.py)`
2.00	9.00	1.02s	5.11s	`create (trade.py)`
8.00	8.00	1.00s	1.22s	`decode_csv_row (csv_testing.py)`
2.00	2.00	0.67s	0.67s	`validate_order (dispatcher.py)`
3.00	3.00	0.62s	0.68s	`__post_init__ (order.py)`
0.00	0.00	0.57s	0.57s	`return_format_order (dispatcher.py)`
1.00	1.00	0.54s	0.54s	`validate_order (validators.py)`
4.00	5.00	0.54s	0.97s	`peekitem (sorteddict.py)`
1.00	14.00	0.53s	4.42s	`stream_csv_requests (csv_testing.py)`
1.00	100.00	0.52s	35.76s	`main (csv_testing.py)`
2.00	5.00	0.46s	1.14s	`__init__ (<string>)`
1.00	1.00	0.43s	0.43s	`__getitem__ (sortedlist.py)`
1.00	80.00	0.41s	29.06s	`_submit_order (api.py)`
1.00	1.00	0.36s	0.50s	`__get__ (enum.py)`
2.00	2.00	0.30s	0.30s	`return_format_trades (dispatcher.py)`
2.00	2.00	0.29s	0.51s	`__call__ (enum.py)`
1.00	45.00	0.25s	18.42s	`match (order_book.py)`
0.00	2.00	0.23s	0.33s	`remove (sortedlist.py)`
0.00	0.00	0.23s	0.23s	`__new__ (enum.py)`
0.00	0.00	0.22s	0.22s	`get_token (session_manager.py)`
1.00	1.00	0.21s	0.21s	`fieldnames (csv.py)`
0.00	0.00	0.20s	0.43s	`__setitem__ (sorteddict.py)`
0.00	2.00	0.18s	0.51s	`__delitem__ (sorteddict.py)`
0.00	0.00	0.17s	0.23s	`add (sortedlist.py)`
0.00	1.00	0.15s	0.16s	`_get_user (api.py)`
1.00	1.00	0.15s	0.15s	`__instancecheck__ (abc)`
0.00	0.00	0.14s	0.14s	`value (enum.py)`

Output of `py-spy` by (Frederickson, n.d.) when running fuzz tests with 1 million lines.

References

Bloomfield, R., O'Hara, M., & Saar, G. (2015). Hidden liquidity: Some new light on dark trading. *The Journal of Finance*, *70*(5), 2227–2274.

Collins, R. (2023). Stock exchange systems design case study [Accessed: 2025-06-14].

Community, F. T. (n.d.). Fix protocol online specification [Accessed: 2025-06-14].

Frederickson, B. (n.d.). Py-spy: Sampling profiler for python programs [Accessed June 2025]. https://github.com/benfred/py-spy

Gedminas, M. (n.d.). Objgraph - draw python object reference graphs [Accessed: 2025-06-17].

Gould, M. D., Porter, M. A., Williams, S., McDonald, M., Fenn, D. J., & Howison, S. D. (2013). Limit order books. *Quantitative Finance*, *13*(11), 1709–1742.

Jane Street. (2017). How to build an exchange [Accessed: 2025-06-17].

Jenks, G. (n.d.). Sortedcontainers: Fast and pure-python implementation of sorted collections [Accessed June 2025]. https://github.com/grantjenks/python-sortedcontainers

Rodola, G. (n.d.). Psutil - process and system utilities [Accessed: 2025-06-17].

Roşu, I. (2009). A dynamic model of the limit order book. *The Review of Financial Studies*, *22*(11), 4601–4641.

Tripathi, A., & Dixit, A. (2020). Limit order books: A systematic review of literature. *Qualitative Research in Financial Markets*, *12*(4), 505–541.