Efficient Market Data Collection for algorithm trading strategy

Student: Aliaksandr Piatrouski

Course: CM3070 - Computer Science Final Project

Github link: https://github.com/Aallzz/cm3070/tree/main

Video recording link:
https://www.youtube.com/watch?v=T5EnQFYpX9I

Introduction

Project is mostly following template of **“CM3010 Databases and Advanced Data Techniques**” course.

This summer I got into algorithmic trading on crypto exchanges, that is a brand new space to me! There are many interesting strategies that operate with order book states and where having access to extensive market data super fast is crucial. Unfortunately almost all strategies are limited to data from a single exchange, which felt like an unnecessary constraint. I began to ask myself, why not leverage data from multiple exchanges to develop this new type of strategy?

This question led me to create a project designed to collect and conveniently use vast amounts of market data across various exchanges and instruments. Initially, this was simply just as simple —gathering data for backtesting. But with time the project quickly evolved into a complex and interesting challenge on its own because of the limitations that I had in terms of money and laptop.

Market data collection requires significant storage and efficient data handling approaches. After researching several options on different aspects of the project like storage, fetching approaches, useful logging etc I had to find a compromise solution that I've described in this project report!

Literature And Professionals Review

Market analysis

Though the project is originally for my own usage I've made a market research to understand where it can go in the future! There are two aspects where it can be used

- Part of algo trading platform
- Data seller

Algorithmic trading strategies platforms

Most of the trading platforms like wundertrading, OKX and other provide you with an option to create bots that react to some predefined by UI signals. It lacks flexibility of incoded strategies that would allow users to do whatever they want.

After a lot of searching I've found quantconnect platform, that actually allows you to do coding strategies and even gives you an option to perform backtesting! Unfortunately I didn't manage to include any libraries to connect to two exchanges simultaneously, not for trading nor for backtesting. Which is the main use case for multi exchange strategies I'm working on.

Market Data Storage platforms

I haven't found many good platforms on google that provide historical market data, but there is one tardis.dev that actually allows you to get data from multiple exchanges, but for a very aggressive price (stating 100$ for one exchange and instrument and other plans). Based on what I'm collecting here with a proper time investment and further iterations this can grow into a 600$/month service.

Socialising the idea

The results of the market research were stunning to me - the trading needs tools that would simplify creating strategies and backtesting across exchanges! That's where I've started asking more questions to people in trading firms about their opinions on my idea for the project that will collect multi exchange data. And what a surprise it was that that's exactly what's happening in HFT companies! The usage of this data is a must for strategies like

- Cross chain arbitrage
- Future spread following
- Delayed update
- Fee based orders

This was just another confirmation that the track is right!

What is the market and me are after

- Cheap solution for collecting market data
- Solution working in a rigid constraints on local memory - 50Gb of free memory
- Easy to onboard new exchanges - good design
- Unnoticeable latency on the exchange data writing to storage - meaning it won't be reflected in the data

Techniques and method analysis

Based on what we are after there are two important bits that we need to analyse for the proper weighted technical decision that we will make in the Design part.

Storage analysis

Later we will need to figure out what type of storage we want to use. I've listed a few options in the table below highlighting them in traffic light that will be used later to make the decision.

	Google Cloud SQL	AWS S3	Server rent (digital ocean) with SQL	Local machine with SQL

Price (a month)	storage: $0.22 per GB write/read: spent 40$ loading 10GB somehow Source: https://cloud.google.com/sql/pricing	storage: $0.023 per GB write: $0.005 per 1k requests read: $0.0004 per 1k requests Source: https://aws.amazon.com/s3/pricing/?p=pm&c=s3&z=4	storage: $1 per GB + machine cost write: $0 read: $0 Source: https://www.digitalocean.com/pricing/managed-databases	storage: $0 write: $0 read: $0
Memory limit scalability - very important	Unlimited	Unlimited	Limited	Limited (50Gb)
Onboarding complexity	Fairly easy	Fairly easy	Takes time, faced permission problems	Easy
Have prior experience	Yes	No	Yes	Yes
Require custom data structure	No	Yes	No	No
Expansion to strategy running	No	No	Yes	No

It's easy to see that having SQL server on a local machine would be ideal if there was no memory constraint for the storage and it's a huge factor that doesn't allow us to use it. Google cloud SQL is quite expensive and I've lost 40$ just playing with it.

The ideal option for this project appeared to be AWS S3 for data storage + Server rent closer to exchanges to reduce the latency for the data collection.

Message queue analysis

I've selected a few options on the internet to choose the one that will store data from the exchanges before it's processed by the program and stored somewhere.

Mostly the data was taken from Kafka vc RabbitMq and ActiveMq vs Kafka. In the future we will see that the queue will handle quite a load based on the number of exchanges, so we should

care about throughput and keep the latency meaningful, though we don't care about it too much after the message is sent to the queue.

	Apache Kafka	RabbitMQ	ActiveMQ
Throughput	Kafka can reliably handle up to millions of messages per second. Source: https://quix.io/blog/apache-kafka-vs-rabbitmq-comparison	RabbitMQ is optimized to handle lower throughputs (thousands or tens of thousands of messages per second). Source: https://quix.io/blog/apache-kafka-vs-rabbitmq-comparison	Respectable latency, not as good as Kafka Source: https://quix.io/blog/activemq-vs-kafka-comparison
Latency	Very low latency (in the millisecond range). Source: https://quix.io/blog/apache-kafka-vs-rabbitmq-comparison	Very low latency (in the millisecond range). Latency increases when high throughput workloads are involved. Source: https://quix.io/blog/apache-kafka-vs-rabbitmq-comparison	Good throughput and low latency (with medium workloads). Source: https://quix.io/blog/activemq-vs-kafka-comparison
Tyring using at python lib	Moderate	Easy	Couldn't make work

The main factor for me appeared to be usability. I could easily make RabbitMQ make work for me, while having some problems with others, even though Rabbit MQ has a lower throughput we are good with thousands messages a second, usually we will get 5-60 messages per instrument a second.

Design

Product

We are delivering a tool in the form of a library, that will listen to the data exchange data efficiently, storing it in the cloud with minimal processing effect of the data.

Domain: Trading. The tool is very important for not just cross exchange trading strategies (as it was pointed out in the previous section), but for one-exchange strategies or in general for data analysis.

Users: Traders, quants (people writing algo strategies). They will need data to backtest the strategies especially if it's cross exchange. We can easily see it based on the data from the previous section (Specifically there is data demand and tools that almost support cross exchange trading, where this tool would shine).

Roadmap

WorkPlan Gantt Chart (key milestone)

Evaluation

Before going into the engineering heavy part, I want to discuss the product success evaluation. It's essentially duplicates of what the market (and me) is after

- We can work with a big amount of data, while locally having only 50Gb. To check this we just need to make sure that the program works.
- We have close to zero latency caused by our processing. To check this we will benchmark trivial and the final solutions.
- Easy to onboard new exchanges. To check this we will see how much time it takes to make a new exchange data collection work.
- Reliability. To check this I will use subjective analysis of how easy or not it is to maintain the system.

Architecture

Constraints - storage

Market data collection is not a memory cheap operation, on the contrary we need hundreds of GB of memory a month to store one month of market data from 3 (to show case) exchanges (file size can be from 300Mb to 3Gb based on the observation). That's why before going to much in the design, we must figure out what we will use for data related work and based on this create the architecture of the collector.

In the Literature Review section in the storage analysis I go with AWS S3 for storage, because we want to be able to save money at scale with a side effect that this will restrict us in how we should store the data and quickly access it and below we will explore it in the details. Also it's a benefit because I'll get my first experience with S3.

Constraints - collection speed

In the scope of this project, I'm not aiming to have the data first, on the contrary for the backtesting we are okay to collect data with a reasonable delay. But the latency of data collection is important for strategies. The speed can be improved if the collection will be using rented servers that are close to the exchange servers. For example, to collect data fast I've rented a server by www.2co.com near Heathrow Airport, very close to Derebit exchange. So we could rent a few servers to different exchanges.

Requirements - scalability on many exchanges and instruments

We want to easily onboard new exchanges and instruments to our data fetcher and this should be done in an efficient and type-safe way

High level Architecture

Now knowing the constraints of the system let's figure out the architecture for our data collection program. The natural focus of the architecture is on read from exchanges and writing to the storage so later it could be used for backtesting.

Writing

Writing market data to S3 storage should optimise the following criterias

1) We do a reasonable amounts of writes and reads to S3, because it's not for free
2) We do provide a file structure that allows us to quickly access the most important data for backtesting. We care about sequential data (by time) from some exchange to some instrument, so the "index" in this file system should be time+exchange+instrument related.
3) We should be cautious of the machine memory size restrictions

I've landed with the following flow design

Pic 1

The file structure comes as a compromise between all three criterias. One day of the data for one instrument is manageable on my machine (3GB file * 3 exchanges * 5 instrument-asset type = 45Gb). It's a bottleneck and if we want to scale within the scope of my only machine we need maybe <half day>.csv granularity.

Step 1 - file creation

Let's go over the file tree

- Server name - as it was pointed out in constraints data collection latency is important for the strategy and we should tell where the data is coming from to strategy could be used on the best server
- Year-month + exchange name - this is a compromise between time and exchange "index"
- Asset type - it's future or options or spot
- instrument - self explanatory
- day.csv - file with Tick data as CSV

Step 2 - file writing

When writing strategies a big part is orderbook, it's very important that we could easily reuse tick data to reconstruct and update orderbook.

The Ttck data has dependency on the exchange being able to provide this data, if the data is not provided it must be transformed to follow the agreed format.

Tick Data type must contain:

- Server Timestamp — long int, unix timestamp since epoch UTC, when the tick was received by the server
- Exchange timestamp(Optional) - long in, unix timestamp since epoch UTC, when the tick was generated by the exchange
- Tick Type — (0 — Bid, 1 — Ask, 2 — Trade) type of the tick
- Update Type — (0 — New, 1 — Update, 2 — Delete, 3 - Sell, 4 - Buy) what to do with the update (update existing level, delete level or that's a snapshot) NEW ⇒ means that's snapshot. Update/Delete ⇒ means incremental order book update
    - If current row = 0 (i.e. NEW), then clear current status of order book and read rows with update type = NEW until update type ≠ NEW
    - if current row = 1/2 (UPDATE / DELETE) then either do orderbook[price][tick type] = size or del orderbook[price][tick type] (unless obv. tick type = trade, as it's only for bid and ask)
- Price — float, level
- Size — float, size on that level

Step 3 - file S3 dumping

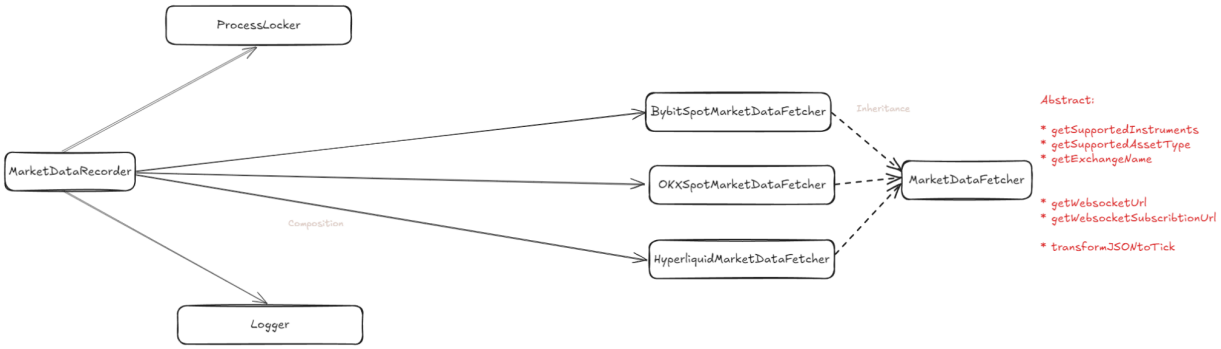
Once the day has finished, we stop writing to the file and send it to S3, deleting it locally to free up space for the new files and remove duplication while dumping to S3. And from here we start all over again.

Reading

Data fetching from the exchanges might be more straightforward than writing, because we don't have many constraints by our own systems, only the exchange ones. A good thing that most crypto exchanges provide websocket AP, that provide close to real time data.

More detailed Architecture

Going more into the details we can represent the recorder - fetcher dependency as below.



Pic 2

Here Market Data Recorder Process is responsible for one and only one fetcher

1) Prepare everything for fetchers to be successfully called

   It means that it's responsible for the creation of the file structure described in the **Writing** section and it maintains the consistency of that file by preventing many fetchers writing to the same file. It can be also considered as a mutex.

2) Efficiently initiate fetching the exchange data to the files

   Fetchers are working all the time listening to the sockets and we need to make sure that they don't interfere with one another and fight over the resources. Therefore the recorder is created as a separate process per fetcher, where fetcher is responsible for getting data from one exchange about one asset type about one instrument.

   The reason why it's done this way is because we want to have concurrency to fetch many things at the same time, but because of python GIL we cannot leverage multithreading and use a multiprocessing approach instead.

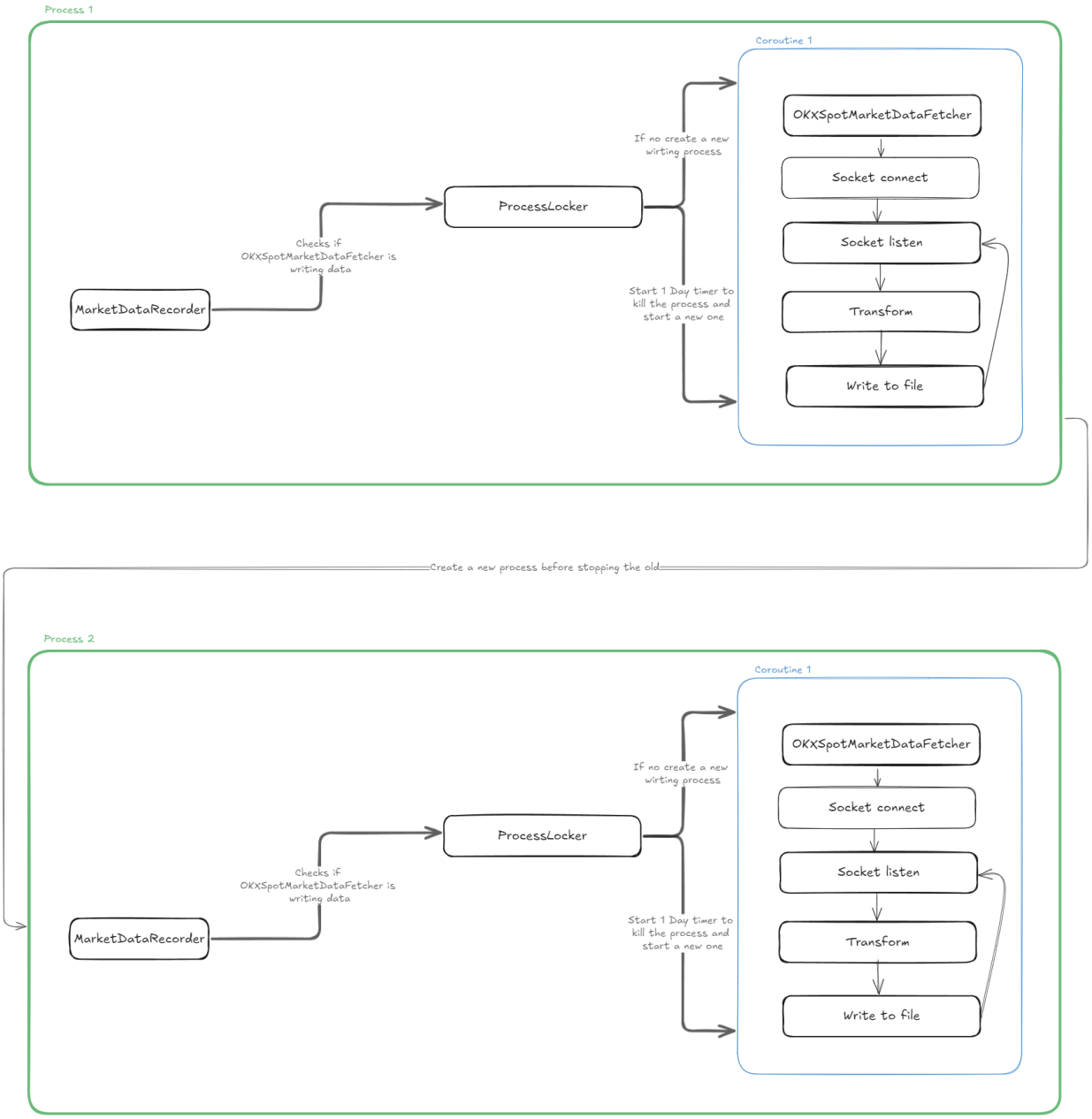
3) Track when it's time to stop writing to the current day file

   When the recorder has started the fetchers it will be responsible for stopping them. Good for us that python asyncio provides the functionality out of the box.

4) Start writing the next day files

   This is achieved by starting a new process for another day using the recorder and killing the existing process. (see a diagram below where Process 2 goes after process 1).

Below you can see this flow schematized with very rough implementation details of the fetcher..



Pic 3

Now that we saw the MarketDataRecorder and understood what it does, let’s understand the MarketDataFetcher responsibilities.

1) Handling connection to the websocket API of the exchange

   It means that fetchers know how to connect and with what arguments to the specific exchange to start collecting data about the instruments

2) Handling datastream from the websocket

   The fetcher is constantly listening to the data and tells how to transform it to the Tick Data type and store it somewhere

3) Minimal latency caused by receiving part

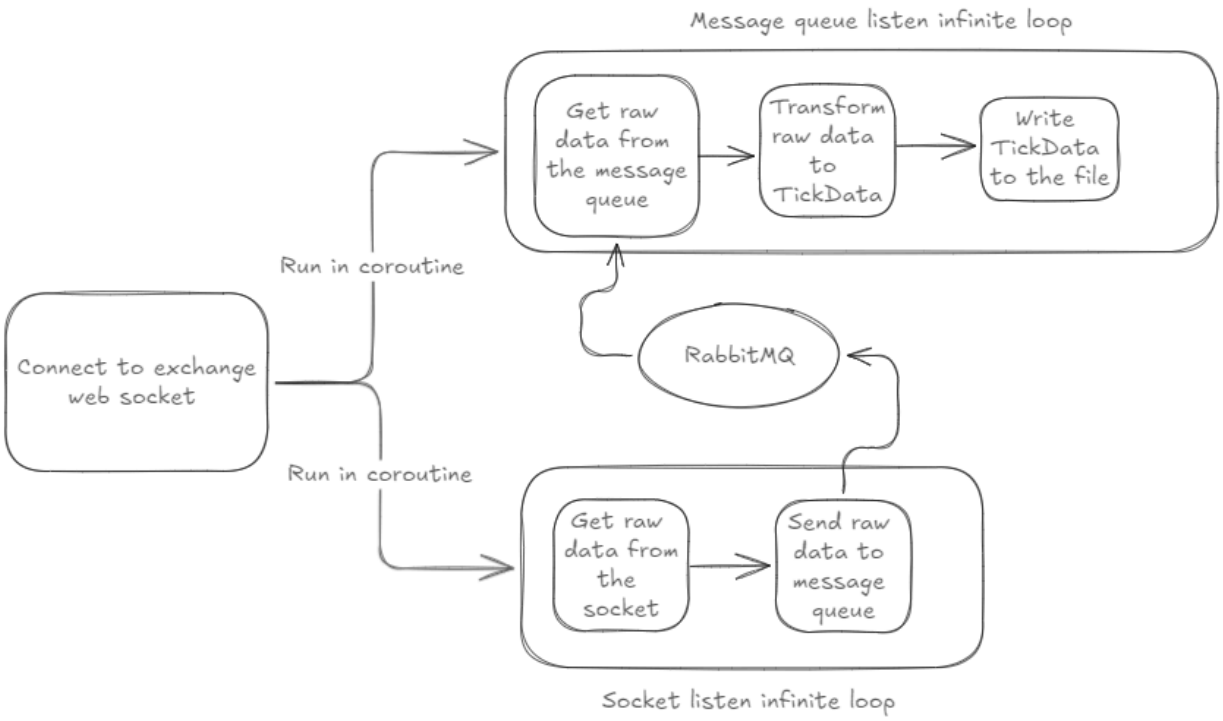
   We want to eliminate the time caused by the transformation of the data from raw data returned by the socket API to Tick Data type. This can be achieved by the introduction of the message queue. This is crucial, because strategies that we are backtesting shouldn't care about time we took to process and write the data to the file storage, in real life that would handle raw data right away.

   Note: In the first iteration I didn't use a message queue that added 30-50ms on top of the receiving time, which is quite a lot seeing that exchange time and receive time were just 0-5ms different.

4) It's very easy to expand to new instruments and new exchanges

   It's achieved by a proper OOP model with type safe abstract functions.

The above can be depicted with the schema below (except the OOP, which is depicted on Pic 2 on the right)



Pic 4

The final element is uploading this data to S3, that is done through scripting because there is no central process that would sync all the recorder processes and understand that it's time to move files.

Implementation

Now that we have a good high level understanding of the architecture we can go deeper into the implementation. Essentially we have three parts as it was described in the Design section: fetching data from exchanges, storing data locally, and storing data in the cloud. So I will cover implementation of those, but will start with the draft approach.

Draft approach

The task of listening to data itself is not hard. Let's put everything for one of the exchanges to just listen to the data and print it to the console.

```
import asyncio
import websockets
import json

msg = \
    {"jsonrpc": "2.0",
     "method": "public/subscribe",
     "id": 42,
     "params": {
        "channels": ["book.ETH-PERPETUAL.none.1.100ms"]}
    }

async def call_api(msg):
   async with websockets.connect('wss://test.deribit.com/ws/api/v2') as websocket:
      await websocket.send(msg)
      while websocket.open:
         response = await websocket.recv()
         print(response)

asyncio.get_event_loop().run_until_complete(call_api(json.dumps(msg)))
```

This solution is good enough already, if we want to stop on unstructured data from one exchange not stored anywhere. But we want more. Going through the exchanges APIs like hyperliqued, okx etc it was easy to see all of the support websocket API so it was the first area of improvement. How do we make it reliable? Then there are many exchanges, how do we onboard them? The data is different, how do we structure? A lot of questions to address already!

When I made the tick record type, this started taking time to transform the data and writing it to a file . I observed that this created a delay between receiving data from exchange to writing increased to 50ms! Meaning that we wait for longer to collect a new piece of data from the websocket! That's where the realisation came that I should use a message queue (async queue would work too, but I didn't want to lose the program async queue if we are losing the process, so used Rabbit).

……

Final solution

Fetching

From the draft approach we've learnt that all exchanges of interest support websocket api for the data sharing. It will play an important role in creating a base class for all the fetchers, because we can use the same mechanism hidden from the implementations. See the declaration of the MarketDataFetcher below:

```python
class MarketDataFetcher(ABC):
    """…

    def __init__(self, instrument: str):…

    async def run(self, process: Callable[[QuantumEdgeTickRecord], None]):…

    async def fetch(self):…

    async def process(self, process: Callable[[QuantumEdgeTickRecord], None]):…

    @abstractmethod
    def transform_json_to_tick_record(self, message: str, receive_timestamp: int) -> List[QuantumEdgeTickRecord]:…

    @abstractmethod
    def subscription_arguments_str(self) -> str:…

    @abstractmethod
    def websocket_url(self) -> str:…

    async def listen_and_process(self, process: Callable[[QuantumEdgeTickRecord], None]):…

    @staticmethod
    @abstractmethod
    def supported_instruments() -> List[str]:
        """…

    @staticmethod
    @abstractmethod
    def supported_instruments_for_test() -> List[str]:…

    @staticmethod
    @abstractmethod
    def supported_asset_type() -> AssetType:…

    @staticmethod
    @abstractmethod
    def exchange_name() -> ExchangeName:…
```

Before going deeper in how the fetcher works using the code, I want to highlight that this interface requires minimal effort from the engineer to onboard a new exchange - here are two example of the fetchers:

```python
"""
Data fetcher that connects to the OKX exchange and fetches spot market data
"""

from typing import List
import json

from python.types.enums.asset_type import AssetType
from python.types.enums.exchange_name import ExchangeName
from python.market_data.fetchers.fetcher import MarketDataFetcher
from python.types.tick_record import TickRecord
from python.market_data.fetchers.bybit.data_transformer import BybitOrderBookDataTransformer


class BybitSpotMarketDataFetcher(MarketDataFetcher):
    """
    Data fetcher that connects to the OKX exchange and fetches spot market data
    """

    def websocket_url(self) -> str:
        return "wss://stream.bybit.com/spot/public/v3"

    def subscription_arguments_str(self) -> str:
        return json.dumps({
            "op": "subscribe",
            "args": [
                f"orderbook.40.{self.instrument}"
            ],
        })

    def transform_json_to_tick_record(
        self,
        message: str,
        receive_timestamp: int
    ) -> List[TickRecord]:
        return BybitOrderBookDataTransformer.transform_json_message(
            message,
            receive_timestamp
        )

    @staticmethod
    def supported_instruments() -> List[str]:
        return ["BTCUSDT"]

    @staticmethod
    def supported_instruments_for_test() -> List[str]:
        return ["BTCUSDT"]

    @staticmethod
    def supported_asset_type() -> AssetType:
        return "SPOT"

    @staticmethod
    def exchange_name() -> ExchangeName:
        return ExchangeName.BYBIT
```

```python
"""
Data fetcher that connects to the OKX exchange and fetches spot market data
"""

from typing import List
import json

from python.types.enums.asset_type import AssetType
from python.market_data.fetchers.fetcher import MarketDataFetcher
from python.market_data.fetchers.okx.data_transformer import OKXOrderBookDataTransformer
from python.types.tick_record import TickRecord
from python.types.enums.exchange_name import ExchangeName


class OKXSpotMarketDataFetcher(MarketDataFetcher):
    """
    Data fetcher that connects to the OKX exchange and fetches spot market data
    """

    def websocket_url(self) -> str:
        return "wss://ws.okx.com:8443/ws/v5/public"

    def subscription_arguments_str(self) -> str:
        return json.dumps({
            "op": "subscribe",
            "args": [{"channel": "books", "instId": self.instrument}],
        })

    def transform_json_to_tick_record(
        self,
        message: str,
        receive_timestamp: int
    ) -> List[TickRecord]:
        return OKXOrderBookDataTransformer.transform_json_message(
            message,
            receive_timestamp
        )

    @staticmethod
    def supported_instruments() -> List[str]:
        return ["BTC-USDT", "ETH-USDT"]

    @staticmethod
    def supported_instruments_for_test() -> List[str]:
        return ["BTC-USDT", "ETH-USDT"]

    @staticmethod
    def supported_asset_type() -> AssetType:
        return "SPOT"

    @staticmethod
    def exchange_name() -> ExchangeName:
        return ExchangeName.OKX
```

It’s quite easy to see that the only meaningful part that requires some implementation in the transformation from raw data from websocket to TickRecord, which is a great achievement!

Now going back the the fetcher, setting all configurations aside there are three important methods

fetch	process

```python
async def fetch(self):
    """
    Listen to the market data and process it
    """
    connection = await aio_pika.connect_robust()
    async with connection:
        channel = await connection.channel()
        await channel.declare_queue(self.queue_name, durable=True)

        ws_factory = WebSocketFactory(self.websocket_url())
        ws = await ws_factory.connect()

        await ws.send(self.subscription_arguments_str())

        self.logger.log_info(
            "Start infinite loop to fetch data from websocket.")
        while True:
            ws_data = await ws.recv()
            self.logger.log_with_sampling(
                f"Received data from websocket: {ws_data}"
            )
            ws_data_with_timestamp = json.dumps({
                "data": ws_data,
                "receive_timestamp": round(time.time() * 1000)
            })
            message = aio_pika.Message(
                body=ws_data_with_timestamp.encode(),
                delivery_mode=aio_pika.DeliveryMode.PERSISTENT
            )
            await channel.default_exchange.publish(
                message,
                routing_key=self.queue_name
            )
```

There is a library for the asynchronous communication with RabbitMQ called aio_pika.

As it was pointed out in the Draft Approach, we want to have an ability to get messages from the previous process and to do this we need to use aio_pika.RobustConnection that will reestablish the connections. For this we get the channel with instrument description (exchange+asset+instrument) as aio_pika.connect_robust(). Ordinary AsyncQueue in python wouldn't be able to deliver this.

Then we do a connection to the websocket (The class used here is coming from OKX API, I found it very convenient so didn't implement anything on top of that) creating a subscription sending configuration in the first message.

Then we go over data from the websocket, add a timestamp when we have received it to the raw data and give this raw data to the aio_pika! This way the only delay here is creating a message + sending it to the queue via async API, that won't block the main execution. In practice it blocks by 0-5ms.

```python
async def process(self, process: Callable[[TickRecord], None]):
    """
    Process the data from the queue
    """
    connection = await aio_pika.connect_robust()
    async with connection:
        channel = await connection.channel()
        queue = await channel.declare_queue(self.queue_name, durable=True)

        async def transform_and_process(message: aio_pika.abc.AbstractIncomingMessage) -> None:
            try:
                async with message.process():
                    self.logger.log_with_sampling(
                        f"Received data from queue: {message.body}")
                    ws_data_with_timestamp = json.loads(message.body)
                    records = self.transform_json_to_tick_record(
                        message=ws_data_with_timestamp['data'],
                        receive_timestamp=ws_data_with_timestamp["receive_timestamp"]
                    )
                    for record in records:
                        process(record)
            # pylint: disable=broad-except
            except Exception as e:
                self.logger.log_error(f"Error processing message: {e}")

        self.logger.log_info("Start consuming data from the queue.")
        await queue.consume(transform_and_process)
        try:
            # Wait until terminate
            await asyncio.Future()
        finally:
            await connection.close()
        self.logger.log_info("Stop consuming data from the queue.")
```

In the process function we get the data from the message queue that was passed in the fetcher, but here we don't really care about being fast, we can take out time to do everything we need.

We get data from the queue, transform it to TickRecord and give it to the process functions, that tells where to write this data (what file). Also this part is heavily covered with logs, so we could analyse what kind of data we have (I'm sampling this data a lot to not pollute the log channel) and detect if something goes not as expected, so it could be easily fixed.

run

```python
    async def run(self, process: Callable[[TickRecord], None]):
        """
        Run the fetcher and process the data courutines
        """
        self.logger.log_info("Start fetching and processing market data.")

        await asyncio.gather(
            asyncio.create_task(self.fetch()),
            asyncio.create_task(self.process(process))
        )

        self.logger.log_info("Stop fetching and processing market data.")
```

Run functions just make sure that fetch and process functions work at the same time using asyncio.

Important to note that all fetchers require string to TickRecord transformers. This one really depends on the exchange data and I don't want to look too deep into this here.

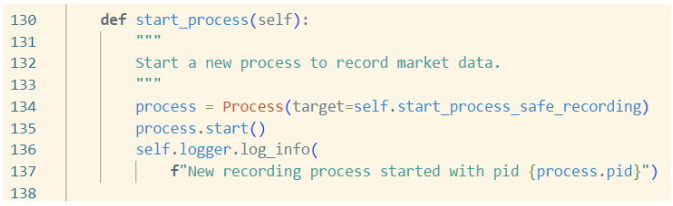
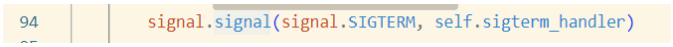
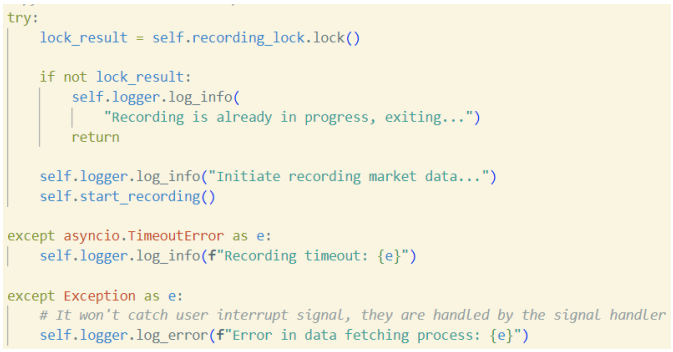
Local recording

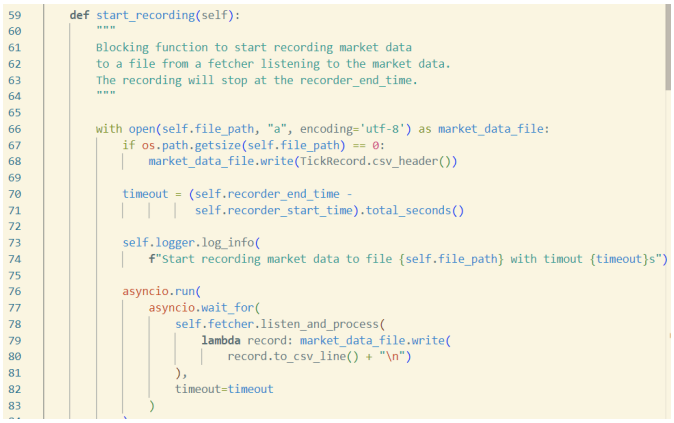
Recording is a much bigger process then just fetching. It requires us to make sure that all the recordings are happening correctly, to the right places and no external interference. One of the primary goals that I want to achieve (you can see it in the evaluation success criteria) with recorder is that it can easily handle many exchanges at the same time. As it was pointed out in the Design section, python doesn't support multithreading due to GIL, so instead we will use multiprocessing.

Let's see the declaration of the MarketDataRecorder class

```python
class MarketDataRecorder:
    """
    A class to record market data to a file from a fetcher listening to the market data.
    """

    def __init__(self, fetcher: MarketDataFetcher, end_time: datetime.datetime):…

    def start_recording(self):…

    def start_process_safe_recording(self):…

    def start_process(self):…

    def sigterm_handler(self, _sig, _frame):…
```

Essentially the way it works is described below

Script main		When we run the script we go over all the fetchers and start the recording process. We are using a sort of composition design principle here.
Start process		The objective here is to create a process for the new recording.
Start safe recording		It's created to make sure that everything is ready to run the important functionality of the recording (more further)
Start safe recording - kill signal handling		This is to make sure that if the process has to be killed we handle mutex (locking) correctly.
Start safe recording - locking		This is to make sure no two processes are running at the same time, because it will do something weird with data. The locking is done via a record in a txt file with a description of the fetcher. Files are a very good way to communicate between processes, so I've used them for mutex. You can also see that asyncio.TimeoutError is handled here - this is an exception that

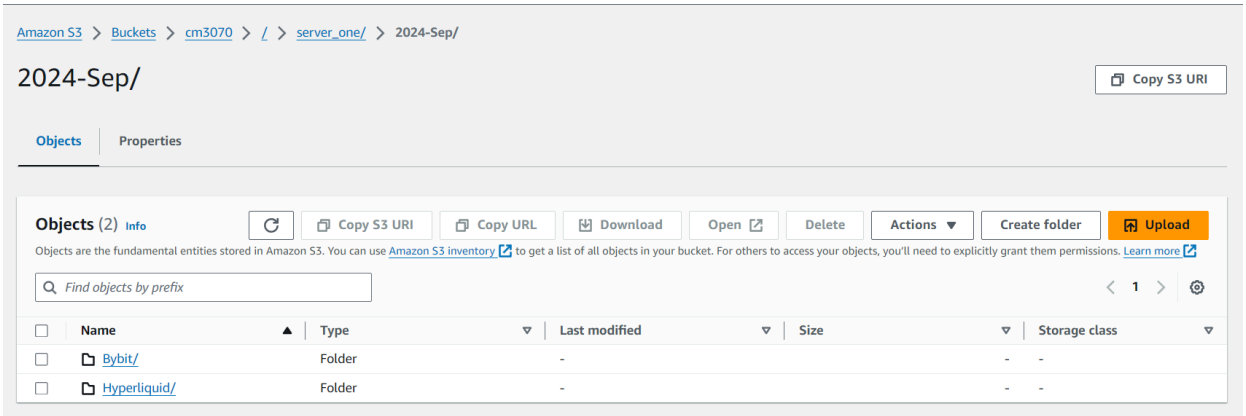
		goes from the actual recording functions (see below)
Start safe recording - next day process creation		In the end of the day (TimoutError exception) we just create a new process that will be working with a file for the next day - this is built with the intention that if something unexpected has finished we could start with a new list the next day. (one example would be unexpected file removal where fetcher is writing)
The actual recording		Here we open a file that was prepared for us by MarketDataRecorder constructor and start the timer till the end of the day, when the fetching will be writing the data down to the file. The time limit is implemented using `asyncio.wait_for` that throws TimeoutError when timeout is reached.

S3

This part is supposed to live as a daemon that dumps all the collected data from the previous day. It's implemented using a cron tool. When the time is come (3 am - to make sure that all the recording processes are finished) and it takes all the files with yesterday name and puts them in S3.

```bash
#!/bin/bash

export S3_BUCKET="s3://cm3070/"
export DATA_SYNC_DATA_DIR="./data/"
export DATA_SYNC_SYNCED_DIR="./synced_data/"
export DATA_SYNC_LOG_FILE="./aws_sync_log.txt"

upload_and_move() {
    # Find files matching the pattern and execute the following
    find "$DATA_SYNC_DATA_DIR" -type f -name "$(date -d "today" '+%-d').csv" -exec bash -c '{
        file="$1"

        # Get relative path from DATA_SYNC_DATA_DIR
        rel_path="${file#./data/}"

        # Create directory structure in DATA_SYNC_SYNCED_DIR
        mkdir -p "$(dirname "$DATA_SYNC_SYNCED_DIR/$rel_path")"

        # Upload to S3, maintaining folder structure
        aws s3 cp "$file" "$S3_BUCKET/${rel_path}"  && echo "$(date "+%Y-%m-%d %H:%M:%S") - Uploaded: $file" >> "$DATA_SYNC_LOG_FILE"

        # Move file to synced_data, maintaining folder structure
        mv "$file" "$DATA_SYNC_SYNCED_DIR/$rel_path" && echo "$(date "+%Y-%m-%d %H:%M:%S") - Moved: $file to $DATA_SYNC_SYNCED_DIR/$rel_path" >> "$DATA_SYNC_LOG_FILE"

        # Schedule deletion from synced_data after 3 hours
        echo "rm \"$DATA_SYNC_SYNCED_DIR/$rel_path\"" | at now + 3 hours
    }' _ '{}' \;
}

# Get the full path of the current script
SCRIPT_PATH=$(readlink -f "$0")

# Check if the script is already scheduled, if not add it to crontab
if ! sudo crontab -l | grep -Fq "$SCRIPT_PATH"; then
    # Append the new cron job to the existing crontab with the full script path
    (crontab -l 2>/dev/null; echo "0 3 * * * sudo $SCRIPT_PATH") | crontab -
fi

# Run the upload and move function
upload_and_move
```

Then in S3 we can see those files



Reliability

Logging and monitoring. In the previous section you could notice that all the code is covered with logs. This is important because it allows you to detect problems early or monitor for some unexpected behaviour. You can just `tail -f logs.txt` and it will provide it. The logging is implemented by wrapping the python logging in a custom class, where I can easily control the format, allowing it to be easily parsed due to the format. Also it gives me a good way to sample

the data. For example we don't want all the data to be in logs from the exchanges, just a few to see that it looks alright (especially useful when onboarding new exchanges).

```python
class MarketDataFetcherLogger:
    """
    A class to log messages for the MarketFetcherLogger class.
    """

    def __init__(self, exchange_name: str, supported_asset_type: AssetType, instrumen
        self.exchange_name = exchange_name
        self.supported_asset_type = supported_asset_type
        self.instrument = instrument

        self.logger = logging.getLogger("MarketDataFetcher")

    def log_info(self, message):
        """
        Log an info message with a prefix.
        """
        self.logger.info("%s %s", self.log_prefix(), message)

    def log_error(self, message):
        """
        Log an error message with a prefix.
        """
        self.logger.error("%s %s", self.log_prefix(), message)


    def log_with_sampling(self, message, sample_rate=0.00005):
        """
        Log a message with a sampling rate.
        """
        if random.random() < sample_rate:
            self.logger.info("%s %s", self.log_prefix("sampled"), message)

    def log_prefix(self, extra_message=""):
        """
        Create a log prefix based on the fetcher class and the current process id.
        """
        # pylint: disable=consider-using-f-string
        return "[%s:%s:%s:%d%s]" % (
            self.exchange_name,
            self.supported_asset_type,
            self.instrument,
            os.getpid(),
            f":{extra_message}" if extra_message else ""
        )
```

Safe exit. When the process is being killed, the sigterm_handler of the data recorder will make sure that lock for the fetcher is released.

Testing. The testing was done in "production", meaning that I used the actual data to see that the system flows work. For this I've implemented a method like supported_instrumes_for_test,

so that I wouldn't have to download all available instruments data, but only the 1-2 per fetcher type. Also it's very easy to monitor

- pgrep -f "python python/market_data/data_recorder.py" - to see the recorder processing (process per fetcher)
- rabbitmqadmin list queues - to see the queues where fetchers are writing data and how many messages are there
- tree data/ --du - to see the data we recorded locally

Setup

This step is the creation of an extensive readme in the github repository. I don't want to go too deep into this in the document, you can check the github for this, but the main objective here was to help developers by providing the following sections

1) Before start - what you need to install (bash commands) before starting the tool or starting the development
2) Start - how to run the tool and a brief description of the output
3) Tips - useful commands that help in testing and maintaining the system. Also it includes solutions to the problems I have encountered while working on the tool.
4) Market Data Fetchers - description of what it is with a tick record format
5) Onboarding a new fetcher - steps to onboard a new fetchers and what fetchers are already implemented with a basic description of the data they provide.

Evaluation

Resources Scalability

One of the primary goals of the project was to build a market data collection tool that can work efficiently on a local machine with limited storage—50GB in this case. This constraint drove many of the design decisions, particularly around storage management and the use of cloud resources. The choice of AWS S3 for long-term data storage and a local file system storage for temporary storage was a critical design choice.

The tool successfully achieves this goal by only keeping the most recent data locally and uploading older data to S3 once it is processed. This ensures that the system never exceeds the 50GB limit, no matter how much data is being collected. By batching data and uploading it

periodically, the project also minimizes the number of writes to the local storage and ensures that the most critical data remains accessible without overloading the system.

From a resource scalability perspective, the architecture allows for smooth handling of multiple exchanges and instruments without saturating the available storage. However the extension will go at the price of updating the frequency of the file uploads to S3. The file size will get smaller as the number of exchanges and instruments grow and it's not handled automatically right now.

Architecture Scalability

Another key goal was architecture scalability. Given that the tool is designed to support multiple exchanges and instruments. The system uses Python multiprocessing to ensure that each data fetcher can run independently without being blocked by the Global Interpreter Lock (GIL). This allows the tool to scale horizontally by simply adding more processes to handle data from different exchanges or instruments.

The architecture is also highly modular, with each data fetcher being relatively easy to onboard. The base class for fetchers simplifies the process of connecting to new exchanges, as demonstrated by the fact that only minimal effort is required to transform raw data from the exchange's API into the desired tick record format. This object-oriented design makes it easy to add support for additional exchanges in the future, which will be crucial for future scalability.

In terms of the cloud infrastructure, the use of RabbitMQ for message queuing ensures that the system can handle high throughput without significant latency. However, as the project scales to more exchanges and instruments, network latency, and bandwidth costs may become more significant, especially. Even now uploading 1 day of data to S3 takes around a minute. A potential improvement for future iterations could involve introducing more robust load balancing.

Data quality

The quality of the data being collected is a critical factor, especially for algorithmic trading, where even small inaccuracies in historical market data can lead to incorrect backtesting results or missed trading opportunities (like collection time here). In this project, the focus on minimal latency between data receiveal from exchanges and queueing the tick for processing helps ensure that the data is not delayed. We could see that we are saving a few dozens of milliseconds using this approach compared to writing at the moment of receiving in the current scale, that would only go worse with the data amount from an exchange increase. The choice to

use a message queue to buffer incoming data before writing it to disk ensures that even during periods of high data throughput, the system does not lose any data.

However, the project could benefit from more rigorous testing to ensure that the data remains accurate under a variety of network conditions. In particular, simulating network delays or outages and observing how the system handles reconnections and data loss would be valuable tests.

Reliability

Reliability is crucial for any tool used in the financial sector, where downtime or data loss can result in significant financial losses. This project addresses reliability in several ways.

- The use of RabbitMQ as a message queue ensures that even if the local machine or storage system experiences an issue, data that has been collected from the exchanges is not lost.
- The use of S3 for long-term storage further enhances reliability by providing a durable and highly available storage solution.

The tool also includes several reliability features in its design, such as locking mechanisms to prevent multiple processes from writing to the same file and cron jobs that automatically upload data to S3 at the end of each day. The system handles process failures gracefully by restarting fetchers and ensuring that data collection continues even if one component fails.

But there is a huge amount of work that the project would benefit from for reliability tooling. Right now we can't tell what process is responsible for which recording, all the commands are in readme and not as a nice command line tool, there is no alerting system etc.

Conclusion

Overall, the tool developed in this project successfully meets its initial goals of resource and architecture scalability, while also maintaining high data quality and reliability. The use of cloud storage, message queues, and multiprocessing ensures that the system can scale to multiple exchanges and instruments without overloading the local machine and if it does it's easy to modify the occurrence of S3 dump. Moreover, the system's modular design makes it easy to onboard new exchanges and maintain reliability through robust error handling and logging.