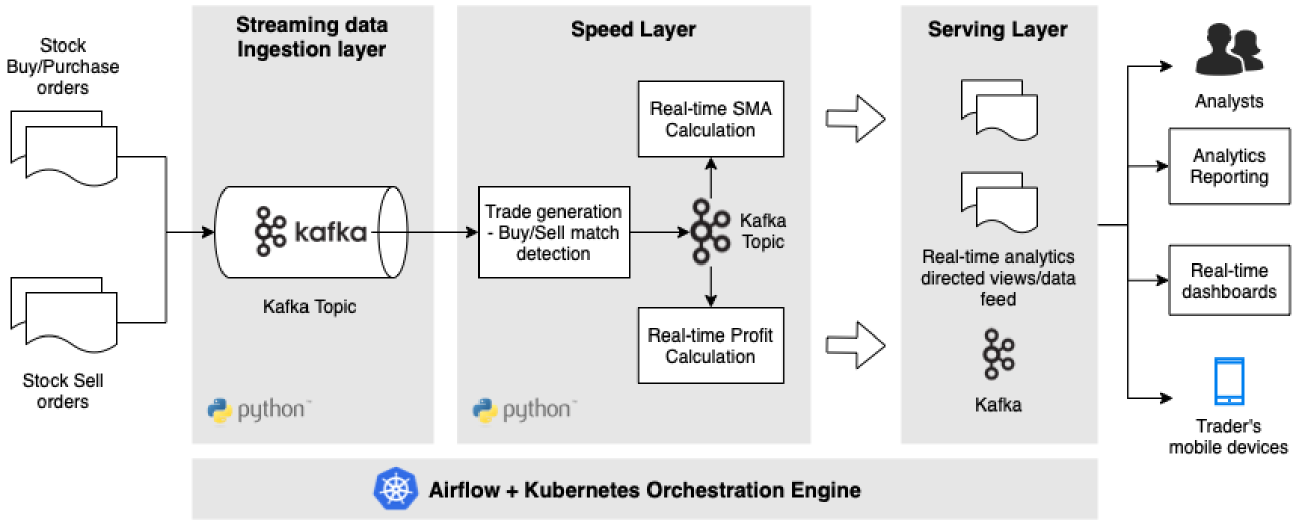
Stream Processing and Analytics

Electronic Stock Exchange:

Exercise 1: Architecture diagram
Tech Stack - Python, Pandas, Kafka, Airflow, Kubernetes
The overall objective of this module is to facilitate day trading of stocks and perform streaming data analytics and statistical analysis on the stock trade data to be consumed by traders on the fly.


Stock Buy/Purchase orders
Stock Sell orders
Streaming data Ingestion layer
kafka
Kafka Topic
python
Speed Layer
Real-time SMA Calculation
Trade generation - Buy/Sell match detection
Kafka Topic
Real-time Profit Calculation
python
Serving Layer
Real-time analytics directed views/data feed
Kafka
Analysts
Analytics Reporting
Real-time dashboards
Trader's mobile devices
Airflow + Kubernetes Orchestration Engine

The above diagram gives an overview of the technical architecture of the Electronic Stock Exchange system. Python has been leveraged as the primary coding language across the platform.

- For this implementation, the Kappa architectural paradigm has been employed to cater to the requirements. Incoming stock purchase and sell orders are received and a real-time **Ingestion Layer** has been designed using open-source stream-processing platform, Apache Kafka. The incoming order feed is received in the input Kafka topic with a field in the input describing whether it is an order of buying or selling a stock.
- Further, after the ingestion phase, the **Speed Layer** consists of a few stream processing modules to process and transform the input data into usable insights; it contains a process for trade generation(match-making algorithm) which detects and matches orders of buying to selling to result in a trade. The generated trades are then passed onto other modules using a different Kafka topic to produce the required statistics on the trades i.e., SMA and profit calculations.
- All these results are then made available to the end-users through the **Serving Layer** by means of processed results and real-time insights which can be pushed to traders' mobile devices using Kafka and mobile API calls. A final layer of HDFS storage can be included to maintain intermittent storages of required results in this layer in case of future business requirements.
- For the purpose of **Module deployment and orchestration**, a combination of Apache Airflow with Kubernetes is being suggested since orchestration of the streaming-data process workflows can be easily managed using Airflow, which is again an open source platform. It has a native functionality which helps in deployment of processes on Kubernetes without much hassle (can be accomplished using the KubernetesPodOperator). This also helps in dependency management since the module is packaged and containerised for deployment.

Exercise 2: Order Producer

Code file has been attached with the document (**order_gen.py**)

As seen in the previous explanations, orders of both Buy and Sell transaction types(*trxn_type*) are published to a Kafka topic from the ingestion layer and are received in the Speed layer. Below is a sample order data feed which gets received here as a JSON object.

Ex:

```
{
        'id': 1000001,
        'instrument': 'TCS',
        'trxn_type': 'BUY',
        'quantity': 5.0,
        'price': 15.0,
        'order_timestamp': '2023/03/12, 23:48:37’
}
```

```
{'id': 1000001, 'instrument': 'TCS', 'trxn_type': 'BUY', 'quantity': 5.0, 'price': 15.0, 'order_timestamp': '2023/03/12, 23:48:37'}
```

Here, the ‘id’ attribute refers to the auto-generated unique order-id associated with each order; there is also an order_timestamp which gets associated with each order for audit purposes; Price, Quantity, Transactiontype and Instrument can be made user inputs which can extracted as input variables from the mobile interface.

Exercise 3: Match Maker

Code file has been attached with the document (**trade_gen.py**)

In the Speed layer, a module is defined to match the BUY and SELL orders to each other and result in a successful trade. Here, we have employed a simple match making algorithm which compares the orders on different parameters.

- As a first step, the orders are split into 2 different datasets - BUY orders and SELL orders. These datasets are intermediately maintained within the process itself using pandas data-frames.
- Secondly, based on the instrument of trade, quantity and corresponding price, both these datasets are compared and matching records are considered as successfully completed trades. Further, to make sure that there are no multiple matches for a single because of a prior queue, any duplicity in trades are removed.
- Thirdly, if any trade is seen to be successful, then both their BUY and SELL records are removed from the intermediately stored data-frames. The python program for this module has been attached.
- Finally, the successful trade along with both the order-ids resulting into the trade is published further into another Kafka topic and also offered as part of the serving layer.

Also, for the purpose of this exercise we have used an equivalence of 3 attributes of orders for trade generation. A further improvement on this can be the usage of only Instrument and Price as the matching keys - quantity can be updated(i.e., -ve quantity if it’s a SELL and +ve quantity if it’s a BUY) on the fly to include partial matches and more accurately generate trade records.

Exercise 4: Exchange – Simple Moving Average Calculator

Code file has been attached with the document (sma_calculation.py)

Exercise 5: Exchange – Profit Calculator

Code file has been attached with the document (profit_calculation.py)

Assumptions/Observations/Considerations

As part of this assignment, we have made the below assumptions and designed the solutions with the corresponding infrastructure:

- Since we have only produced the input data for this assignment, we have restricted it to only a few dimensions/features for the purpose of demonstration. With increasing dimensions, the match-maker algorithm and further data transformation logics might need updates.
- We have assumed an open-source solution for this assignment; with managed services like AWS Kinesis the architecture can modified and implementational difficulty can be reduced.
- We have assumed the availability of a Kafka Cluster and an Apache Zookeeper instance to be installed and running with the deployments; even in our demonstration we have both these services running on our local system.
- Usage of Airflow or Kubernetes is just added as a suggestion for deployment and overall pipeline orchestration. These functionalities have not been implemented as part of the assignment but we feel it would be a good addition to the infrastructure as part of the overall module.
- We also observed that using a configuration of Kafka to reset offset to “earliest”, we can read all the messages in the queue from the first message that has been sent into the topic. This can be used if there is a case of failure in any of the stream and analytics layers.
- In case of complete module delivery, security implications and other parameters might be considered. For this assignment, we have considered it to be out of scope but it would definitely be an important part in the overall system architecture.