Background

To design a sample retail order management system which has the capability to process a couple of millions orders per day.

Requirements

- Build a ROM application which is generating random order in above format and push the message to Kafka (Local). Number of orders created should not be less than 1000.
- Write a spark job to read the order data from kafka and aggregate orders by to location.
- Populate the order data in time series format so that it will be useful for analytical purpose

Scope

- Proposed design covers only the prototype of simple order management flow and not considering the entire lifecycle of order management.For example, warehouse management, logistics management, delivery service, payment service is not taken into consideration.
- Proposed design is not bound to any particular cloud provider.
- No particular PAAS service from the cloud is chosen in order to keep it generalised in nature, whenever we find a fitment of proposed solution to migrate to SAAS service, that can be adopted.
- Details of Monitoring, Orchestration are not part of scope although reference for the same is done based on the dependency of the objective in the production deployment plan section.
- Coding is done as a prototype to achieve the main objective thus not every aspect of validation and check point is not implemented. For example, taking care of duplicate orders is not taken care of.
- Security aspects like kerberos / SSL implementation to cover the design principles has not been implemented neither in setup nor in coding but keeping the security in mind the components were chosen which support and can adhere to AAA policy of data governance.

Design goals

Elasticity

- Should scale seamlessly when needed.

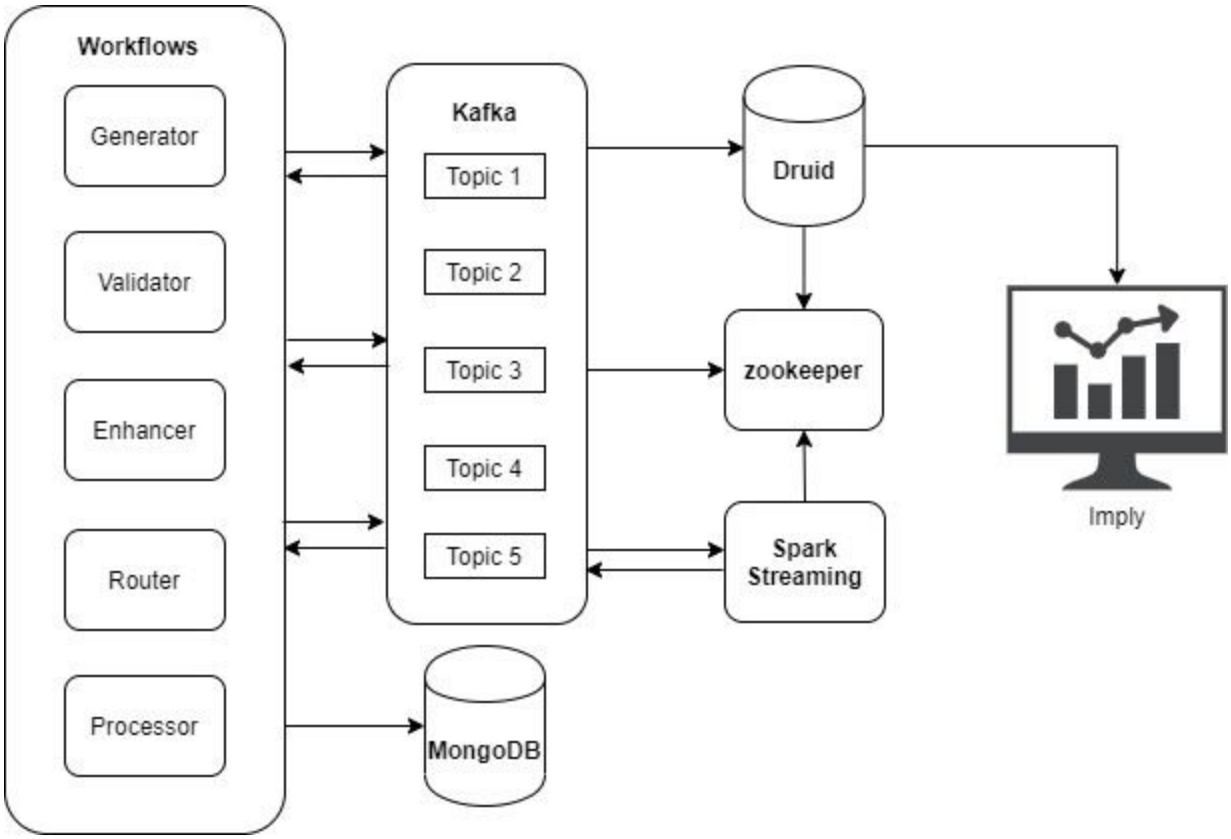
Data security

- Provision to adhere to AAA policy of data governance.
- Provision to ensure minimal impact case of security lapses.
- Provision to implement data governance

Data accessibility

- At every transformation point, every data needs to be accessible easily for any other additional computation.

High level design



Simple Order Management System is divided into following components:

- **Event Generation/Transformation**
  The objective of this is to generate, transform the events which in our case in orders. Design implements the workflows which are captured within the scope and not all which will represent a complete order management system. The one which is been incorporated are as follows:
  - **Order generator**: This workflow is designed to generate random orders consisting of products. This is run as a batch job so that each batch it will generate 1000 orders, if needed more we can do the same by changing the variable.This can be further enhanced to make the numbers of orders configurable which has not been done in this release.
  - **Order validator:** This workflow validates each order to check if the order consists of any ips which is listed as fraud ip. As of now only validation is done against one vector. There could be others which can be considered as validation vectors like duplicate orders, payment defaulters, delay in fulfillment of orders, order cancellation etc.
  - **Order ernicher:** This workflow enriches the to_location and from_location id's to their respective location names. There can be other enrichment capabilities built like IP address to region mapping which can be helpful for logistics, delivery services, demand management etc.
  - **Order Router**: This workflow based on the order type this will route the traffic onto respective channel

- **Order Processor:** External order processor workflow consumes the order and convert the order into JSON object to be

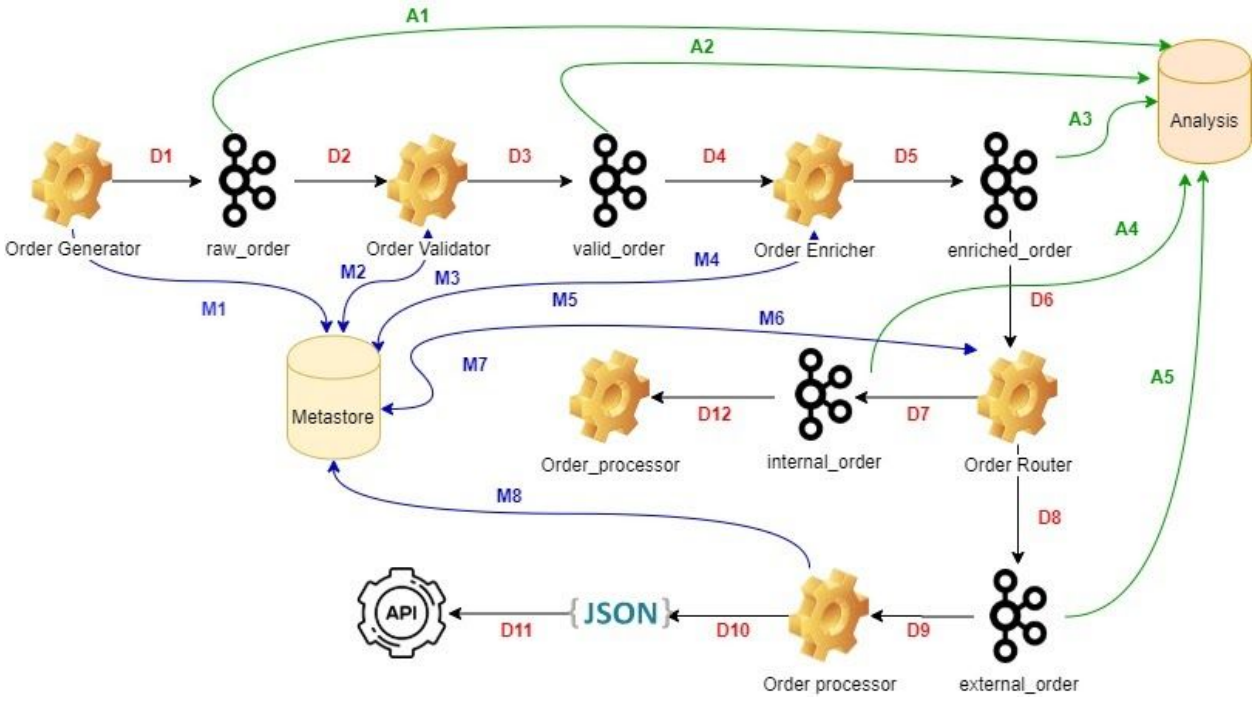
➢ Event collection
- All the orders which are generated needs to be published onto a queue so that it can be consumed by other workflows for enrichment, transformation thus need to fulfill the following requirement:
  - Low latency
  - high throughput
  - Reliable design to handle failures
  - Scalable
- Kafka is chosen as solution which fulfills all the above requirement
- Each workflow reads from a topic transforms the data and publishes back to kafka.

➢ Event processing
- The orders generated need to be processed in large scale and aggregated to transform the data to provide different insights. Thus Apache spark is chosen as the solution which provides the following benefits.
  - Spark executes much faster by caching data in memory across multiple parallel operations
  - Enables to process the query which will help in joining the data between multiple streams and computing the data with speed.

➢ Event store
- Store all the processed data in a time series datastore which can be used for analysis.

➢ Event Analytic platform
- Platform which will provide an interface to visualize and analyse the order data.

Workflow - Data Flow



D1: Order Generator generates the random data and publishes the order onto raw_order topic
M1: Order Generator updates the order status on metastore.
A1: Druid kafka stream supervisor consumes the data from kafka and populates onto Druid.

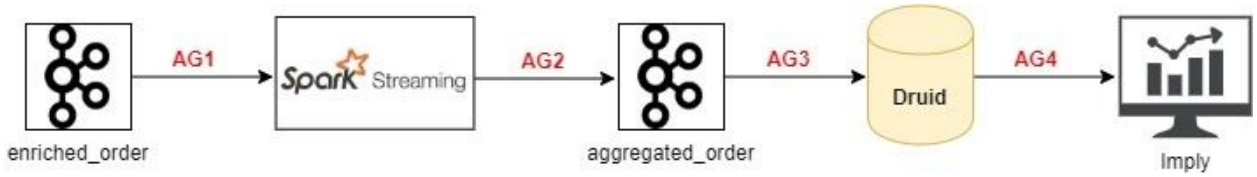
D2: Order validator workflow consumes the raw_order data.
M2: Order validators look up at the metastore for fraud ip.
D3: Order validator publishes the valid orders onto valid_order topic.
M3: Order validator updates the order status on metastore.
A2: Druid kafka stream supervisor consumes the data from kafka and populates onto Druid.

D4: Order Enricher consumes the data from valid_order topic
M4: Order Enricher looks up at the metastore for location names respective to to_location and from_location ID.
D5: Order Enricher publishes the valid orders onto enriched_order topic.
M5: Order Enricher updates the order status on metastore.
A3: Druid kafka stream supervisor consumes the data from kafka and populates onto Druid.

D6: Order Router consumes the data from enriched_topic
M6: Order Router looks up at the metastore to check if the order needs to be routed internally or to an external client to process.
D7: Order Router publishes the valid orders onto internal_order topic for all the internal orders.
D8: Order Router publishes the valid orders onto external_order topic for all the external orders.
M7: Order Enricher updates the order status on metastore.
A4: Druid kafka stream supervisor consumes the data from kafka and populates onto Druid.
A5: Druid kafka stream supervisor consumes the data from kafka and populates onto Druid.

.
D9: External order process consumes the data from external_order topic
D10: External order process produces a JSON which in turn can be sent as API request to external end point for processing
M8: Order Processor updates the order status on metastore.

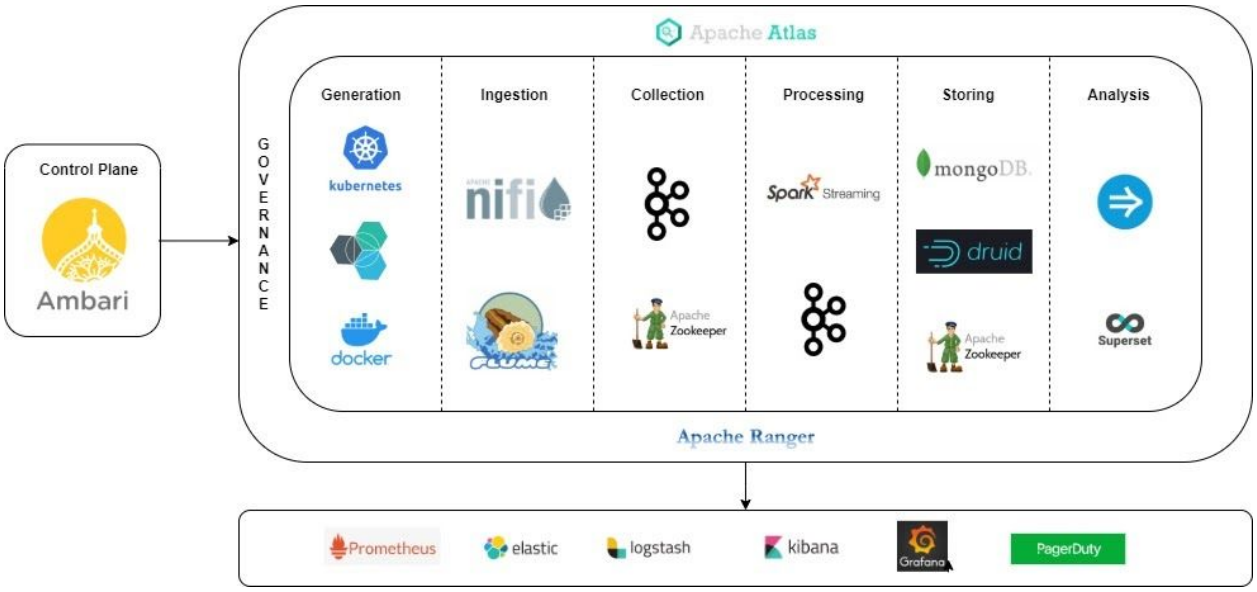
Aggregation - Data flow



AG1: Aggregator Job consumes the enricher_order from kafka as a stream creates data frames, selects the data needed for computation and applies the aggregation accordingly.
AG2: The result is published as write stream to aggregated_order topic
AG3: Druid kafka stream supervisor consumes the data from kafka and populates onto Druid.

AG4: Data cubes are created in the Imply and further analysis is enabled via Imply.

Production rollout plan



Security:

Authentication

- All the components of the pipeline supports kerberos for authentication

Authorization

- Apache Ranger service is used to set up and implement role based ACL's.

Encryption

- File system encryption to take care of data on REST.
- SASL_SSL to support data encryption in transit.

Concerns / Challenges

- In Kafka there is no guarantee that there won't be data loss especially to offset_commit what is chosen. If the application stops processing the data after consuming the data and with auto offset commits the offset this will lead to data loss or if the kafka service goes down and application processes the data this will lead to data duplication.
- Data replication between regions is a challenge with kafka mirror maker service.