Introduction

This document outlines a proposals for a scalable architecture for a real-time alert processing platform for LSST:UK capable of handling the data rates expected during the lifetime of the LSST project.

In order to make the system scalable the architecture is designed from the ground up to avoid high value single instance compute resources wherever possible.

A core design concept behind this is to take advantage of the inherently parallel aspects of the problem and to distribute the processing to collections of small independent micro-services, mapping these onto computing resources in a cloud compute environment.

Kafka streams

Multiple Kafka streams are used throughout the architecture as the data distribution component, providing a loose coupling between independent computing resources typical of a micro-services architecture.

Kafka streams provides built in support for two ways of distributing the data to consumers.

The standard delivery method for Kafka provides a rolling buffer for the stream, enabling each group of consumers to consume data at different rates without interfering with each other.

The ‘at least once’ delivery method distributes the data between a group of consumers, guaranteeing that each alert will be processed by one at least one of the consumers in the group.

Using a combination of these delivery methods enables us to deploy a network of inter-linked processing components, adjusting the level of parallelism to suite the processing load, latency and bandwidth requirements at each stage.

Service coupling

Using Kafka connections to the data transfer between nodes in our system gives us the loose coupling we need to enable us to use different technologies to to implemented different components within the system and for the different components to operate at different speeds.

Flexible buffering

Using the buffering capability of Kafka between processing stages will allow us to optimize compute resources to meet the different latency requirements for each science case in a flexible manner.

For example, a watch list component designed to trigger automated follow-up observations by a robotic platform may need to have as low a latency as possible, evaluating the criteria and issuing the follow-up observation alert soon as possible after the input data arrives. At the other end of the scale, producing jpeg thumbnail images for the website is not a time critical process, and can accommodate a fairly high latency between the time the input data arrives and the time the thumbnail image is eventually produced.

The first stage of our system should be a Kafka service that provides a local rolling buffer of the incoming data stream from the upstream source (ZTF/LSST). This service would provide an initial buffer for the input data, and insulate us from network issues between our system and the upstream producer.

In our example the rapid response trigger and the thumb nail generator can subscribe to the same Kafka buffer service. Each component would be free to process data at the rate that best suited their use case.

Kafka uses a simple offset marker for each client to keep track of which messages that client has seen. When a client acknowledges receipt of a block of messages, Kafka advances the marker to indicate that that block of messages have been seen by that client., When the client requests the next set of messages, Kafka uses the stored offset to select the next block of messages.

By storing a separate offset for each group of clients, the same Kafka service can accommodate widely different data rates to different clients. Providing data to both the rapid response trigger and the thumbnail generator at the appropriate rate for their use case.

The proposed architecture repeats this pattern throughout the system, using local Kafka services to provide a buffer between each stage of the data processing. This provides a simple way to implement a loose coupling between each stage in the processing pipeline that gives us much more flexibility in how the individual layers are implemented.

Kafka partitions

The degree of parallelism available for each stage in the pipeline is determined by the number of partitions configured for the Kafka topic providing the data input.

Kafka stores the data for a topic in a set of partitions, defined when the topic is created. The Kafka client and server components distribute the data between themselves based on the number of partitions.

Consider a simple example of a single server hosting a topic configured with two partitions.
The server will store data for the topic in two directories on the filesystem, one for each partition, distributing the data evenly between the two partitions.

If a single client connects to the server and subscribes to the topic, the client will receive all the data from both partitions, alternating between blocks of data from the two partitions.

If a second client subscribes to the same topic using the same group identifier, then Kafka will spread the data across the two clients, sending data from one partition to each client.

If a third client subscribes to the same topic using the same group identifier, it will not receive any data. The limiting factor is the number of partitions defined for the topic. By itself the Kafka service will not split data from a partition to distribute parts of it to different clients.

If we increase the number of partitions for the topic to four, then the server will spread the data between the available clients, with two of the clients receiving data from a single partition, and one of the clients receiving data from two partitions. Again, Kafka will not split a partition to send fractions of a partition to different clients.

If we increase the number of partitions further, to eight, then the data will be spread more evenly between the three clients, two of them receiving data from three partitions, and one receiving data from two partitions.

The same distribution rules apply to the data distribution between Kafka servers. If we keep the number of partitions set at eight, but we add a second server to the system, then the data for the topic will be distributed evenly between the two servers, each handling data for four of the eight partitions.

The distribution to the three clients remains the same, two of them receiving data from three partitions, and one receiving data from just two partitions.

It should be clear from this (simplified) description that the more partitions defined for a topic the more configurable the parallelism and data distribution is possible.

The current set of evaluation tests have been configured to distribute the data as sixteen partitions distributed across four servers and each test has been configured to use four clients with four threads running in each instance. Further testing is planned to explore the overhead of increasing the number of partitions to 32, 64 or 128 partitions per topic.

Note that the mapping between which client gets data from which server is not fixed, the clients and servers act together as a group to balance the data throughput of the system. As a result, which of the three clients receives only two partitions with of data may change over time as the components work together to balance the system. In addition, if one of the three clients were to fail or drop out of the group, then the system would automatically re-balance the data distribution to spread the data between the two remaining clients.

Kafka as a component

A common pattern in Kafka deployments is to have a single logical instance of Kafka with all of the data analysis processes transferring data to and from the same Kafka service.

However, based on our experience with using Kafka to handle ZTF alert data, the performance of our Kafka services is limited by the I/O bandwidth of network and disc access. This becomes even more significant if the services are deployed on a cloud compute system with network based storage, in which case network bandwidth of the physical system becomes the limiting factor for both data transfer and storage access.

Our use cases form a simple directed graph, without loops or changes in direction. In which case it makes sense to deploy more than one Kafka system, corresponding roughly to layers or stages in the processing pipeline, and to control the placement of these so that we optimize the network connections between the individual Kafka instances and the data processing nodes.

As an example, consider a single Kafka service, comprised of four Kafka server instances, and two stages of data processing, each comprised of four data processing nodes.

In this deployment, the first set of data processing nodes would read their input from the initial Kafka topic, process the data in some way and then push their results back into Kafka as a new topic. The second set of data processing nodes would read their input from this intermediary topic, process the data and again push their results back into Kafka servers as a third topic.

If all three topics are handled by the same logical Kafka services, and hence by the same set of Kafka server instances, then there is a danger that the data transfers to and from the worker nodes will traverse the same set of physical network connections multiple times. In which case we are building in a potential bottle neck in the design.

To avoid this kind of problem, it is better to base our design on having multiple logical Kafka services from the beginning. The first step towards this is to define a Kafka ‘service’ as a logical component containing a Kafka service and its associated Zookeeper service, managed by a orchestration tool such as docker compose or Kubernetes.

The data processing pipeline can then be implemented as a series of processing nodes linked by a Kafka buffer components between each stage. The details of how the components are distributed can be dealt with at a lower level as part of the orchestration layer. The key design pattern to establish at this level is to treat each of the Kafka buffers between processing stages as just another pluggable component rather than re-using one big Kafka service.

Processing component

To support a range of different use cases the system should be able to to support a wide variety of different technologies for implementing processing components.

Python may be suitable technology for implementing many of the data analysis components, on the other hand Java may be a better technology for implementing some of the more technical components such as the catalog cross match and watch list components. Some users working on statistical analysis may be more familiar with using R to implement their algorithms.

To support dynamic configuration and management of the Kafka streams, topics, partitions etc