

# Introduction

This document outlines a design for a scalable architecture to implement 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 resources wherever possible. To achieve this, the design takes advantage of the inherently parallel aspects of the problem to distribute the processing to small independent micro-services, mapping these onto computing resources in a cloud compute environment.

Many of the ideas in this design come from our experience of working with the OGSA-DAI distributed data access platform developed by EPCC at the University of Edinburgh. In particular the concepts of using REST web-service control interfaces to connect together data processing components (Activities) and build an interconnected pipeline (workflow) that operates on a stream of data rows (Tuples) are core parts of the OGSA-DAI architecture.

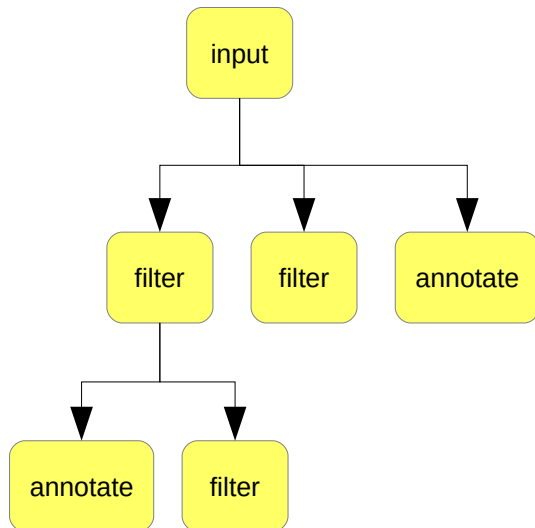
The design outlined in this document makes use of current technologies to implement a similar architecture, taking advantage of recent developments in cloud compute platforms to enable greater use of concurrent execution on multiple machines.

However, we are aware that recent developments in a number of key technologies, in particular the Apache Spark Structured Streaming platform means they already provides much of the functionality described in this design. In which case it may be better to consider adopting the Apache Spark Structured Streaming platform as-is as the basis for the LSST:UK platform and concentrate on developing extensions to provide the additional functionality needed to meet the full requirements for the LSST:UK system.

This document outlines a number of components of the original design that may be applicable in implementing the data transport layer behind an Apache Spark streaming deployment. Other aspects of the system, particularly the components and interfaces seen by an end user are probably better provided by the Apache Spark platform.

# Tree of nodes

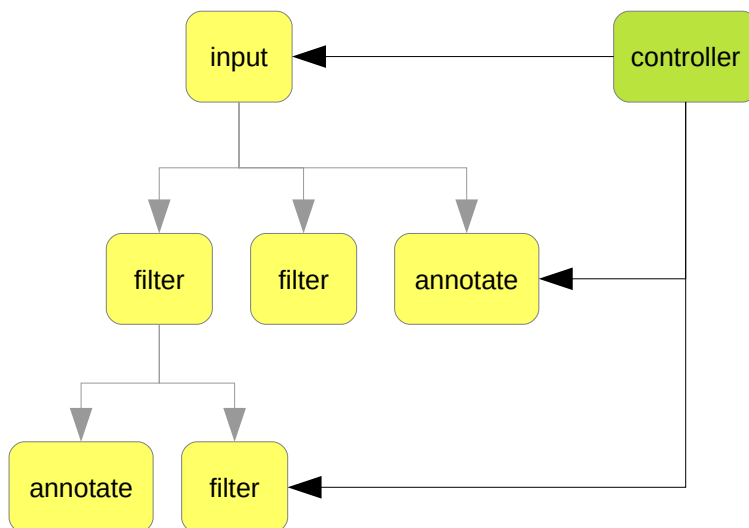
The system can be viewed as a tree structure, with the top level being the initial input of alert messages from the upstream source.



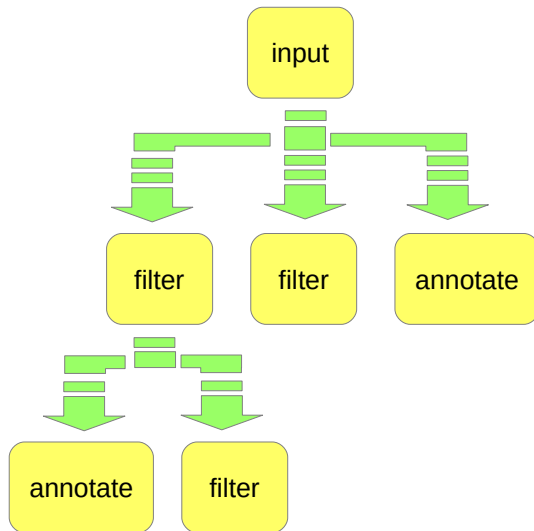
A node in the tree represents a processing step that applies an operation to each alert message in turn.

Each node can either filter the input messages, selecting and passing on those that meet a specific criteria, or generate a new output type based on the content of the input message.

Each node is a web-service component in its own right, with a REST based command and control web service API.

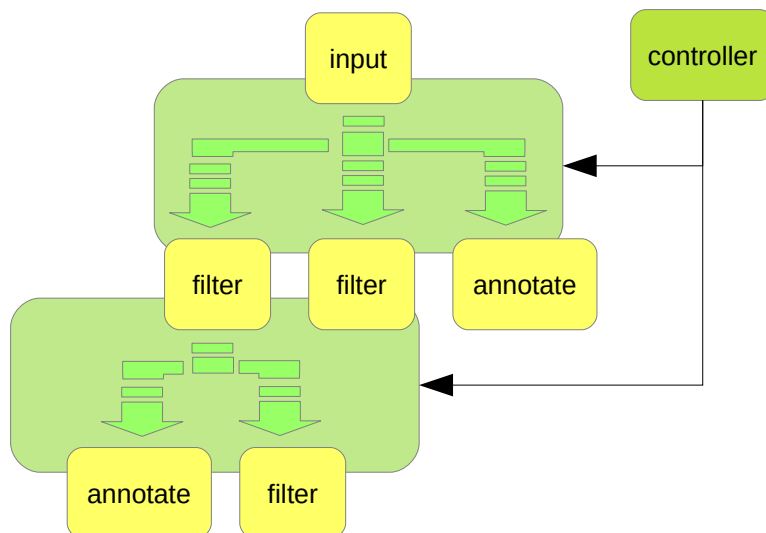


Nodes are connected together using Kafka streams acting as FIFO buffers to provide a loose coupling between the nodes.



Each Kafka buffer is also a web-service component in its own right, with a REST based command and control web service API.

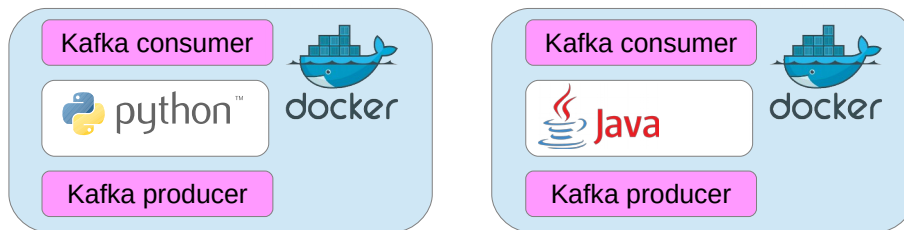
The resulting network of nodes and buffers can be deployed and managed using a client library that connects to the REST web service API on each node.



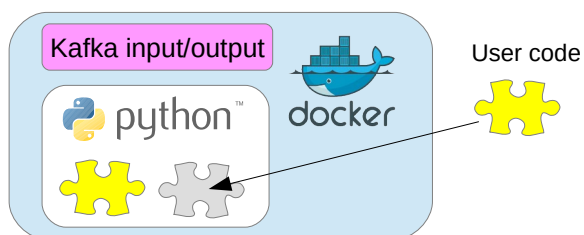
Together these components provide a toolkit that enables users to create their own pipeline of nodes to filter and process the input data stream, connecting together existing node types or by creating their own new types of node.

## Node templates

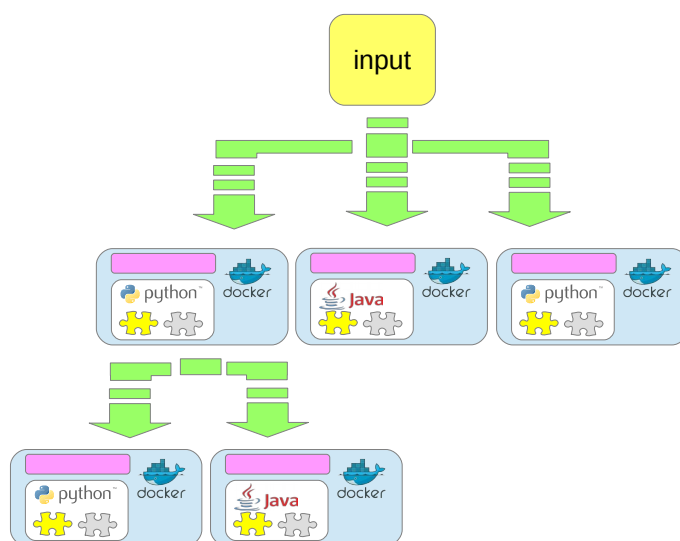
In order to simplify the process of creating new types of node we would provide a template container for each programming language or environment. The templates would implement the node controller functionality and all of the Kafka consumer and producer connections.



The end user would only need to provide the code that implemented the required algorithm. Filling in the space labeled “your code goes here”, reading input messages from a data source interface provided by the container, applying the required processing and writing the results to an output interface.

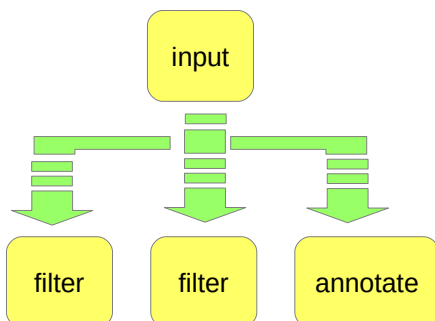


Overall the architecture could be considered as providing a “function as a service” platform, providing all of the infrastructure and components for building a workflow.



# Kafka services

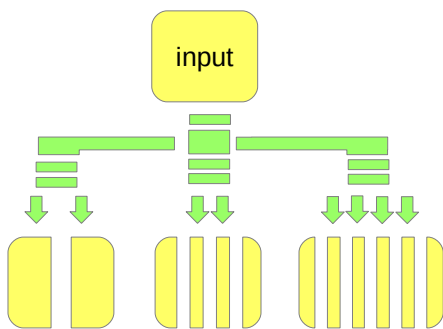
Kafka services are used throughout the system as the data distribution layer, providing a loose coupling between nodes in the tree.



Kafka implements a rolling buffer for a stream of messages, enabling each consumer, or group of consumers, to process the data at different rates without interfering with each other.

Multiple clients can subscribe to a data stream, and Kafka will ensure that all of the messages are received by all of the clients.

Alternatively, if a group of clients subscribe using the same identifier then Kafka will distribute the messages between the members of the group, ensuring that each message is processed by one client within the group.



Using a combination of these delivery methods enables us to create a tree of processing nodes with a buffer between each layer and different levels of concurrency within the layers.

If the components within a logical node all subscribe to the stream with the same group identifier, Kafka will automatically distribute the messages between the set of concurrent processes for that node. Distributing the data between concurrent processes in this manner requires no additional configuration of the Kafka clients or services beyond setting the group identifier when subscribing to the topic.

Using Kafka buffers between processing stages also allows us to optimize compute resources to meet the different latency requirements for each science case.

At the top of the tree is a Kafka service that provides a rolling buffer (~days) of the incoming data from the upstream source. This provides an initial buffer that insulates the rest of the system from network issues between our system and the upstream producer.

Below the top node multiple child nodes can subscribe to data stream, all processing data at different data rates depending on their requirements.

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 (~sec) after the input data arrives.

At the other end of the scale, producing the thumbnail images for the website is not a time critical process, and can accommodate a relatively long delay (~hrs) between the input data arriving and the thumbnail image available on website.

By keeping track of the messages seen by each client, the same Kafka service can accommodate

different data rates for different clients. Providing data to both the rapid response trigger and the thumbnail generator at the appropriate rate for each of those components requires no additional configuration within the Kafka service.

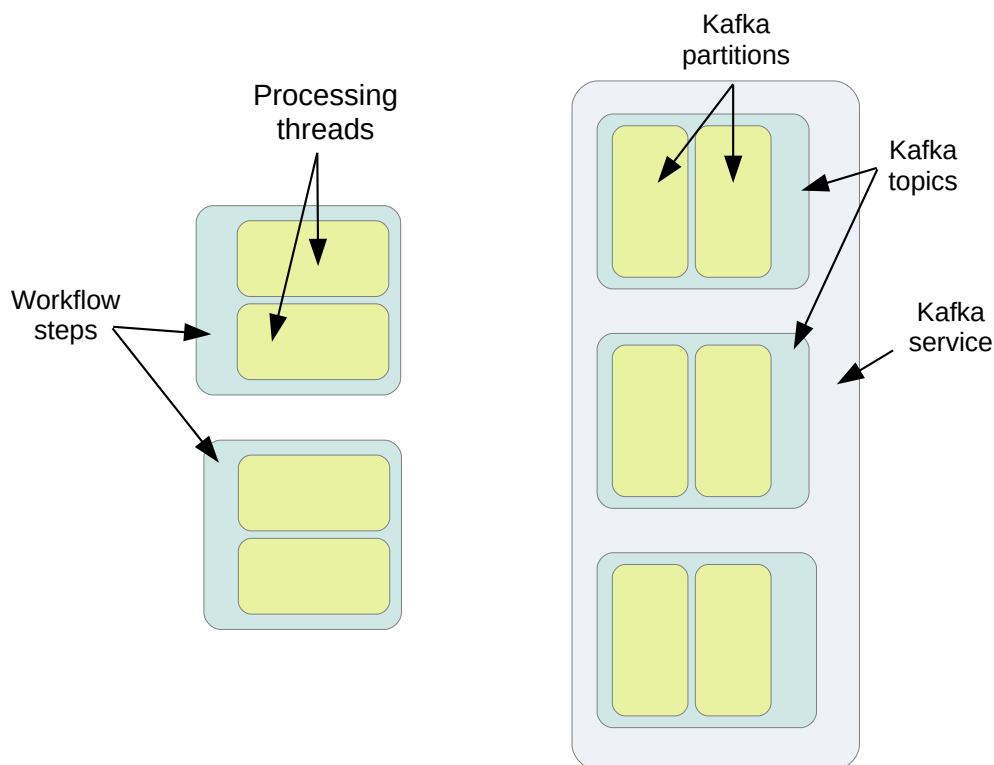
## Kafka components

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

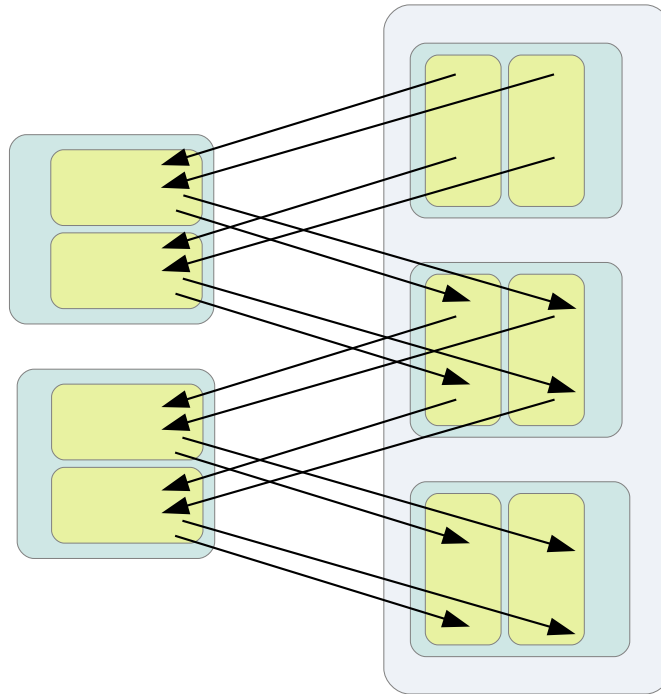
However, based on our experience with using Kafka to handle ZTF alert data, the performance of our Kafka services is limited by the I/O bandwidth of network and disc access. This becomes even more significant if the services are deployed on a cloud compute system with network based storage, in which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

Many of our use cases form a simple directed graph, without loops or changes in direction. In which case it may make 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 two server instances, and two workflow steps each containing two execution threads.

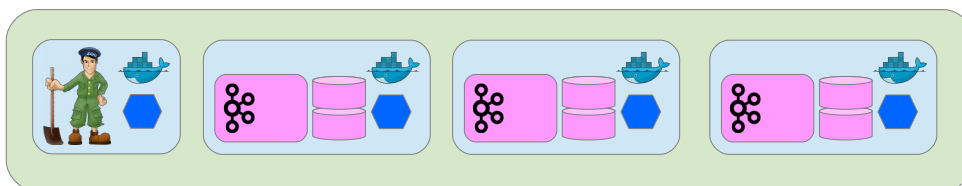


In this deployment, the first set of data processing nodes would read their input from the initial Kafka topic, process the data in some way and then push their results back into Kafka as a new topic. The second set of data processing nodes would read their input from this intermediary topic, process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



If all three topics are handled by the same Kafka services on the same compute resources, then there is a danger that the data transfers to and from the workflow nodes will traverse the same set of physical network connections multiple times, building in a potential bottle neck in to the design.

To avoid this problem the design uses separate Kafka instances to provide the data buffers. In order to support this, we can define a Kafka ‘service’ as a logical component containing the Kafka servers and the 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 Kafka buffers, using these Kafka component packages to deploy separate Kafka instances at each layer. This enables us to control the placement of the Kafka service instances to avoid crossing the same physical resources multiple times.

# 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 Kafka clients and servers will act together as a group to balance the data throughput of the system. Which of the clients receives which partitions may change over time as the clients and servers work

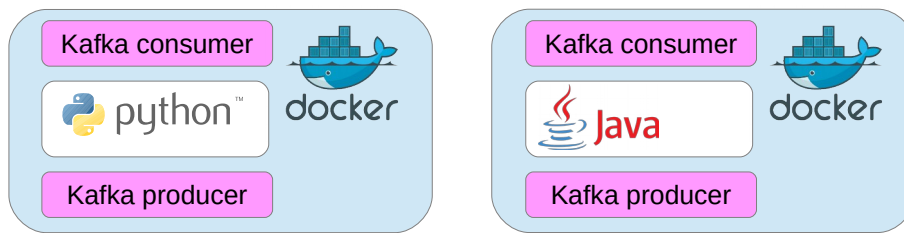


together to balance the system.

This allows Kafka to cope with different clients operating at different speeds, and if one of the clients fails or drops out of the group then the system would automatically re-balance the distribution between the remaining clients.

## Node containers

The architecture refers to containerized components for both the data processing nodes and the Kafka buffers. The most obvious technology choice for this is to use Docker containers orchestrated by a container management platform such as Kubernetes.



However the design is not necessarily linked to any particular technology for implementing these components. Abstracting the details of the containerization layer behind web service interfaces, enabling the platform to change the underlying containerization technology with minimal impact on the rest of the system, particularly the end user code in the processing nodes.

## Future directions

Recent developments in a number of key technologies, in particular the Apache Spark Structured Streaming platform mean the already provide much of the functionality described in this design.

As a result it may be better to consider adopting the Apache Spark Structured Streaming platform as-is as the basis for the LSST:UK platform and concentrate on developing extensions to provide the additional functionality needed to meet the full requirements for the LSST:UK system.

An example of this approach is the Astronomy eXtensions for Spark project (AXS), which uses the existing functionality of Spark to provide the basis of their system and concentrated on developing the extensions to the platform needed to meet their science requirements.

Based on this, and our experience of developing similar systems, our current recommendation is to concentrate our effort on evaluating the capabilities of the Apache Spark Structured Streaming platform with a view to adopting it as the basis for the LSST:UK platform.

Some of the technical details outlined in this document may be useful in developing the components needed to extend the Spark system. For example the deployment of the Kafka data buffers as packaged components and the Kafa data partitioning. However, it is unlikely that developing a complete bespoke system as outlined in this design is a viable solution.