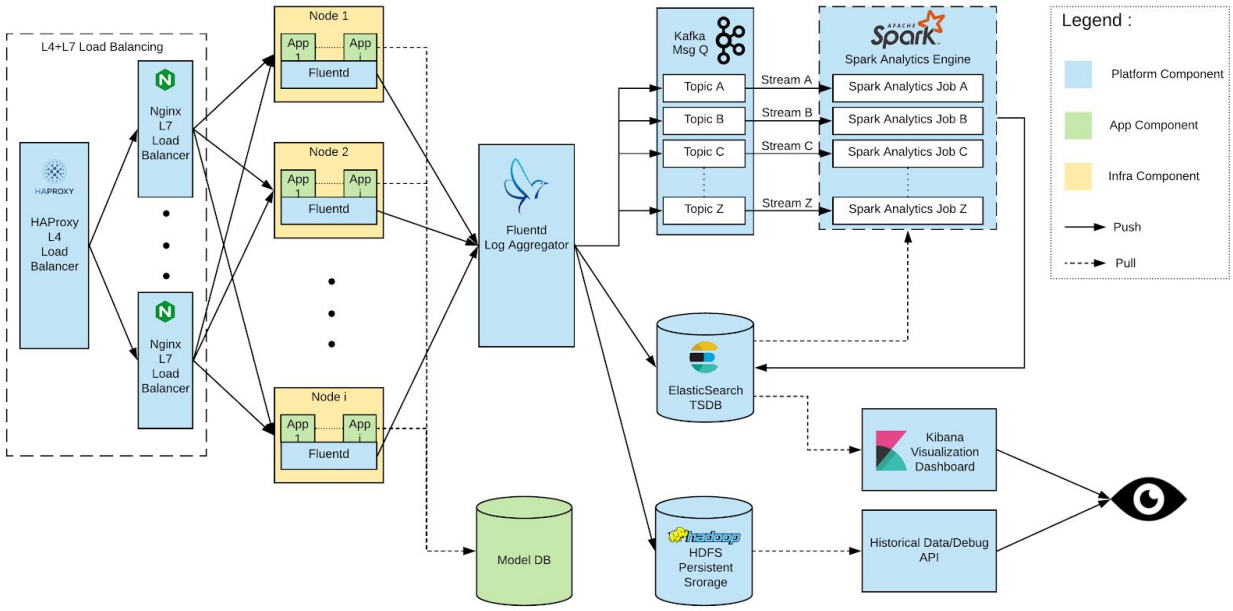
Introduction

This document offers two slightly different architectural solutions for our problem. These two solutions offer a tradeoff which will be explained and compared later in this document. Since the two architectures are fairly similar, the majority of various design decisions will be explained for the first solution. As for the second solution, only the difference will be highlighted.

Stream/Push/Event-Driven Architecture



Platform Components

- **HAProxy**: A L4 load balancing solution which will be responsible for handling massive volume of read/write requests at transport level
- **Nginx**: A L7 load balancing solution which will be responsible for handling massive volume of read/write requests at application layer
- **Fluentd**: A log aggregation solution responsible for aggregation, filtering, buffering and formatting of various types of logs within the system
- **Kafka**: Provides a message Q that is responsible for broadcasting messages to various other components (mainly spark jobs) and act as an event dispatcher for the system
- **Spark**: A cluster-computing framework responsible for the bulk of analytics work. Various ML operations would be perform by spark jobs to produce the desired metrics
- **Elasticsearch**: Used as a time series database (STDB) to record various

- **Hadoop HDF**: Works as a long term persistent storage where logs are collected for future internal use such as debugging and further analysis.
- **Kibana**: data visualization dashboard for Elasticsearch

Workflow Design

At the ingress edge of the system, resides the load balancing module which comprises L4 and L7 load balancing solutions. L4 load balancing is provided by HAProxy. Note that a hardware solution is highly desirable and maybe even recommended i.e. ALOHA Load Balancer. The L4 load balancer(s) distributes the load over a number L7 load balancers. These could be purely software solutions or possibly hardware solutions. These load balancers distribute the load at the API level. We use Nginx servers, they could be deployed on worker nodes or better yet have their own standalone servers. Note that this load balancing task could be delegated to Kuberenetes or any underlying cluster orchestration system that might offer such solutions. **Note that both L4 and L7 solutions can have their own local fluentd installed so that they too can send their events and metrics to the log aggregator**.

Furthermore, fluentd will be installed on each worker node in the cluster. All logs to **std_out** and **std_err** from all apps running on the node would be collected and sent to the fluentd aggregator.
fluentd aggregator will ingest and format logs. It will further filter and split the logs into 3 main streams.

1. **Kafka stream**: Logs are published to a variety of topics on kafka. Which log would end end up on which topic is decided by a set of rules and filters on fluentd's log aggregator
2. **Elasticsearch stream:** "Almost" all logs are sent to Elasticsearch where they might be used by sparks jobs for ML purposes or to be queried by the user to reflect various metrics. Some might be discarded if they are irrelevant to our analytics goals.
3. **Hadoop HDFS**: "All" logs are sent to Hadoop for a long term, persistent storage where they can be later accessed and asses for development and debugging purposes

Note that **most messages will be replicated on all three of these streams**. We will mainly focus on the **Kafka** stream. Each Kafka topic is designated for a unique analytic task for example producing a metric that represents average RTT in the last hour. Let's call this topic AVG_RTT. These messages would be consumed by a spark job (or a collection of tasks in AVG_RTT **consumer group**) that undertakes the analysis operation required to produce such a metric. Once a new value for the metric is achieved, the park job records the metric into Elasticsearch. Note that spark jobs can also query previous data from Elasticsearch if the current analysis requires it so.

The rest of the design is more or less self explanatory through the Figure. Kibana accesses Elasticsearch to provide the user with the visualization of the metrics and other aspects of the time series data.

We have also provided an API for internal uses where a developer can peek into the HDFS storage and find older logs for debugging purposes.

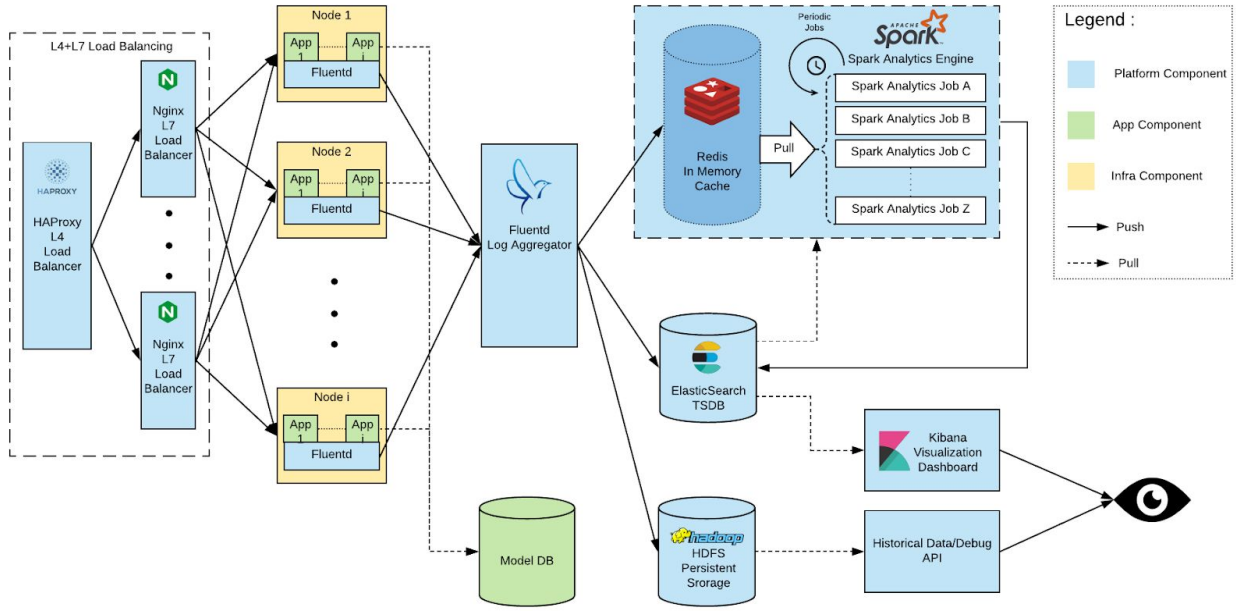
Scalability and Availability

We suggest scalability be delegated to the underlying cluster orchestration system such as pod scaling in Kubernetes or Autoscaling in AWS. Implementing your own auto scaling solution is not impossible but is definitely very difficult. Here is the short version of the effort required for such a solution.

- A metric collection solution must be added to the system. The best solution I can think of is prometheus. Prometheus offers metric exporters for a wide variety of products as well as hardware components (https://prometheus.io/docs/instrumenting/exporters/).
- These metrics must be sent to a **ResourceManager** component were performance thresholds are checked against incoming metrics to capture any performance shortcomings.
- ResourceManager then is responsible to bring up new resource and deploy another replica of the troubled component on it

However, it is important to note that, if there is an auto scaling solution, this architecture allows scaling of each component completely independent of other components

Batch/Pull/Periodic-Task Architecture



Difference

In this solution, instead of kafka pushing streams of logs to spark jobs, logs are cached in an In Memory DB, e.g. Redis, where they would be consumed in batches. The jobs are scheduled each hour to undertake the data analysis task by querying a batch of logs corresponding to a certain metric stored within one hour timespan and producing the desired metric.

Tradeoff

The reason I offered the second solution was specifically due to the following requirement:

Provide metrics to customers with at most one hour delay

The first solution (although a superior solution in my mind) makes it difficult to verify whether this requirement is met or not. Imagine the following scenario. Let's say the volume of the logs on one of the kafka queues e.g AVG_RTT start to increase to the point that there are more logs produced on the topic than they are consumed by the spark jobs responsible for consuming it. Now this means that the delay between the time that the log was originated from the source to the time that it is consumed by the spark job keeps extending. Eventually, we might have a

situation where the metric corresponding to AVG_RTT for 2:00-3:00 pm time period is produced at 4:30, a 30 minutes delay. Obviously, the solution would be to scale up spark jobs in AVG_RTT consumer group. However, to realize that we are even in such a situation in the first place is very difficult. It would require additional checks, logic and complexity in the spark job as well other places in the system.

However, in the second solution, we run spark jobs as scheduled tasks and time how long they take to consume the batch of logs corresponding to AVG_RTT metrics stored between 2:00-3:00. If the spark job takes longer than an hour we know for sure we are lacking resources for this specific stream and thus can scale up the corresponding spark job.

The tradeoff is that, in the first solution, if we properly implement the scaling mechanism for the spark jobs or under lower loads, the metrics would be available as soon as they could theoretically be. No need to wait for the periodically scheduled job to start

Another difference of this design is that it decouples log production from log consumption. If we decide to add a new metric to the analytics system, all we need to do is to create a new spark job where as previously we had to add the corresponding Kafka topic as well as the filtering mechanism in the Fluentd aggregator.

The tradeoff is that, in the second solution, spark job has to implement querying logic and might cause conflict if the DB schema changes.