by **RAJIV KURIAN (HTTPS://WWW.SIGNALFX.COM/AUTHOR/RAJIV/)** | Posted in: **ENGINEERING (HTTPS://WWW.SIGNALFX.COM/BLOG-CATEGORIES/ENGINEERING/)** • **INTEGRATIONS (HTTPS://WWW.SIGNALFX.COM/BLOG-CATEGORIES/INTEGRATIONS/)** • **MONITORING (HTTPS://WWW.SIGNALFX.COM/BLOG-CATEGORIES/MONITORING/)**

This was originally published as a guest post on the Confluent blog (http://www.confluent.io/blog/how-we-monitor-and-run-kafka-at-scale-signalfx).

Why We Chose (and Love) Apache Kafka

SignalFx is used to monitor modern infrastructure, consuming metrics from things like **AWS (https://www.signalfx.com/amazon-rds-monitoring/)**, **Docker (https://www.signalfx.com/docker-monitoring/)**, **Elasticsearch (https://www.signalfx.com/elasticsearch-monitoring/)**, and **Kafka (https://www.signalfx.com/kafka-monitoring/)**, and applying analytics in real time. We've relied on Kafka for the core of our infrastructure since the beginning.

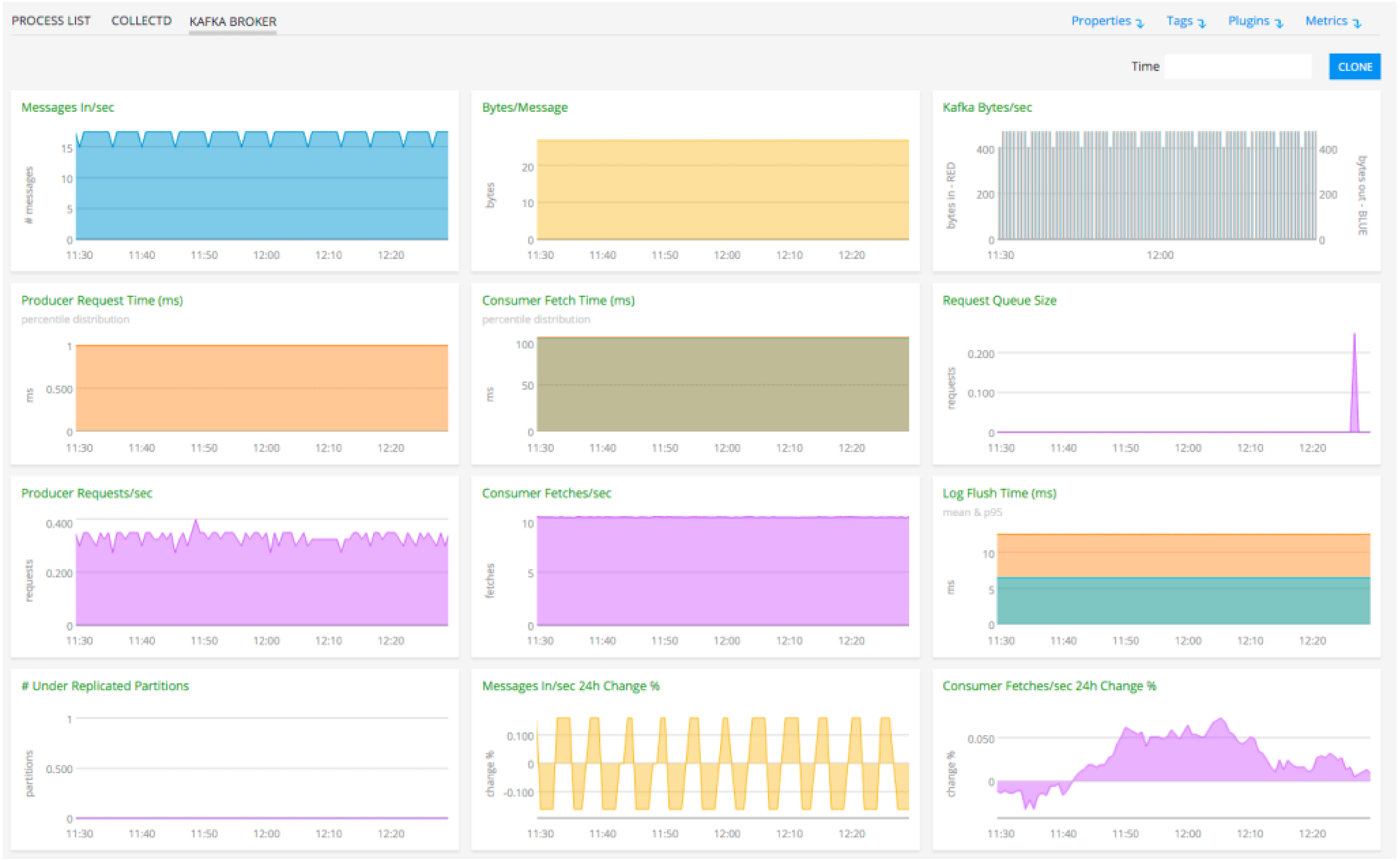
Our engineering team has decades of experience with every kind of streaming or messaging platform out there. Kafka's unique ability to combine *high throughput with persistence* made it ideal as the pipeline underlying all of SignalFx. Throughput is critical to the kind of data SignalFx handles: high-volume, high-resolution streaming times series. And persistence lets us smoothly upgrade components, do performance testing (on replayed data), respond to outages, and fix bugs *without losing data*.

SignalFx does on the order of *70+ billion messages per day* with:

- 27 brokers
- 1000 (approx) active partitions
- 20 (approx) active topics
- 1M+ messages / sec (and growing daily!)

One huge advantage of Kafka has been the fantastic community that's built up around the project, from independent developers to all the people at Confluent. We've found that for every hurdle we've run into, the community has been extremely responsive in either helping us correct some mistaken pattern on our part—or to fix bugs and add roadmap items that address the needs of environments like SignalFx.

In this post, we'll go over some lessons learned from monitoring and alerting on Kafka in production, at scale, in a demanding environment with *very high* performance expectations.



Operating Kafka

Instrumentation: Collecting Metrics

We use a **GenericJMX (https://collectd.org/wiki/index.php/Plugin:GenericJMX)** collectd plugin to collect metrics **exposed (https://kafka.apache.org/documentation.html#monitoring)** by Kafka via **JMX (https://en.wikipedia.org/wiki/Java_Management_Extensions)**, particularly for **brokers and topics (http://docs.signalfx.com/en/latest/integrations/collectd-info.html#using-collectd-metrics)**. You can also instrument the clients themselves but we haven't found that necessary (so far). But we do do something interesting there: wrap the client in a layer that's instrumented so that we know what services in SignalFx are producing or consuming messages and of what size. This is a relatively standard Kafka pattern: adding additional functionality on top of the client to expose more capabilities.

If you use collectd and the GenericJMX plugin configured for Kafka, SignalFx provides built-in dashboards displaying the metrics that we've found most useful when running Kafka in production. Since topics are set by you when you set up Kafka, for per topic metrics we provide templates where you can insert your topic names.

`Bytes In`	`Bytes Out`	`Messages In`
`Active Controllers`	`Request Queue`	`Under Replicated Partitions`
`Log Flushes`	`Log Flush Time in ms`	`Log Flush Time in ms - 95th Percentile`
`Produce Total Time`	`Produce Total Time - 99th Percentile`	`Produce Total Time - Median`
`Fetch Consumer Total Time`	`Fetch Consumer Total Time - 99th Percentile`	`Fetch Consumer Total Time - Median`
`Fetch Follower Total Time`	`Fetch Follower Total Time - 99th Percentile`	`Fetch Follower Total Time - Median`

Investigation: Log Flush Latency and Under Replicated Partitions

The most important metrics we track are:

- Log flush latency (95th percentile)
- Under-replicated partitions
- Messages in / sec per broker and per topic
- Bytes in / sec per broker (collected as a system metric from each broker using collectd)
- Bytes in / sec per topic
- Bytes / message

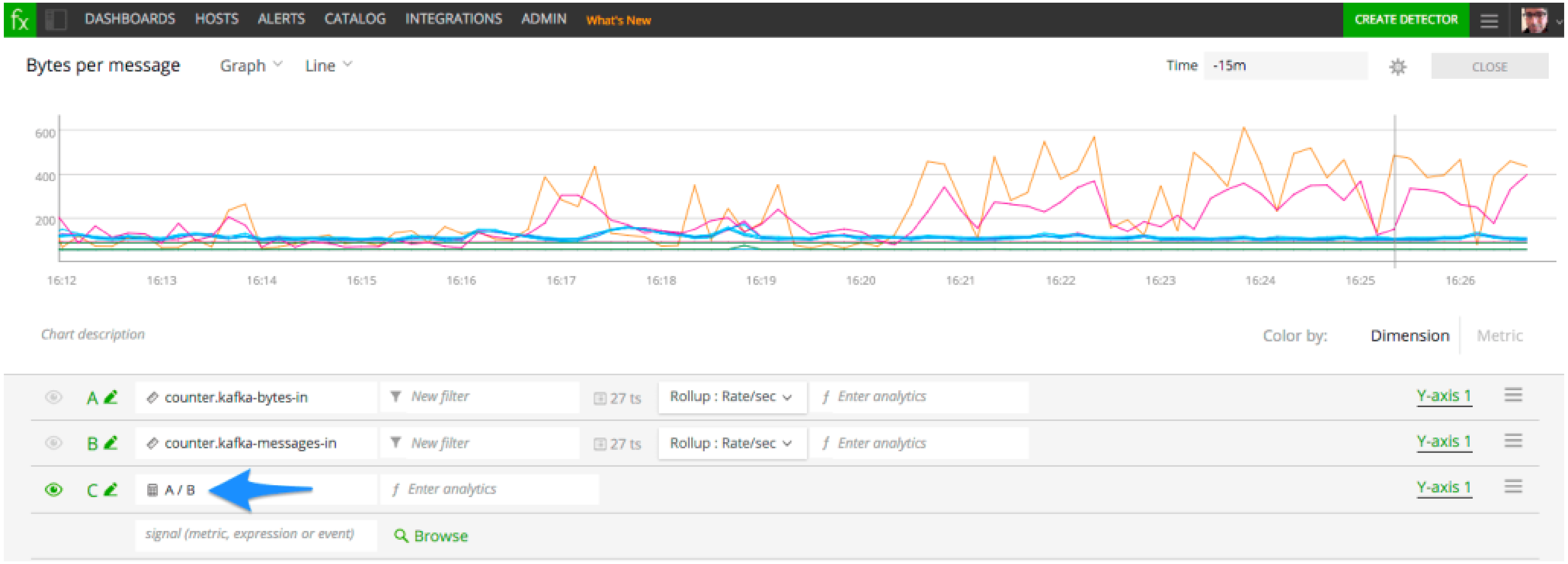
We've found log flush latency and under replicated partitions to be the leading indicators that we need to pay attention to what's going on and prepare to investigate a new bug or regression.

Log flush latency is important, because the longer it takes to flush log to disk, the more the pipeline backs up, the worse our latency and throughput. We monitor for changes in the 95th percentile. When this number goes up, even 10ms going to 20ms, end-to-end latency balloons and all of SignalFx is affected. impacted.

Under-replicated partitions tells us that replication is not going as fast as configured, which adds latency as consumers don't get the data they need until messages are replicated. It also suggests that we are more vulnerable to losing data if we have a master failure.

Changes in these two metrics generally lead us to dive into the other three metrics.

Messages and bytes in tell us how well balanced our traffic is. if there is a change in their standard deviations, we know that a broker(s) is overloaded. Using messages in / sec and bytes in / sec, we derive bytes / msg.



A counter.kafka-bytes-in | New filter | 27 ts | Rollup : Rate/sec | Enter analytics | Y-axis 1
B counter.kafka-messages-in | New filter | 27 ts | Rollup : Rate/sec | Enter analytics | Y-axis 1
C A / B | Enter analytics | Y-axis 1
signal (metric, expression or event)

Bigger messages have a higher cost in performance, because it takes a proportionally longer time to process those messages and can cause the pipeline to back up. A sudden, positive change in the message size could indicate a bubble in the pipeline or a fault in an upstream component. A trend of larger message sizes over time suggests an unintended architectural change or an undesirable side effect of a change to another service causing it to produce larger messages.

Real example:

- We found a bug in our code where we'd increased message size unintentionally
- We were preallocating 256 byte arrays to serialize messages, which was way too large
- Everything slowed down
- But we saw the larger message size on the chart

- Tracking down and reducing the allocation to just what was needed (about 32 bytes) immediately reduced network I/O by 4x

There are other metrics important for monitoring Kafka that don't come from Kafka directly. For example, our messages themselves have timestamps to track approximate latency from producers to consumers. Increases in end-to-end latency between services can indicate a Kafka issue, since it's usually the largest contributor to inter-service latency for SignalFx. We look forward to timestamps coming to Kafka messages **soon (https://cwiki.apache.org/confluence/display/KAFKA/KIP-32+-+Add+timestamps+to+Kafka+message)**!

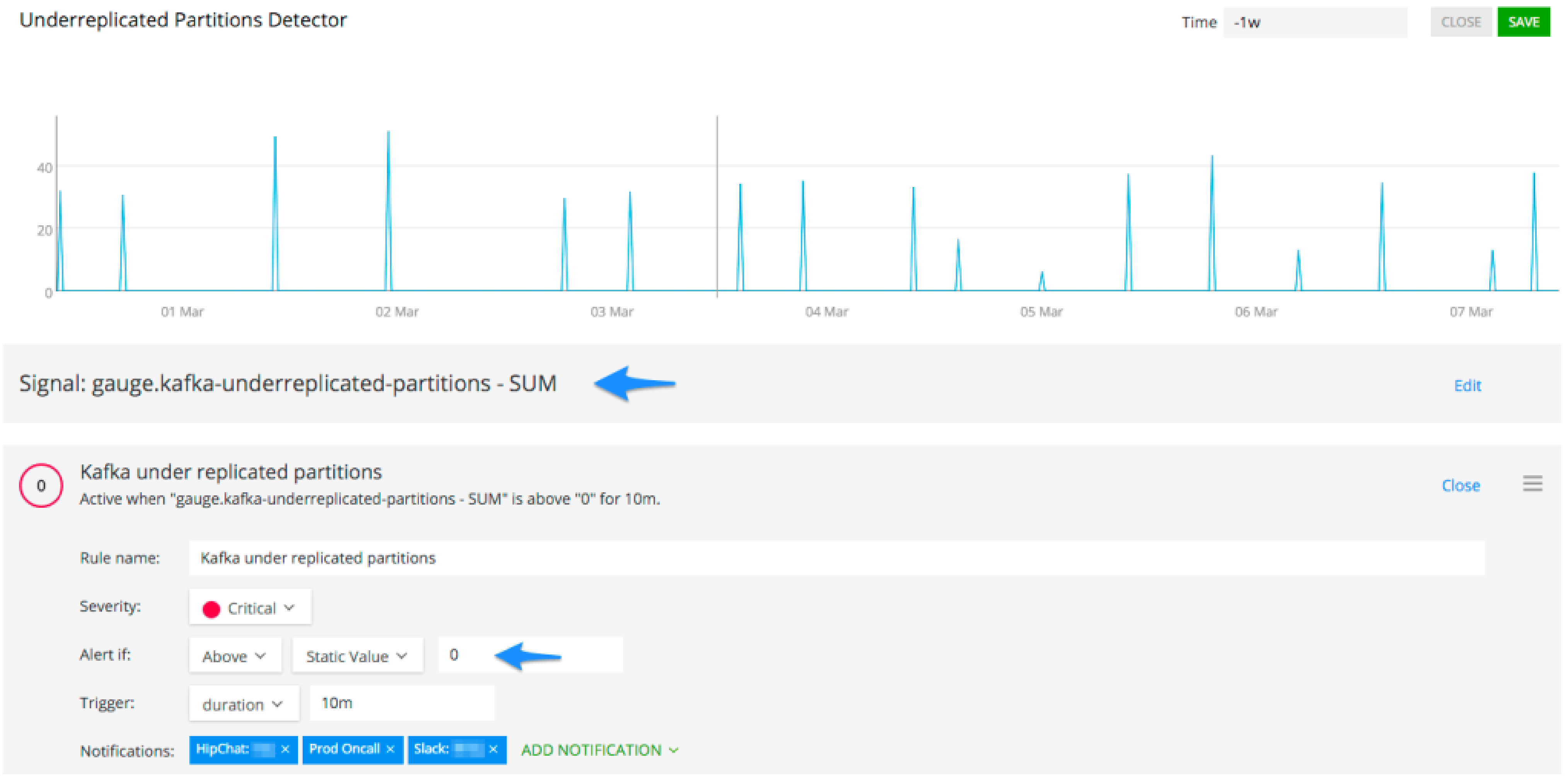
Here's another example of an operational issue we experience early on:

- In one of our consumers, the bytes in was high but it was barely getting any messages. It was listening to an empty topic.
- Kafka has a feature, for efficiency, where a consumer sends a request saying "give me messages on this topic" and Kafka will park your request until there is at least a certain amount of data (configurable) to reply back with, which is good batching
- But if your log was completely empty, it would immediately reply with an empty message which isn't actually empty–we found a bug in Kafka
- So continuously having this request-response caused the network in for the consumer to be high–because even an empty response can be quite a bit of data on the wire (depending on the number of partitions involved)
- We filed the **bug report (https://issues.apache.org/jira/browse/KAFKA-3159)** and it got fixed fast!

Alerting: Focusing On Leading Indicators

From our experience over the last two years, we've found that *it's most useful to notify on alerts for the two leading indicators: Log Flush Latency (95P) and Under Replicated Partitions*. And investigation usually leads to something at the broker level. We've never really hit a cluster-level issue, which is a testament to how well Kafka's been designed.

Any under replicated partitions at all constitute a bad thing. So for this we use a simple greater-than-zero threshold against the metric exposed from Kafka.



Underreplicated Partitions Detector

Signal: gauge.kafka-underreplicated-partitions - SUM

0 Kafka under replicated partitions

Active when "gauge.kafka-underreplicated-partitions - SUM" is above "0" for 10m.

Rule name: Kafka under replicated partitions

Severity: Critical

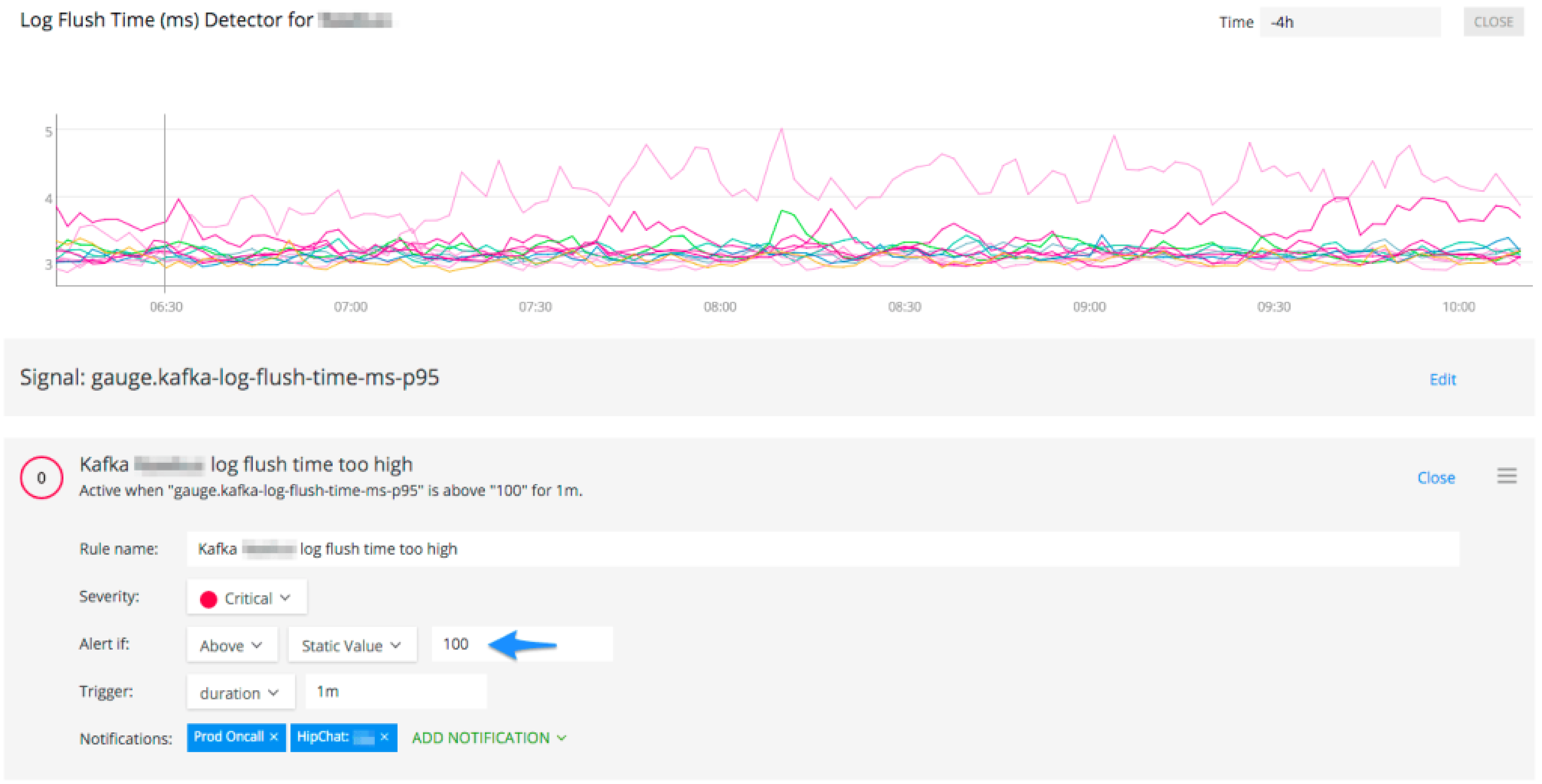
Alert if: Above Static Value 0

Trigger: duration 10m

Notifications: HipChat: [redacted] Prod Oncall Slack: [redacted]

Log flush latency is a little more complicated. Because some topics are more or less latency sensitive, we set different alert conditions on a per topic basis. Each broker's metrics have metadata that we apply (as key value pairs of property:value) to identify the topics impacted.

For example: raw customer data being ingested is highly latency sensitive, so it gets a 100ms threshold.



Log Flush Time (ms) Detector for [redacted]

Time: -4h

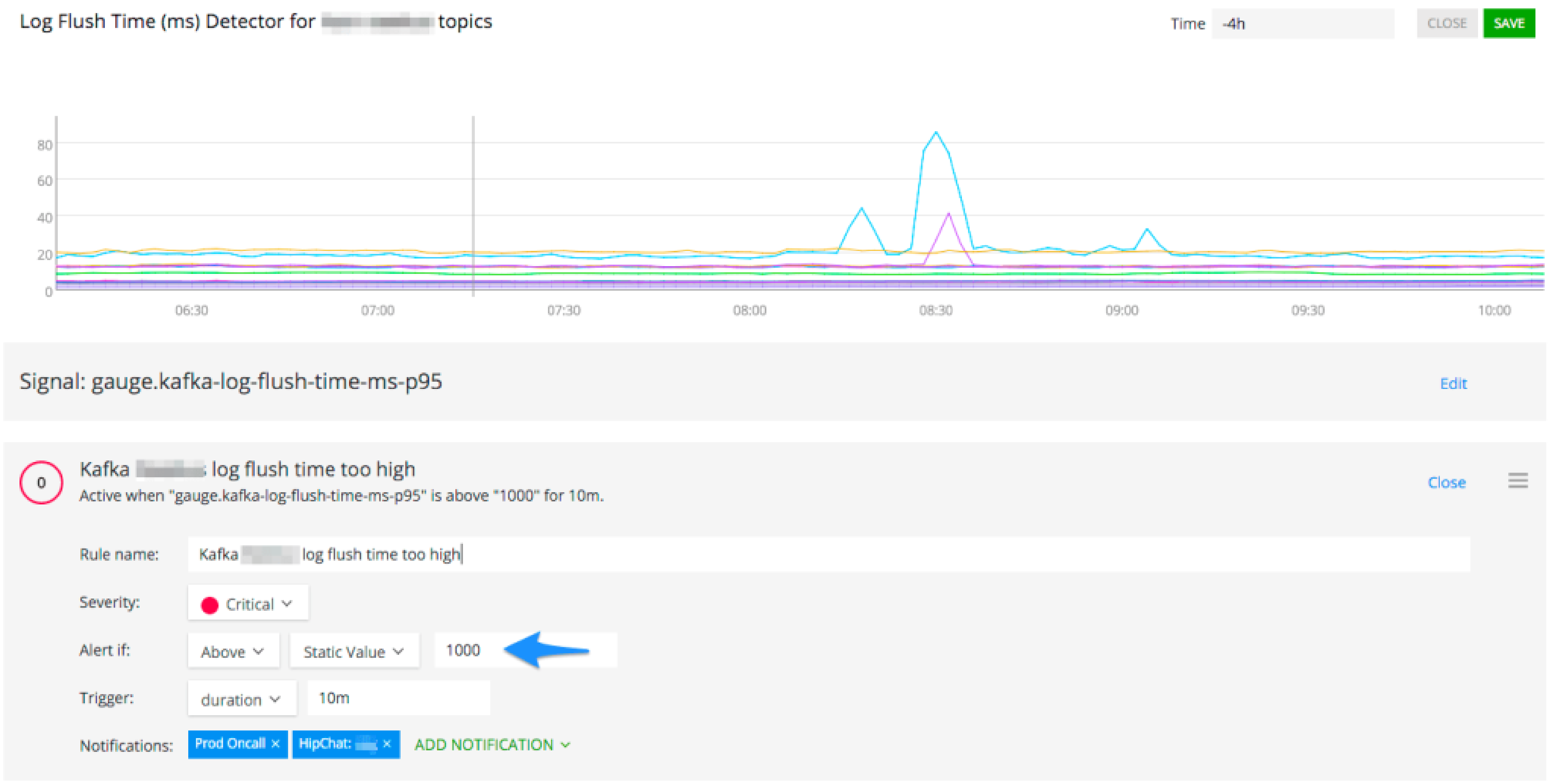
Signal: gauge.kafka-log-flush-time-ms-p95

0 Kafka [redacted] log flush time too high

Active when "gauge.kafka-log-flush-time-ms-p95" is above "100" for 1m.

Rule name:	Kafka [redacted] log flush time too high
Severity:	● Critical
Alert if:	Above · Static Value · 100
Trigger:	duration · 1m
Notifications:	Prod Oncall × · HipChat: [redacted] ×

But email can wait plenty of time, so the threshold is orders of magnitude higher.



Log Flush Time (ms) Detector for [redacted] topics

Time -4h

Signal: gauge.kafka-log-flush-time-ms-p95

Kafka [redacted] log flush time too high

Active when "gauge.kafka-log-flush-time-ms-p95" is above "1000" for 10m.

Rule name:	Kafka [redacted] log flush time too high
Severity:	● Critical
Alert if:	Above / Static Value / 1000
Trigger:	duration / 10m
Notifications:	Prod Oncall × / HipChat: [redacted] ×

Finally, we also alert on having less active controllers than expected, since this is a clear signal that we have a big problem.

Scaling and Capacity

Scaling Kafka is *involved*.

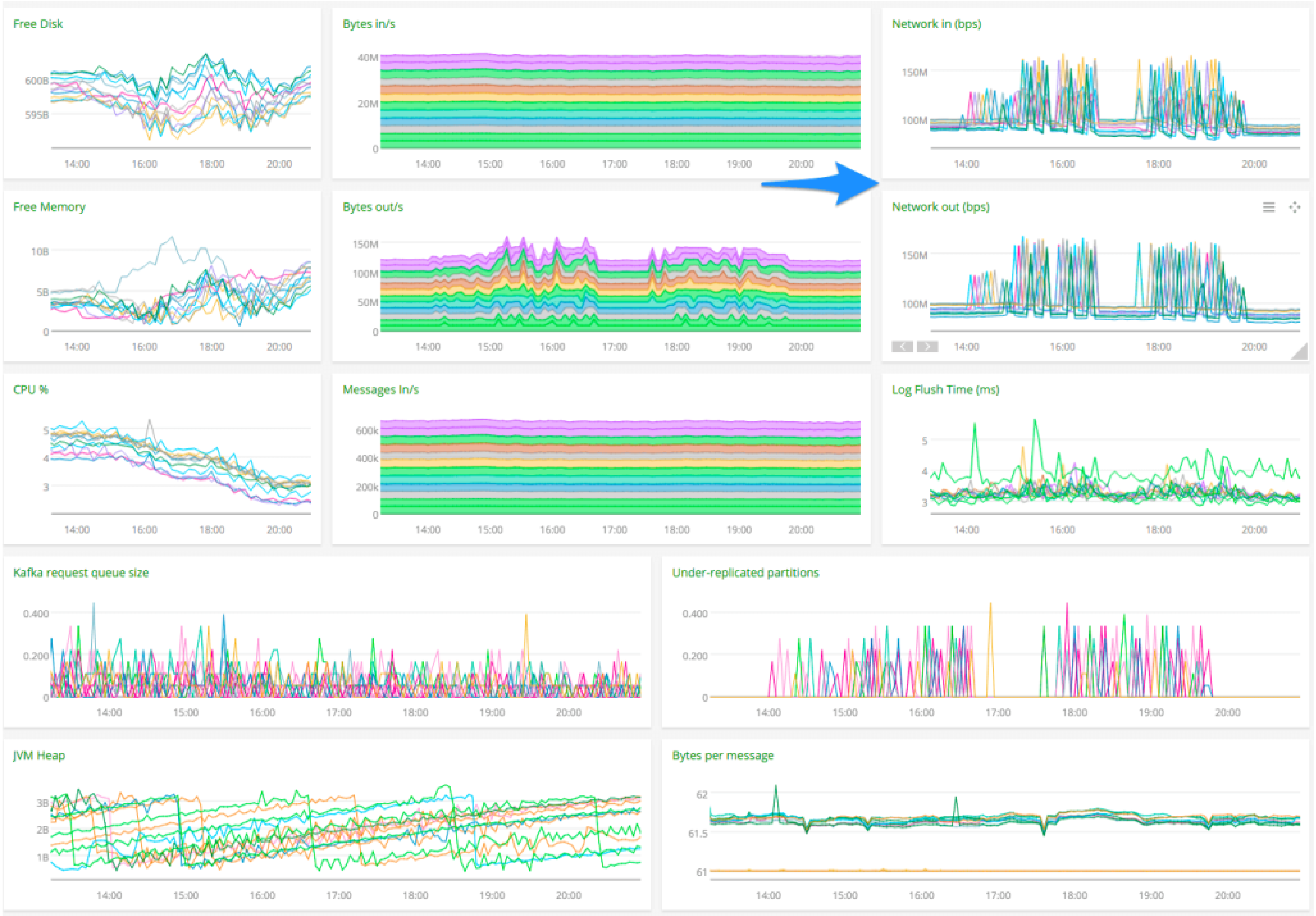
The adding capacity part is easy. But re-balancing topics/partitions across brokers can be quite hard. For smaller or simpler setups, Kafka can generate an assignment plan for you that provides even distribution across brokers. Which is fine if your brokers are homogenous and co-located. *This does not work well if your brokers are heterogenous or spread across data centers.* So we manually manage the process. Fortunately, Kafka takes care of the actual movement of data, given the partition to broker assignments. And with the expected addition of rack/region awareness, Kafka will soon allow for this kind of spreading of replicas across racks and regions.

This is where the pain comes in. Say you have a lot of traffic on one topic and are adding capacity for it. The topic partitions have to get spread across the new brokers. Although Kafka currently can do quota-based rate limiting for producing and consuming, that's not a applicable to partition movement.Kafka doesn't have a concept of rate limiting during partition movement. If we try to migrate many partitions, each with a lot of data, it can easily saturate our network. So trying to go as fast as possible can cause migrations to take a very long time and increase the risk of message loss.

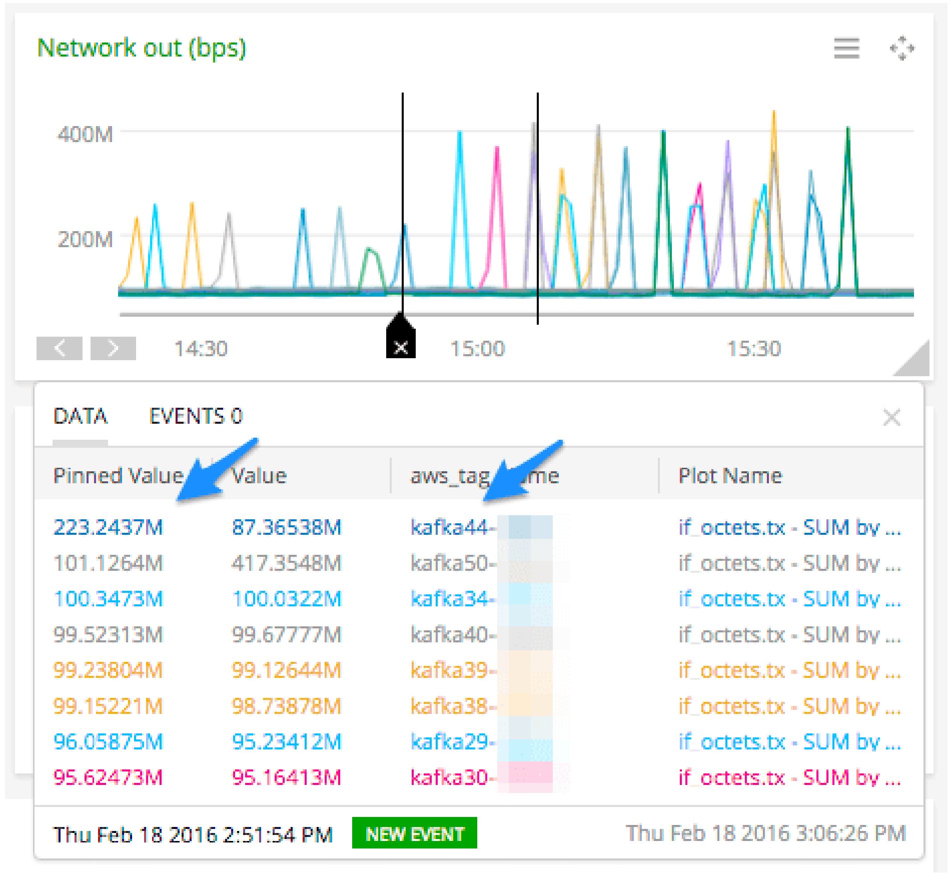
This issue will be obviated soon, as we're expecting Kafka's built-in rate-limiting capability to be extended to cover partition data balancing. In the meantime, to reduce migration time and the risks, we end up moving one partition at a time, watching the bytes in/out on the source and target brokers, as well as message loss. We use those metrics to control the pace of rebalancing to minimize message loss and resource starvation, thus minimizing service impact.

Here's the dashboard we observe, with the network in/out in the charts on the top right.

SignalFx **(https://www.signalfx.com)**


Free Disk
600B
595B
14:00 16:00 18:00 20:00
Bytes in/s
40M
20M
0
14:00 15:00 16:00 17:00 18:00 19:00 20:00
Network in (bps)
150M
100M
14:00 16:00 18:00 20:00
Free Memory
10B
5B
0
Bytes out/s
150M
100M
50M
0
Network out (bps)
150M
100M
CPU %
5
4
3
Messages In/s
600k
400k
200k
0
Log Flush Time (ms)
5
4
3
Kafka request queue size
0.400
0.200
0
Under-replicated partitions
0.400
0.200
0
JVM Heap
3B
2B
1B
Bytes per message
62
61.5
61

We'll zoom in on a particular time to see the parallel network in / out activity. Here you'll see that network out on kafka44 is about 220 million bits per second and network in on kafka29 is about 220 million bits per second. You'll also note that we wait for that network activity to go down to baseline before starting the next migration.



Pinned Value	Value	aws_tag_name	Plot Name
223.2437M	87.36538M	kafka44-	if_octets.tx - SUM by ...
101.1264M	417.3548M	kafka50-	if_octets.tx - SUM by ...
100.3473M	100.0322M	kafka34-	if_octets.tx - SUM by ...
99.52313M	99.67777M	kafka40-	if_octets.tx - SUM by ...
99.23804M	99.12644M	kafka39-	if_octets.tx - SUM by ...
99.15221M	98.73878M	kafka38-	if_octets.tx - SUM by ...
96.05875M	95.23412M	kafka29-	if_octets.tx - SUM by ...
95.62473M	95.16413M	kafka30-	if_octets.tx - SUM by ...

Thu Feb 18 2016 2:51:54 PM Thu Feb 18 2016 3:06:26 PM

For now, the process is manual. But we're looking forward to completely automating it in the future, like we've begun to do with Elasticsearch. Our approaches to both follow the same general pattern:

- Baseline the use case to understand performance trade-offs (e.g., learn that small changes to Kafka log flush latency have large impacts on end-to-end service latency)

- Find the unit of data that can be operated on without causing performance issues
- Make sure that it's evenly distributed (or accessible as if it were), even if not random
- Break up large or long running processes that run across those units into smaller components to prevent overload, bottlenecks, and resource starvation so app level performance is not impacted by service level changes (like scaling and rebalancing)

Where scaling is hard, dealing with capacity is quite easy. *Kafka is quite predictably bound by memory and network, not CPU.*

Some heuristics we've learned:

- Priority is on memory, followed by good disk I/O (use SSDs), and lastly CPU
- We're on AWS and it just so happens that AMIs with large memory capacity and good disk I/O characteristics have so much CPU that we never even task it If there was an option for a lower CPU, high memory and high disk I/O image, we'd take it.
- Kafka uses the page cache heavily: you're always writing and consuming from the end of the log, if consumers are keeping up with producers (the normal case)—so the more memory you give it, the more data can stay in page cache. More memory = more cache = less disk access.

Conclusion

Kafka is awesome. Speed and accuracy, in every way, are critical to SignalFx and the main things our customers depend on. The throughput performance of Kafka is far superior to all the other messaging or data pipeline platforms we've tested for our use case. Combined with the persistence features, this makes Kafka a core part of our infrastructure.

SignalFx is excited to have joined the Confluent Partner Program and looking forward to providing Kafka users coordinated solutions that meet the needs of the most demanding environments. As Confluent continues to provide increased visibility into the inner workings of Kafka and greater capabilities through the Confluent Platform, SignalFx will continue to harness that data for deeper monitoring and alerting.

This is a great community to be a part of and we hope everyone who runs Kafka finds this useful. **Check out our on-demand webinar about how we run Kafka at more than 70 billion messages per day. (https://www.signalfx.com/monitoring-kafka-performance-capacity/?utm_source=blog)**

Join our live weekly demo on cloud monitoring » (http://info.signalfx.com/live-demo/?ori_ref=/blog/how-we-monitor-and-run-kafka-at-scale/)