Logging @ Klaviyo: How we log 10 billion events a day!

Author: Alex Frier

Claps: 36

Date: Apr 17

If there is one universal pain point that unites all software engineers, it would probably be diagnosing and debugging problems with the software they write and the infrastructure they deploy. Logging has always been a go-to solution for engineers to debug their code and understand how itâ€™s behaving in production. And this doesnâ€™t apply just for software *you* are writing, but every application running on your servers, the operating system (or *systems* in a world with containerization) you run your software on, and the software vendors you integrate with. Every technology company that scales from a handful of people with a small codebase running on a few servers to a large codebase running on thousands of servers â€” with each server likely running hundreds of processes â€” one day faces the question: How to store, query and analyze the logs generated by these services at scale?

Log data is extremely valuable, not only for software engineers to debug issues, but also for security engineers to detect and analyze anomalous and suspicious events, and for data scientists to identify trends. But how do you store petabytes of log data in a cost-efficient manner, while making it easy for your engineers to access and query the data?

Iâ€™m a member of the team responsible for our logging infrastructure. Here at Klaviyo, thatâ€™s our Security Engineering team, one of four subteams within our SRE (Site Reliability Engineering) group.

Soâ€¦what kind of volume are we talking about?

Great question. Our logging pipeline processes approximately ***9.5 billion*** log events a day.

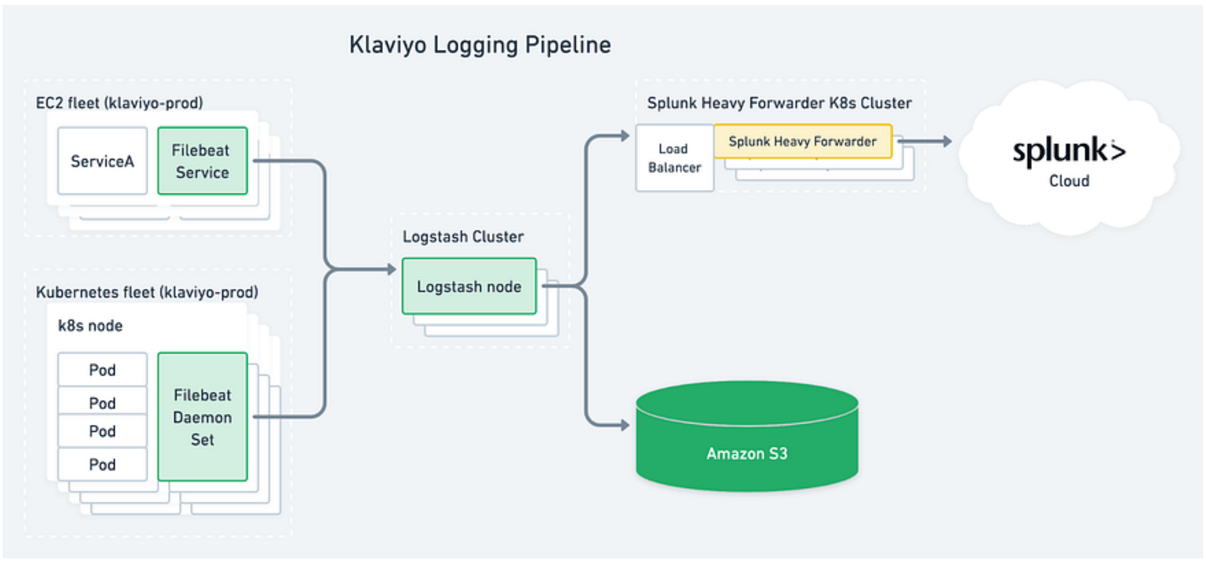
Wow, thatâ€™s a lot. Where do all of these log events come from?

All over the place. On an average day, we have 8,000 servers running in our production AWS cloud environment. Thatâ€™s a mix of managed Kubernetes clusters and EC2 auto-scaling groups. (During our busy season of peak traffic, Black Friday and Cyber Monday, the host count goes much higher.) A vast majority of the log volume originates from these compute resources and the many applications that run on them. Some of these logs are Linux operating system events (syslog, auth logs, etc.), while others are generated by events from the Klaviyo web application and other layers of that stack. We also ingest logs from various external vendors, such as Cloudflare, and other internal applications and services. Finally, we collect TCP logs from select points in our network.

Alright, so what does this logging pipeline look like?

To drastically oversimplify: Filebeat for log shipping, Logstash for log processing, Splunk for further parsing/storage/querying, and Amazon S3 for long-term archiving.

Hereâ€™s a high level view:



[Filebeat](#) runs persistently on a server, watches a set of log files, and ships them to your desired location. We run Filebeat as an agent on every EC2 instance across our production compute resources, and as a DaemonSet on every Kubernetes cluster (meaning there is a Filebeat pod on every node in the cluster). Filebeat harvests logs on those devices by following the log files we specify in its configuration. All of this orchestration is done via Puppet (and soon Argo CD for Kubernetes) â€” Puppet is used to install, configure and upgrade Filebeat, as well as to keep the service running.

Hereâ€™s a snippet from our Filebeat configuration on Kubernetes, where we instruct Filebeat to harvest log messages from all files ending with `.log` located in the `/var/log/containers/` directory.

```
filebeat.inputs:
- type: log
symlinks: true
paths:
- /var/log/containers/*.log
processors:
- add_kubernetes_metadata:
host: ${FILEBEAT_HOST}
in_cluster: true
default_matchers.enabled: false
matchers:
- logs_path:
logs_path: /var/log/containers/
```

[Logstash](#) is an application for log filtering, parsing and transformation. Our Logstash cluster has about 100 instances on a normal day (and more during our peak Black Friday / Cyber Monday season). We use Logstash to standardize incoming log events by mapping data to certain fields and transforming the outgoing log message into a format expected by the final destination. Logstash also allows us to clone a single log message where needed, so it can be transformed into multiple desired formats and sent to multiple destinations.

Below is a sample of a Logstash configuration file for our log output plugin to Amazon S3. This sample is actually the Embedded Ruby (.erb) template we built so Puppet can dynamically generate an output for every type of log, since logs from different sources go to different S3 buckets.

```
output {
<% @logstash['outputs'].each do |log_type, opts| -%>
  if [@metadata][log_type] == "<%= log_type %>" <% unless opts.dig('skipms
<% if opts.dig('s3', 'bucket') -%>
  s3 {
    bucket => "<%= opts.dig('s3', 'bucket') %>"
    prefix => "%{[@metadata][s3][prefix]}"
    time_file => <%= opts.dig('s3', 'time_file') or 5 %>
    codec => "<%= opts.dig('s3', 'codec') or 'json_lines' %>"
    temporary_directory => "/tmp/logstash-<%= log_type %>"
    canned_acl => "<%= opts.dig('s3', 'canned_acl') or 'private' %>"
    id => "<%= log_type %>_output_S3Output"
  }
<% end -%>
  â€¦
}
```

A ***Splunk Heavy Forwarder*** is a local Splunk instance. We have a relatively small cluster of these forwarders to allow us to do specific log event parsing and transformation before they are sent to Splunk Cloud. This cluster serves two main purposes: assign each incoming log event an index that can be queried in Splunk; and remove potentially sensitive data from incoming log events. We put a load balancer in front of these forwarders to allow for equal distribution of incoming events across the cluster.

We can also use this layer of the pipeline to configure settings exclusive to Splunk indexing. Hereâ€™s an example of a stanza from props.conf for our Django logs:

```
[django:app]
AUTO_KV_JSON = false
DATETIME_CONFIG =
LINE_BREAKER = ([\r\n]+)
MAX_TIMESTAMP_LOOKAHEAD = 44
NO_BINARY_CHECK = true
SHOULD_LINEMERGE = false
TIME_FORMAT = %Y-%m-%d %H:%M:%S,%3N
category = Application
pulldown_type = true
REPORT-JSONapp = JSON
BREAK_ONLY_BEFORE_DATE = true
```

The final destinations are then ***Splunk Cloud*** and ***Amazon S3***. Splunk Cloud provides an accessible and user-friendly interface for our employees to query, analyze and alert on the vast wealth of log data we store. Amazon S3 is used for more durable, long-term storage of events and can be queried via Amazon Athena.

Was the logging pipeline infrastructure always like this?

No, it has gone through many changes and iterations just like the rest of Klaviyoâ€™s infrastructure.

Here are two examples of lessons we learned that led to the current configuration.

In the past, we had multiple Logstash clusters separated by the type of incoming logs. We had one Logstash cluster for our application and web server logs, one for Kubernetes logs, and one for all other logs â€" and this was just for our production environment.

The thought behind this was that it would allow us to logically isolate log filtering, parsing and shipping (and the supporting configuration code) into separate EC2 clusters. It would also reduce Logstash startup time via a reduction in the number of log events Logstash needed to synchronize in order to determine the point where it left off prior to shut down.

In practice, while all of the above was true, maintenance became cumbersome. This is because we ended up needing to heavily customize Filebeat configuration on all of our hosts in order to ensure different types of logs were being sent to the right place. Additionally, it made scaling the Logstash clusters up and down more tedious, since we needed to individually determine scaling needs for each cluster.

This is what led to the eventual merging of all of these different clusters into one cluster (minus our test cluster in our development environment).

Another lesson and change was related to load balancing. We initially put load balancers in front of our Logstash hosts. Thatâ€™s a standard pattern here, and seemed like the right way to distribute incoming log events among Logstash hosts. However, that turned out to be overkill and introduced another source of potential failure. The reason is that our Filebeat configuration already specified a round-robin system of log event shipping. Filebeat randomly selects a number of Logstash hosts from a predefined list and tries to initiate a connection with each host until one accepts the connection. Filebeat therefore already takes care of load balancing. It may not be the most efficient mechanism, but it is resilient. We decided to remove the Logstash load balancers. (We do use standard AWS auto-scaling behavior to replace unhealthy instances and, since there is no load balancer, we configure for each instance to assume one of the â€œvanityâ€� DNS names reserved for our Logstash hosts.)

With the amount of data being processed by this pipeline on a daily basis, what kind of challenges have you run into?

Iâ€™m glad you asked! We ran into a particularly nasty bug that haunted us for almost a year. The TL;DR is â€” after a year of reports from engineers across our organization regarding logs missing from Splunk, with seemingly no common trend between the missing logs â€” we discovered a bug where all logs originating from our Kubernetes clusters which contained the characters D, E, B, U or G, were getting dropped by Logstash! Read on to learn about how we finally managed to track this down.

When we first onboarded Kubernetes logs to Splunk Cloud, our total data ingestion began to exceed a terabyte daily. At the time, this was over our Splunk licenseâ€™s daily capacity, so we needed to trim down the log volume. As we dug in, we saw a huge number of DEBUG and INFO log events. These types of log messages are generally more informational, and not indicative of errors or exceptions occurring in our applications. In order to ensure we were not going over our daily log volume capacity, we configured Logstash to drop DEBUG and INFO log events.

Interestingly enough, for about a year after this change was made, we would occasionally receive reports of Kubernetes log messages missing from Splunk â€” these were for log messages that were not of type DEBUG or INFO. Engineers reporting the missing log messages would confirm that these messages were indeed being written into log files on disk, but then observe that somewhere between the original source machine and Splunk Cloud, the message would disappear. Another interesting quirk was that this problem was only ever reported with logs that originated from our Kubernetes clusters. We dug through many Logstash configuration files with no luck at first.

There seemed to be no common pattern or trend in the missing log messages, other than the fact that they all originated from our Kubernetes clusters. One day, we decided to go through our Kubernetes Logstash configuration, thinking that it must have something to do with this configuration. This turned out to be right, as we ended up circling back to the original if statement we added for dropping DEBUG level logs. We discovered the bug in this code sample below:

```
if [message] =~ "[DEBUG]" {
  drop { â€¦ }
}
```

At first glance, the snippet looks straightforward â€” if the log message contains the string â€œ`[DEBUG]`â€�, then we drop it. However, once we looked closer at the operator used to perform this check (=~), and compared that to Logstash documentation, we realized something â€” this is the operator for matching regular expressions! We had mistakenly thought it was the operator for checking that a message contains a substring. (Yesâ€¦ I knowâ€¦ using =~ for matching regex is commonâ€¦ but humans make mistakes!)

In regular expressions, encapsulating characters in square brackets means match IF any of the characters in the brackets are found. This is known as a â€œcharacter classâ€� or a â€œcharacter set.â€� So â€œ`[DEBUG]`â€� was being interpreted by Logstash not as a literal string, but as a regular expression. Any incoming log message that contained any of the letters D, E, B, U or G was getting dropped by Logstash! We started to take a look at some of the reports of missing log messages we had received over the last year, and noticed that all of them had one of these characters somewhere in the log message â€” a capital â€œEâ€� in a log message for an exception, etc.

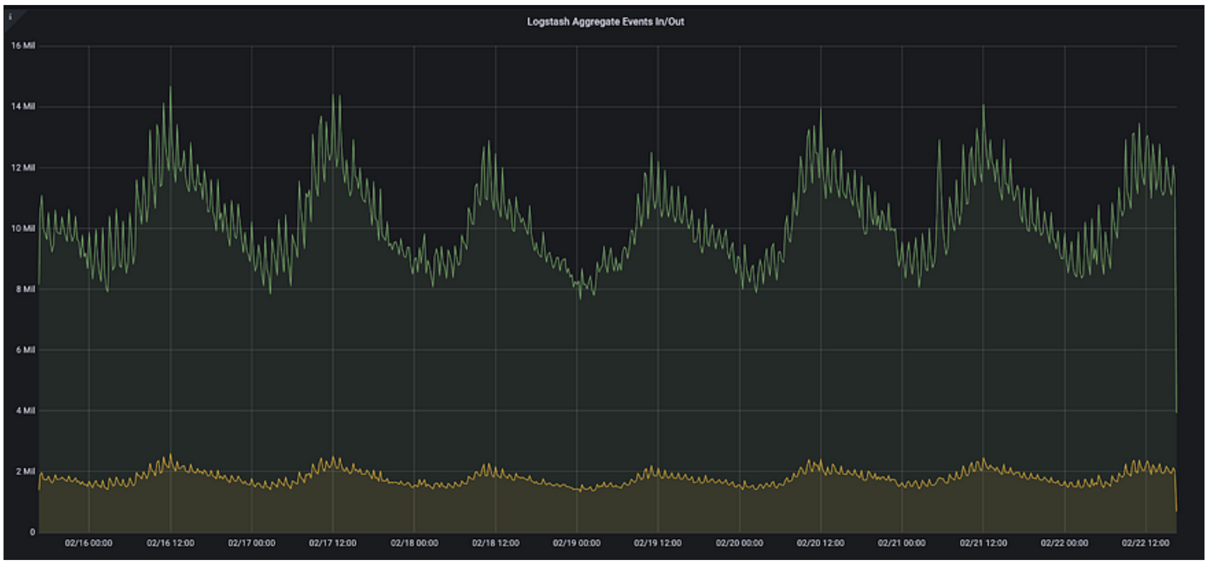
The fix was easy, escape the square brackets:

```
if [message] =~ "\[DEBUG\]" {
  drop { â€¦ }
}
```

Part of the reason why we missed what may seem like an obvious bug for so long is that Kubernetes was not widely adopted here at the time, and so the problem didnâ€™t become visible or urgent until more of our services migrated to Kubernetes.

Is log volume predictable? How do you know that Logstash is scaled properly?

Yes, it generally is. Unsurprisingly, log volume increases when activity in our system increases. When email and SMS sends are going out at high volume, log volume also goes up. We generally see the highest log volume around noon Eastern Time.



As mentioned above, we also have certain times of the year where traffic to Klaviyo is significantly higher than normal. Usually, this traffic begins the week before Black Friday (when many businesses begin crafting and sending out their holiday marketing emails/SMS), and concludes on Christmas. In particular, the weekend of Black Friday and Cyber Monday (BFCM) is of most concern.

To prepare for BFCM, we calculate the number of hosts needed to handle the estimated increase in log events. (Horizontal scaling of our logging pipeline is much easier and safer than vertical scaling â€" no need to test with different instance types or worry about restarting all of the nodes in our auto-scaling group). For example, during BFCM 2022, we scaled to 300 Logstash hosts. Scaling up to this level allowed us to handle the spike in traffic. The cost was worth it when you weigh it against the risk of losing logging information or causing problems for our application hosts (see below) at such a critical time.

What challenges do you struggle with today?

As always, there is so much that could be innovated and improved on, but only so much time in a day or even in a quarter. Right now, if I had to select our biggest challenge, itâ€™s determining what to do in the event of a disaster where our Logstash cluster is down, and backpressure is applied to Filebeat. The two potential consequences here are cascading failures across our production infrastructure, and/or severe loss of logging data.

Log messages form an unending stream. Log messages are being generated every millisecond from dozens of processes, each running on thousands of hosts. Log messages are critical data â€" they tell a story of how our infrastructure and software is running, and of what events are occurring. Losing this data, even the most negligible amount, means a potential obstacle for software engineers and security engineers in debugging and investigating incidents. This means that the infrastructure supporting the logging pipeline is critical, and it is expected that it will always be up and running. So what would happen if it wasnâ€™t up and running? Or if a downstream step in the logging pipeline failed, for example, our Splunk heavy forwarders?

The answer is not great at the moment. If our Logstash infrastructure were to go down, Filebeat agents across our fleet would continue to attempt to round-robin between Logstash hosts, trying to find a running Logstash instance. Since Filebeat would fail to publish events, it would begin to consume memory for in-flight log events and file handles. Over time, if Filebeat resource utilization were not constrained, it could begin to exhaust memory on the host, or hog memory resources needed by other critical services. If this happened, the host could become unhealthy, and any applications being served on that machine could become overwhelmed or even go down. This could happen on a large scale if Logstash were down for too long and could cause significant damage.

The current alternative to storing log events in memory is setting a maximum number of active events in Filebeat. This would mean Filebeat would effectively stop harvesting logs until downstream destinations become available again. In this scenario, there is a possibility of experiencing log data loss. This is because log file rotation is independent of log event shipping. Log rotation is configured on an individual service-by-service basis â€" for example, we may keep up to 7 syslog log files. When it is time to rotate the main syslog log file, the oldest of these 7 files gets deleted. If this occurred while Filebeat was experiencing this backpressure, there is a chance that Filebeat would have met its maximum queue size before harvesting all log lines from the oldest of these 7 files.

Luckily, the SRE Security Engineering team has monitoring and alerting in place to prevent a scenario like this from running out of control. As soon as a certain threshold of active Filebeat events is crossed, a team member is paged. However, while it is unlikely, this does not entirely remove a scenario where we would need to make the tough choice to begin sacrificing log data in exchange for keeping Klaviyo up and running.

This is one of our main challenges at the moment. How do we mitigate a catastrophic event of our core applications facing downtime, while also ensuring no crucial log data is lost? It is a balancing act and it is not a problem that will be solved in a day or two, but this is one of the issues our team constantly thinks about. Some mitigations could include log spooling (both on and off disk are options) and/or increasing redundancy of our Logstash infrastructure. However, we have lots of priorities, and logging is only one of many areas our team is responsible for.

Closing

Logs are some of an organizationâ€™s most valuable data, especially when made easily queryable. Weâ€™re proud of our logging infrastructure. Itâ€™s been an invaluable and critical resource to our security and engineering teams. That being said, thereâ€™s still more to do to keep up with scale, make our tools even better, become more efficient, and mitigate potential nightmare scenarios.

If you found this topic interesting and could imagine yourself tackling these types of problems, our team is hiring a Senior Site Reliability Engineer. Our team is growing and weâ€™d love to find talented people to help us upgrade and optimize our logging infrastructure!