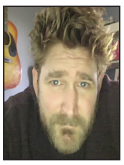
iVoyeur
Flow 3

DAVE JOSEPHSEN



Dave Josephsen is a book author, code developer, and monitoring expert who works for Sparkpost. His continuing mission: to help engineers worldwide close the feedback loop.
dave-usenix@skeptech.org

If you take any sort of guided tour of Paris, you are likely to hear references to "The Great Flood of 1910," wherein the Seine rose to a depth of eight meters above its normal height, buried the city in water, and shut down critical infrastructure like freshwater and heating-oil delivery for a month.

Rivers have backed up and flooded cities since time out of mind, but this flood makes for particularly great data-engineering metaphor fodder because the water never managed to overflow the tops of the quay walls lining the river itself. In other words, primary queue cardinality was within threshold.

Instead, the city was flooded from below by way of the recently enlarged and fortified sewer system that ran from every direction into the Seine. I suppose you could say that the hotpath bypassed the queue. Ironically, the infrastructure most prized by city planners, like train stations and hospitals, which had the best-engineered sewer access, were hit the worst. Their basement grates spewed water like the geysers of Yellowstone, rapidly flooding and spilling into the streets until the streets themselves became waterways.

In some areas of the city, firefighters used boats to rescue stranded people from second-story windows, as engineers constructed a city-wide series of wooden catwalks to enable residents to reach shelters and sources of food and fresh water.

Here's the thing: if you've never read anything about the history of Paris, the city was supposedly an untenable *mess*, until Napoleon III put it into the hands of a gentleman named Georges-Eugène Haussmann. "Baron Haussmann" would spend 20 years becoming the most unpopular guy in France as he demolished the medieval firetrap the city had been in order to singlehandedly re-architect it into the city we more or less recognize as Paris today.

The "grand rearchitecture" of the city included a herculean refactoring of the dense labyrinth of pipes, sewers, and tunnels beneath the streets into the most modern and robust sewer system in the world. The system provided the city's freshwater supply, steam heat, and oil pipes to power the streetlights, as it simultaneously swept away rainwater and waste. The sewers were such a source of pride that bureaucrats of the time used their own pet euphemisms to make them sound less like sewers and more like re-election.

Haussmann himself compared them to bodily organs. "The underground galleries," he said, "are an organ of the great city, functioning like an organ of the human body, without seeing the light of day; clean and fresh water, light and heat circulate like the various fluids whose movement and maintenance serves the life of the body; the secretions are taken away mysteriously and don't disturb the good functioning of the city and without spoiling its beautiful exterior."

It's fortunate Haussmann died before his miraculous "underground galleries" buried the city chest-deep in human waste and river water. Had he been there to see it, I'm sure it would have been the facepalm heard around the world.

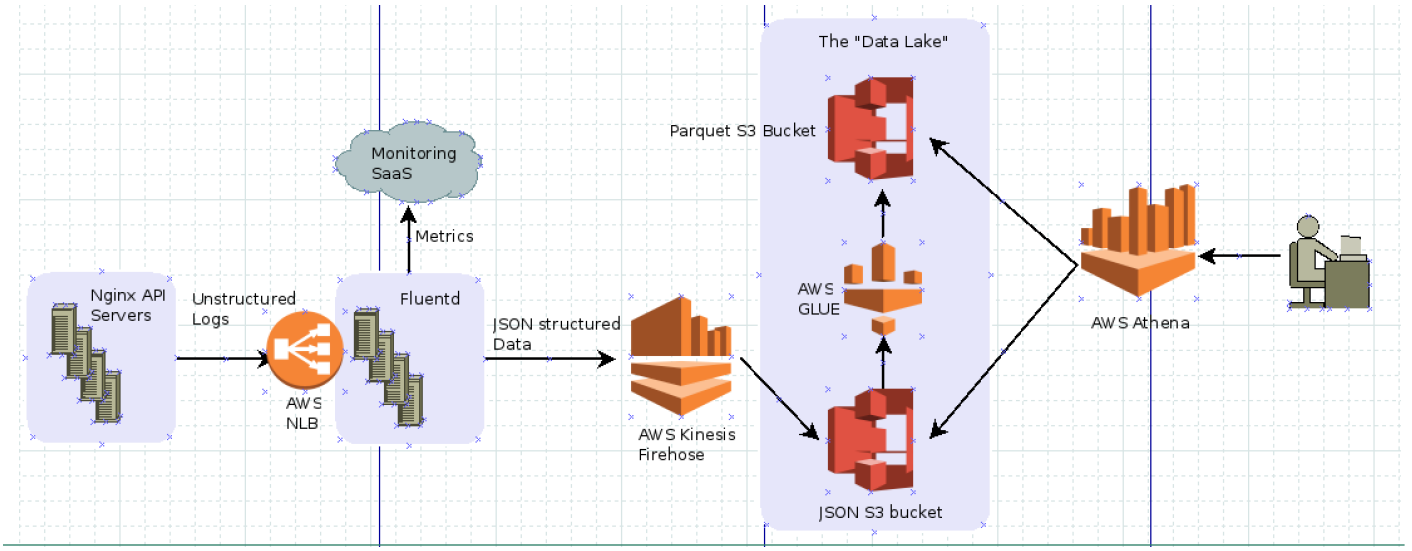


Figure 1: Sparkpost's "Internal Event Hose" data pipeline

I suspect that anyone who has seriously worked with data pipelines or distributed systems can probably relate; an overabundance of input can have extreme and unforeseen effects on asynchronous processing systems.

The Flow, Part Three

This is the third article in my series about our API-query data pipeline, so you, dear reader, could certainly be forgiven not knowing just what the heck I'm going on about. Let's pause, therefore, for a moment of reflection. In Figure 1 you can see the pipeline in its entirety.

In my last article, we spoke about the first data transformation, which takes place inside Fluentd, to change raw log data into structured JSON. We learned about how tags and message routing works inside Fluentd and about Fluentd's buffered output plugins. I also mentioned that we were using the Prometheus plugin to extract some metrics from Fluentd and shared some cardinality graphs from our production monitoring system, Circonus.

Merely enabling Prometheus in your `tdagent.conf`, along with its `outputmonitor` plugin, gives you all the visibility you need to detect backups inside Fluentd of the sort tour guides in Paris are *still* talking about a century later.

```
<source>
  @type prometheus
</source>
<source>
  @type prometheus_output_monitor
</source>
```

Upon restarting `td-agent` (the Fluentd demon), a `wget http://localhost:24231/metrics` will yield myriad stats on every registered output plugin, like these two counters of messages emitted per output plugin (`sns` and `firehose` for us):

```
fluentd_output_status_emit_count{plugin_id="object:3f86dc5b80cc",type="amazon_sns"} 570277.0
fluentd_output_status_emit_count{plugin_id="object:3f86d9833444",type="kinesis_firehose"} 10109263509
```

Fluentd also has a filter type Prometheus plugin, which you can use in your routing configuration to extract metrics directly from the data as it passes through. We use this to break down the cardinality of the various types of API calls that are occurring within our Nginx data. Here's the configuration blurb:

```
<filter firehose_parsed.**>
  @type prometheus
  <metric>
    name outgoing_msg
    type counter
    desc Outgoing messages
  </metric>
  <labels>
    type ${type}
  </labels>
</filter>
```

This filter catches all messages tagged with "firehose_parsed" and increments a counter metric named "outgoing_msg" that—crucially—is labeled with the *value* of the message's `type` attribute. In other words, as each message is routed through this filter, Fluentd literally uses the value of `msg.type` to create the

Prometheus metric label. Hence, when we `wget` the reporting socket, we get output metrics that break down the cardinality of each type of API call our customers are currently making:

```
...
outgoing_msg{type="get_sending-domains"} 31190473.0
outgoing_msg{type="get_subaccounts"} 33089429.0
outgoing_msg{type="get_webhooks"} 527765630.0
outgoing_msg{type="auth_request"} 58139133.0
outgoing_msg{type="get_users"} 144456173.0
outgoing_msg{type="4xx_error"} 193923362.0
...
```

In a proper Prometheus shop, we'd be using the Prometheus server to slurp up all of these metrics and report on them, but for better or worse, our monitoring solution of choice lies in another direction, so I wrote a small shell script that performs the polling and reformatting. Omitting the error handling, it's really just two lines...

```
INPUT=$(curl -k -ss -m "${TIMEOUT}" "${URL}")

echo "${INPUT}" | grep -v '^#' | sed -e 's/{.*="/`/' -e 's/"} //'
-e 's/ //g' -e 's/^fluentd_//' -e 's/`\([^`]\+\)$/ n \1/'
```

If you squint at it hard enough you'll see it transforms the output into *backtick* separated lines of the style: `outgoing_msg`get _sending-domains`31190473.0. I know. Backtick separation. Don't get me started.

When we first architected this data pipeline we carefully read up on the various AWS streaming event services, compared their limits and tradeoffs against our workload, and decided that SNS was the best fit for us. We installed the most popular version of the SNS Fluentd plugin, gave it our configuration particulars, and watched everything collapse and fail in a Parisian-esque epic flood of traffic.

We eventually discovered two overlapping problems. The first, which I mentioned in my last article, was the SNS Fluentd plugin we found didn't support buffered output, meaning, among other bad things, that it didn't support threading and completely blocked the entire Fluentd process as it tried to flush 11,000 messages to SNS every second.

The second problem was that the SNS service itself doesn't have a bulk-send endpoint, so every message emitted equates to a single HTTP connection. It's surprisingly easy for little details like this to be obscured by frameworks and plugins and abstraction. Engineers who know AWS very well are fond of saying things like *there are no limits to SNS*, and asking around, I heard myriad utopian tales of shops pushing hundreds of thousands of 140-byte messages per second into SNS without breaking a sweat.

Well, it turns out, the real-world limit on SNS is the number of HTTP connections you can reliably make per second from your sending instance's ENI. I'm not really sure what that number is (it no doubt varies by instance type), but I'm here to tell you, for us, it was smaller than 11,000 divided by three instances.

Rather than attempting to scale up or out, we took a look at AWS Kinesis Firehose, which has a bulk-send endpoint capable of ingesting batches of over 100 messages in a single HTTP call. This was a WAY more efficient and reliable means of feeding data into AWS. Bonus, the Fluentd Kinesis plugin is well supported, buffered, and supports threading.

We experimented with lambdas attached to our firehose to transform the JSON log data directly in to Parquet but eventually decided that we wanted a copy of the data in both JSON and Parquet, so we pointed the firehose directly at an S3 bucket. Kinesis automatically partitions this data up for us into minute-sized chunks, ready for Athena to parse through them.

To make the final hop into columnar data format, we rely on a combination of custom-written code, Apache Spark, and AWS Glue. Spark's PySpark (http://spark.apache.org/docs/latest/api/python/index.html) library makes it simple to `sqlContext.read.json()` our JSON data from S3 into a Spark DataFrame (https://spark.apache.org/docs/latest/sql-programming-guide.html), and from there `df.write.parquet()` it back out to a new S3 bucket in Parquet format. We use AWS Glue to schedule our PySpark code as an ETL job that runs hourly (five minutes after the hour, to give firehose a sufficient buffer of time).

I find it difficult to articulate the extent to which this data has enriched my life as an engineer, but I'll give you an example from last week, wherein someone noticed that we appeared to be bouncing an order of magnitude more email than normal, which everyone found...worrisome.

I first checked whether there was a pattern of increased bounces for our top-tier receivers. This sort of thing has happened in the past when Gmail, for example, implemented some new, aggressive, and ill-conceived filtering technology.

```
select dt, count_if(routing_domain='gmail.com') as google,
count_if(routing_domain='yahoo.com') as yahoo,
count_if(routing_domain='hotmail.com') as hotmail
from "glue-data-lake-usw2-prd".eventlog_parquet
WHERE bounce_class=21
AND dt >= '2018-09-01'
group by 1;
```

With this Athena query, I was able to get a day-by-day breakdown since September 1 of email we bounced to the top three providers and verify that we were *NOT* in fact bouncing more mail than normal. This query took three minutes to complete and scanned around 100 GB of data (Athena queries cost $5 per TB scanned).

What, then, could account for the increase in bounce traffic?

```
select count(dt),raw_reason
from "glue-data-lake-usw2-prd".eventlog_parquet
WHERE bounce_class=21 and dt between '2018-10-01' and
'2018-10-21'
group by raw_reason
order by count(td)
LIMIT 10;

select count(dt),raw_reason
from "glue-data-lake-usw2-prd".eventlog_parquet
WHERE bounce_class=21 and dt > '2018-10-21'
group by raw_reason
order by count(td)
LIMIT 10;
```

With these two queries I was able to enumerate the top 10 reasons that email bounced in the period before the change was noted, and then again in the period after the change was noted. I discovered that there was indeed a difference between these two lists. The first looked like:

```
454 4.4.4 [internal] no MX or A for domain
554 5.4.4 [internal] Domain Lookup Failed
"550-Requested action not taken: mailbox unavailable
550 invalid DNS MX or A/AAAA resource record"
451 Your domain is not configured to use this MX host.
```

While the second looked like:

```
454 4.4.4 [internal] no MX or A for domain
554 5.4.7 [internal] message timeout (exceeded max time, last
      transfail: 454 4.4.4 [internal] no MX or A for domain)
554 5.4.4 [internal] Domain Lookup Failed
554 5.4.7 [internal] exceeded max time without delivery
```

As you can see, some new, timeout-related error messages have overtaken the first and fourth most common error message in the logs. As it turns out, our engineering teams had implemented a new suite of error detection code and had miss-classified these timeout messages as bounce-class messages, which in turn caused a reporting error.

While this particular example turned out to be a false-alarm rather than a flood, I think it serves to illustrate how capable our new log data pipeline is at helping us deal with the deluge.

I think that pretty much wraps up my series on our Data Pipeline at Sparkpost, and along with it, my overspilling (sorry) of river-related metaphor. Until next time.