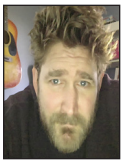
iVoyeur
Flow, Part II

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

The bells of Basilika Sankt Kastor clang—a nagging reminder behind me that I should be in Cochem right now, exploring castles like a proper tourist. But my imagination has been hijacked, and so I sit in Koblenz, having failed to switch trains when I realized—looking at the railway map—that this was the city of Deutsches Eck, where the Mosel empties into the Rhine.

The Rhine is the second longest river in Europe (behind the Danube), and yesterday, 100 miles north of here, I watched as a long, low jalopy-looking riverboat meandered up to its bank in Dusseldorf and launched, like a fanout-algorithm, a small flock of half-a-dozen bicycles—mother and children—toward the farmers market, their baskets full of empty shopping bags.

The wide flat deck of the boat was laden with the typical boat-crap-trappings that you would expect to see on the deck of a riverboat, but there were also things foreign to that environment, like a large wooden dining room table with seven chairs, an Iron Man Big Wheel, and lush green potted plants. Through the window of the wheelhouse I could see crayon art and action figures adorning every sill as if on the lookout for inclement weather.

It was love at first sight.

And so here I sit, watching the riverboats navigate the confluence of these two great rivers, most of them laden with cargo or tourists, but some—about one in twenty—serving as someone's home afloat, headed who knows where. I imagine them unhurriedly drifting from town to town, suffering the world to move around them until they come spilling out into the North Sea, and maybe then turning right to explore Amsterdam's maze of canals, or perhaps left, hugging the coastline as far as Le Havre and the mouth of the Seine. I can't help but wonder how the 4G reception is along the great rivers of Europe.

In the US we don't really have any rivers like the Rhine anymore—unencumbered by hydroelectric necessity. Our closest analog is probably a thing called the "Great Loop" [1], which is more of a bucket-list, check-box-excursion sort of thing than a place to live. People who navigate it are called *loopers*, and they traverse a 6000-mile circular "system of waterways" (many of which are man-made) with soulless hyphenesque names like *The Atlantic Intercostal Waterway* and the *Tennessee-Tombigbee Canal*.

I will spare you my rant on the absurd irony of the pork-barrel excavation of navigable waterways in America, in the aftermath of our insane spree of pork-barrel river-damming, and confine myself to pointing out that despite, or maybe *because of* its lack of poetic romance, the American Great Loop, with its overabundance of locks, too-low bridges, VHF signaling, and flood control, is actually far better suited a metaphor to streaming data pipelines and data engineering than rivers like the Rhine.

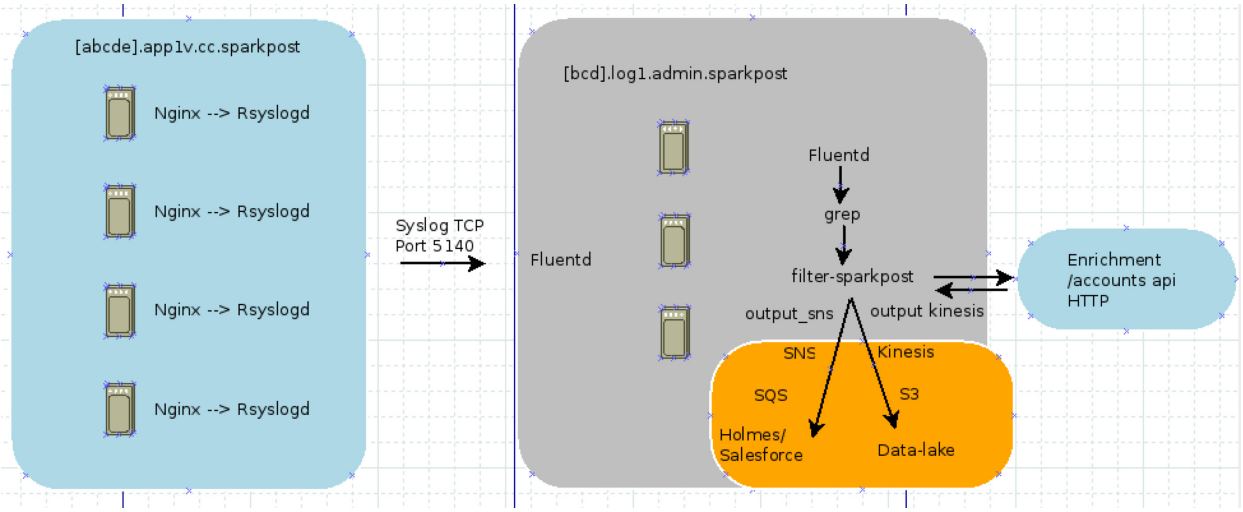


Figure 1: The IEH pipeline architecture

The Loop is a messy, complicated, and artificial route whose waters do not always flow smoothly, in the expected direction, or even at all. Indeed, if data engineering were as pleasant as the great rivers of Europe, STEM graduates would be starving in the streets. But as it is, few people outside of those demented few who yearn for nautical headaches have ever heard of the Great Loop, in just exactly the same way as you, my dear demented reader, are still working your way through this intro in anticipation of my getting to the juicy data engineering bits.

The Flow, Part Two

In my last column, I introduced you to our *Internal Event Hose* ingestion pipeline and *Data Lake* projects at Sparkpost, which together, provide an SQL-like query interface into various types of cherished log data. We covered how schema-at-read systems obviate the need for a proper database and enable us to query any at-rest data set, and we learned about columnar data stores, which make our SQL query engines practical and inexpensive to use (in both the computational and pocket-book sense).

In this article we'll look at what the Great Loop navigators would call *the first leg*, where our log data gets reaped from the instances and undergoes its first transformation into structured data. One of the many well-spring sources of critically important data-flow for us is that of our API servers, whose Nginx processes are configured to send their access logs not to a text file or a syslog server but, rather, to a local UNIX socket like so:

```
access_log syslog:server=unix:/var/run/msys-nginx.sock,facility
=local0 api;
```

Listening to the other side of that FIFO is the local `rsyslogd` process, which is carefully configured to disable all limits, and forward all messages via `syslog/TCP` to an environment-specific logging cluster, which resides behind an Elastic Load Balancer at the split-horizon DNS name: `log.sparkpost`.

```
$SystemLogRateLimitInterval 0
$SystemLogRateLimitBurst 0
$IMUXSockRateLimitBurst 0
$IMUXSockRateLimitInterval 0
$ModLoad imuxsock
$ModLoad imklog
$AddUnixListenSocket /var/run/sys-nginx.sock
local0.* @@log:5140
& ~
```

The day we turned on IEH (Internal Event Hose) in production was (not at all coincidentally) the same day we learned the practical limitations of Nginx's `syslog` outputter, `rsyslogd`, and the UDP `syslog` protocol itself. We service around 11,000 API calls per second in our production environment, a number too great for each of the aforementioned technologies in their original configurations. So Nginx was moved from `syslog`-direct logging to UNIX socket, `rsyslogd` had all of its annoying rate-limits disabled, and `udp/syslog` transport to the logging servers was replaced with `tcp/syslog`.

Listening on port 5140/tcp on the logging cluster is a log-processing framework called Fluentd. You may think of Fluentd as an event-router. You provide routing targets and addressing, and Fluentd routes incoming events accordingly. Our high-level architecture looks like the diagram in Figure 1.

In the configuration, a source block defines a listening port. Routing instructions in Fluentd-land are called *tags*, so the following source block listens for `syslog` protocol on `tcp/5140` and tags everything that arrives as routable to *firehose*.

```
<source>
  @type syslog
  port 5140
  protocol_type tcp
  bind 0.0.0.0
  tag firehose
</source>
```

The *firehose* tagged event's first stop is to a built-in Fluentd plugin called *parser*. The parser plugin's job is, predictably, to parse each plaintext line into a JSON blob of named fields. To do this, it needs a Ruby-syntax regular expression with named fields. For our particular Nginx log format, the Fluentd config looks like this:

```
<match firehose.**>
  type parser
  key_name message
  format /^(?<ts>.*?) "(?<remote_addr>.*?)" (?<response_code>\d+)
    "(?<request>(?<method>.*?) (?<path>.*?) (?<version>.*?))"
    "(?<key>.*?)" (?<key_type>.*?) (?<customer>.*?)
    (?<username>.*?) (?<response_time>.*?) (?<bytes_sent>.*?)
    (?<length>.*?) (?<tenant_id>.*?) (?<subaccount_id>.*?)
    "(?<upstream>.*?)" "(?<user_agent>.*?)" (?<cache_status>.*?)
    "(?<entity_id>.*?)"$/
  tag firehose_parsed
</match>
```

Every plugin begins with a `match` or `filter` parameter that names tagged `fluentd` should route to it. At this point, given the big hairy regex, you might be wondering about the computational overhead of Fluentd, and my answer would be that the system is internally threaded, partially implemented in C, and surprisingly resource-efficient. Although we initially had problems getting the traffic load stable across the network boundaries (as mentioned), we've had no problem running our workload on a set of three (one-per-AZ) modest instances.

Once a given log line has traversed the parser plugin, it exists in a parsed state to the rest of the plugins in the chain. In other words, we can now refer to the individual fields of our log lines using the names we assigned them in the regex we provided to the parser plugin. For example, I can see a given event's response code by specifying `event['response_code']`.

You'll notice the events are now tagged *firehose_parsed*. These get routed to the next filter in our config, which is a custom filter that we wrote ourselves in Ruby (all custom Fluentd filters are Ruby).

```
<filter firehose_parsed.**>
  @type sparkpost
</filter>
```

A simple enough configuration, since Fluentd doesn't know anything about it other than to import our Ruby script and provide it with events via the pre-ordained `filter` function, as described in the Fluentd documentation on writing custom plugins [2].

Our custom filter performs a slew of business-oriented tasks on the event flow. First, we use it to scrub the Nginx data, deriving new attributes from existing ones. For example, the `event['path']` contains the entire path from Nginx, including things like CGI query parameters. In our custom Fluentd filter, we can split these out like so:

```
fixed_path = String(message['path'].split('?')[0]),
query_string = String(message['path'].split('?')[1]),
```

Some events represent API calls that are *special* in a business-sensitivity context, like new-user sign-ups or account deletions. Our custom plugin extracts these events, *enriches* them with data derived from follow-up API calls, and then forwards them to Salesforce and other internal tools by way of AWS SNS (Simple Notification Service).

```
<match ieh_enriched.**>
  @type amazon_sns
  flush_interval 5
  num_threads 20
  buffer_type file
  buffer_path /tmp/td-agent/amazon_sns
  topic_name internal-event-firehose-prd
  aws_region us-west-2
  sns_message_attributes_keys {"enriched":"enriched", "event_
type":"type"}
  add_time_key true
</match>
```

These *special* events are tagged `ieh_enriched` from within our custom plugin. You might notice that there is quite a bit of buffer configuration in this SNS output block. Although we haven't had scalability problems with Fluentd itself, we have found that handoffs to external services like AWS SNS and Kinesis can be fragile. It's taken some time to get the buffer settings locked in for our particular workload, and you should expect a similar experience.

You might be curious about the `sns_message_attributes_keys` parameter. This parameter implements AWS-side SNS filtering [3]. I bring it up because there are two widely used third-party Fluentd filters today. One of them is an unbuffered (read: dangerous) plugin that supports SNS filtering, and the other is a buffered plugin that does not support filtering. What the world in fact needs, is a single, buffered, SNS plugin that supports AWS-side filtering.

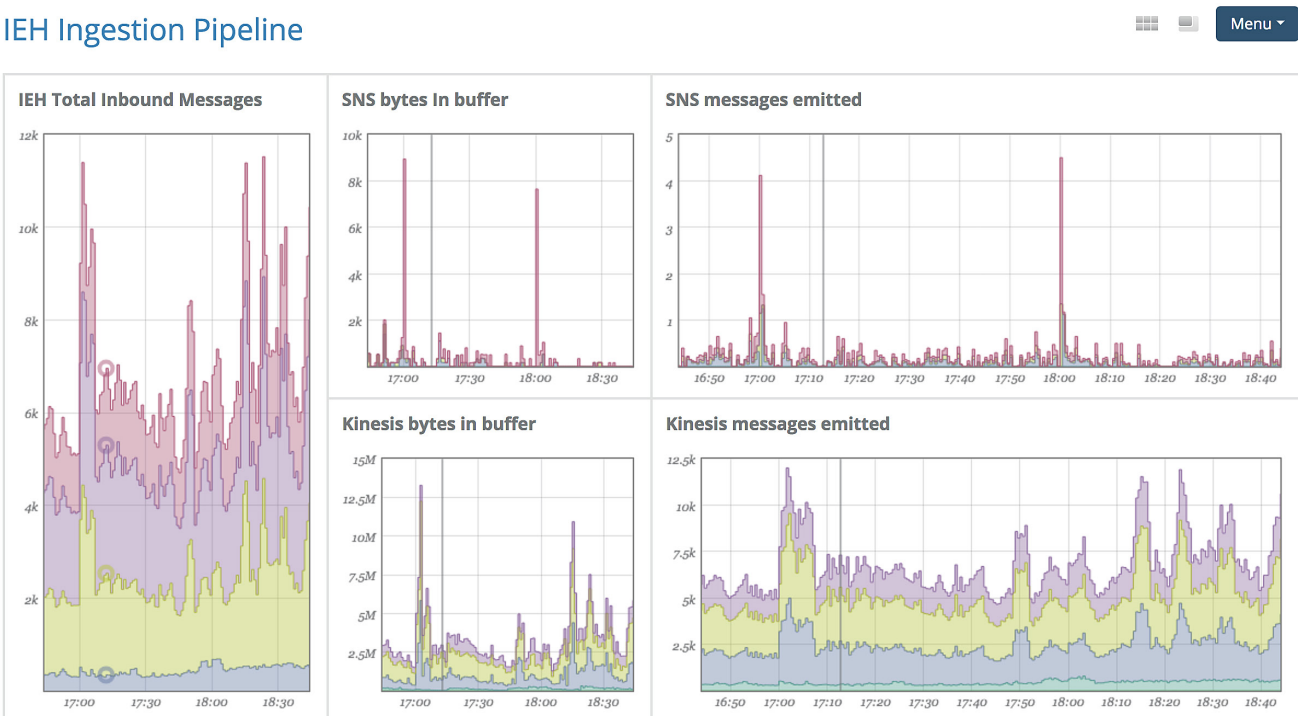


Figure 2: IEH metrics, derived from the Fluentd-Prometheus plugin

To that end, I've forked and extended the buffered plugin with filter support for our workload at Sparkpost and have PR'd the result back upstream [4]. Hopefully, by the time you actually care about this it'll be merged in.

Finally, every event, regardless of whether it is special or not, gets forwarded to the data-lake via AWS Kinesis Firehose. These events keep the `firehose_parsed fluentd` tag and are routed to this output config block:

```
<match firehose_parsed.**>
  @type kinesis_firehose
  region us-west-2
  delivery_stream_name internal-event-firehose-prd
  append_new_line true
  num_threads 64
  flush_interval 1
  buffer_type file
  buffer_path /tmp/td-agent/kinesis_firehose
  buffer_chunk 8388608
  buffer_queue_limit 512
</match>
```

This plugin is an AWS-supported plugin for Fluentd [5] and works very well. We still needed to carefully balance Fluentd's buffer behavior to our workload. Some things to point out here are the `append_new_line` feature, which places a new line between each event rather than just firing a huge incomprehensible JSON blob of 100 smushed-together events into Kinesis, which subsequent data tooling like Glue will not be able to parse. I point this out to you as someone who had to perform a manual retroactive data-reload on several weeks' worth of incomprehensible JSON data.

A few words about Fluentd buffers: first, study the diagram in the Fluentd documentation [6]. There are buffers, and there is a queue, and they are different entities with unique behaviors, log-errors, and configurations. In production you want to use file-based buffers. They are fast enough for sub-second data flushes (remember, buffered output is threaded) and survive better in the event of an instance/server failure. Finally, consider your buffer chunk sizes carefully, especially how long it takes to fill a chunk, because internal buffering can be a source of massive time-delay for low-frequency events.

I'm out of space for this issue, but we've gotten through pretty much all of *the first leg* save for our favorite subject: monitoring. The easiest means of introspecting Fluentd's behavior is using the Prometheus plugin for Fluentd. You don't need to be using Prometheus to use the plugin—in fact, I'm currently using Circonus to visualize metrics from it as you can see in Figure 2.

Next time, I'll walk through the (quite excellent) Prometheus plugin's configuration and start you out on leg two of our journey, where we'll stream our parsed event data into S3 and use Apache Spark to transform it into Parquet format.

Take it easy.

References

[1] The Great Loop: https://en.wikipedia.org/wiki/Great_Loop.

[2] Fluentd plugin docs: https://docs.fluentd.org/v1.0/articles/api-plugin-filter.

[3] AWS SNS filtering: https://docs.aws.amazon.com/sns/latest/dg/message-filtering.html.

[4] Fully buffered SNS plugin: https://github.com/miyagawa/fluent-plugin-amazon_sns/pull/11.

[5] Fluent plugin for Amazon Kinesis: https://github.com/awslabs/aws-fluent-plugin-kinesis.

[6] Fluent buffers: https://docs.fluentd.org/v0.12/articles/buffer-plugin-overview.

Statement of Ownership, Management, and Circulation, 10/01/2018

Title: *;login:* Pub. No. 0008-334. Frequency: Quarterly. Number of issues published annually: 4. Subscription price: $90.

Office of publication: USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

Headquarters of General Business Office of Publisher: Same. Publisher: Same.

Editor: Rik Farrow; Managing Editor: Michele Nelson, located at office of publication.

Owner: USENIX Association. Mailing address: As above.

Known bondholders, mortgagees, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages, or other securities: None.

The purpose, function, and nonprofit status of this organization and the exempt status for federal income tax purposes have not changed during the preceding 12 months.

Extent and Nature of Circulation			**Average No. Copies Each Issue During Preceding 12 Months**	**No. Copies of Single Issue (Fall 2018) Published Nearest to Filing Date**
a. Total Number of Copies *(Net press run)*			2436	2775
b. Paid Circulation *(By Mail and Outside the Mail)*	(1)	Mailed Outside-County Paid Subscriptions	974	888
	(2)	Mailed In-County Paid Subscriptions	0	0
	(3)	Paid Distribution Outside the Mails	693	652
	(4)	Paid Distribution by Other Classes of Mail	0	0
c. Total Paid Distribution			**1667**	**1540**
d. Free or Nominal Rate Distribution *(By Mail and Outside the Mail)*	(1)	Free or Nominal Rate Outside-County Copies	77	77
	(2)	Free or Nominal Rate In-County Copies	0	0
	(3)	Free or Nominal Rate Copies Mailed at Other Classes	18	17
	(4)	Free or Nominal Rate Distribution Outside the Mail	374	330
e. Total Free or Nominal Rate Distribution			**469**	**424**
f. Total Distribution			**2136**	**1964**
g. Copies Not Distributed			300	811
h. Total			2436	2775
i. Percent Paid			78%	78%
Electronic Copy Circulation				
a. Paid Electronic Copies			468	466
b. Total Paid Print Copies			**2135**	**2006**
c. Total Print Distribution			**2604**	**2430**
Percent Paid (Both Print and Electronic Copies)			82%	83%

I certify that the statements made by me above are correct and complete.

Michele Nelson, Managing Editor 9/28/18