Saltside Engineering
Field notes from the team behind Tonaton.com,
Follow

Our Failure Migrating to Kubernetes

Adam Hawkins Follow
Jun 2, 2017 · 6 min read

My team attempted to migrate one application tier to Kubernetes this week. Unfortunately we aborted because increased timeouts between applications and our own inability to debug and resolve the issue. This post is about what our wrong, what we know, and what may be wrong. My team needs help debugging this issue so please respond (beers on me!) if you can help.

The Setup

20 odd applications compose our entire product. We use Apache Thrift for communication between internal services. The migration goal was to move a subset to Kubernetes.

Our approach was to create one Kubernetes `LoadBalanacer` service to expose all Thrift services to our existing infrastructure. I'll simply refer to these things things as "[the] ELB" and "apollo" going forward. Unique port numbers on the ELB identify each service. We use Thrift with the binary protocol over TCP. Also our Kubernetes cluster exists in separate VPC which peered with apollo's VPC. Once the service was created and everything reported healthy, then we updated apollo application configuration to find the services in Kubernetes. Technically this happens by setting environment variables like `FOO_THRIFT_URL=tcp://the-k8s-load-balancer-host-name:FOO_SERVICE_PORT`. This worked fine in our initial tests so we decided to move forward switching over more production traffic. If things caught fire it's easy to rollback.

Lastly some particulars about the cluster. We used Kops 1.6.0 to build our cluster. Here's the version:

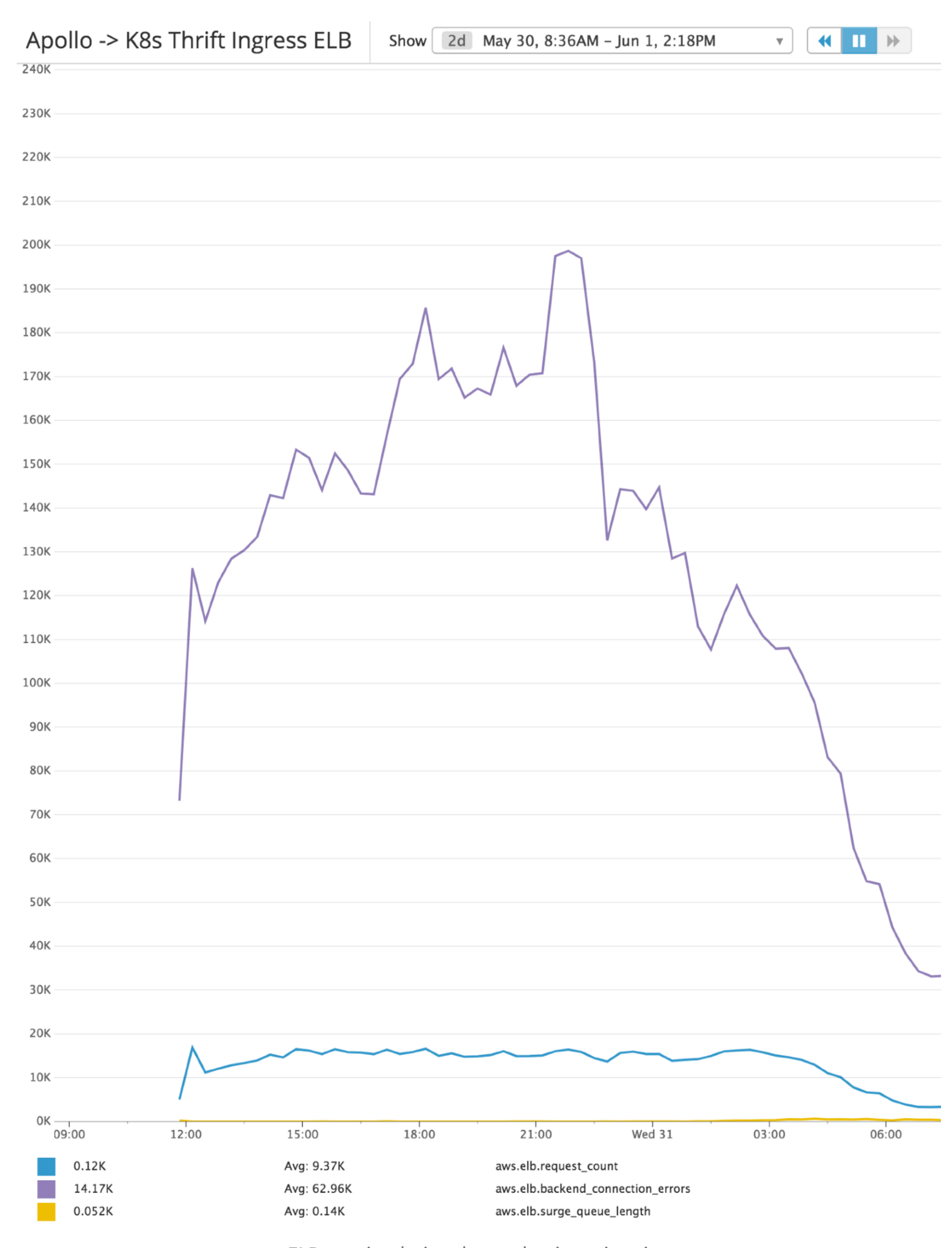
```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"5",
GitVersion:"v1.5.6",
GitCommit:"114f8911f9597be669a747ab72787e0bd74c9359",
GitTreeState:"clean", BuildDate:"2017-03-28T13:54:20Z",
GoVersion:"go1.7.4", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"5",
GitVersion:"v1.5.7",
GitCommit:"8eb75a5810cba92ccad845ca360cf924f2385881",
GitTreeState:"clean", BuildDate:"2017-04-27T09:42:05Z",
GoVersion:"go1.7.5", Compiler:"gc", Platform:"linux/amd64"}
```

What Happened

We triggered a apollo puppet update to switch traffic to the Kubernetes. We noticed an absurdly high `BackendConnctionErrors` count the ELB. AWS' definition is:

> *The number of connections that were not successfully established between the load balancer and the registered instances. Because the load balancer retries the connection when there are errors, this count can exceed the request rate. Note that this count also includes any connection errors related to health checks.*

Here's our data from this time period:



ELB metrics during the production migration

The data starts when traffic switched over (~12:00). The purple line is `BackendConnectionErrors` and blue is `RequestCount`. AWS defines `RequestCount` as:

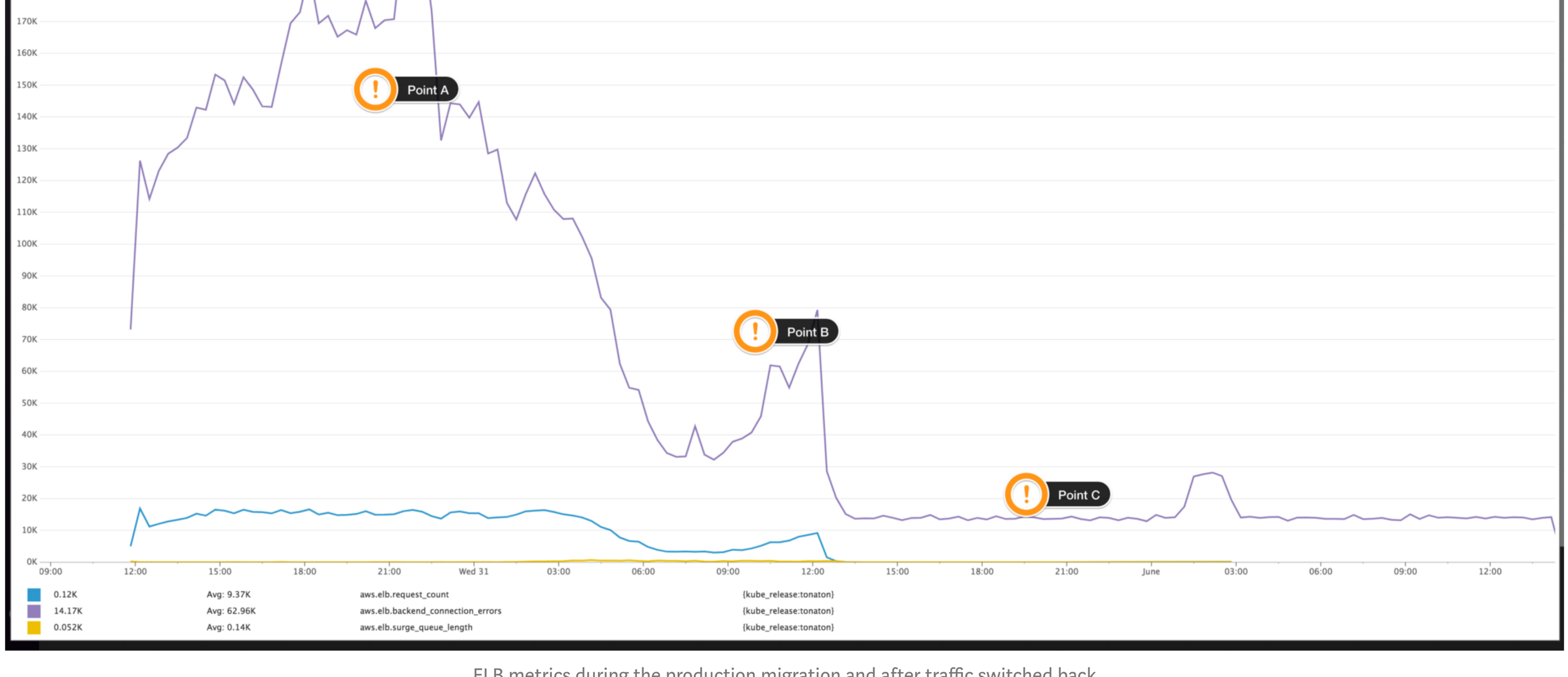
> *The number of requests completed or connections made during the specified interval (1 or 5 minutes).*

My understanding is there ~10% successful connections depending on time. Or in other words, **there are a ton of failed connections.** Initially this did not correlate with other error related metrics. Our backend architecture uses RESTful JSON APIs clients (web application, android, iOS) processes as proxies/orchestrators to multiple internal Thrift services. Thus Thrift RPC failures usually propagate as 5xx in the HTTP API tier.

Initially these bad connections did not correlate with those metrics. We let things run for about 12 hours to see if anything changed or if we could debug it. The next afternoon we realized the 5xx in the HTTP API tier and latency increased since the switch over. We rolledback at this point.

Observations

We attempted to debug the situation while the cluster served traffic. We could not conclude the root cause, but we did make some interesting observations. Consider the following graph.



ELB metrics during the production migration and after traffic switched back

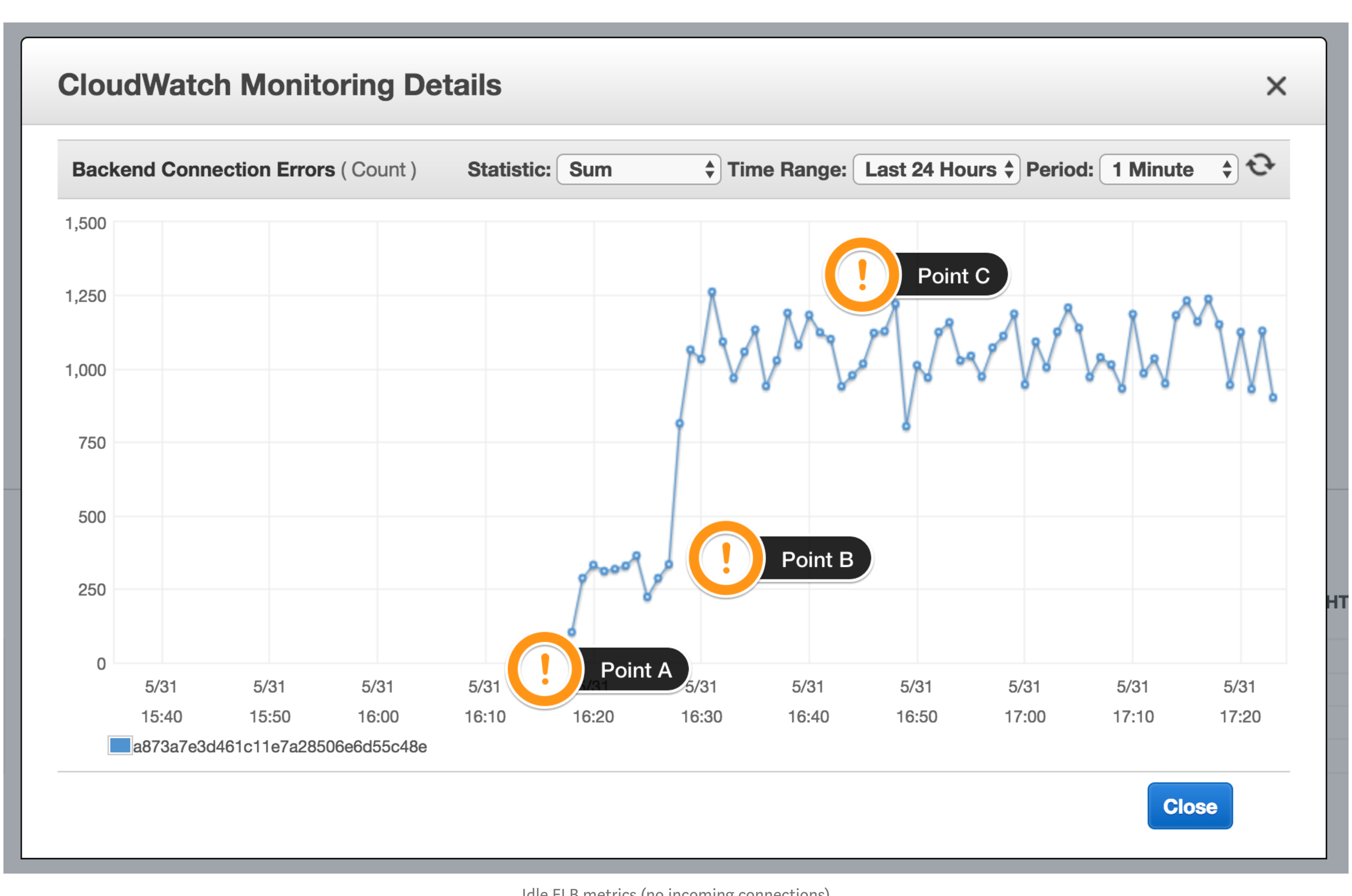
- Point A: This downward trend correlates directly with the our active users. Users increase throughout the day until a sharp nightly drop off. No changes were made to the cluster or applications during this time. Curiously the `RequestCount` does not correlate with the `BackendConnectionErrors` at all. I'd `RequestCount` (the blue line) to have it's own peak (since more users equates to more RPCs and thus more connections). However the `RequestCount` is relatively stable until the lowest traffic level (~08:00 on the graph). One could also interpret this as the cluster can only support ~15k connection requests.
- Point B: Sharp increase corresponds to more active users coming online the next day. Note the small increase in `RequestCount`.
- Point C: We switched traffic back to apollo at ~12:30 graph time. You can see the `BackendConnectionErrors` and `RequestCount` drop off at that point. It's interesting that `BackendConnectionErrors` does not drop to zero. Instead it stays constant at ~11K *even* when there are no incoming connections.

My assumption is that `BackendConnectionErrors` should approach zero under normal operations.

Debugging Experiments

My goal is to understand the metric and diagnose the root cause. I am waiting on a AWS support ticket clarifying what's included this metric. Here are my own observations until then.

I wanted to confirm the number of `BackendConnectionErrors` in an idle copy of the same application. I installed a fresh copy of our Helm chart and watched the ELB metrics. Here is the result.



Idle ELB metrics (no incoming connections)

Point A is t-zero. The `BackendConnectionErrors` reports immediately. Te The value is ~300 until Point B. I enabled cross AZ load balancing at Point B. This added 6 instances to the ELB, which in turn proportionately increased the `BackendConnectionErrors`. Point C onwards covers the ELB in an idle state. The `BackendConnectionErrors` stay constant.

This leads me to conclude that `BackendConnectionErrors` are not related to actual connection requests in any way. The numbers are connection chatter between the ELB and cluster nodes. Here's the math behind the metric.

The ELB in question has 9 nodes behind it. I'll fudge this to 10 to make the math easier. The ELB has a single health check at a 10 second interval. This means 10 connections every 10 seconds, or 60 a minute not counting any retries. Consider the data between Point A & B. There were 3 instances registered with the ELB. That equals 3 connections every 10 seconds, or 18 (3 * 6) connections a minute. `BackendConnectionError` is ~300 during this time frame. That's off by one order of magnitude. You need to add one order of magnitude *and* double it again to approximate ~300. Consider Point C. The number of instances increased from 3 to 9. This equates to 9 connections every 10 seconds, or 54 (9 * 6) connections a minute. That number is not at all close to the ~1,000 in that period. Something is off here. The numbers do not support the hypothesis that numbers are connection chatter between the ELB and cluster nodes.

We've been using ELB's for TCP load balancing in Apollo for years without issue. I checked the `BackendConnectionErrors` for Apollo load balancers. There have been virtually zero for months. This corroborates the conclusion that there is something *specific* about the cluster nodes registered with the ELB.

My Hypothesis

My current hypothesis is the `BackendConnectionError` floor generated enough overhead to increase connection time. This lead to an Thrift RPC timeouts as connections timeout and longer times to acquire successful connections. This explains the HTTP tier latency increase and 5xx HTTP tier increase.

This hypothesis does not leave me in a good place for a few reasons.

- Are connection resets or other early close scenarios covered in this metric?
- The math to support health check only error chatter does not add up — not even close. How in the world are there *so* many connection errors?
- Which part in the network chain is the problem? Kubernetes `LoadBalancer` services work in combination with `NodePort`. Every node is assigned a unique port which forwards to the correct pod/port. Each cluster node is registered with the ELB. Each declared service port maps an ELB port to the Kubernetes `NodePort`. This creates a flow like client -> ELB -> instance Node port -> `kube-proxy` (and other Kubernetes magic with `iptables`) -> pod -> container. There are too many components for me to debug with my limited understanding of the networking internals.
- TCP ELB access logs are not helpful. There is no information about errors, only about IPs and latencies.

I need your help debugging this issue. I suggest the best place to start is on the instance itself or by watching application logs and noting any sort of connections. I do not know how to start debugging the instance itself. My question is: what commands or tools should I use to poke around the network stack to debug this issue or should I look elsewhere?

Hopfully the team can resolve this quickly resolve this and get on with the migration.

Thanks for reading. Good luck out there and happy shipping!

Docker Kubernetes DevOps AWS

WRITTEN BY
Adam Hawkins
Follow
Code DJ! Tech: https://hawkins.io. Podcast: https://smallbatches.fm.

Saltside Engineering
Follow
Field notes from the team behind Tonaton.com, Bikroy.com, and Ikman.lk