honeycomb.io

Do You Remember, the Twenty Fires of September

Over September and early October, Honeycomb declared five distinct public incidents, for various reasons and of various severities. As far as we are concerned, the whole month was part of a broader operational burden, where over 20 different issues came up to interrupt normal work. A fraction of them bubbled up to having a public impact that was noticeable and declared, but most of the significant work would have been invisible. A retrospective would be incomplete if we considered the incidents as distinct entities rather than part of a longer connected sequence.

This series of incidents occurred in a context of continuous change. From August to September:

- The amount of data ingested by Honeycomb grew by roughly 40% (due to both individual customer growth and overall customer count),
- The platform team was working on a migration of our infrastructure from Chef-managed instances to containerized deployments
- The development of new features and integrations kept at a steady pace.

Most of the challenges described here come from a pattern of accelerated growth and scale, highlighting performance degradation and brittleness in our stack in non-obvious ways—particularly when multiple components are hitting inflection points at the same time on multiple dimensions, and there's no clear way to single out any particular slow part. We'll describe the various events that composed into that work—and omit some less relevant ones for brevity even though they were part of a challenging workload—before doing a review of the lessons that can be learned from our experience.

One additional thing to keep in mind as you read these is how many of these individual incidents or near-misses feed into each other to progressively paint a more complete picture, where we finally have enough data to explain everything that happened.

Incidents and near misses

Kafka Rebalancer wedges

Every Tuesday, an automated task we have shuts down one of our Kafka brokers in each environment. This has become standard practice with multiple services that have a fixed cluster size to ensure we are able to replace them. On August 31, the Kafka auto-balancer, which takes care of moving partitions to keep load even, got stuck. A replacement broker came up, but no partitions were assigned to it.

We got a non-paging alert from a trigger telling us that some partitions were under-replicated. That tends to happen right after scheduled replacements, and along with other operations going on, we believed it to be normal. It was only later, on September 1, that we noticed the Kafka replication trigger was stuck in alerting mode.

We finally detected the replacement broker receiving no traffic. Thinking there was something wrong with it, and knowing it had held little data and was leading no partition, we took it out. Then the replacement's replacement got stuck as well, and we knew something was odd. Fortunately, none of this affected any customer since all our Kafka partitions are replicated on 3 brokers in different availability zones and could still tolerate more failures. After help from Confluent's support, on September 3, we found a procedure that unwedged everything.

A bad SSD blamed on a big customer

On Saturday, September 11, a single node of our distributed columnar data storage engine, retriever, started seeing an elevated rate of file system errors that suggested a failure of its solid-state drive. Retriever nodes operate in redundant pairs, so data storage was not impacted by this failure. Both nodes in a retriever pair, however, participate in answering queries for a particular subset of events. During this degradation, the disk errors caused queries handled by this retriever node to incur a performance penalty of a couple seconds. The issue was noticed on Monday morning, and after investigation, the offending node was replaced, restoring service to normal.

The investigation was made more complex by recent discussions where we had wanted to try going to bigger instances, which anchored responders into thinking this could be a capacity issue. Since a large customer was the most impacted, we assumed they were to blame for the overload.

Eventually, we would find that this fault was part of a broader series of failures that were noisy throughout the month and related to a kernel bug around file systems. We have, however, needed to go through many of these failures to see a pattern emerge and to investigate.

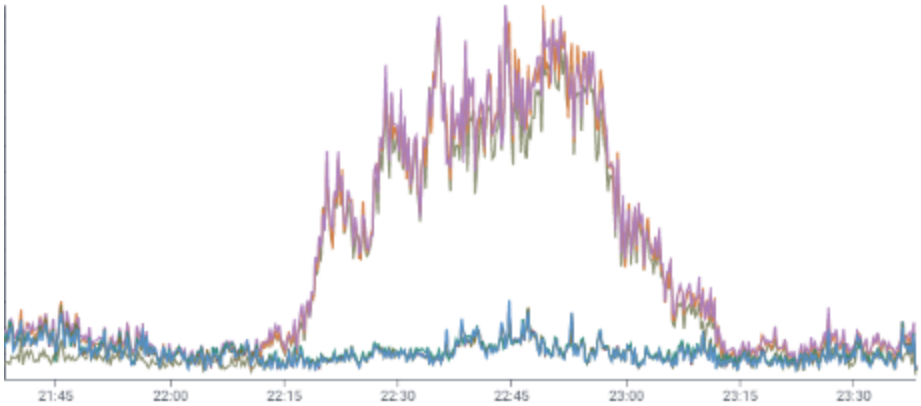
Some lessons from this investigations that may be of general interest:

1. Large customers can be red herrings. The incident happened early on the East Coast, where a specific customer starts business and ramps up quickly. They were hit hardest, to the point where it looked like they were causing the damage. The correlation was thought to be causation when it was not. “Normal” has different meanings at different times.
2. Hardware problems can be hard to detect with the way we instrumented our systems because we seek problems in the data we have first. Things hidden in dmesg are surfaced only after we exhaust more accessible tools.
3. While we thought only a major customer was impacted, all communications surrounding the incident were kept internal, with remediation focused on managing their load. Once the incident was re-framed as a general disk fault, we shared it publicly and reoriented our response.

Overloading our dogfood environment

Through the summer and until mid-August, various optimizations were added and limits raised in our stack, which drastically increased our burst capacity for read queries.

Since then, we had seen a few alerts where multiple components would alert across environments, but without a clear ability to explain why that was.



On September 16, it happened far more violently than we had experienced before. We later found out it was caused by another large customer issuing multiple costly queries, which

collectively read about 3.3 billion column files (1.55 petabytes) in a short time, over S3. Each of these accesses to S3 in turn generate access logs.

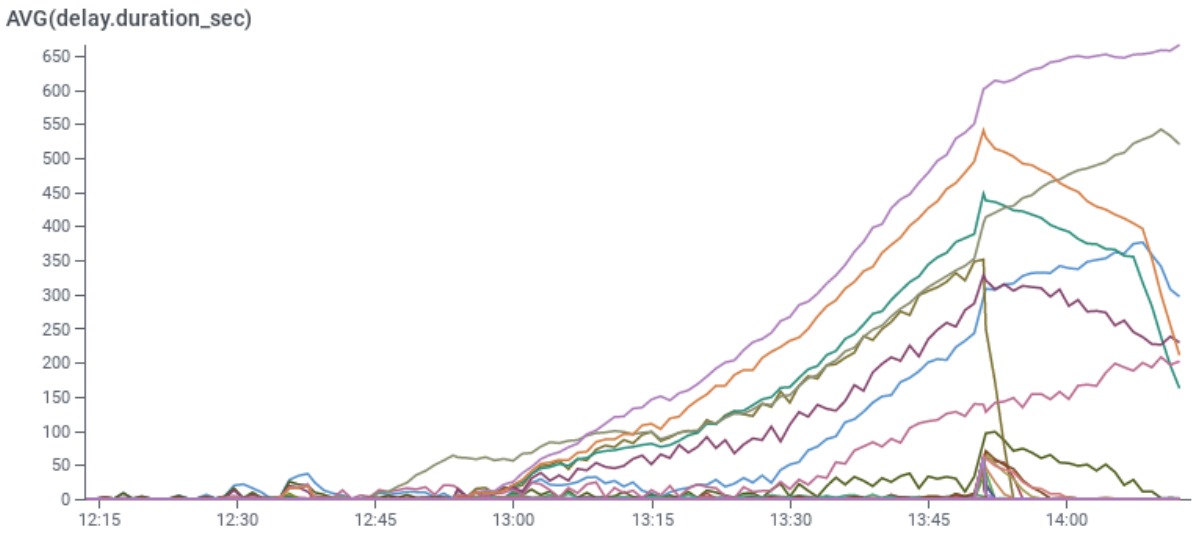
What happened then is our dogfood ingestion pipeline, generally seeing far less traffic, was under-provisioned for this spike. This is effectively equivalent to a Denial of Service (DoS) attack. When this happens, other clients trying to write to the dogfood API endpoint (all production components) get delays and possibly errors until auto-scaling catches up.

Amid the confusion, the volume and back-pressure caused ingestion delays and even crashed production Kafka reporters to Honeycomb, and reports going to third-party tools were brought down with them, making it look like full production outage to our redundant alerting systems.

Beagle processing delays

Beagle analyzes input streams for service level objectives (SLOs) data. Its auto-scaling works by being CPU bound. Partition imbalance, SLO definition imbalance, and network throughput are all different things that can contribute to CPU being a poor proxy metric for its load. We knew about this, but felt it wasn't worth the cost of implementing a custom CloudWatch metric source when CPU worked tolerably, only to get rid of it with our container migration that would soon use a new scheduler.

Previously when beagle would warn of being behind on one or two Kafka partitions, we'd manually scale up its auto-scaling group, which fixed the problem with minimal effort. On September 21, the problem looked very different:



The big bump at 13:50 matches an increase in capacity where some partitions got better, but some still got worse. This is a sign to us that this isn't related to the scale of the consumers.

All of a sudden, a lot of unrelated partitions were lagging behind and struggling, and scaling up gave no immediate results. Scaling yet again seemed to improve things a bit, but the catch-up rate was below our expectations. Eventually, the cluster caught up and stabilized again, without any specific intervention. It did, however, divide our attention.

Migrating dogfood retrievers

On September 21, after a few weeks of running retrievers in hybrid mode between EC2 and EKS in our dogfood environment, we completed the rollout of EKS retrievers. EC2 retrievers were scaled down. Everything seemed fine—until the next day, when we woke up to alerts stating that records, segment data, and columns couldn't be written to disk.

This meant that the entirety of dogfood retrievers was out of disk space and couldn't even write down metadata, and cascaded into other alerts throughout the platform.

What happened (but we didn't know at the time) was that in containerizing retrievers, we accidentally omitted to transfer cronjobs that ran only on EC2 retrievers. One of these is a task that orchestrates our data's life cycle. In a nutshell, all data on retrievers goes through a sort of long-lasting garbage collection for database files that range from their creation, to S3 upload, to deletion through aging out. By not having it running, retriever instances kept accumulating data until they were entirely out of room.

We usually get warnings about disks filling, but got none in this case because we believe retrievers on EKS (which use hostPath mounted volumes from the parent host to store their data) don't see that disk usage reported in Kubernetes metrics. So any early warning that could have let us know things were getting dire was not there.

We guessed that something might have gone wrong due to the retriever migration. Not knowing what it was, we decided to boot EC2 instances again to run whatever was missing. We eventually spotted the cronjob issue and started manually clearing disk space to salvage instances. This failed because as soon as we'd free space, most Retrievers would write back to it before we could clear enough to let the garbage collection run.

This, in turn, generated a lot of noise on kibble, our environment that monitors dogfood (which monitors production), which also ran out of disk space. Unlike dogfood, it was due to generating so much traffic in a short time for its tiny cluster size that it ran out of space before we could even run a GC lifecycle on it.

After failing to free up space, we saw that our new EC2 instances were healthy and had run their own life cycle tasks to completion. This meant that we could now swap the dogfood EKS instances to let new ones take over by fetching correct state (written by EC2 instances) off of S3.

Kibble was salvaged by manually deleting the data from the dataset that was spammed by logs. We had no better solution, and we knew that for the last hours, all the logs were garbage, so we took the loss.

While this was going on, a production host ran out of disk space as well, but it was due to a bad disk (again!) and was easy to fix. Still, for a brief period of time, all three environments (prod, dogfood, kibble) were reporting retrievers with filled disks at the same time, for different reasons. No customer data or performance was impacted at any point, but this was still an all-hands-on-deck situation.

We scheduled an incident review because a lot of interesting stuff happened there despite having no production impact. Unfortunately, we did not even have time to finish the incident review before we got interrupted by yet another incident.

Further beagle processing delays

On September 23, beagle alarms tripped once more. We initially blamed high CPU variance, but after looking further into Kafka, found out that the rebalancer got wedged once again:

```
broker: 1262      leading: 7      non-leading: 12     total: 19
broker: 1263      leading: 7      non-leading: 14     total: 21
broker: 1264      leading: 7      non-leading: 13     total: 20
broker: 1266      leading: 6      non-leading: 15     total: 21
broker: 1267      leading: 6      non-leading: 16     total: 22
broker: 1268      leading: 8      non-leading: 11     total: 19
broker: 1269      leading: 0      non-leading: 1      total: 1
```

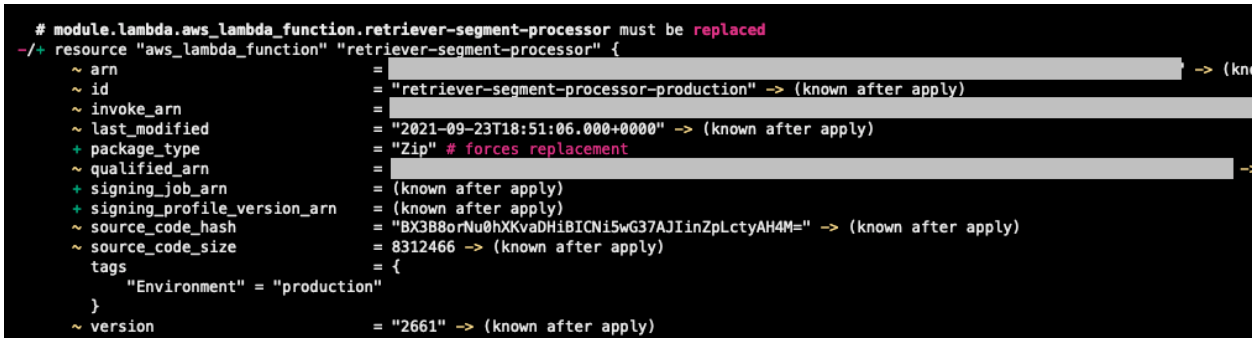
This had caused things to go out of balance and thought this could have overloaded some leaders. We reused the procedure we had developed earlier that month to fix it. The rebalancing nearly caused some Kafka partitions to run out of disk space, so we dropped our non-tiered retention from 3 hours to 2 hours.

Without us knowing about it, the previous day's dogfood migration issues repeated the DoS incident effect that shut down Kafka's production metrics pipeline, which silenced all the data that would usually warn us of under-replicated partitions. The radio silence meant we only found out through indirect signals related to performance.

The dropped retention brought enough room to rebalance the cluster and eventually fixed beagle's lag. We decided to add alerting and make a complete runbook to detect and manage future rebalancer wedges. This alarm has proven useful a few times already.

Lambda deleted by Terraform

On September 23 while running the Dogfood Disk Exhaustion retrospective, we got interrupted by another odd issue, where a seemingly routine Terraform cloud deployment deleted the reference to our production lambda worker for all retriever reads.



We still don't know why the package type changed and why that forced a replacement (which puts in a placeholder). The actual lambda is written by our regular deploy mechanism, which was re-run manually to force a resource to be put in place.

It took a short while for the system to stabilize again, and we added a protection in the terraform file to prevent it from accidentally being deleted again. This is an interesting event because it interrupted corrective work for other issues, and is part of a pattern of ongoing pressures that made it difficult to keep up with and improve our overload situation.

Kafka scale-up

On September 24, the beagle processing delays kept happening, but this time we knew the Kafka cluster to be balanced. However, we detected that some Kafka partitions seemed to be lagging behind others.

After plenty of debugging in a Zoom call (we were getting fed up with these issues), we discovered that our Kafka cluster's brokers were silently being throttled over network allowances by AWS:

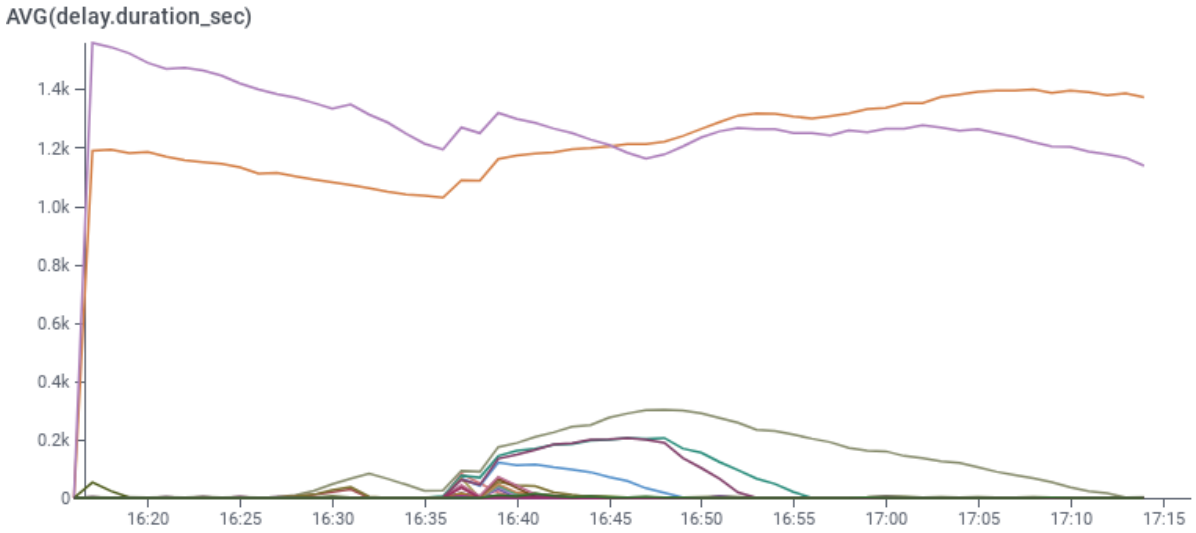
```
Every 2.0s: dsh -c -M -g kafka-production 'sudo ethtool -S ens5 | grep bw_out_allowance_exceeded' | sort

kafka-0008c0bce67996f8b:      bw_out_allowance_exceeded: 8641416
kafka-0188e44ff6caa1d76:      bw_out_allowance_exceeded: 3580811
kafka-020367088f8744f45:      bw_out_allowance_exceeded: 10825493
kafka-04647a50e73e41b90:      bw_out_allowance_exceeded: 4762976
kafka-0b9125f3e14bca660:      bw_out_allowance_exceeded: 4373466
kafka-0c5d3ff84c29bd6a9:      bw_out_allowance_exceeded: 1820860
```

We made an agent to extract the values and to start accumulating data, and comparing it to other services showed that the Kafka instances were being impacted at a far higher rate than others.

We manually checked other network values and decided that our Kafka cluster needed to be scaled up vertically to get onto instances with more network capacity.

As we were planning to grow the cluster, beagle started lagging again and had a hard time recovering, so we decided to fast-track the migration of the most impacted partitions by shifting them from an older smaller instance to one of the new ones.

We then decided we'd transfer data slowly over days with rebalancing off since we wanted to leave old instances nearly empty and the new ones full.



We left things stable on Friday, and completed the migration on Monday and Tuesday (September 27-28). At some point on the 28th, beagle kept being delayed some more, and our end-to-end tests started firing one again.

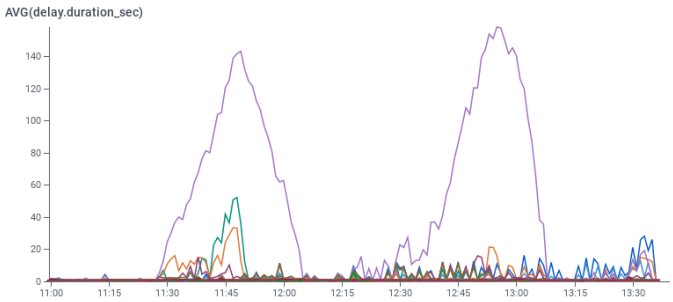
We found out that the latter was caused by many retriever partitions "double-consuming," which means that both retrievers in each pair for a partition is multiple seconds behind in reading from Kafka. This is tolerated by readers, but it means the data Honeycomb users see is either temporarily incomplete (because their data may be on partitions at different levels of progress) or missing (because it's late on all partitions). We posted a public status for this since it was customer-impacting.

We quickly found out that the issue was partially caused by having turned on the autobalancer back on for the Kafka migration (so it would move partitions from a removed older instance onto newer ones), and having it move partitions back onto the smaller instances we were still

planning to cordon off. We canceled the migrations and turned the balancer back off except for instance replacement. This let all consumers catch back up.

Understanding beagle processing delays

We hoped that completing our Kafka migration would solve the beagle processing delays once for all, but on Wednesday (September 29), they happened once again. It now became clear that this issue shouldn't be caused only by Kafka being overloaded since we had added over 50% extra capacity.

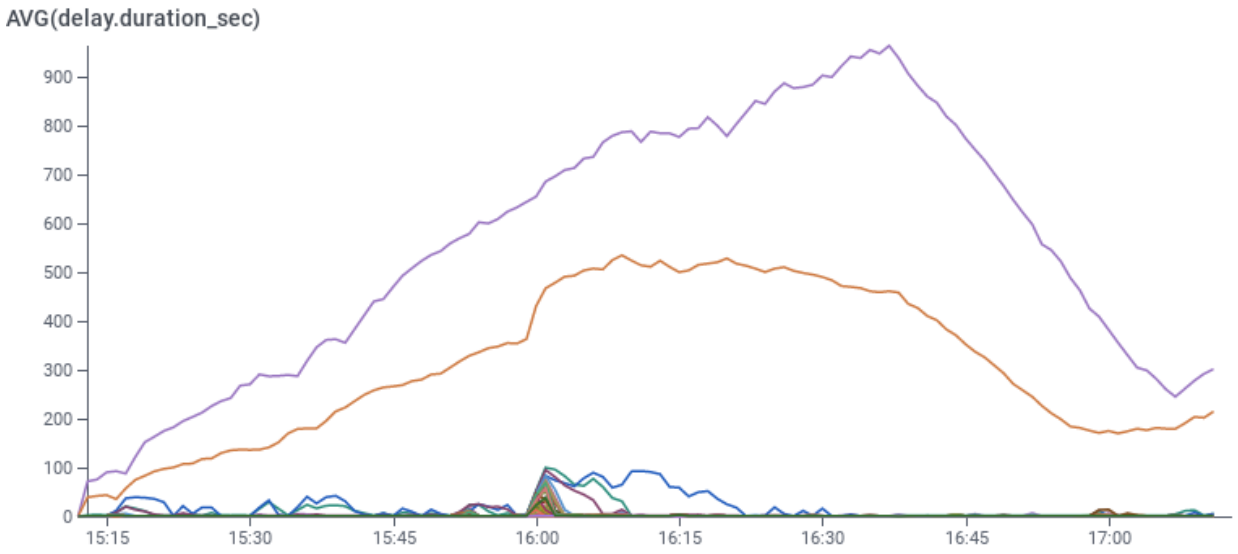


We once again found out that leaders were left unbalanced:

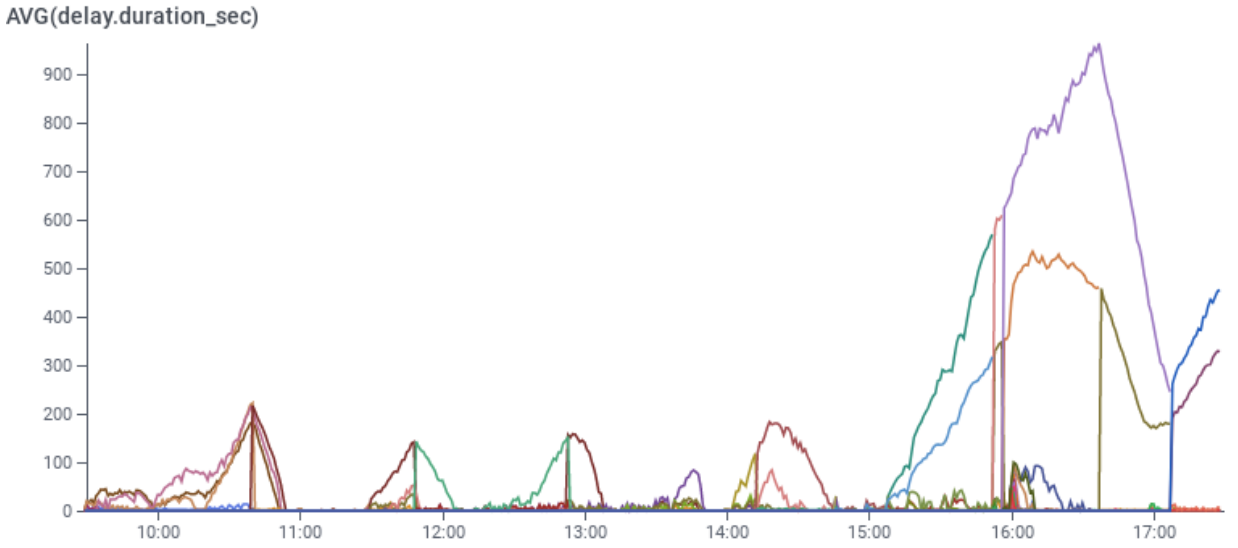
```
broker: 1271      leading: 8      non-leading: 13     total: 21
broker: 1272      leading: 7      non-leading: 14     total: 21
broker: 1273      leading: 6      non-leading: 14     total: 20
broker: 1274      leading: 7      non-leading: 13     total: 20
broker: 1275      leading: 8      non-leading: 23     total: 31
broker: 1276      leading: 5      non-leading: 5      total: 10
```

The rebalancer had died once again, and we had no metrics to fuel early alerting because again, the reporters from Kafka had died as well. We kicked them back up, planned an upgrade that would solve the crash issues, and quickly juggled leadership on partitions and migrated some to once again get things in balance.

Everything was catching up as we came closer and closer to being fully balanced, but once the rebalancing was complete, beagle latency worsened again. Therefore, the balance alone couldn't explain the performance issue.



At one point, we got an alternative version of the graph where instead of grouping by partition, we grouped both by partition and by beagle consumer. And now things looked fun:

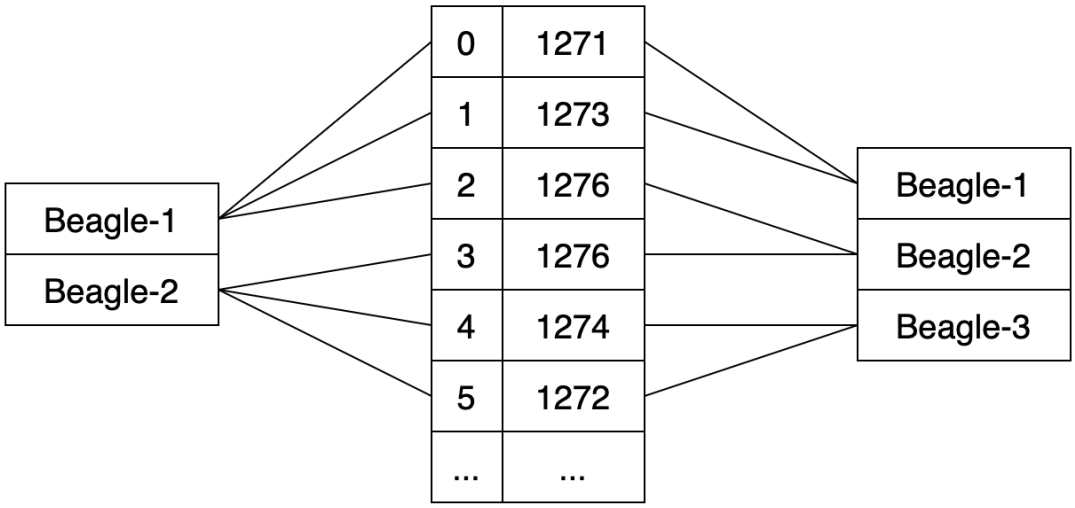


The choice of partitions that beagle would consume from could cause problems. We started having a bit of a divide on the team, debating whether the problem was beagle or Kafka itself. Both looked like they under-utilized their resources and both could go faster, but they just would not.

A few experiments over the day confirmed that the issue seemed to be around pairs of leaders. The Kafka cluster has a mostly random-looking leader assignment. Beagles' consumer group

would however assign them sequentially. Whenever the same leading broker got some of its partitions assigned to a single beagle 2 or 3 times, its consuming performance got worse.

This was hard to detect (because the data for both lives in different systems) but easy to test (just shuffle leaders). It explained why scaling up would often fix the problems, but also why sometimes scaling up or shuffling leaders made things worse even if balance was expected to be much better that way.



This, we found out over the course of days, is because Kafka consumer groups always open only a single connection to any leader, regardless of how many partitions are going to be fed from it. In turn, this creates a point of contention where the connection buffers of the Sarama library themselves (and the speed at which we consume them) bottleneck all traffic. This explains the behavior we saw of both sides of the equation sitting idle while there was more work to do. Many assignment strategies exist: range (the one we used), sticky, and round-robin.

Rather than shifting the strategy, we tried various settings to increase buffers and throughput over a few days, which at this point seemed to hold up. We also had a back-up solution of using smaller Beagle instances and using them in a fixed pool of one per partition; we did not need to use it, but it was planned to buy us some peace after a rough operational month if we needed it.

Scaling up retrievers

What became apparent once we understood the beagle issues is that retrievers themselves were also having scaling problems, since both Kafka and beagle were individually explained. We planned a scale-up, adding roughly a quarter extra capacity by scaling horizontally on September 30.

Before then, we wanted to double-check whether scaling vertically would also make sense. Retriever had been way over-provisioned for over a year, and we had only scaled it up marginally earlier in 2021 to make sure we still knew how. So we weren't quite sure where the limits were in the system, and it seemed we were hitting bad inflection points.

We booted a single larger instance (m6gd.2xlarge → m6gd.4xlarge) and let it steep overnight to see if things would be better with it, but it proved inconclusive. While rolling out the instance, we also noticed something we named "blinking," where retrievers would fall in and out of their partition assignment for a few seconds at a time, something that should never happen.

We started following the steps to scaling out retriever that we had put in a runbook. One significant gotcha about this runbook was that at the time it was written, beagle did not yet exist. The scale-up steps were added after the fact, but never tried under production workloads with empty partitions, so we knew there could be a risk around it.

Even before we got to the beagle steps that involve inserting a configuration for the missing partitions, we started seeing crashes and delays. We guessed that the most likely reason was that there was no data in beagle, and decided to complete the scale-out ASAP.

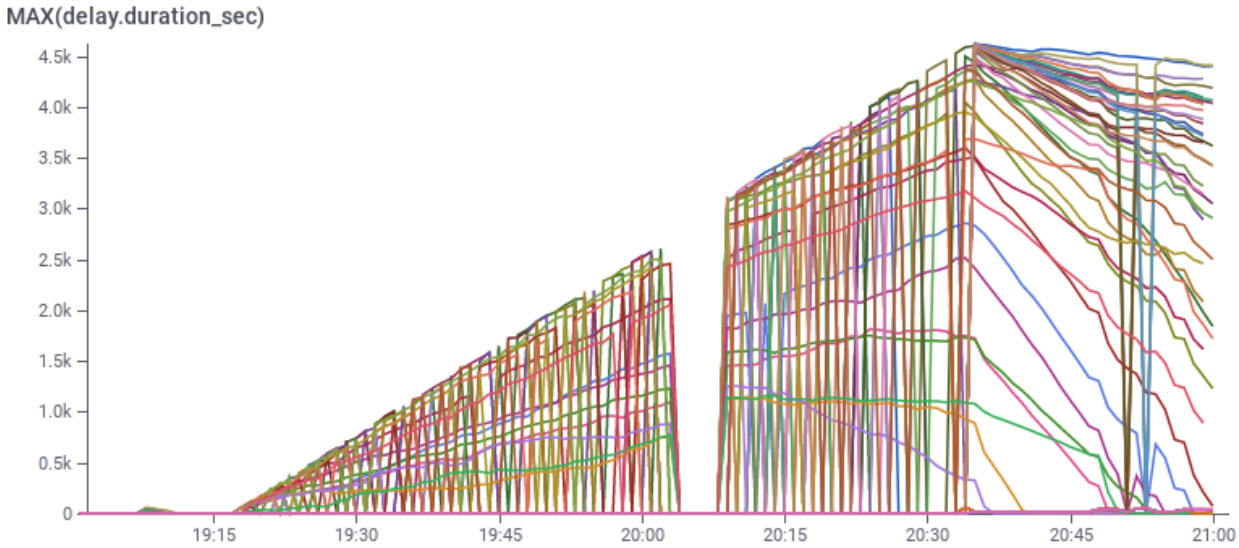
In the hurry, we made many small mistakes. We were supposed to go to 56 partitions (0..55), but ended up setting only 53 of them up at first. This required a back and forth in scaling and record injection. Another one was that the runbook told to introduce records in the beagle SLO database:

```
("beagle", "honeycomb-prod.retriever_mutation", <N>, 0, "manual")
```

Unfortunately, the proper topic is "honeycomb-prod.retriever-mutation" with a -, not a _. We did not notice this when writing the runbook, when crafting the queries, when reviewing them before applying them, when applying them, and even when doing the first one or two audits of the table after things were going bad.

The overall end result was that beagle struggled and crashed in a loop until we managed to stand up the whole pipeline end-to-end and data started flowing in, which caused an SLO processing delay outage. At some point, we corrected all the little oddities and traffic started flowing through.

Once we caught up, we backfilled the SLO data and all customers' service was reestablished properly.



All the new partitions, which had no enterprise customer that would use SLOs on them yet (the final rebalancing is still manual) showed over 4 hours worth of delay and were not recuperating. What we found out was that this was related to a lot of small issues we wouldn't have encountered in other circumstances:

- Partitions with no SLOs would not correctly mark their progress when consuming data
- Some customers were doing heavy load testing, where they created hundreds of datasets, sent a high volume of messages, and then deleted them

The latter in particular was problematic because we do look up datasets in a database, and then cache the results. But when a dataset is missing, we cache nothing. This is generally not a problem when the consumers are up to date because you don't get messages for a very long time after their dataset is deleted. However, in this case we got backlogged by several hours on some partitions and this drastically slowed down the ability to consume anything at all. The bad behavior was invisible until other things were also going bad, and it made them worse.

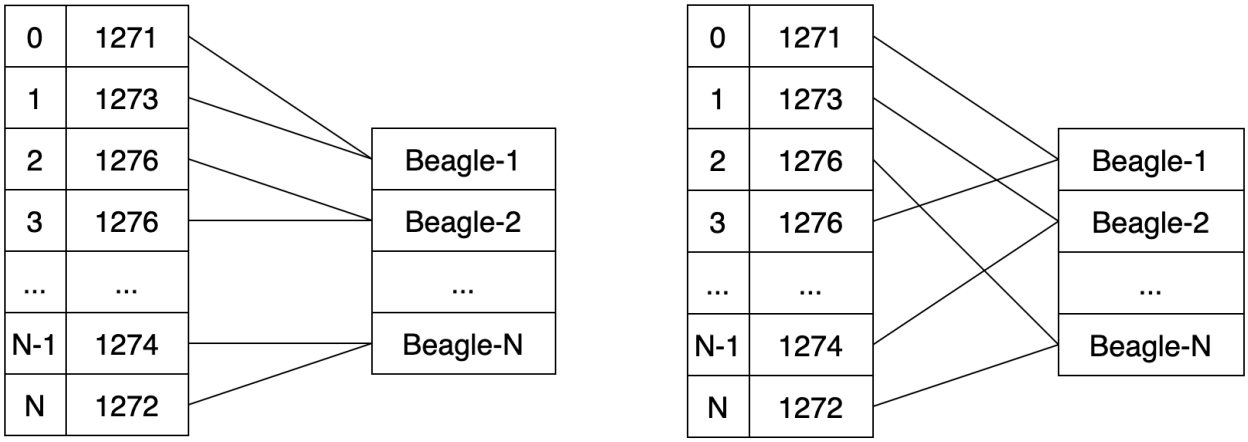
Pending patches, we created fake datasets in the database (attributed to our internal teams) to let beagle catch up by filling its cache, then deleted them again. We also created fake SLOs on all of our internal end-to-end datasets that are pinned to specific partitions.

During the next hours, the patches to properly cope with each issues have made it to production and we got stable again. We also took the opportunity to migrate heavy partitions away from overloaded existing ones onto new ones. We would then do a trickle of smaller retriever rebalances over the next few days, and benefit from the improved capacity.

Final beagle instability

Scaling up partitions meant that the new ones, all assigned in a series, were far less loaded than the older ones. This caused a severe imbalance where the last of the beagles would have no work to do, and drag down the average CPU consumption for the cluster, causing more autoscaling woes (on first the image below)

We ended up fixing it by changing the allocation strategy to be round-robin, which at least would spread the load more equally across all Beagles, and things got back to being acceptable.



We also scaled up the beagle cluster size to a fixed, larger size, which had proven stable regardless of the day of the week or time of the day. We have, however, found out that as we add instances to the cluster, rolling restarts cause larger disruptions to the consumer group that tries to shift load around, and are running experiments with a Sticky strategy to reduce interference. Finally, and more recently, we changed our deploy strategy to completely ignore rolling restarts. See this Twitter thread by one of our engineering managers about it.

September stretches into October

Things didn't quite end there. After analyzing the gain on scaling, we saw that we mostly only gained one month of growth room, maybe less, depending on how fast our customers grow.

We ended up having to cover a few extra issues in October already, all in its first week:

- Horizontally scaling retrievers again to buy room rather than just be okay
- Discussions around what the scaling strategy should be for datasets and bits of continuous expansions
- A troublesome database migration that flushed indexes aggressively and caused blips

- More frequent request interruptions during retriever failures causing potential query problems for customers
- Triggers reaching a point where they sometimes and inconsistently require a long time to work, which required further investigation and highlighted stuck SQL queries that we're currently trying to pin down
- Keeping on cycling retriever instances to avoid file system corruption issues, which was finished in the later weeks of October
- More stress tests by some customers, causing surprises. Dataset deletion and re-creation could reset some limits and scale markers that could lead to overload:
    - One interesting case happened while cycling retrievers: One would not successfully bootstrap because files were seen as missing when a partition had been manually deleted until its peer ran its backup task and cleaned up segments expectations
    - Discovering limitations around the throttling and rate-limiting mechanisms applied to teams and/or datasets
- Requiring emergency surgery on init files in production because an experiment to try and drain retriever connections more effectively on deploys went awry, and a regular deploy could have crashed the whole cluster
- RDS CPU alerts firing and hinting at another vertical scale-up required there, which we ended up doing. Specifically, we ended up improving our ingest performance seemingly by a lot by scaling up, which indicated that we were starting to see it act as a chokepoint that could slow down some queries

We're now looking more stable than during late September, but it's obvious we have more lessons to learn and more limits about our system to discover. For example, growing our container fleet has highlighted more stress in our usage of AWS' SSM, with limits needing to be raised.

Lessons learned and things to keep in mind

Scaling of individual components

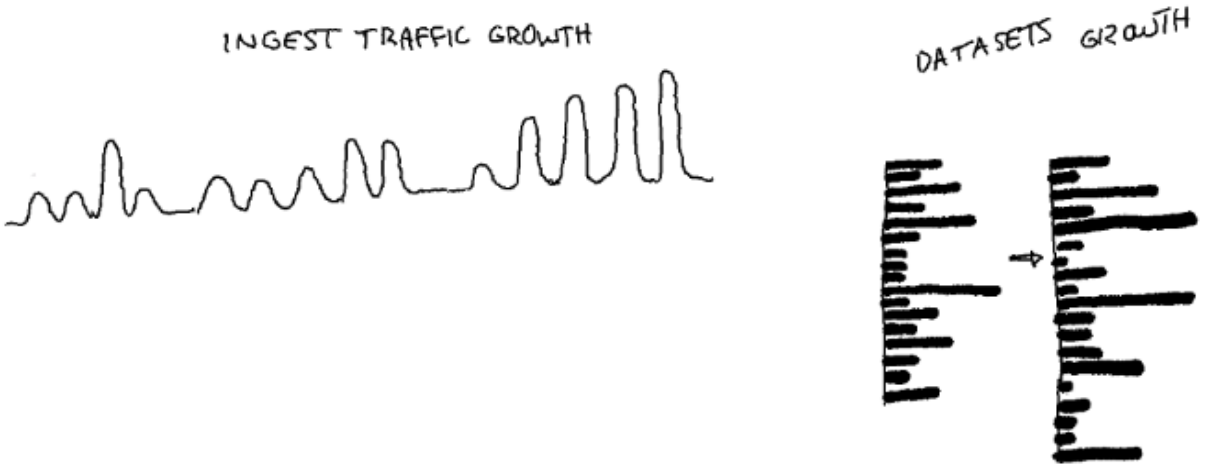
We've had something close to 40% growth in ingest traffic between late August and the end of September. In hindsight, It looks like we haven't been proactive enough, but my understanding of it is really that while some of us had ideas about what some of our scaling limits were, nobody had a clear, well-defined understanding of it, and of all the dimensions.

During the month, we ran into scale issues around:

- Networking and throttling of data packets that were previously unknown and invisible to us
- Limitations in the abilities and stability of the Kafka rebalancer
- Surprising abilities of our production cluster to overload our dogfood cluster in ways that left longer-lasting impacts to production instrumentation (Kafka monitoring)
- Bottlenecks and points of contention around consumer groups in our Kafka client libraries
- Sequential bottleneck in retriever consumption
- Memory and CPU limits around retriever's ability to serve some particularly large queries
- Manual rebalancing of partitions and tool-assisted rebalancing were nearing their toil acceptability levels
- The frequency of deploys was increasing and their effect was inflated and, therefore, more visible in our SLOs

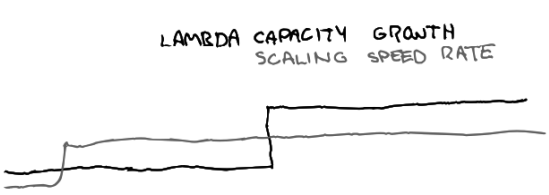
If we're lucky, we would hit these one at a time, but we unfortunately got in a situation where various types of pressures (likely a related to having to scale many different types of customers all at once) just showed up at once, and disguised themselves as each other.

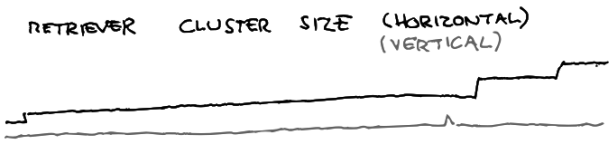
All of it came from rapid customer growth in a short time span.

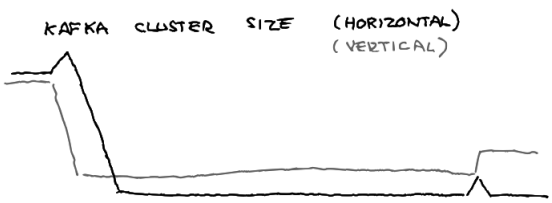


Overall traffic does not scale uniformly across customers, whose datasets aren't uniformly distributed either and may have implications around other services. Beagle consumes all messages of all datasets, but the count and costs of SLO means the scaling shape is distinct than what retriever needs. Interactive queries hit lambdas often and can bottleneck there, whereas triggers are nearly fully on the hot sets and entirely within retrievers.

As we add features, a growth in customer ingest and querying patterns lead to distinct scaling patterns for various components. To add to the challenge, it's sometimes unclear if the limitations highlighted in scaling are due to a bottleneck that would be solved more effectively by scaling vertically or horizontally (or based on some other dimension). This gives us a sort of scaling profile as follows over the last few months:







For each component impacted, its own growth and ability to scale either vertically or horizontally is a function of both costs, awareness of bottlenecks, expected growth models, and so on. Lambda grew capacity in a very stepwise manner that changed volume downstream of it, even in different environments. Retriever's throughput boundaries were mostly unknown, and we needed to experiment to see what would be most effective. Kafka is expected to be fixed in number since the Spring 2021 migration due to licensing structures, except when scaling up vertically, where, for safety, we boot a peer group of larger instances and migrate traffic off. Beagle stayed mostly stable and is now fixed in size because that seems reasonable, but could have gone smaller vertically to grow wider horizontally.

Combinatorial scaling

The real upcoming challenge we're facing, aside from just scaling things in foreseeable directions (more customers mean more partitions) is having to consider when we're going to have our future scaling plans run into each other and cancel each other:

- Scaling retrievers horizontally on writes may make reads more costly or likely to hit high 99th percentile values, which in turn means some larger customers' datasets may need vertical scale

- Vertically scaling retrievers does not necessarily address load issue that beagle could one day see, and scaling horizontally does dilute its autoscaling metrics and require fancier approaches
- Increasing scaling ability in one environment can cause ripple effects in other ones that are loosely coupled to them due to second-order effects, on entirely different dimensions with distinct failure modes
- Extra indexing or internal logical partitioning of datasets would improve the ability to handle triggers and queries, but could cause more load on RDS instances that handle columns and make rate-limiting fuzzier
- More large customers mean more edge cases exercised more frequently and more of these weird interplays clashing in the future
- Deploying more often makes each deploy safer, but as deploys to retrievers cause minor interruptions, these accumulate to having a visible effect on our reliability as well
- Our own observability tooling is running into new hurdles as GROUP BY limits in Honeycomb queries mean we can't see the work of all our partitions or all hosts at once, and its accuracy will only shrink over time.

These last few weeks are probably the clearest signal we have yet of where a lot of our limits lie and where we need to start planning growth adaptation in a composable approach, rather than a more local, per-service vision.

Experience and tempo

It has been surprising how often one of the new incidents highlighted something we did not understand in a previous one or that a previous incident held the keys to solving one of the new ones. This can sometimes feel like the story about the old Chinese farmer, but really should reinforce the idea that all incidents are learning opportunities.

We believe maintaining the ability to adapt to production challenges comes from having a sustainable pace. Not too active, not too sparse, a bit like exercising to stay healthy. It is possible that a knee-jerk reaction where we over-scale the system to ensure we never have issues in the foreseeable future only gives us more time to forget about some operational issues, and makes it easier to turn a healthy amount of it into a dormant long memory.

To put the analogy another way, seeing a piece of wood bend can be a good signal that it's nearing its limits. As load increases, it's useful to keep ourselves familiar with the signals and ways various loads bend the system.

One of the things that was called out in the Platform Team weekly meeting was that we were running at a rather unsustainable pace during most of the month. Lots of work was dropped and, as the incidents recurred, they got longer and more frequent, and the amount of context

available and required to handle them increased dramatically. This in turn meant the people handling many of the incidents felt better equipped to handle the other ones that kept happening. We were entering a self-reinforcing loop in the worst way possible.

We learned that the ability to have downtime and hand-offs that transfer that context from coworker to coworker does become necessary to keep operational burdens sustainable, and calling out such situations to force a shakeup can be effective.

It's worth pointing out that we do believe there is plenty of optimization potential in most of our code bases, which would let us do more with the same cluster and instance sizes. These optimizations generally take longer to put in place than scaling up does, so being caught in a situation where multiple components approach the edge of their performance envelope at once means we can be forced to scale to buy time to optimize properly at a later point, but only if pressure lets up.

If we operate too far from the edge, we lose sight of it, stop knowing where it is, and can't anticipate when corrective work should be emphasized. But if we operate too close to it, then we are constantly stuck in high-stake risky situations and firefighting. This gets exhausting and we lose the capacity, both in terms of time and cognitive space, to be able study, tweak, and adjust behavior of the system. This points towards a dynamic, tricky balance to strike between being too close to the boundary and too far from it, seeking some sort of Goldilocks operational zone.

While we don't have a perfect recipe for this balancing act, we do believe that a focus on learning from all production woes plays an integral role in keeping that balance and maintaining long-term system (and our team's mental) health.