honeycomb.io

Kafka Migration and Lessons Learned

Over the last few months, Honeycomb's platform team migrated to a new iteration of our ingest pipeline for customer events. This was not the first time we've done this, and while we did not fundamentally change how our system works, we did nevertheless introduce significant changes into a core component of our system around which a lot of our practices had stabilized.

Our migration to this newer architecture did not go over too smoothly, as can be attested by our status page—between February and March. There were at least three instances of ingest latency shooting high enough for us to communicate it with customers, one of which lasted over 12 hours. There were also roughly half a dozen additional near misses where our team was paged and reacted quickly enough to avoid major issues, but that still took their toll.

As such, I wanted to share with our readers what went on and the surprises we encountered during this migration, along with the lessons we learned.

The old systems

Architecture

Before we get started, a quick refresh: Honeycomb consists of our ingest workers (called *shepherd*), Kafka, query workers (*retriever*), and frontends (*poodle*). We also run *dogfood* for analyzing production's telemetry, and *kibble* to analyze *dogfood*.

We've had multiple variations of Kafka clusters over the years:

- We tried various instance types and storage strategies, but always needed SSDs (whether EBS-backed or local) to hit proper latency targets.
- Retention was initially of 4+ days but got as low as 12h based on improvements that were made to retriever's crash recovery mechanisms.

The important characteristics we were looking for were related to reliability and latency (to prevent backlogging, we were aiming for <10ms from produce to consume), and how important our safety margins were. At the end of 2020, we determined it was important to have the following:

- At least 24 hours of data availability in case of any retriever mishaps (e.g., corruption, data loss, bad deploys that take a while to notice) as a disaster-recovery buffer. The longer the retention, the better.
- Roughly one hour of rapidly re-playable buffer to cover retriever restarts from hourly snapshots. When a retriever restarts, it fetches data from S3, saved hourly, and re-synchronizes the rest from Kafka.

The trade-off costs were related to a few main concerns:

- **Storage costs.** We pay a premium for faster disks for our latency needs, but only needed ~1h of it to be this responsive in normal scenarios.
- **Instances count.** We mostly needed a lot of instances to get all the disks we needed, which meant a ton of them were sitting idle with under-utilized networking, RAM, and CPU while we needed to add more and more data.
- **Operational costs.** Since customer load shifts and changes with time, load and partition management was mostly manual and tedious to do..

Following this evaluation, we decided to improve our cluster once more by trying to find a better balance across all these trade-offs.

The new system

Planned architecture

These challenges led us towards Confluent 6.0 Tiered storage, which promised to drastically reduce our costs by offloading cold Kafka segments to S3. This would have let us achieve the following:

- Keep SSDs for the “warm” data (used for replays).
- Move all the colder stuff to S3 and pay only for storage there.
- Increase our retention for mishaps to up to five days.
- Change the scaling of the cluster from being triggered by disk size (expensive) to CPU, RAM, and network characteristics of the instance. This manifests immediately as lowering the number of instances from 36 to 6 (of a different type).
- Move to instances that use Graviton2 processors, with which we previously had great results.
- Have access to nicer features such as auto-rebalancing, which promises to do away with manual cluster partition broker management, and rack awareness, which promises to lower cross-AZ costs.
- Get access to Confluent Control Center, which improves automation by offering an API for cluster maintenance instead of command line tools.
- Obtain support and better tooling from Confluent folks.

The migration looked like a slam dunk. At the time this project was about to ship, we frequently firefighting to maintain our data retention to its bare minimum and often had to compromise on it for short periods of time to prevent cascading failures.

Rollout and surprises

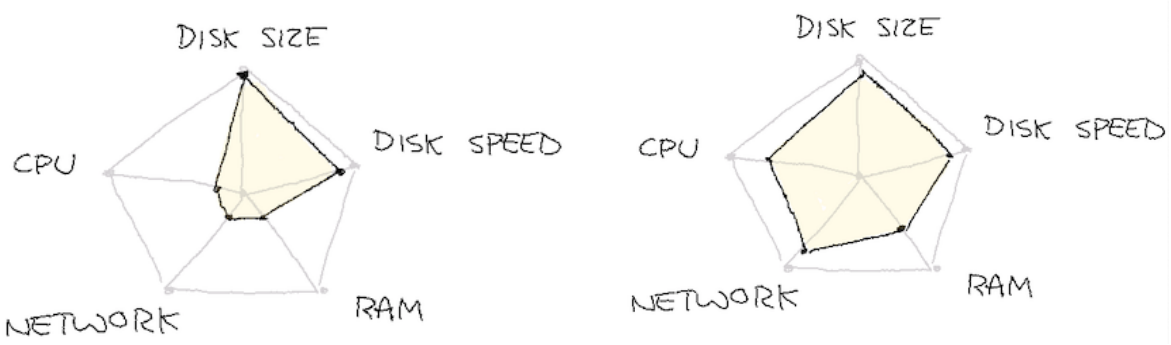
We found lots of things going a bit wrong here and there, and a few things going very wrong all at once during specific outages. I won’t provide a play-by-play of the incidents, but will share some of the significant ones.

Host Replacement

The most significant part of our migration came from turning on tiered storage, progressively uploading all of our longer retention data to S3, and keeping a few hours only locally. This represented an early point of no return: if we started encountering issues, the data wouldn’t be there locally anymore to roll back.

Once that point was reached, we would boot up 6 new hosts of type m6g.xlarge, manually move the topics to them, up until we could get rid of the 36 i3en.xlarge instances.

We had made some quick calculations—which Confluent's tooling confirmed—showing that based on our usage patterns, we could move from the utilization profile on the left (over dozens of hosts) to the one on the right (over only 6):



The move went smoothly in both the dogfood and the kibble environments, where we let this mature for a while.

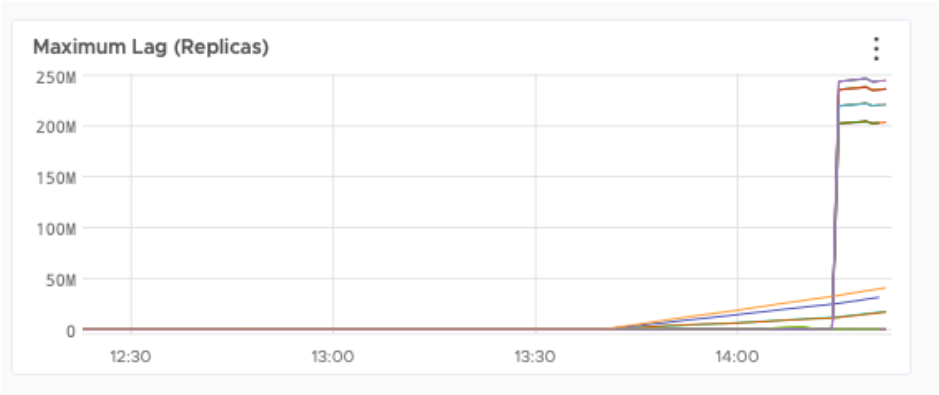
We then moved to production, by migrating partitions to the new hosts. Within an hour, things started going sour as the new hosts seemed slower and our alarms went off around data staleness. We noticed that we were pegging the CPU at 100%. At this point, one of our engineers found out that we had mistakenly migrated to the m6g.**large** instance type—the same one we used in internal environments—instead of the m6g.**xlarge** instance as we had planned for.

This change was subtle enough that careful rollouts over multiple days just made it recede into the background. Since nothing went wrong early on, we put more and more load onto the new hosts until we reached a tipping point.

Of positive feedback loops

It was already too late to back out of the change. We were now running over 40 instances, but ironically wedged ourselves into an under-provisioned state.

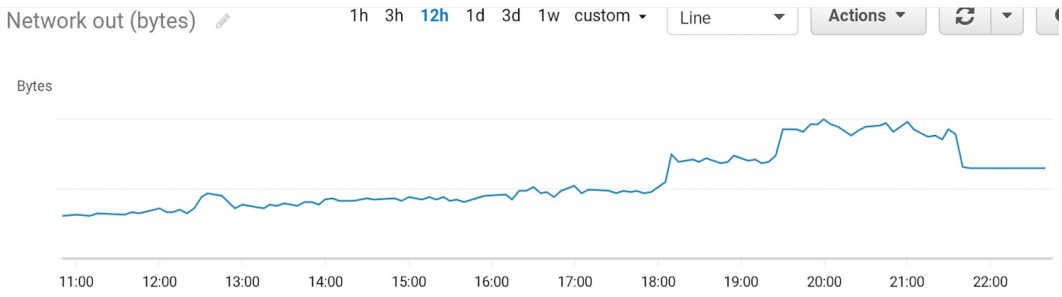
Since we were falling behind, we started moving partitions *back* to the older bigger machines. However, as we started the migration, things got even worse. The new hosts were replicating slower than expected, which worsened delays.



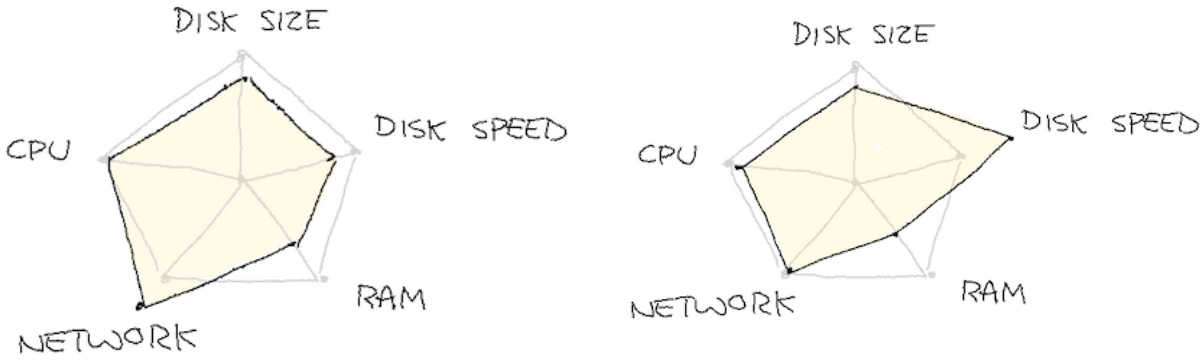
We weighed the cost of letting things catch up versus adding further uncertainty by doing an in-place upgrade of m6g.large instances to c6gn.xlarge—they'd offer better network and CPU, whichever might be the bottleneck, while keeping data in place. The latter felt risky as this isn't a procedure we were super familiar with; we just knew it *was* possible to make it work.

We instead tried letting things catch up, helping them by using administrative functions that let us shift traffic from underpowered instances we mark as read-only, to others that have spare capacity. This would let us write new data only to partitions that were on old hosts until the rest trickled off the new hosts, without major write delays.

We noticed that CPU was *not* saturated anymore. We had been over-using the network, until AWS throttled us and had even less capacity than early on to work with.



It was surprising to find we had been running over our allocated share of resources, completely blowing out our ability to cope with the load we had. It's still unclear right now if the limits we ran into were due to the network or the disk speed (through EBS rate-limiting under similar mechanisms). Both profiles below are as likely, and we possibly hit each one of them in succession:



The end result was that just replicating data *out* of the m6g.large instances was over-saturating the host to the point we were replicating slower than we could write.

Live instance upgrades and ASG issues

As latency kept crawling up, it also propagated to more and more partitions even though we were trying to use the read-only hammer as much as possible to keep things under control. We decided to upgrade instance types live, by marking them as "standby," changing their description, and starting them back up.

We got ready and marked a first instance. The ASG instantly reaped it and added extra pressure to an already slow replication. We eventually managed to re-stabilize the cluster, adjusted the procedure a bit, tried again, and failed again. We decided to just roll the instances normally to resolve things, even if it meant taking a bit longer.

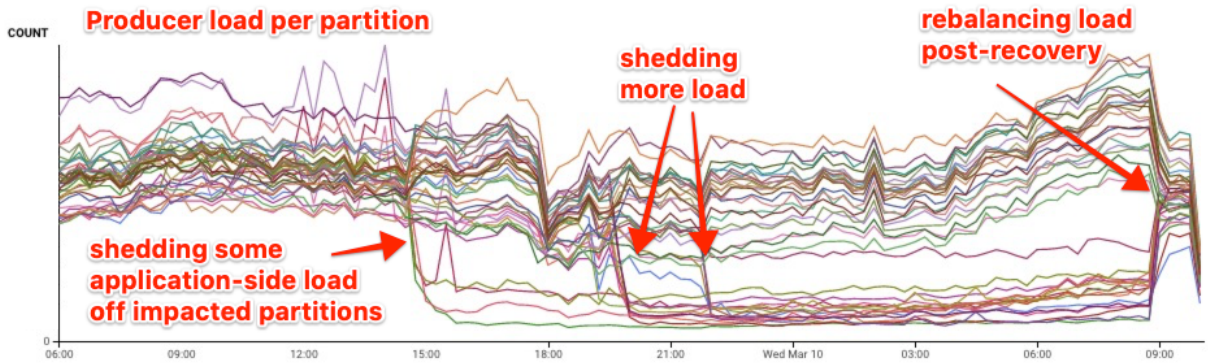
After the switchover to new Kafka instances, we disabled scale-in protection as things seemed stable once again, as part of the initial plan.

However, we had fallen into an imbalance in our ASG where we expected all three Availability Zones (AZs) we're on to be used equally—we expected a 2-2-2 mix but were into a 3-2-1 situation, which the ASG corrected by killing an instance and replacing it to get us back to 2-2-2.

Traffic shaping

I mentioned using read-only partitions as a traffic shaping mechanism. This can usually destabilize the cluster, but was used to the opposite effect during the night, trading off retriever stability against Kafka stability, until everything was running smoothly again.

But as the sun rose, the traffic progressively ramped up. The Kafka brokers were stable, but their consumers' disks (retrievers) were now filling up at a faster rate than usual, risking to cause other failures. We started re-enabling writes, re-shifting load around services to normal levels post-recovery.



Rack awareness piling on

Kafka provides a feature called Rack-Aware Replica Selection, which lets consumers prioritize local replicas rather than leaders. It lets you limit how much cross-AZ traffic you do; you pay the cost once when replicating, and then all consuming is then local. We wanted to try that one, and bundled it in the roll-out.

One of the many, many odd things we noticed during the outage was that the selector was extremely eager to use local replicas. It seemed to insist on using local ones even if they were not up-to-date, while a healthy leader in another AZ was available.

We later found out that the code had a small oversight: in selecting which replica to consume, no criteria existed to restrict consuming to replicas that were in sync. As such, the consumer would gladly pick out a new broker we just booted, without sufficient data.

We have since opened a pull-request addressing the bug upstream with plenty of details and hope to be able to re-enable it in the future.

Auto-balancing capacity

One of the Confluent Kafka features we turned on as part of the roll-out was self-balancing clusters, which promises highly configurable ways of automating the management of partitions to keep everything stable.

For us, given the harsh constraints of too-small instances with over-subscribed resources around EBS and the network, auto-balancing turned out to accidentally become an amplifier for

cluster issues. As replicas got out of sync and everything started lagging more and more, Kafka's own automation tried to reassign partitions across brokers, which were already not keeping up. This cascaded into further leader elections.

We no longer felt confident about what the exact operational boundaries of our cluster were supposed to be. We opened a support issue mentioning our issues and hoping to eventually get a less eager balancing going on.

As it turns out undocumented options for auto-balancing exist, which do just that:
`confluent.balancer.num.concurrent.partition.movements.per.broker=1`

By default, this value is set to 5. While this might be sufficient for many clusters, how appropriate this is depends in no small part on how much spare resources are left available for special operations, given how much they are used for steady state use. Our cluster is also atypical (we use fewer, bigger partitions rather than many small ones) and discussed these things with Confluent when evaluating whether it was workable, but nevertheless got surprised.

We have since managed to find a sweet spot configuration that lets us benefit from the auto-balancer without saturating our resources under common emergency situations.

Other contributing factors

Not that we're short on surprises, but other elements have been mentioned over time that I felt should be included:

- Our metrics and monitoring around Kafka were spread out between various dashboards and providers, with some values imported into Honeycomb. At some point in time, the values across these did not align, and it was difficult to figure out exactly what to trust.
- Initial incidents and near-misses were tricky, and not everyone had the experience required to operate optimally in these situations. They fortunately provided training for the bigger outage.
- Various configuration options exist for Kafka, and our own systems contain lots of possible throttles and controls as well. When seeking explanations, those are all part of the candidate space and bear a cognitive cost even when they don't actively contribute to events.
- Similarly for our recovery options, many ideas were concurrently available, with various levels of risks associated with them. They too bear a cognitive cost and complexify already new and dynamic situations.
- Even after we figured out a working procedure, we still had to wait hours for things to stabilize due to resource constraints.

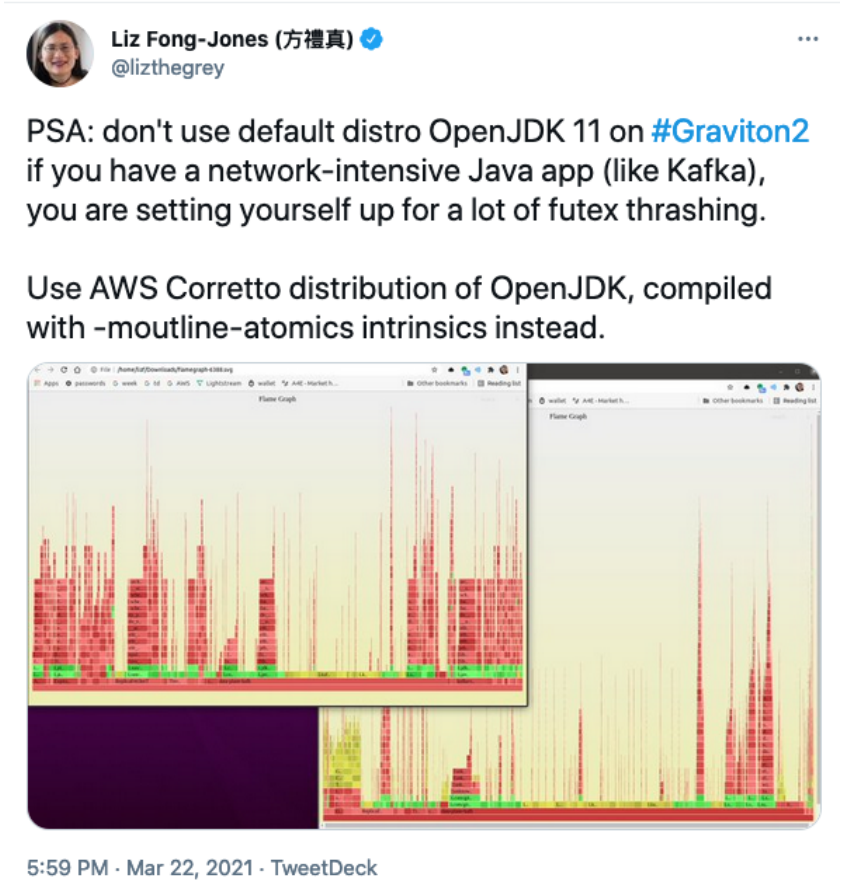
Overall, the incident lasted roughly 14 hours, including over 12 of them in a Zoom call, with the most visible impact being massive delays in some of our customers' data being processed.

CPU saturation

As the dust settled over the incident, we found ourselves with a cluster of nine c6gn.xlarge hosts rather than the initially planned six m6g.xlarge ones. We started analyzing what happened with more depth.

We found that our CPU usage was still higher than expected, even though our new instances were fine when it came to network and EBS capacity.

After experiments around kernel patches, packet processing issues, and various flame graphs, it was pointed out to us (by AWS engineers) that we were not using AWS's OpenJDK distribution, and thus that it was inefficiently using futex locking instead of atomics:

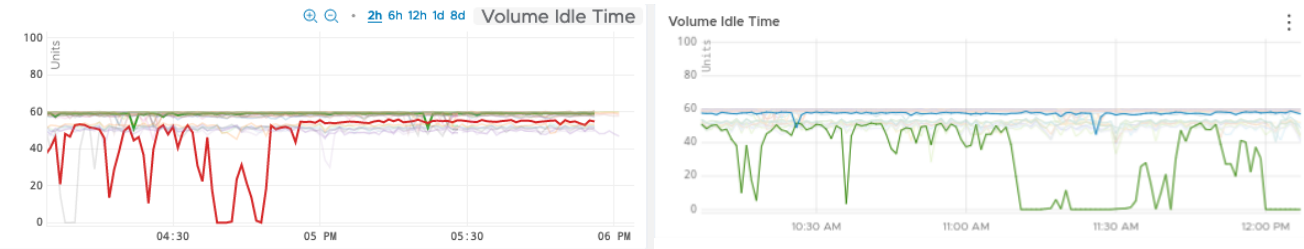


https://twitter.com/lizthegrey/status/1374118661338791939

Following the deployment of these fixes, CPU usage got back under control. Nobody at Confluent had really used production Kafka on Graviton2, and we knew the risks even if things otherwise looked like they were working out of the box.

Further EBS Issues

After a few days of stability, we started having hiccups once again. What we noticed was that one of our brokers leading a few partitions started getting heavy bursts of activity on its EBS volume (within allowed limits), to the point of being unresponsive. After 30 minutes, things came back to a non-critical state, but still with visibly degraded performance.



We took the leadership away from that broker to wait for a diagnosis. We were later given the explanation that the EBS volume "experienced higher than expected latency due to a hardware failure of the underlying storage subsystems," which was then remediated.

Another Kafka broker suffered a similar performance degradation the next day, but this time due to EBS internal partition serving our EBS volume having a significant high load. Safety throttles being triggered, despite being within allowed usage boundaries.

We started having concerns about how often this would be happening, and the sort of imbalance we would create in our clusters by constantly moving partitions around.

Then, as we were practicing migration logic to boot from EBS drives without having to replicate all the data across peers, we started suffering a never-ending series of outages where AWS would reap instances that were barely done booting up. In some cases those had messy interplay with Kafka itself, which brought back some hosts with a truncated log but advertised itself as ISR, and caused some data loss.

After days of constant firefighting, we made the decision to run an emergency migration to i3en.2xlarge instances with a stable local SSD; we had lost trust in the setup we had, attached some EBS drives to copy the data onto local SSDs, and re-stabilized the cluster. While we're still investigating what exactly made our intermediary set-up unstable, we now suspect whatever is behind it was behind the "weird" ASG behaviour and that the instances themselves were having issues.

We picked something bigger than we needed, which offered more stability promises both for normal and disaster situations, even when accounting the durability and persistence of EBS drives, which would have otherwise felt counter-intuitive without our spate of outages.

Lessons learned and things to keep in mind

Plan for bad days

We have a tendency to plan optimizations for steady state scenarios: In other words, *assuming things are right, how close to the edge can we bring performance?* In practice, our performance envelope should keep room for operational concerns, including asking:

- How easy is it to move one topic because of load issues?
- How easy is it to restart one broker without bad effects?
- How long is *needed* to move one broker's entire data set, and how long is *desired*?
- What is an acceptable recovery plan for a full AZ failure where one-third of our brokers vanish? Are the downstream effects tolerable?

While we can't necessarily do this evaluation for each and every component, we aim to answer most of these questions and have clear ways of tackling these situations.

Falling back to manual operations is always an option

When shit hits the fan and automation goes bad, you must have awareness and understanding of how to *manually* do what automation was supposed to do.

Manual partition reassignment keeps popping up as necessary, whether during near misses or actual incidents. Unless we are absolutely 100% sure that automation is correct and that there is no other way to do things, we have to consider the possibility of having to run steps manually when things are on fire.

Operating Kafka requires training and background understanding of how it works and how we structured our version of it. We should consider part of it as background material or simulation-worthy as part of on-call onboarding, rather than something to learn opportunistically.

Risky changes beget riskier changes

When change sets and updates are seen as risky, a lot of small improvements that are "nice to have" tend to get pushed back as they are not seen as worth the risk. Once a big enough change makes its way on the roadmap, we then bundle the small improvements along with it.

The risk with this is that the bigger the change sets are, the more variables are at play, and the more dynamic the conditions are going to be. In general, there are two approaches that sort of exist around this stuff:

- Keep bundling big changes together, but compensate by having extremely exhaustive testing environments and pre-production periods.
- Break down the changes into smaller, more frequent change sets that are more easily manageable.

This is a tendency that seems to happen everywhere in the industry, and keeping it in mind is going to prove useful.

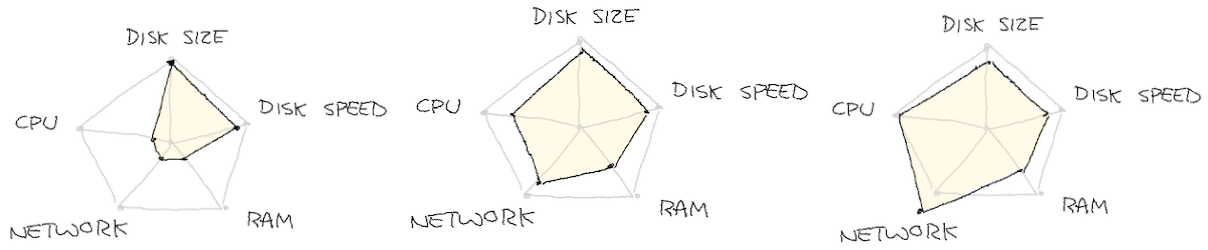
At Honeycomb, we believe the riskier pattern is one where we delay and bundle changes together without any of the safety mechanisms that usually compensate for these delays. We should find ways to force more frequent changes—even if they were artificial or dry-runs—to stay familiar with the operational context of our components.

In this case, many of the features we tried only became accessible with the new Confluent-Kafka packages, so releasing them *early* wouldn't have been possible; only delaying them. The bundling was surprising.

Performance and operational envelopes shift dynamically

Changing the way we run things means we change the performance envelope and operational pressures around them.

In this specific case, we have seen this with our profiles around the Kafka hosts we were running:



Moving from one operating point to the other carries the risk of new types of faults that we hadn't seen before, which were triggered by events that were previously acceptable and playing in the slack capacity we had. We often only discover where limits were when we hit a wall or break boundaries.

Whenever we make this sort of change, we should be prepared—or at least expect that such disruptions are going to be plausible, if not downright frequent. But we learned there are some steps we can take to help avoid something like this from happening again (and these were the steps we took as part of recovery in the following weeks), including:

- Allocating more resources than expected—e.g., bigger instances with more capacity.
- Counteracting the risk of hitting many sensitive pressure points at once by slowly clamping down on resources until you find the early inflection points, and then tackling those one at a time.

We ended up doing these things implicitly when re-stabilizing the system.