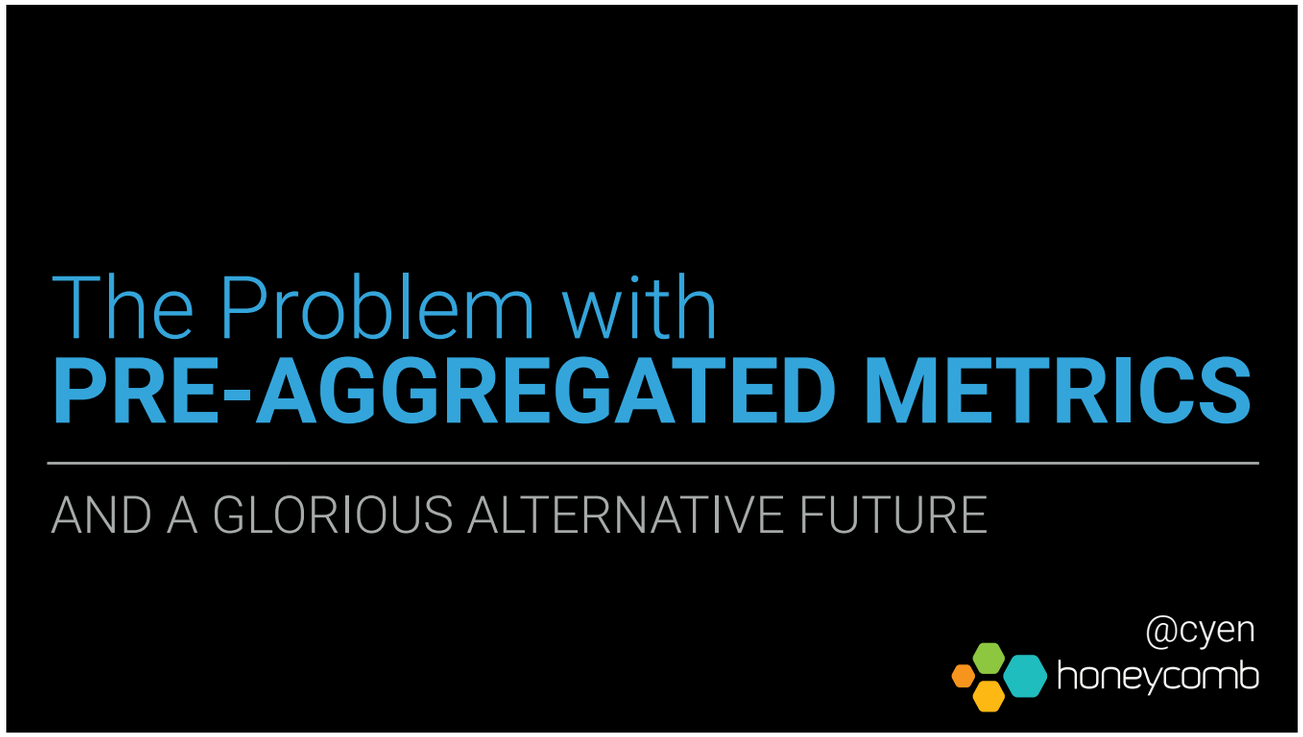
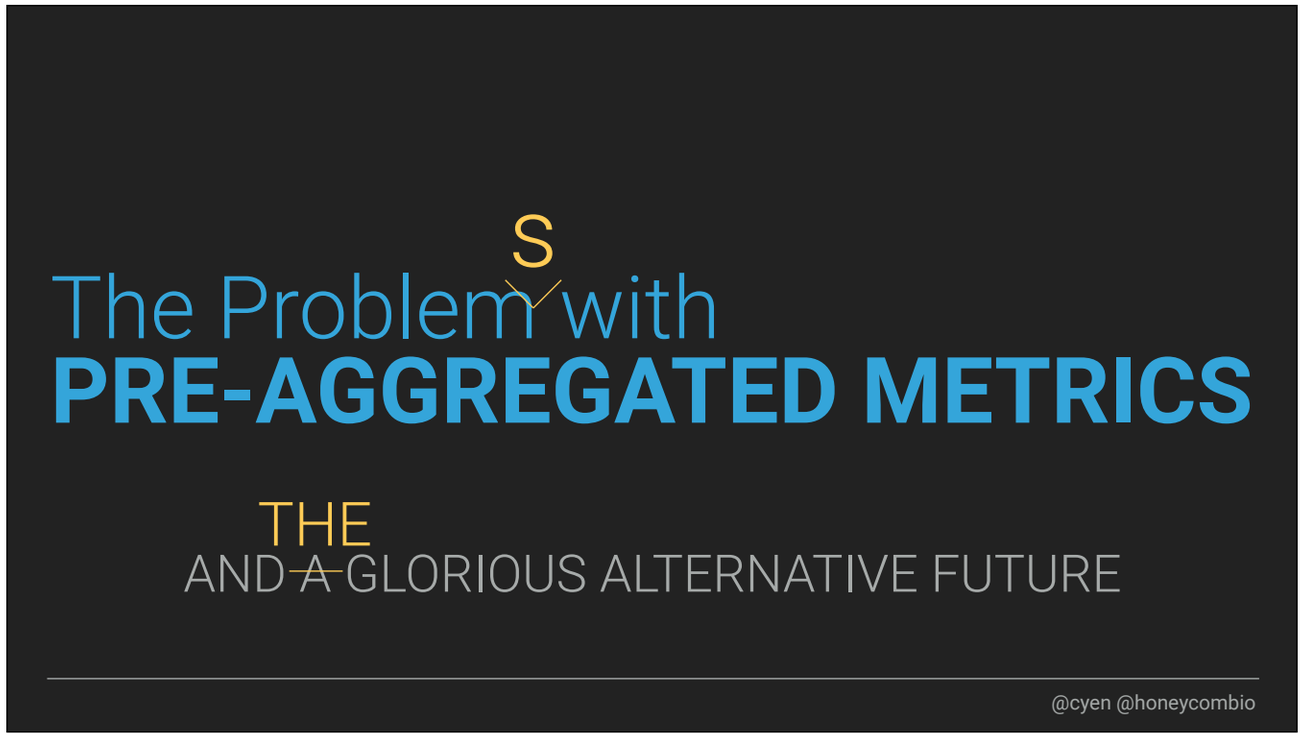
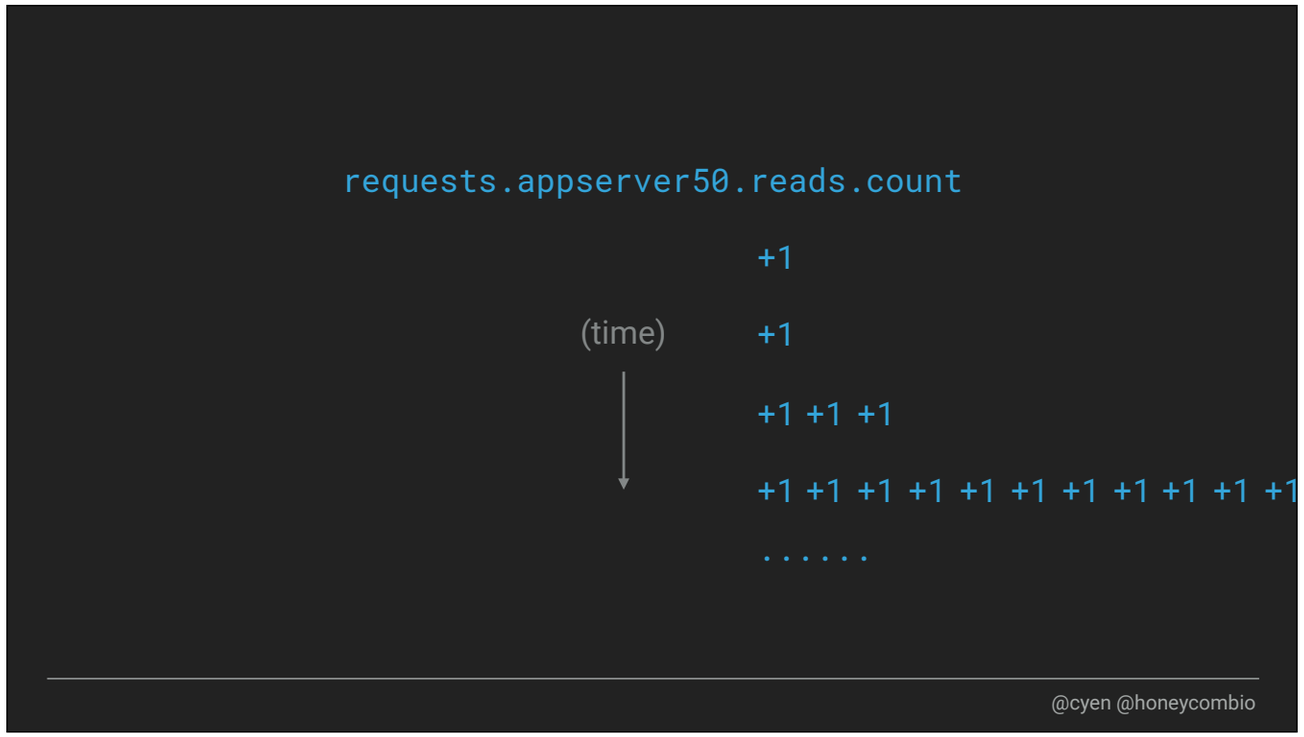


Hi! I'm Christine, with Honeycomb, and I have a real problem with pre-aggregated metrics.



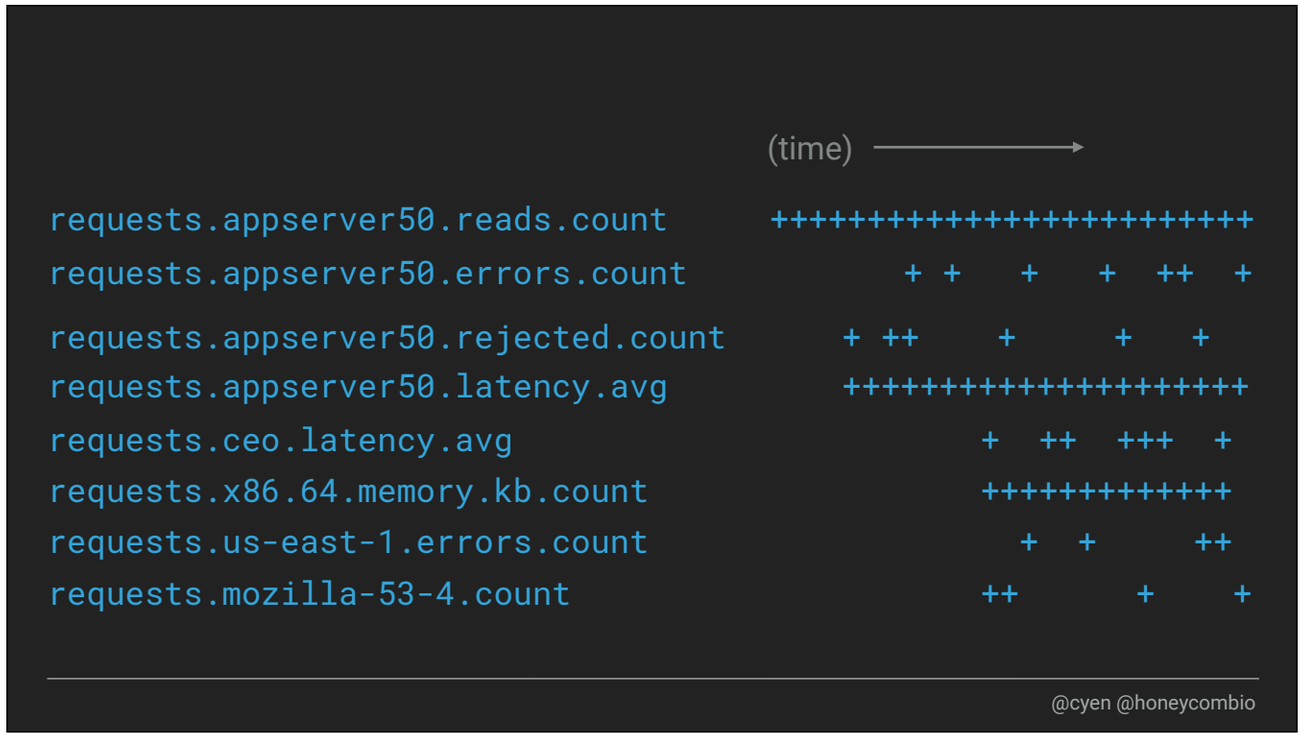
In fact, let me rephrase. I have several problems with pre-aggregated metrics, which I'll run through today, and I have a single vision of the solution to all of our problems. All of my problems, anyway.

Background: I spent a few years at Parse, a Backend-as-a-Service, building out an analytics product that relied on pre-aggregated time series metrics. That decision resulted in a lot of pain and extra work for myself and other engineers supporting the product from operational and customer-support perspectives. This talk is my penance.



I want to start with a disclaimer. Pre-aggregated metrics can be great. They're simple to reason about, simple to add, and can collapse an incredible amount of traffic into some very compact forms…
If your problems are very well understood and very few new problems tend to crop up in your system, then maybe they're all you need. But that's very rarely reality...

Because they're so easy to think about, and so useful when you have the perfect time series in your back pocket,

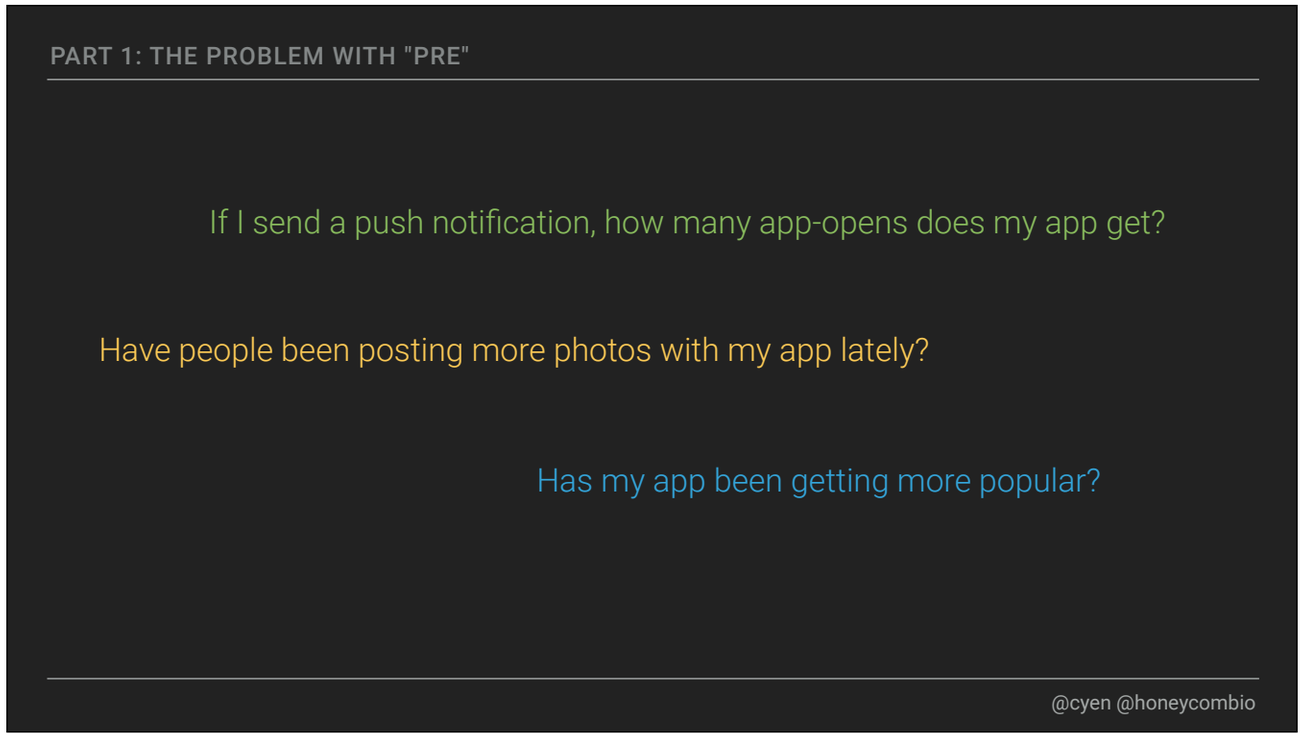


it's incredibly easy to keep adding one or two or ten new metrics to your dashboard. …. But soon you’ve got more metrics than you know how to handle, and you can run into a wall in your understanding of how to use them if you aren't careful.

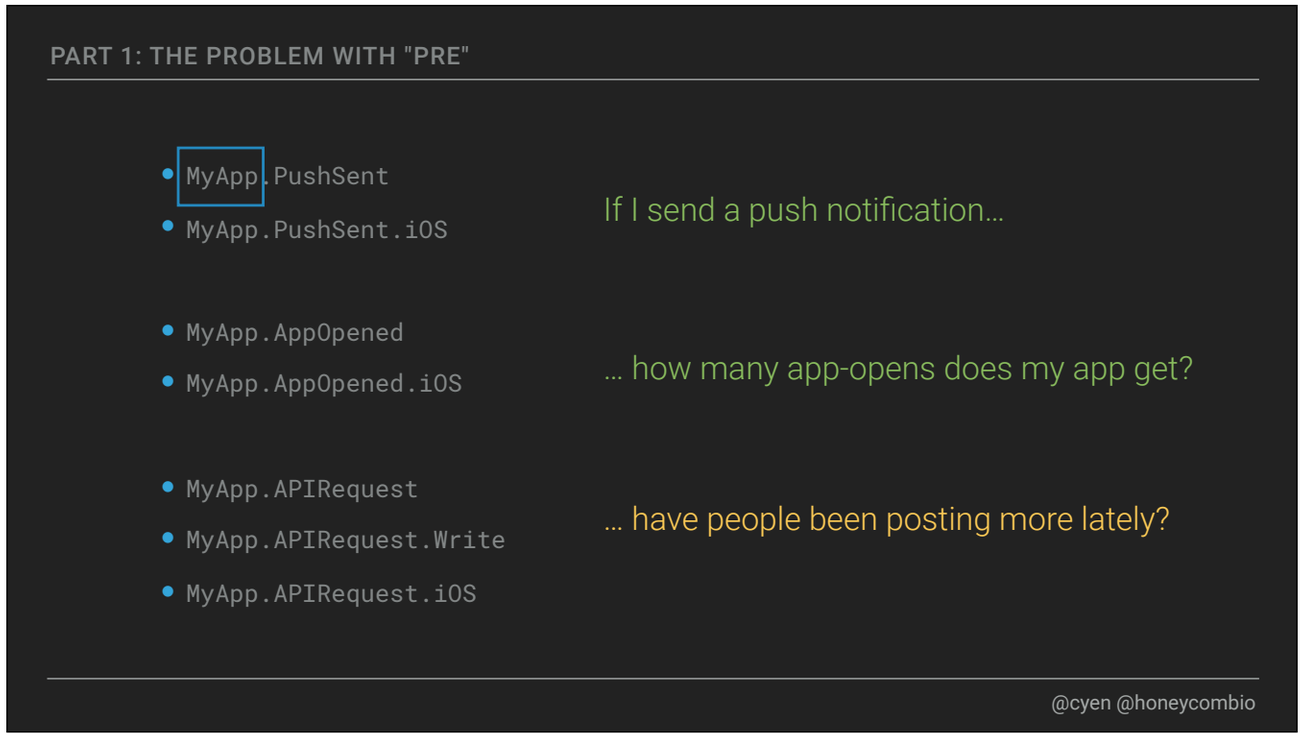
The Problem with PRE-AGGREGATED METRICS

@cyen @honeycombio

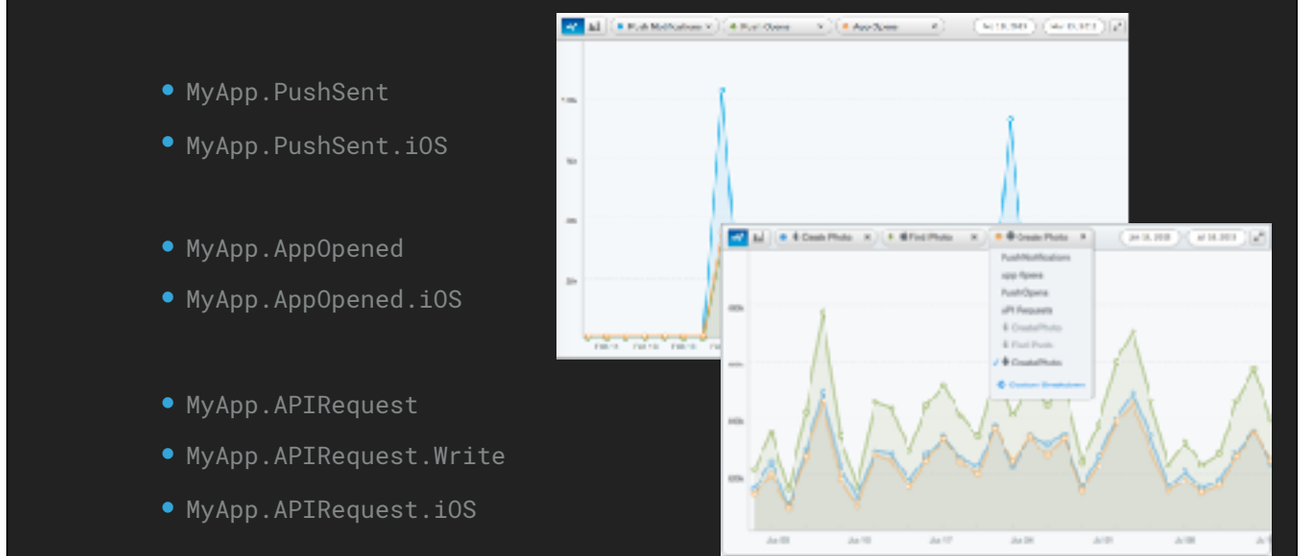
This talk is about all of the ways I've managed to run myself into walls (or seen our customers run themselves into a wall) trying to bend pre-aggregated metrics to my will, and why i now think they're a problem rather than a solution.



So. The whole thing started when we wanted to build a brand-new product for tracking push notifications and api requests passing through our system.



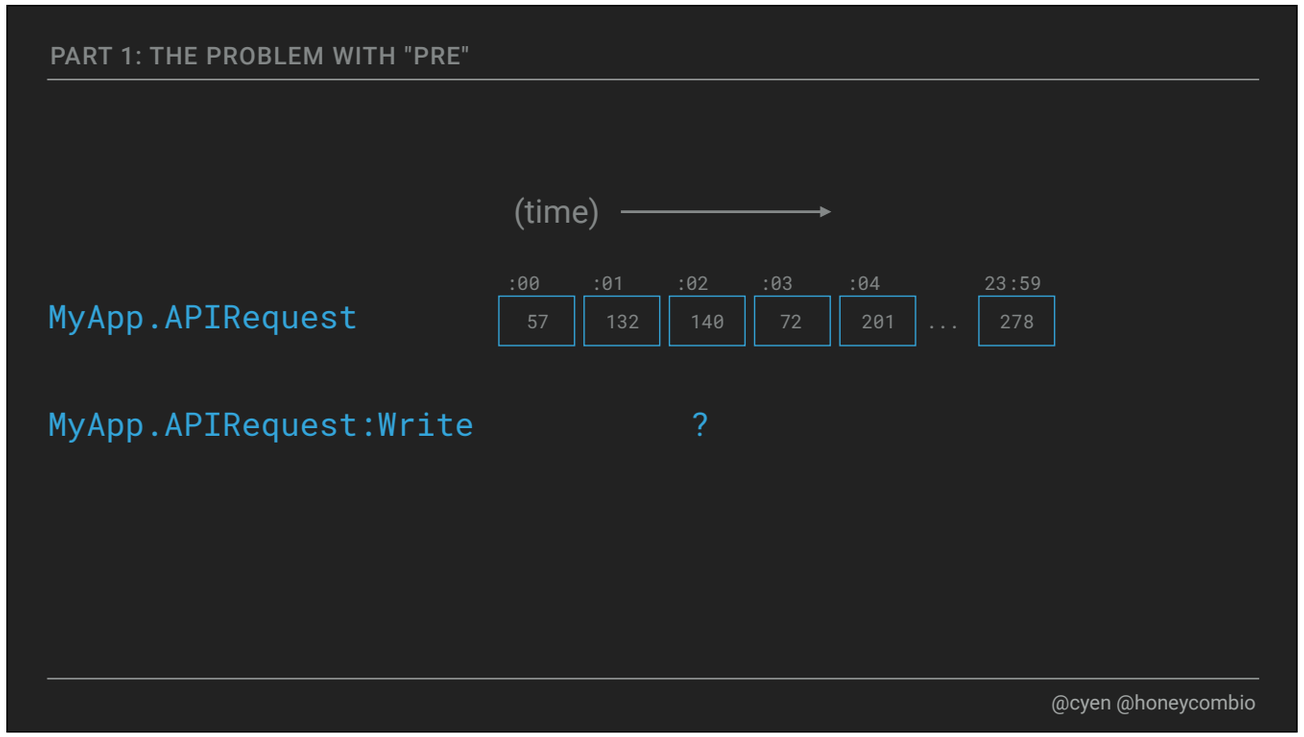
this was a pretty well-defined problem at first. we were a platform so were used to architecting for multitenancy, and we knew what sorts of questions we wanted to be able to answer. this basically required us to track a finite handful of metrics per app -- totally reasonable.



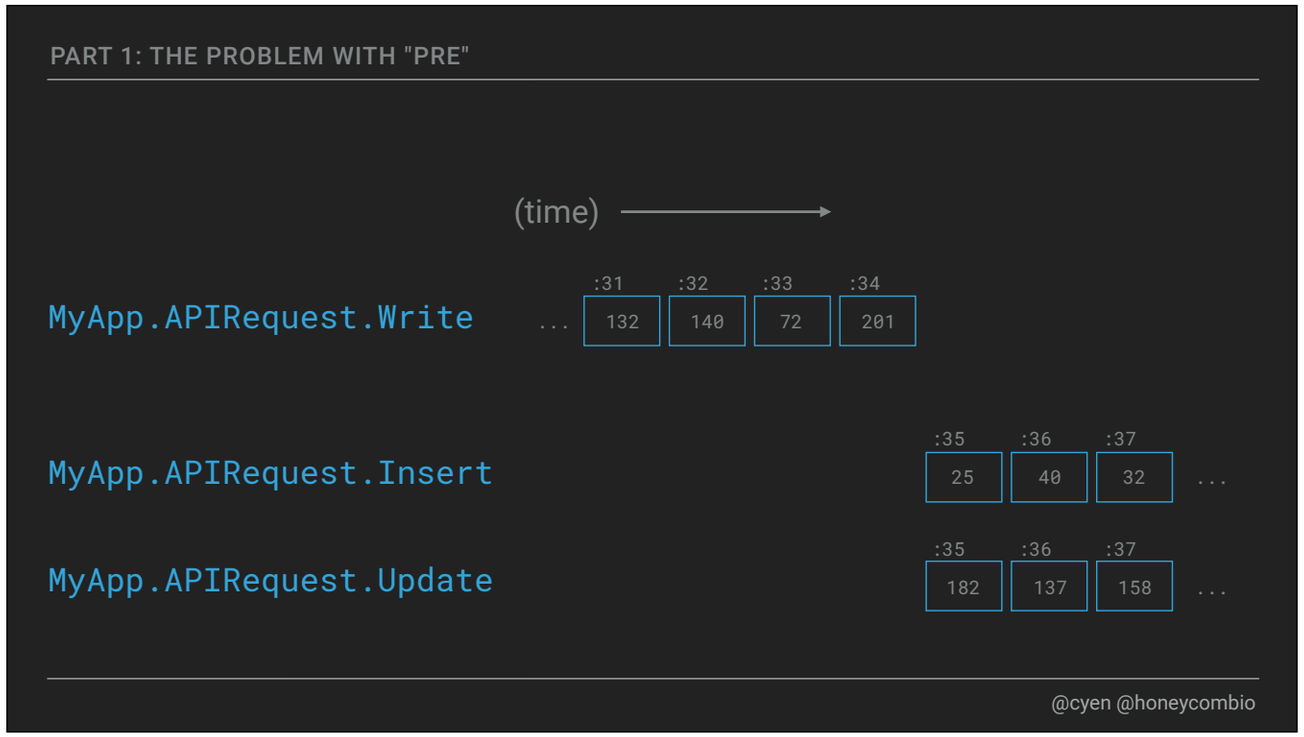
- With these metrics being incremented in per-hour buckets, we were able to drive a gorgeous UI that showed an app's overall traffic over time, how often their push notifications resulted in app opens, the whole shebang.

[PRE] pre-aggregation is also known as write-time aggregation — it means that you define a set of counters you'll keep track of and increment accordingly, and as data flows in, those counters get incremented. Like a set of tally sheets.

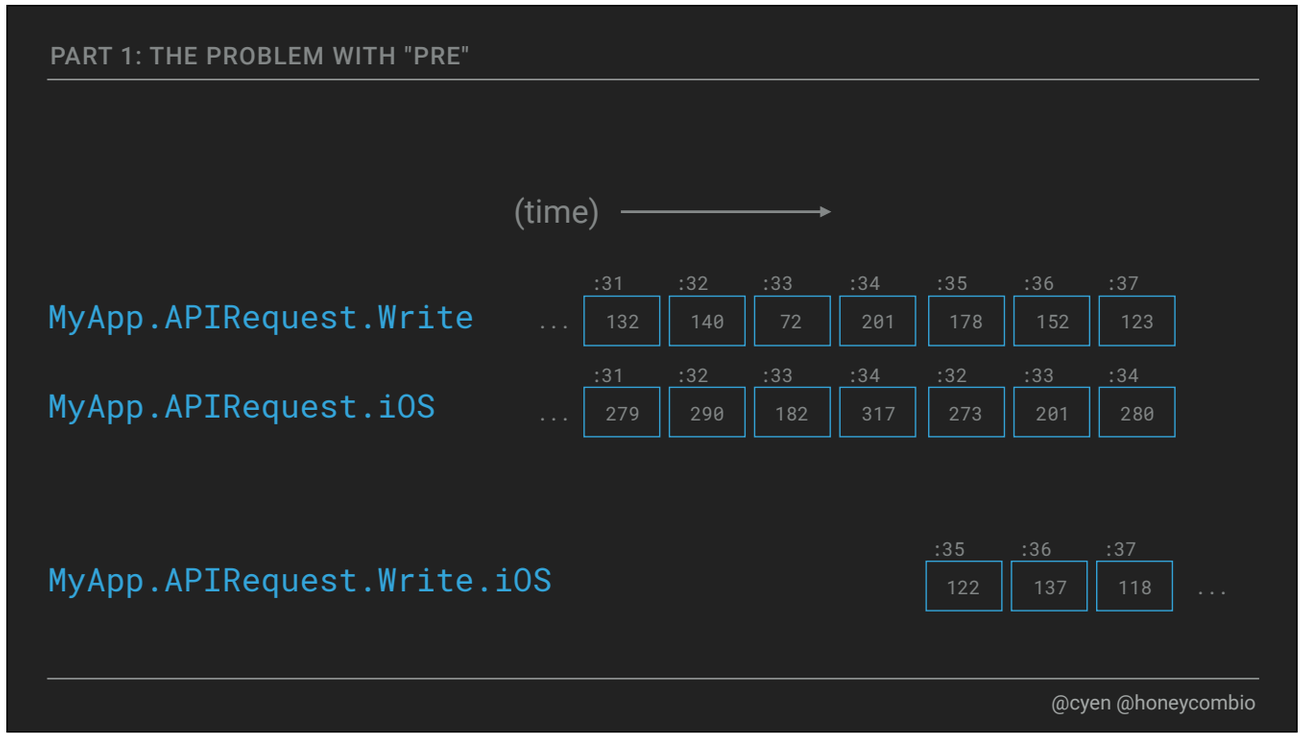
We chose a pre-aggregated approach for a product like this because we wanted to optimize for a compact representation on disk and really simple, speedy reads (if we're just incrementing counters all along, reads just involve plucking a single number (or row of numbers) off disk.)



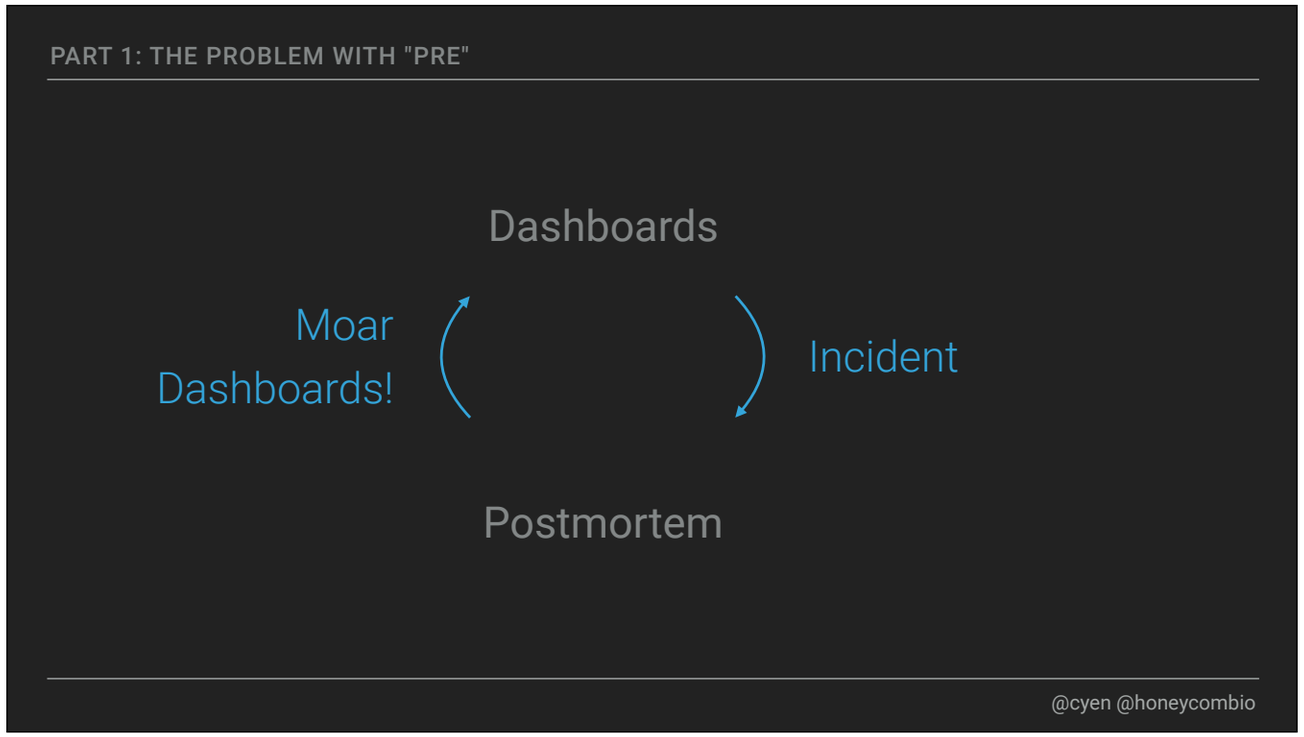
but the fact that we chose to PRE-aggregate these individual API requests/push notifications on the write path meant that it was a pain each time to track something NEW in the system. Having to pick the specific counters to be incremented up front meant that we were agreeing to be constrained by that predefined set of counters.



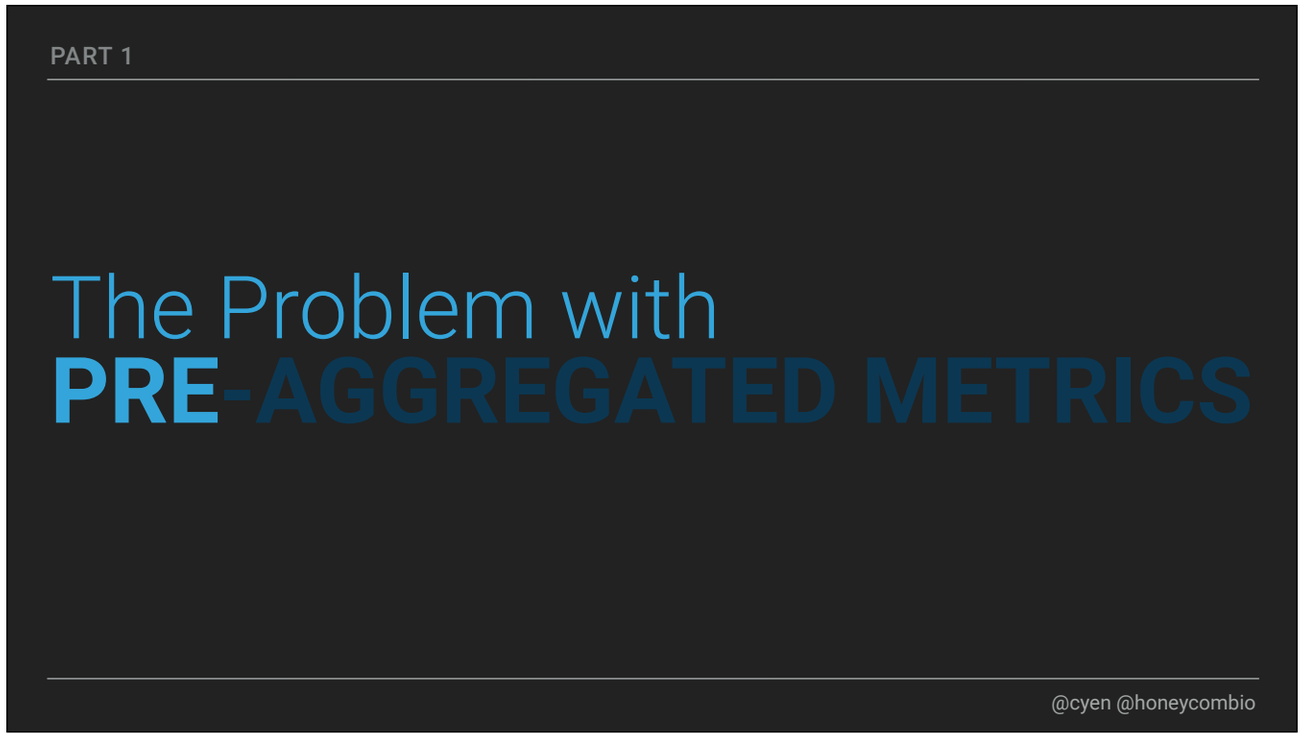
trying to redefine something is just a giant pain (expanding a write into an insert vs update means we have essentially two completely new metrics, with no past data to lean on)



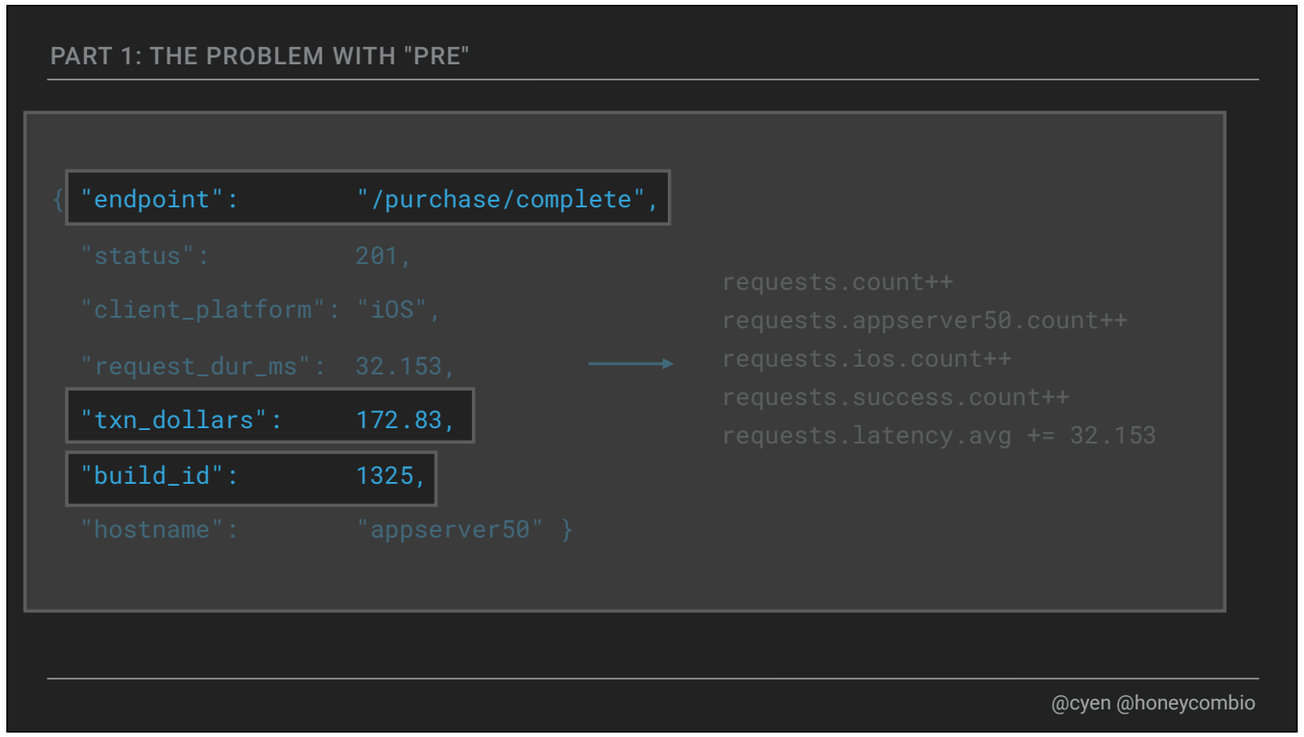
... and in some cases, even when we **were** sending all of the relevant pieces of data, if we hadn't thought to track the cross product of two metrics, we were - again - basically back to square one.



With pre-agg metrics, you’re constrained by the questions that **past** you happened to ask.
And the behavior pattern we fall into, is to end up in a loop of using dashboards to debug… -> incident… -> postmortem… -> more dashboards to scroll through.
=> your dashboards end up being artifacts of past failures, because you’re trying to predict... what you know has already happened. you’re constantly playing catchup.
[05:00]



so this is the first part of my problem with pre-aggregated metrics: the PRE. PRE-defining, PRE-constraining, PRE-cognition. Not being able to ask new questions because of how everything's collapsed at write time.



PRE-determining what counters might be interesting to increment means that… this super-interesting API request (represented here as a bag of keys and values) gets collapsed into to a tiny handful of individual counters.
(and, because pre-aggregation requires us to be so explicit about defining **which** counters to increment, we're only capturing a subset of interesting things on that request. just look at what we're dropping entirely!)

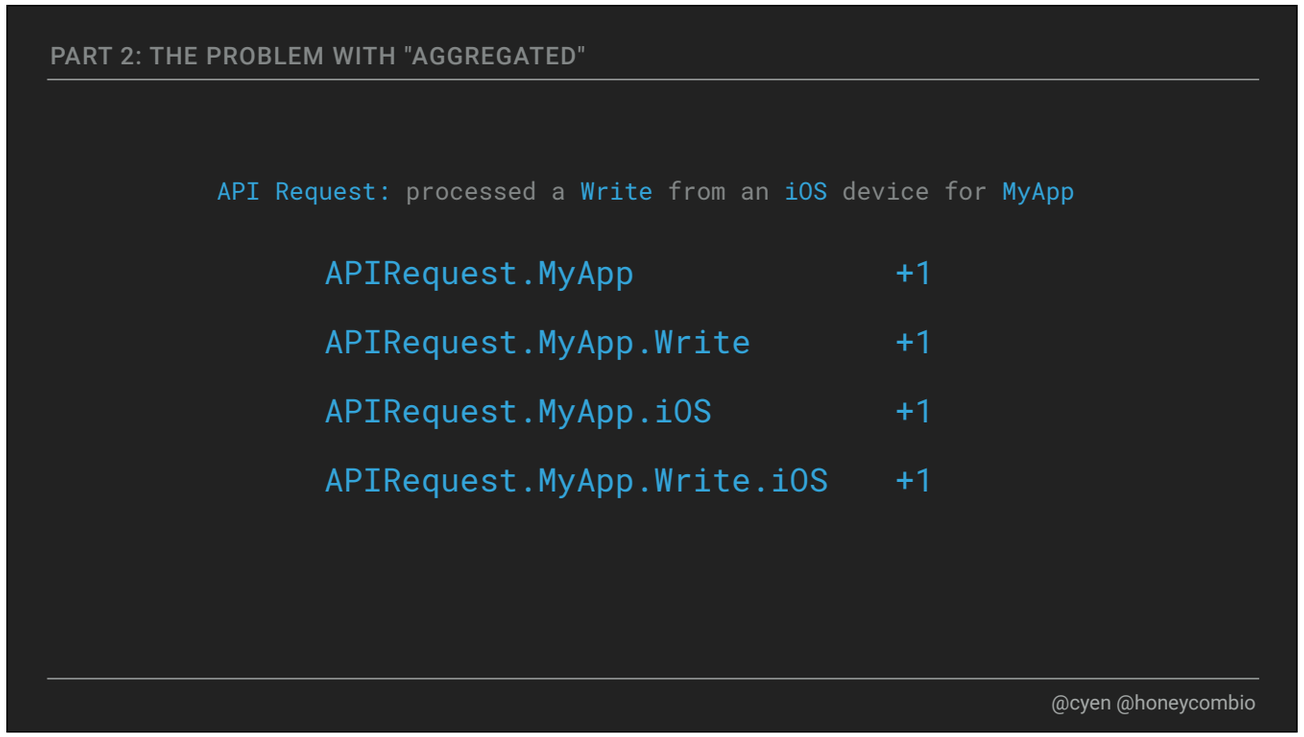
PART 2

The Problem with PRE-AGGREGATED METRICS

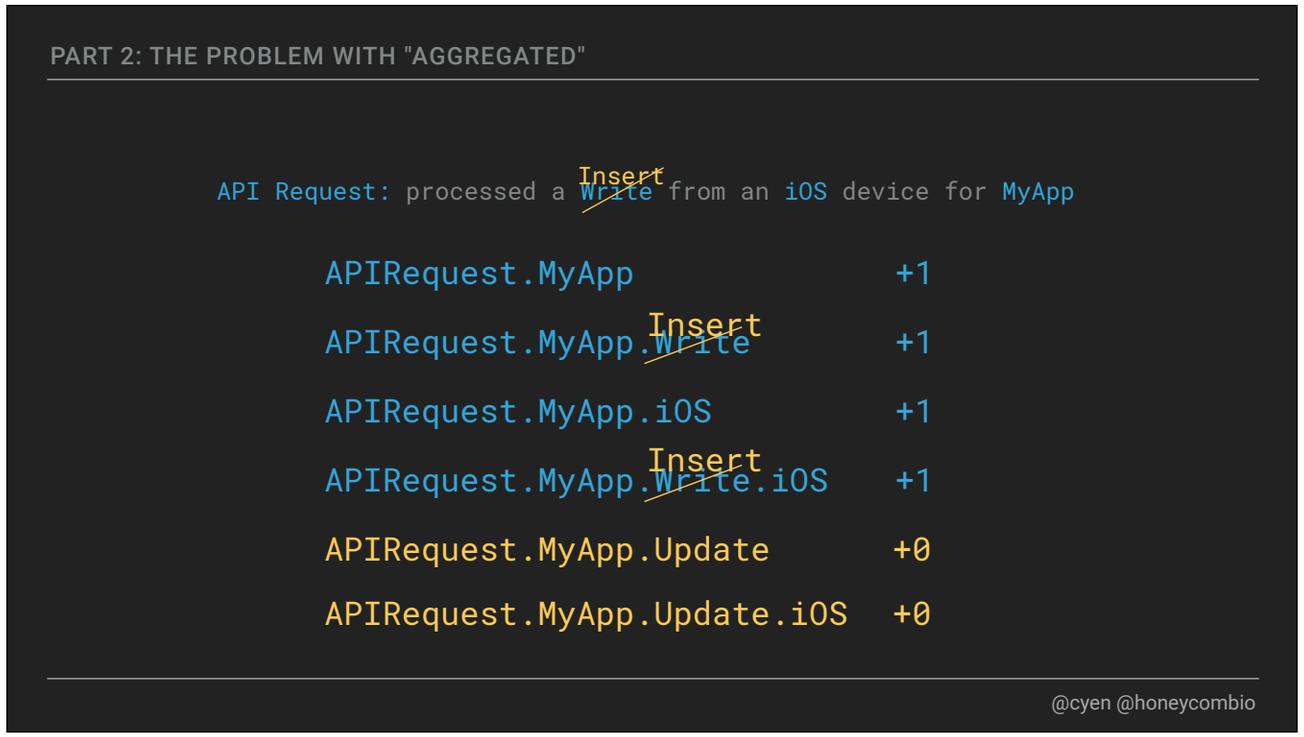
@cyen @honeycombio

hokay. part two - i also have a problem with the AGGREGATED part of pre-aggregated metrics.
Besides the **model** of pre-aggregation fighting us, the **implementation** of them does, as well.

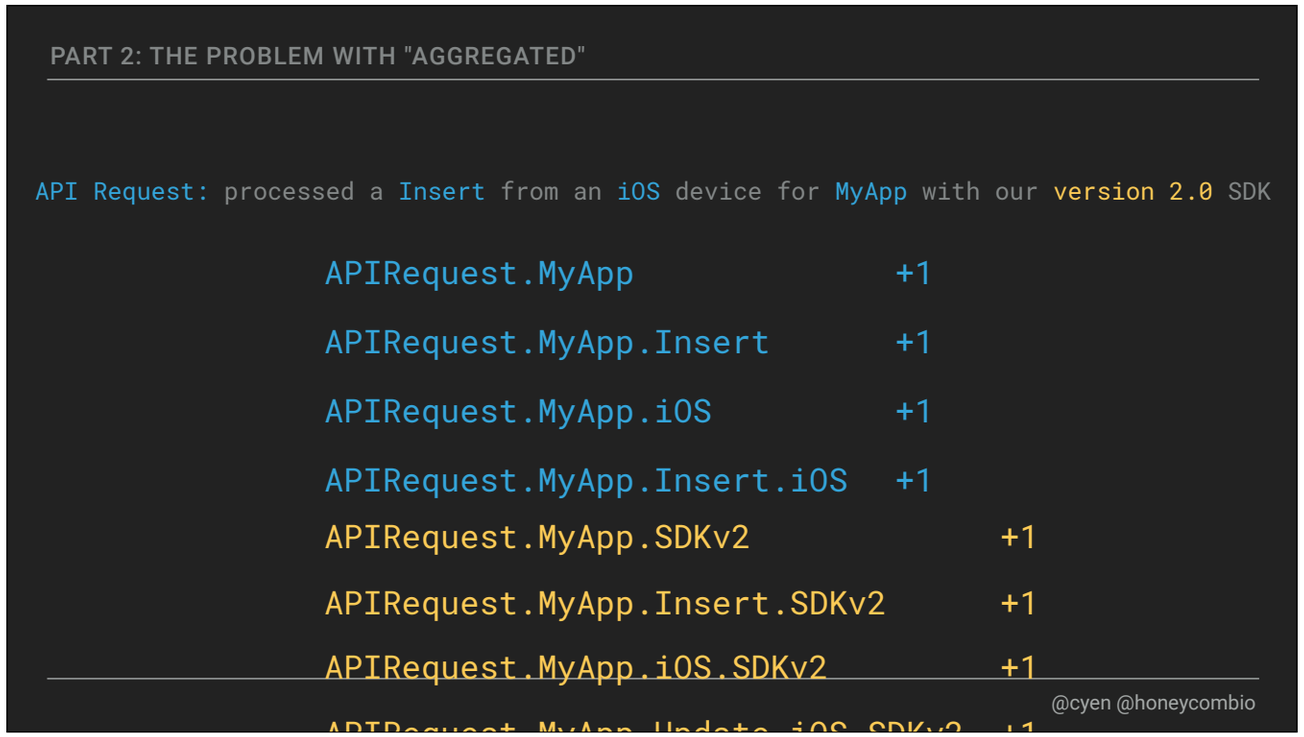
As a little bit of background, the way that pre-aggregation generally works under the hood, some "thing" happens in the wild. This "thing" can typically be represented as a multidimensional set of keys and values.
For example, an API request or database call can be represented by a set of measurements (latency, payload size, etc) and attributes (hostname, method, success).



Typically, the designers of some pre-aggregated metrics system will define a set of metrics to track, like we did above. Attributes are carefully sliced out to define metric names, and then any number of the metrics that describe some part of this multidimensional bag of keys and values will be incremented accordingly.

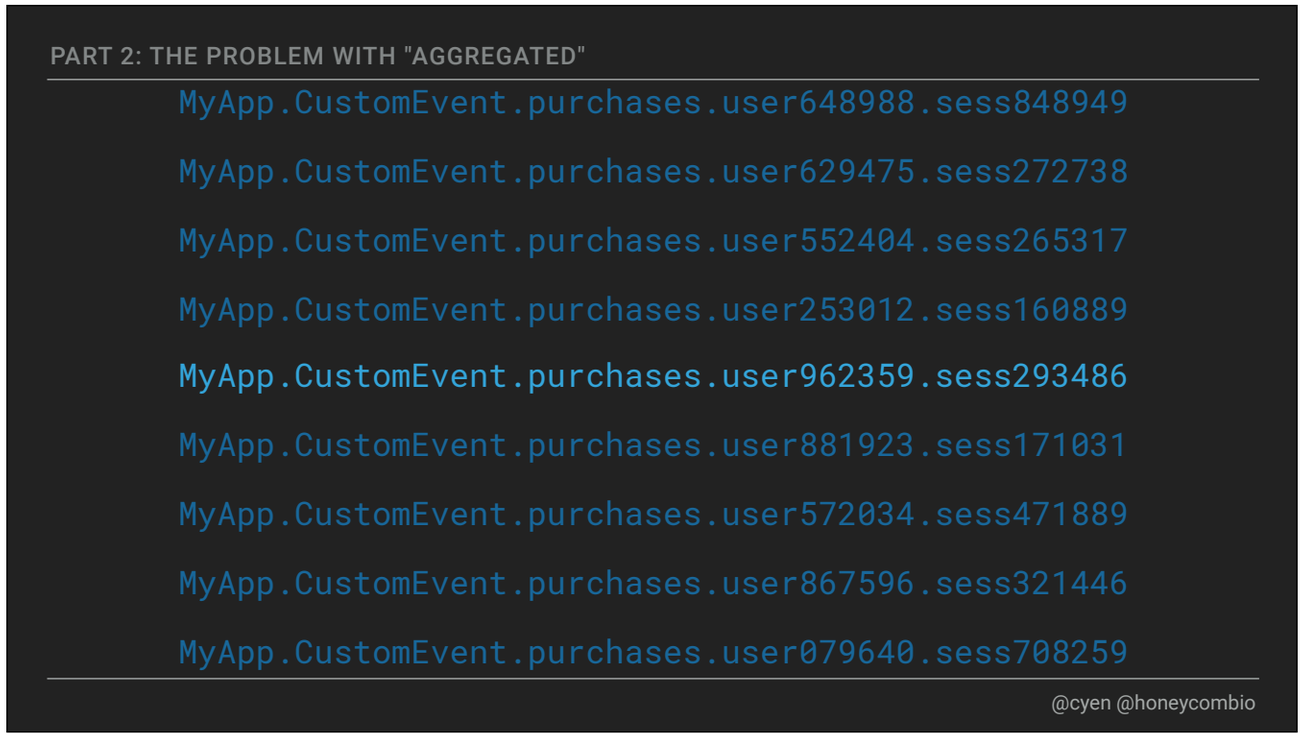


So. When we were architecting our push and API analytics solution at Parse, we were somewhat concerned with this fanout... but not that much, because we were very familiar with the sorts of things being stored and even if we ended up having to replace a "write" time series with "insert" and "update" time series, the change was… manageable.



It was even manageable if we wanted to add a new dimension entirely to these time series - letting us break any of the preexisting time series down by, say, app version. Increasing the number of key/value pairs tracked for a given event resulted in exponential growth. But it was ok! Because we were firmly in control of the sorts of keys and values we were sending and the world was predictable.

... until we decided to throw that all out the window and let customers define custom metrics. [09:00]



we set a limit on the number of total key/value pairs we'd let customers track for a given event, but didn't set cardinality limits up front. so by aggregating these time series at write time, **and** by accepting arbitrary user input, we were suddenly exposed to potentially incredibly expensive traffic.

PART 2: THE PROBLEM WITH "AGGREGATED"

WAYS TO EXPLODE YOUR PRE-AGGREGATED METRICS STORAGE

this style of write-time fanout (and aggregation) meant that the problem, then, is that the number of unique metrics directly affect the amount of storage needed, **not** the underlying volume of traffic. And there are two stupidly easy ways to significantly increase the amount of storage for a system like this:

WAYS TO EXPLODE YOUR PRE-AGGREGATED METRICS STORAGE

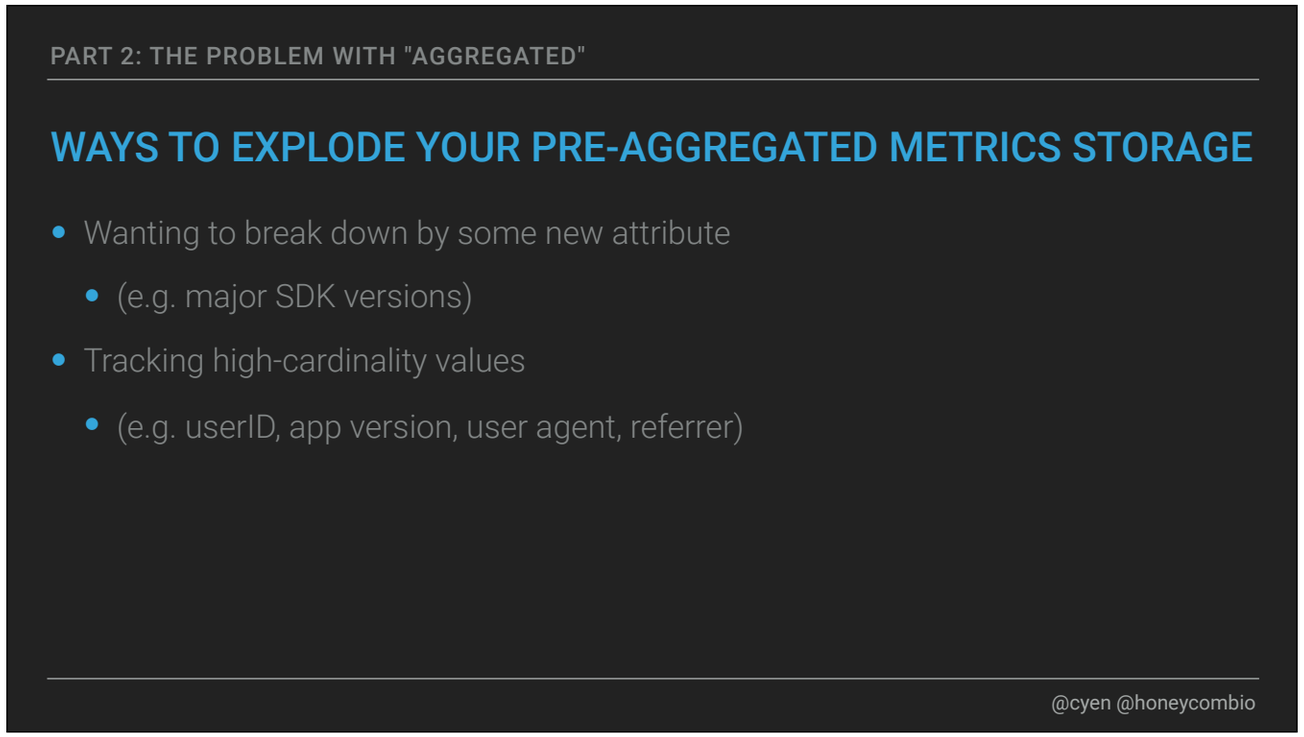
- Wanting to break down by some new attribute
  - (e.g. major SDK versions)

1) realizing that some new important attribute needs tracking. adding new attributes to a set of dimension names may result in a combinatorial explosion of metrics to increment

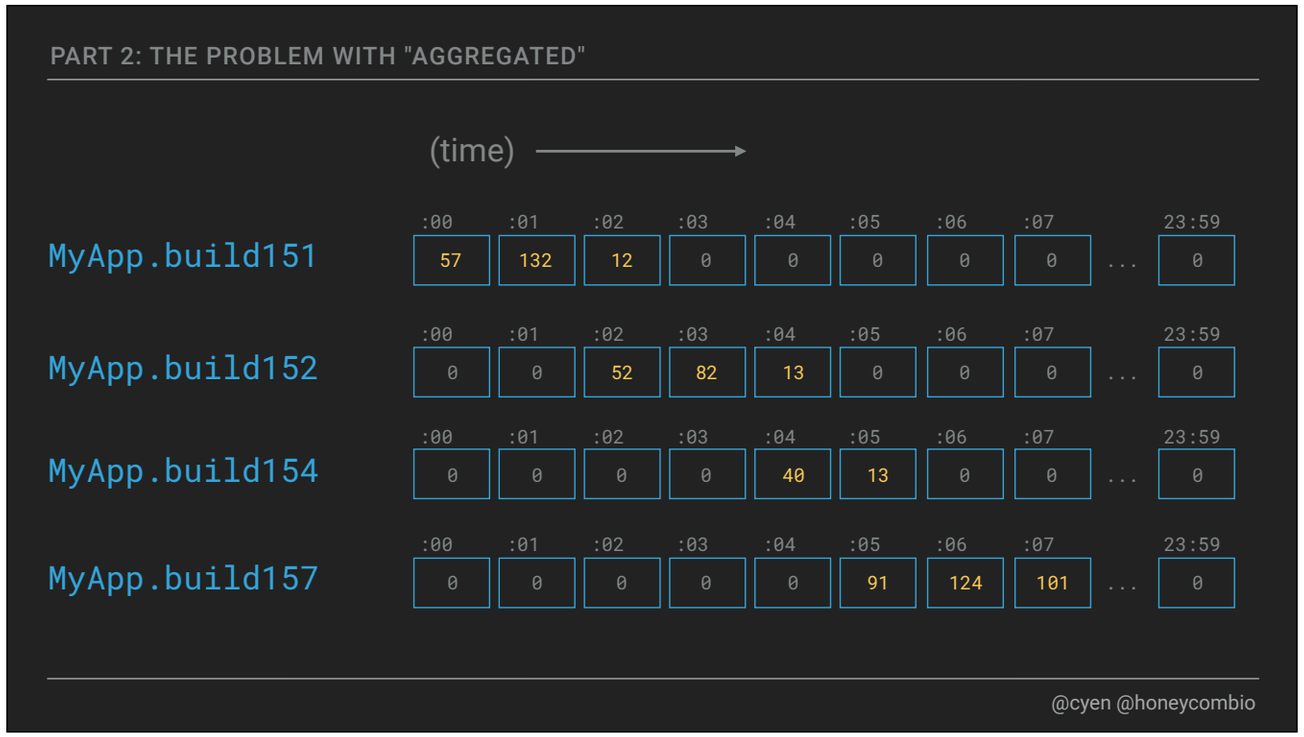
WAYS TO EXPLODE YOUR PRE-AGGREGATED METRICS STORAGE

- Wanting to break down by some new attribute
  - (e.g. major SDK versions)
- Tracking high-cardinality values
  - (e.g. userID, app version, user agent, referrer)

2) ... or, intentionally or unintentionally, tracking an attribute with a very large set of potential values (otherwise known as high cardinality). Common high-cardinality attributes here are things like user ID, app version, user agent, or referrer -- and if the storage system is allocating space to collect a time series for **each** discrete value, well. whoops.

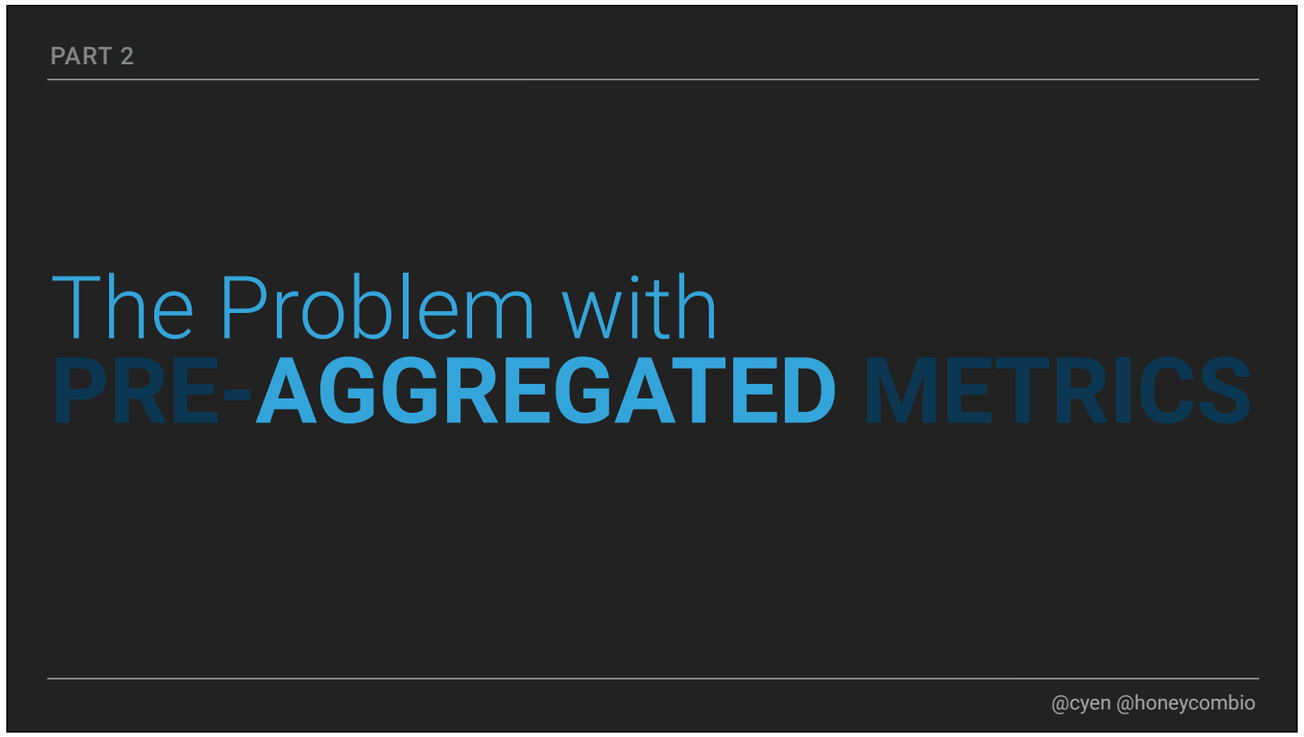


and - as a reminder - why might these sorts of high-cardinality attributes be useful? being ableto isolate a single app's traffic and still get the same averages, percentiles, and breakdowns... lets you drill down to and understand a single terrible segment of your traffic. it's very difficult to do something like that with a pre-aggregated metrics tool because of these cardinality storage constraints.

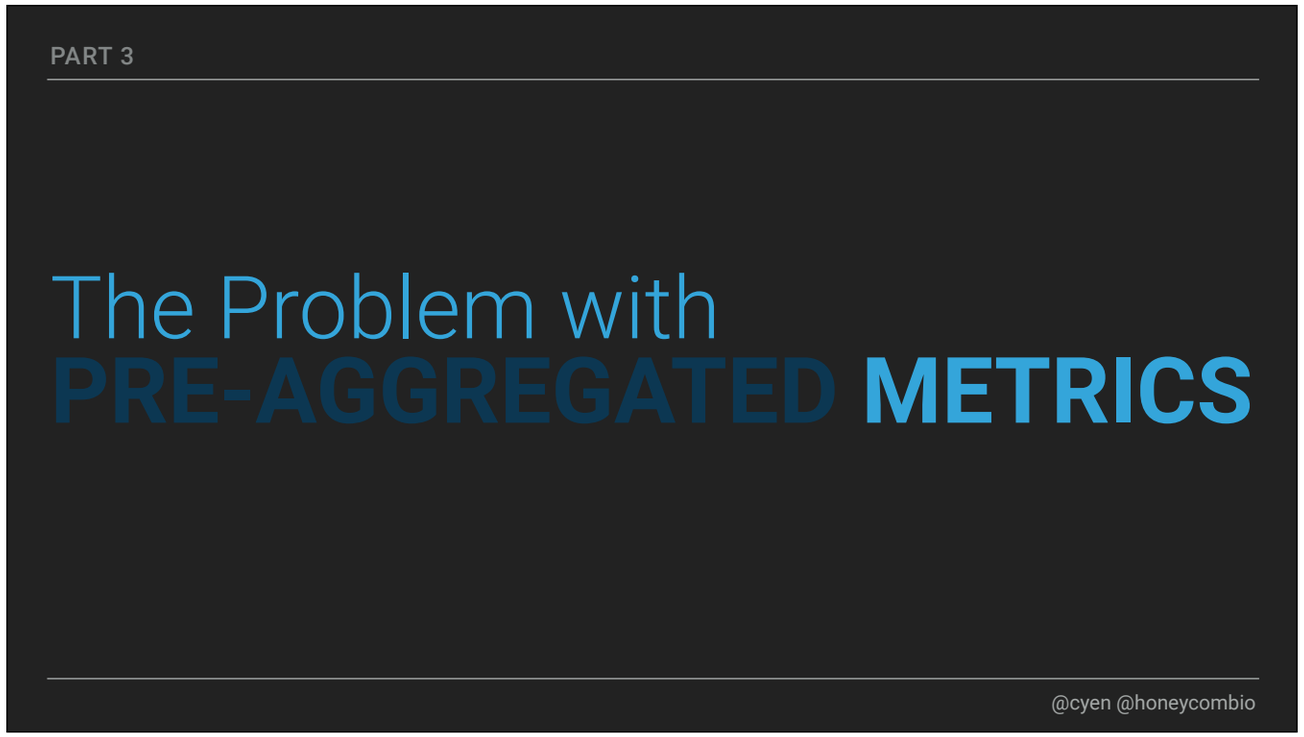


even some key/value pair with a monotonically increasing value would be difficult to support like this. If, say, your system relies on allocating a week's or a month's worth of storage for a time series at once, including something incredibly useful and incredibly volatile like the build ID in the metric key could result in a ton more storage than expected.

try as you might, whatever constraints you want, it’s still stupid easy to accidentally cause this to happen. (e.g. logging string addresses instead of strings)



and this is the truth — when building a preagg metrics product, the problems inherent in the implementation make it so that it's never going to get easier to deal with high-cardinality values.
This storage strategy - while super-efficient when talking about high volumes of data - relies strongly on the number of metrics being, well, countable. Once your requirements cause that number to inflate, you’ve got some hard choices to make.

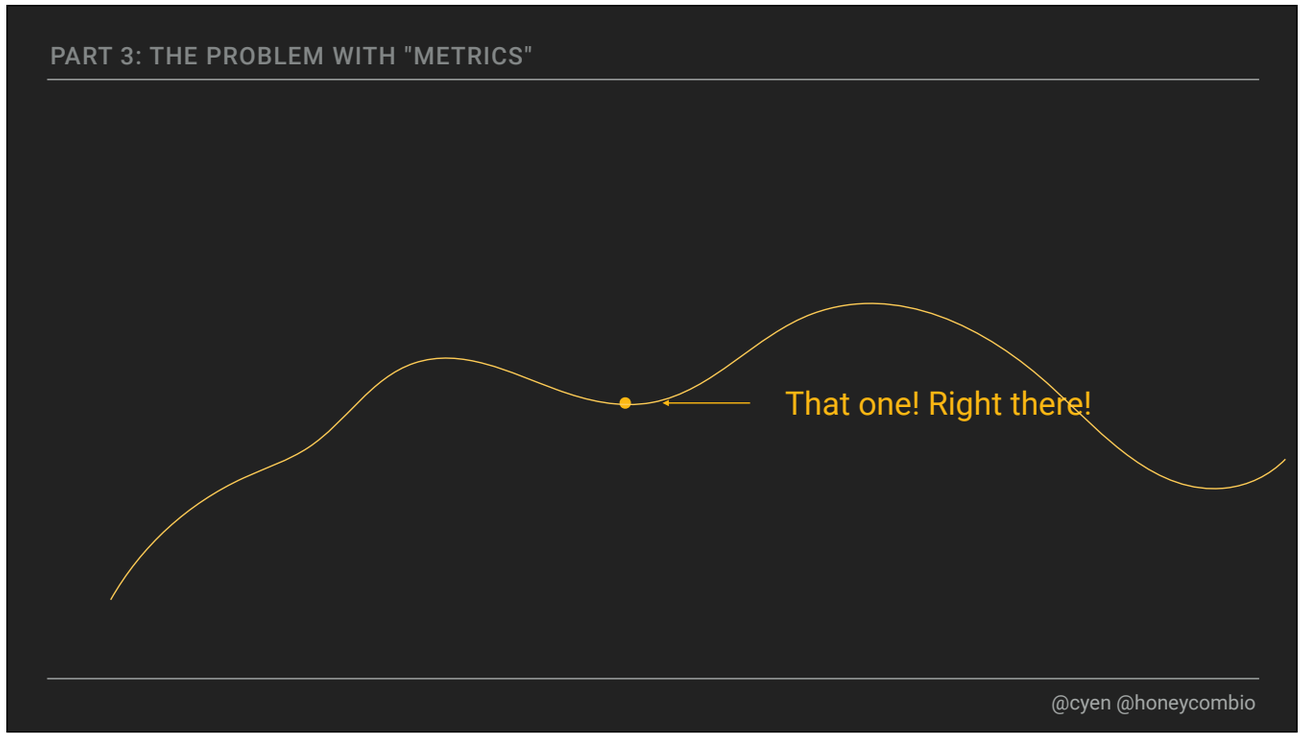


and finally, the last part of my problem with pre-aggregated metrics: the concept of a metric itself. [14:00]

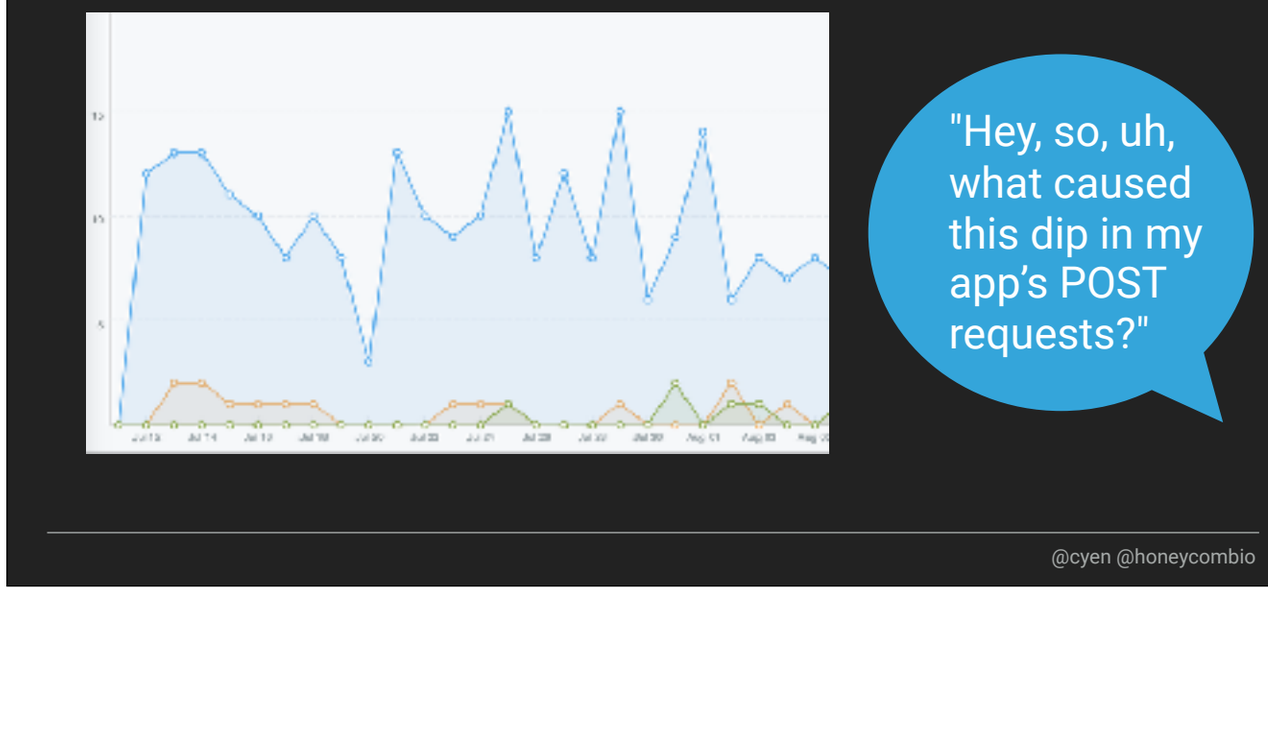
METRIC:

A single measurement, usually tracked over a period of time.

for our intents and purposes, we define a metric as:

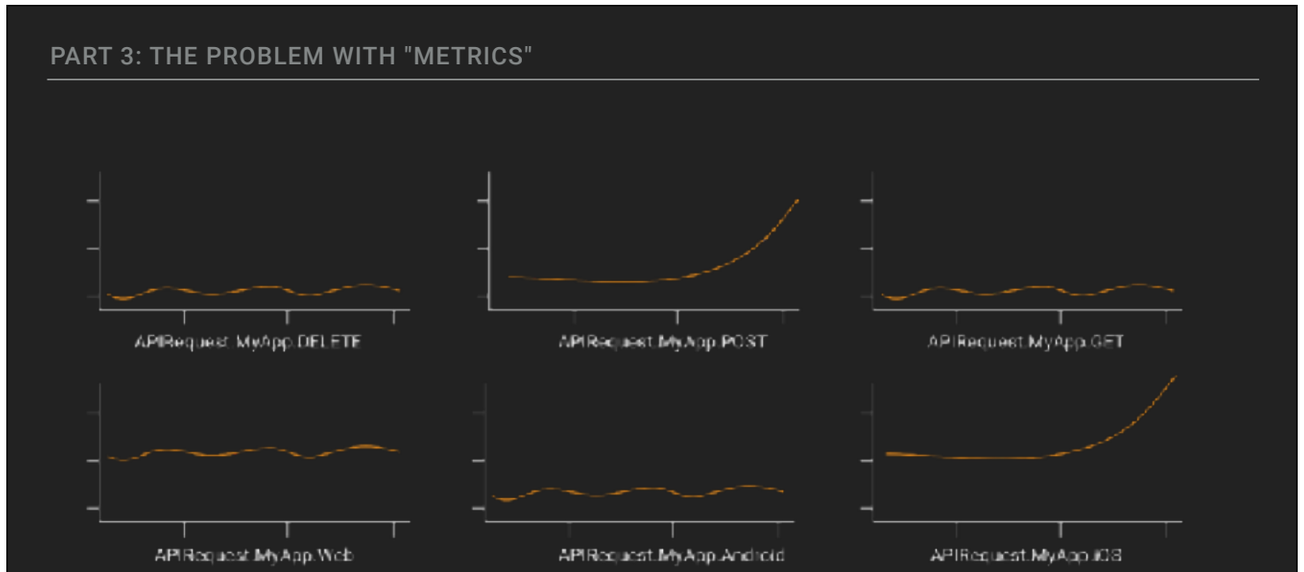


the problem with metrics is just that -- they’re individual, they’re isolated, they’re standalone dots in the big picture of your system. And looking at the progress of standalone datapoints over time means that, while you can get a **sketch** of reality over that time, you’re never really able to reconstruct reality to identify what **actually** happened.

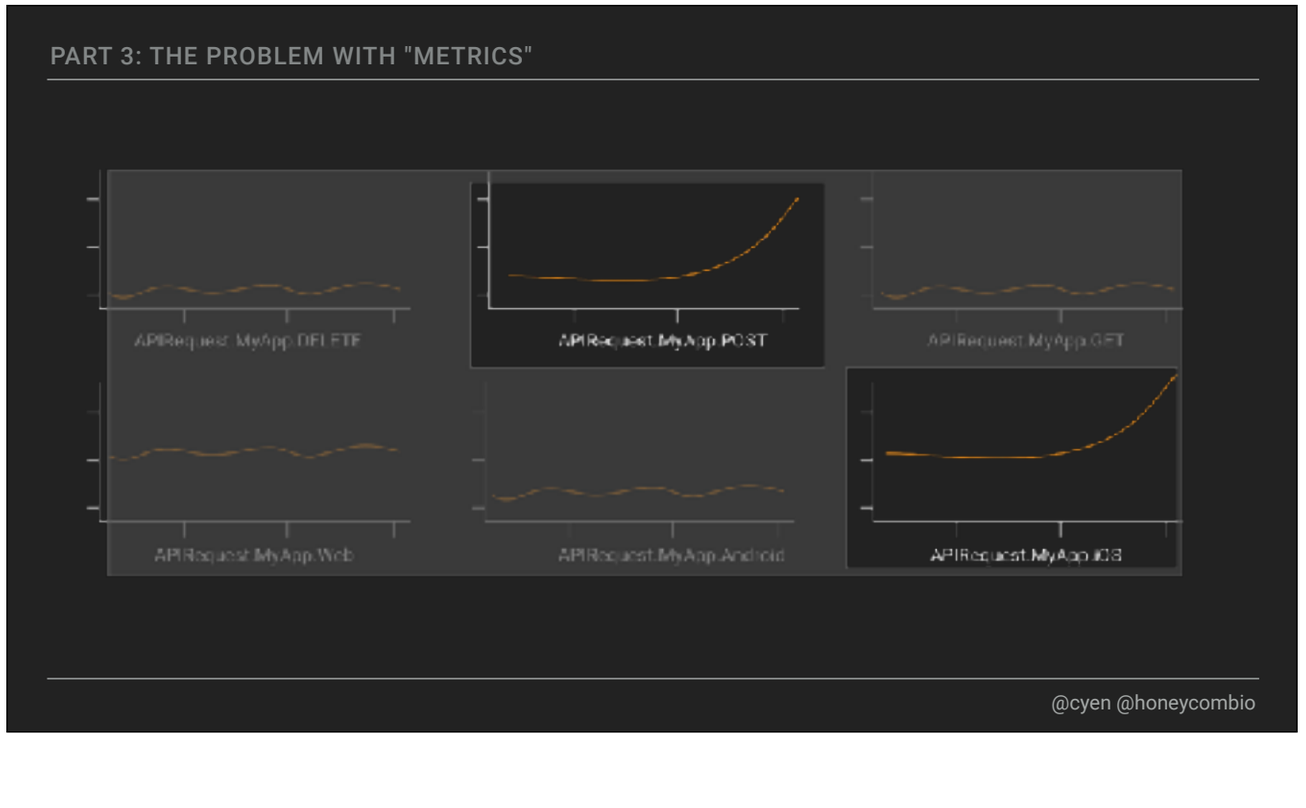


Whoa, we got a little abstract there. What am I really trying to describe?

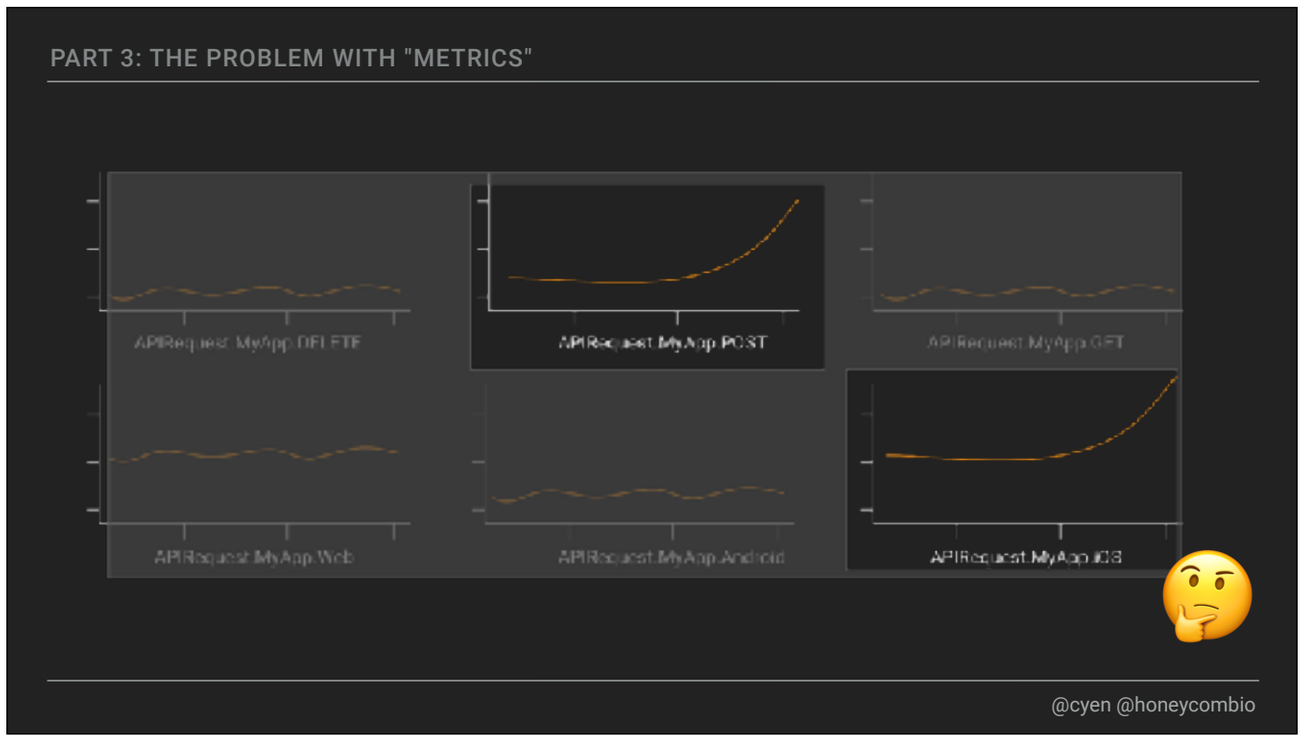
Let's rewind a little bit. When we released this lovely API analytics product, we committed to supporting our developers in any way we could. Which often meant answering as many of their questions as possible.



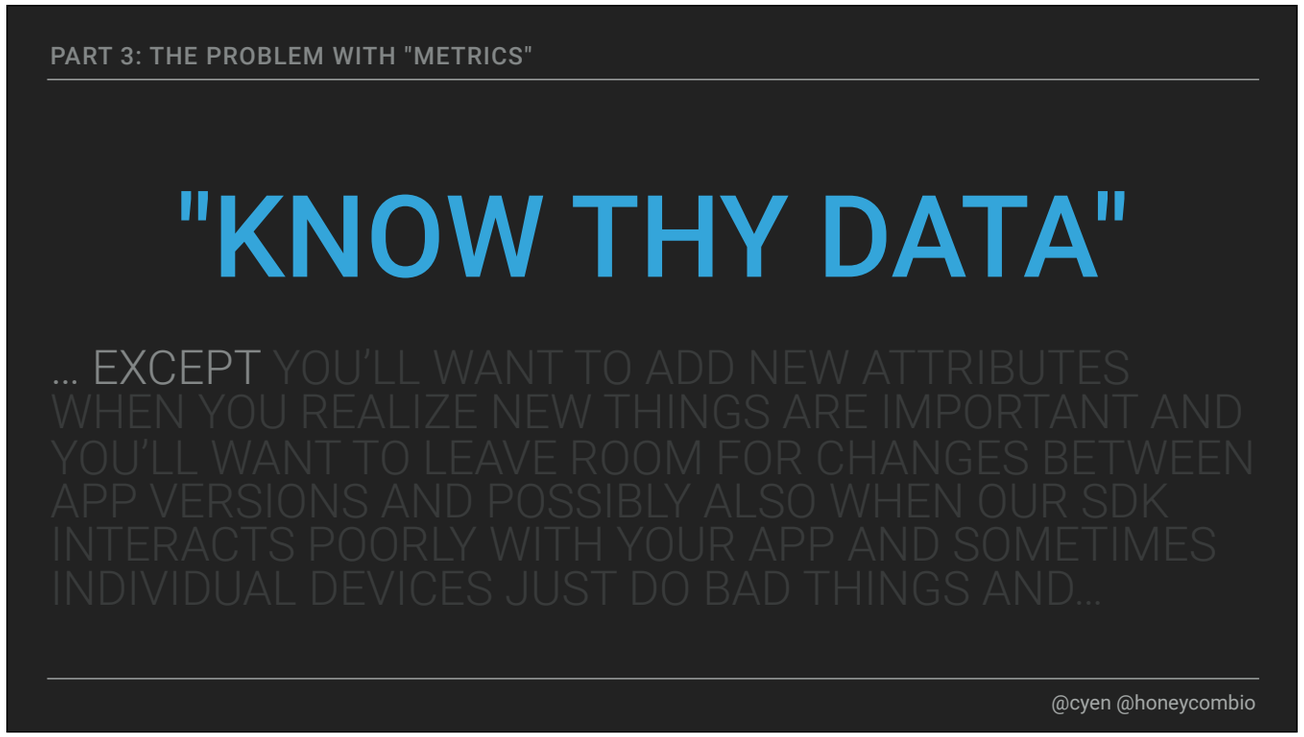
At first, we were frustrated that our users couldn't figure this out themselves. We worked so hard to provide them breakdowns!



Couldn't they tell that if their POST API requests looked like this, and their iOS requests looked like that, then obviously their iOS app activity was somehow correlated with the increased POSTs, more than their other clients?

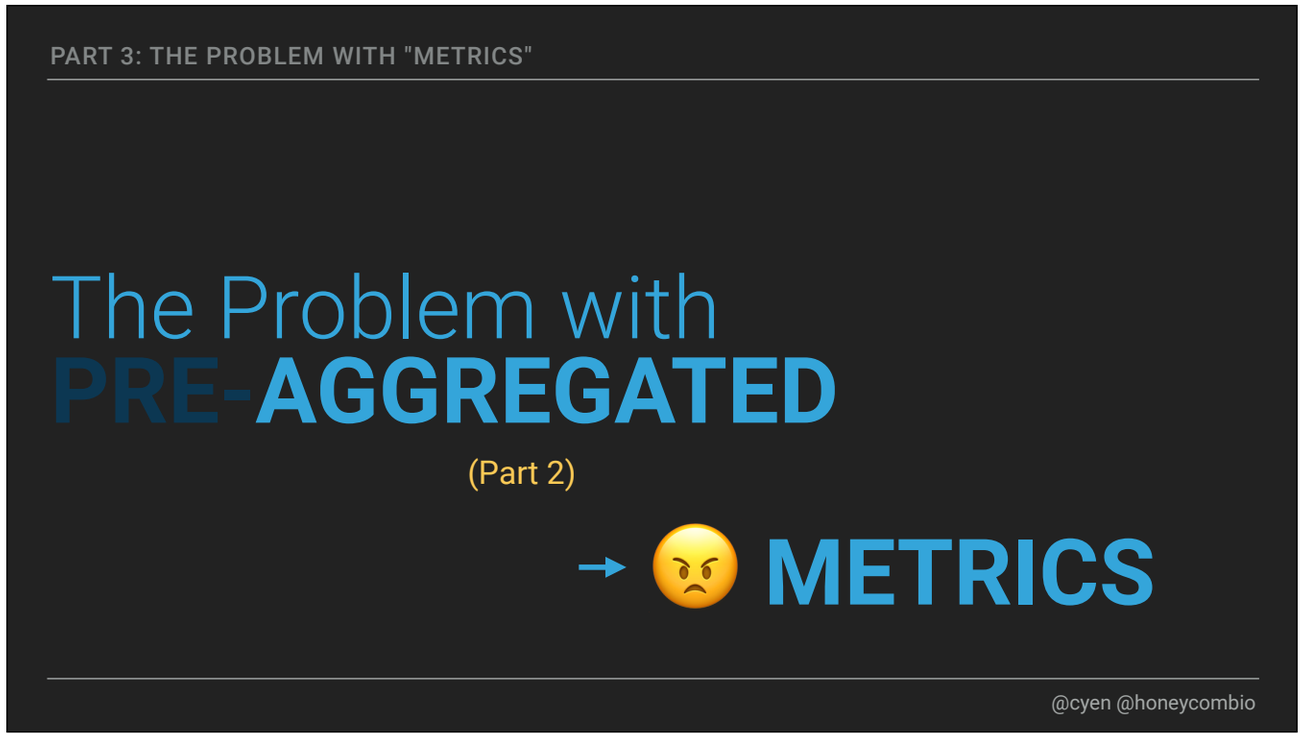


Super obvious. Except when the change was due to a specific app VERSION, or a specific app version on a specific OS version, in which case, thinking that it affects all iOS POSTs a little bit instead of a specific version a lot... well. Who could expect us to give them all of the answers?



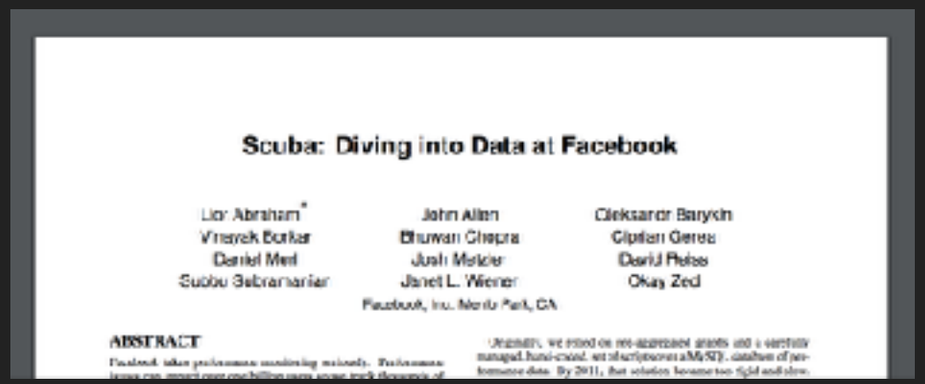
Our whole system was predicated on the idea that folks knew their data well enough to answer this themselves! "Know thy data", and all that - we went around telling users that they should be able to anticipate useful breakdowns and track them ahead of time.

Except, if your system is reasonably well-built and robust, new problems have new causes. And we know that from part 1, the problem with "PRE," it's a pain to be constrained to the sorts of questions and graphs that you anticipated back when you first set up your system's instrumentation.



... And we know from part 2, the problem with "AGGREGATED," that there are practical limits to how many cross products we should store -- even if we anticipated them ahead of time!

Parse + facebook



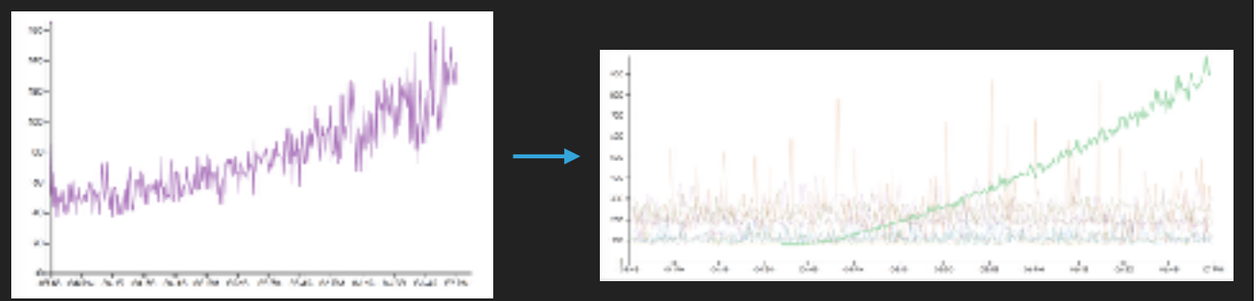
Scuba: Diving into Data at Facebook

Lior Abraham* John Allen Oleksandr Barykin
Vinayak Borkar Bhuwan Chopra Ciprian Gerea
Daniel Merl Joeri Metzler David Reiss
Subbu Subramanian Janet L. Wiener Okay Zed

Facebook, Inc. Menlo Park, CA

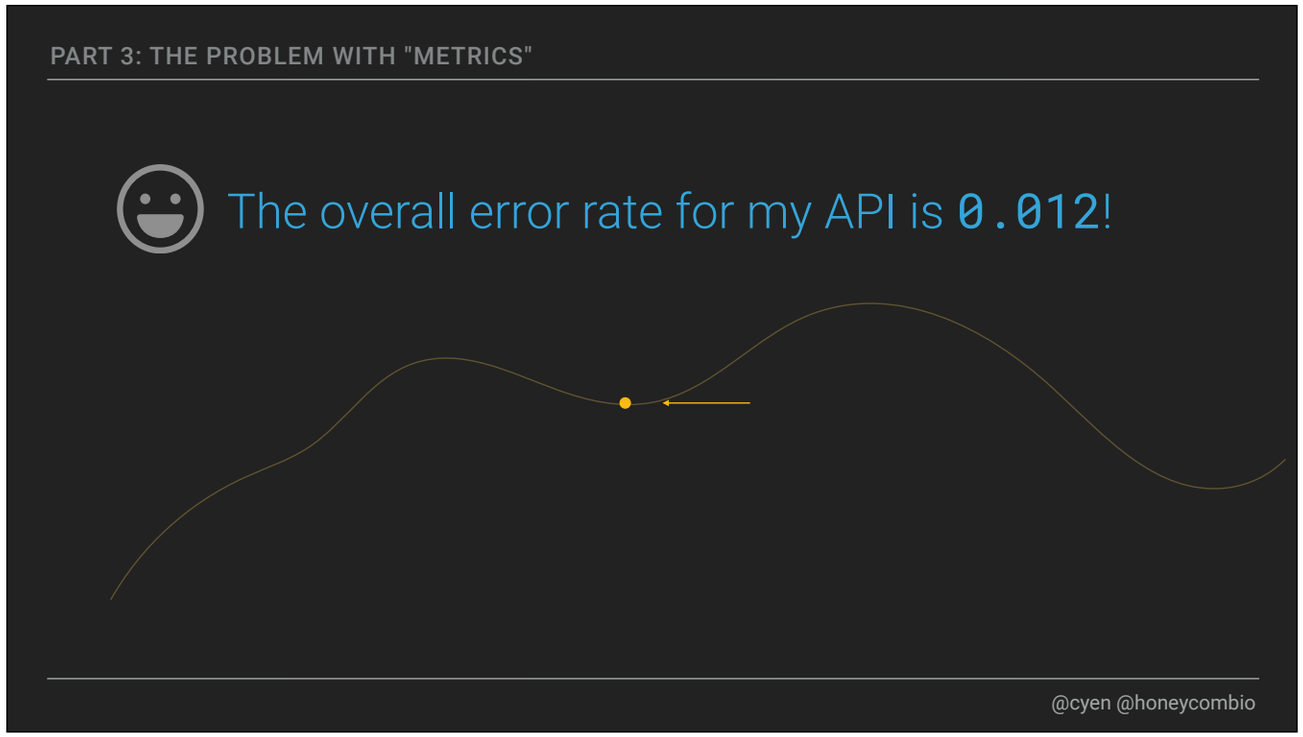
ABSTRACT

So the way this story ends is that, after we were acquired by Facebook, we were able to start sending all of our api logs to their internal tools - one of which was called Scuba. It was an in-memory datastore that was hyper-optimized for speed (sometimes sacrificing 100% accuracy) while retaining the ability to look at raw samples of events.

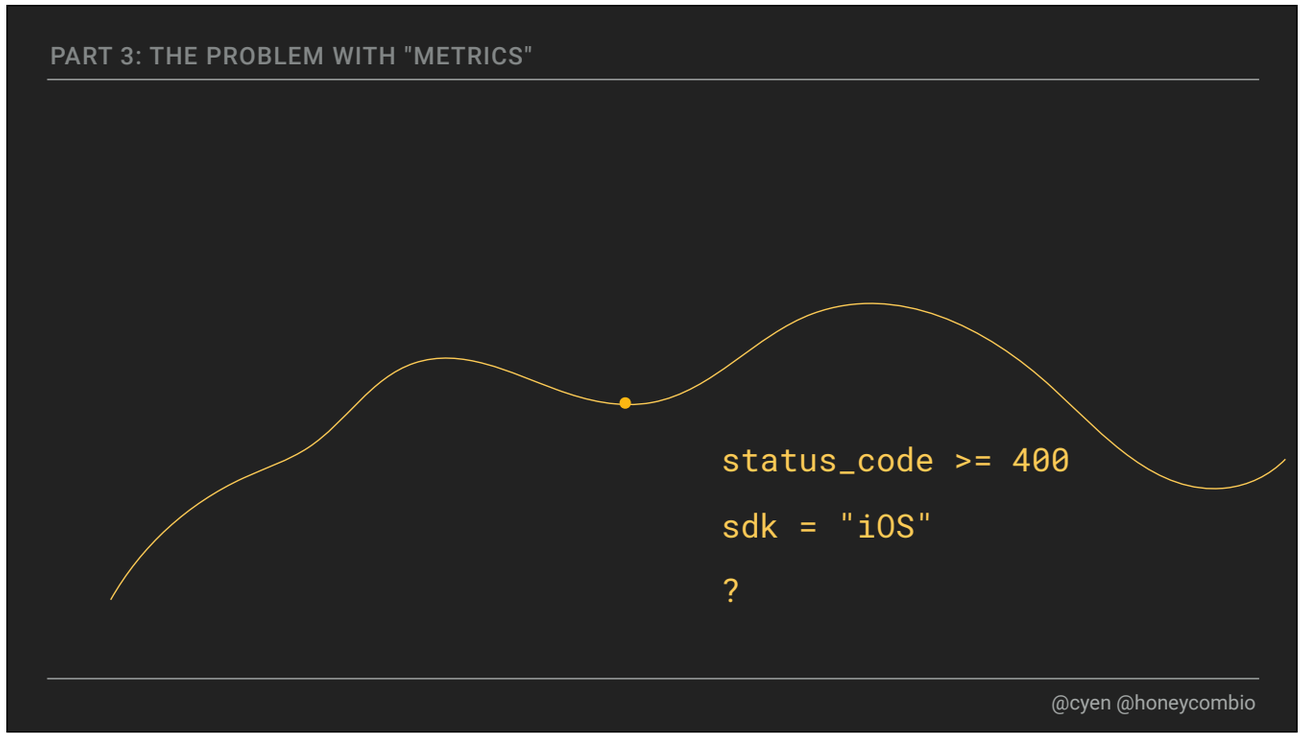


This let us, a platform which, at the time, was serving about a hundred thousand different apps' traffic, with different read/write loads, slice down to look at an individual app's traffic (even if it was only doing a couple requests per second among the flood of overall Parse traffic), and figure out - oh, hey, there's this other attribute that seems correlated with that increase in POSTs.

(And we did eventually appreciate the irony of having to fall back on our internal analytics tool to support our... external analytics product.)



The real problem with metrics is that, once they've been pre-aggregated, or rolled up, they're these individual points that you can't unroll to get **context**.

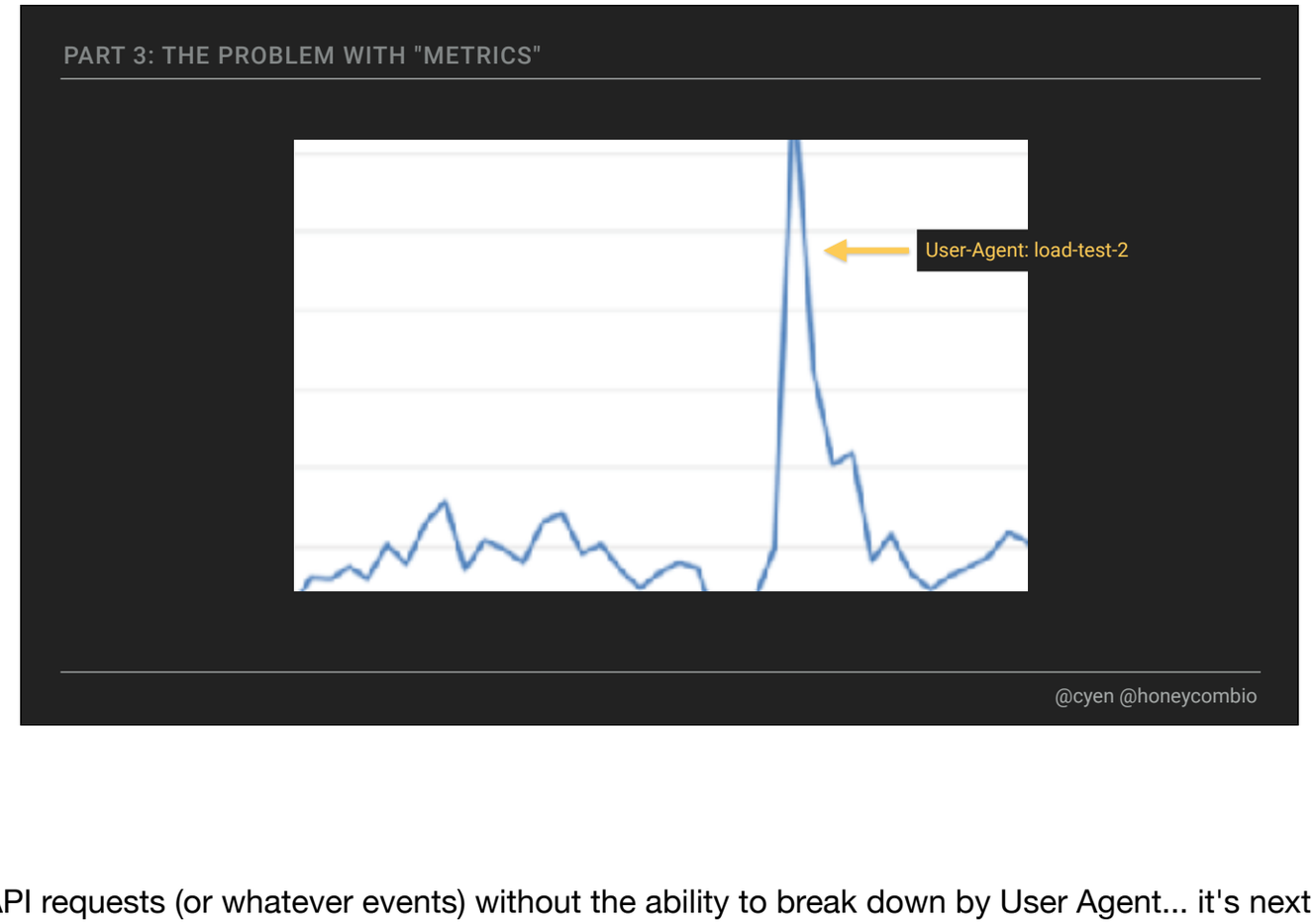


You can’t really iterate or explore - you can't side-eye that number, and say "OK, and what happens if I filter out requests below some threshold, and only look at requests from iOS devices?"

PART 3: THE PROBLEM WITH "METRICS"

@cyen @honeycombio

When you're stuck with the questions you thought to ask in the past, with the strict cardinality limitations that afflict these types of datastores, then you're stuck staring at your graphs of metrics without any ability to dive deeper into the data itself, to figure out whether or not this spike is worth worrying about.

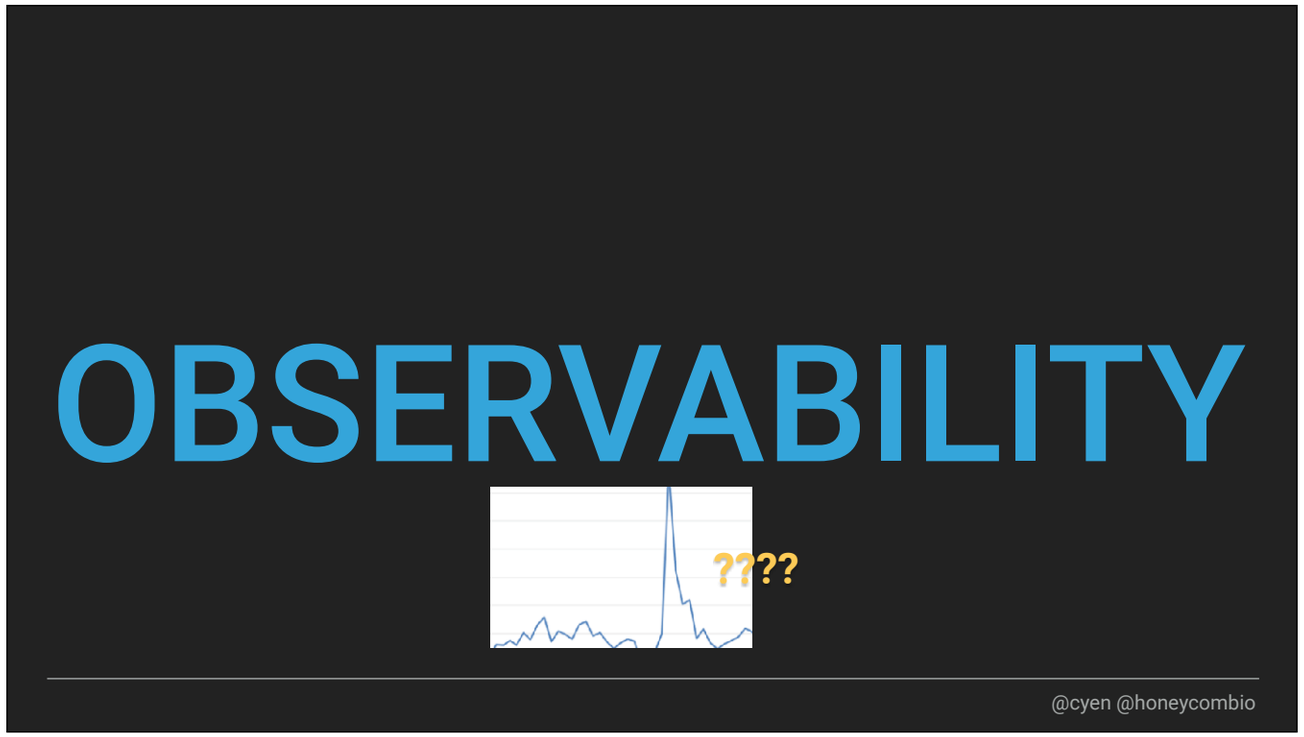


Once we've pre-aggregated all of our API requests (or whatever events) without the ability to break down by User Agent... it's next to impossible to understand our reality with data.
Turns out, this spike is just due to a load test being run… but we only know that because we were the ones who ran it, not because our graph was able to tell us. [20:00]

THE FUTURE?

@cyen @honeycombio

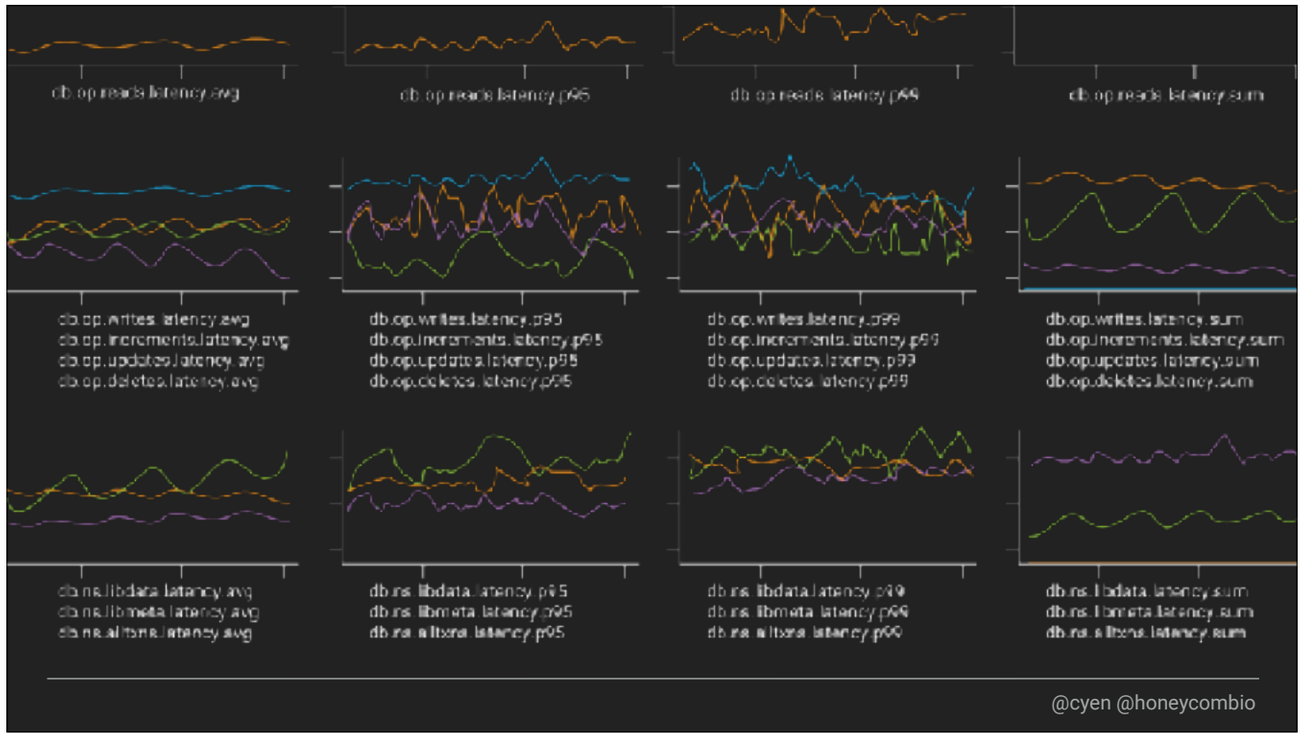
so. what are our alternatives?
at parse, we had facebook's scuba to let us get aggregates and graphs quickly and easily while retaining all the context we needed



we, in the non-facebook world, need to learn to expect more from our tools. Observability has to be more than just logging plus time series — I refuse to accept that the state of the art involves falling back to distributed grep after time series graphs fail to actually answer our questions.

(THE OBLIGATORY DISCLAIMER)

Remember my disclaimer from the very beginning, though — pre-aggregated time series tools are great **sometimes.** there are some sorts of metrics that are probably worth capturing in this opaque-but-very-performant manner, just to alert you when something's off: overall system latency, overall error rate, maybe something like overall purchases if you're an ecommerce site. Things like Key Performance Indicators, or the sorts of graphs you'd put on a single screen on the wall to watch all day long.



... but for when you need to dig in, we need to remember that producing and scrolling past hundreds or thousands of similar graphs, with no way to dig in and iterate and explore... just isn't the right way to debug. It isn't the right way to refine our understanding of things, or to find out **more**.

THE FUTURE

OBSERVABILITY AND NEXT-GEN TOOLS

What **will** help us dig in and iterate and explore?

OBSERVABILITY AND NEXT-GEN TOOLS

- Event-based systems

Event-based systems, for one (aka, structured logging backed by column stores and optimized for speedy reads on analytical workloads. It’s 2017; stop relying on Printf statements and fulltext search to debug systems.).

THE FUTURE

OBSERVABILITY AND NEXT-GEN TOOLS

- Event-based systems
- Approximation algorithms

Embracing approximation algorithms. We need to embrace the fact that super-fast and mostly-right is totally OK, and is way better for our use case — incident response and realtime observation of a system — than tools from the data science world that get you a perfectly accurate answer… tomorrow.

THE FUTURE

OBSERVABILITY AND NEXT-GEN TOOLS

- Event-based systems
- Approximation algorithms
- Intelligent sampling techniques

@cyen @honeycombio

Intelligent sampling techniques - "sampling" can be a dirty word (which is probably the basis of a whole 'nother talk), but along the lines of working with approximation algorithms: if we can accept that, at incredibly high volume, 10% or 5% of traffic is sufficiently representative, then awesome, go with that. If we want to take it one notch smarter, keep 100% of all errors and 1% of all successful requests. You'll get full resolution into requests that are more likely to be interesting while still getting a reasonable sketch of "normal" traffic.

OBSERVABILITY AND NEXT-GEN TOOLS

- Event-based systems
- Approximation algorithms
- Intelligent sampling techniques

For reference, at Facebook, Parse sent all of its API traffic through Scuba and sampled at somewhere between 10-25%; Instagram used Scuba as well, and sampled *its* traffic at something like 0.1%. (Don't quote me on this, it's been a few years since I checked those numbers, but they're the right orders of magnitude.)

THE FUTURE

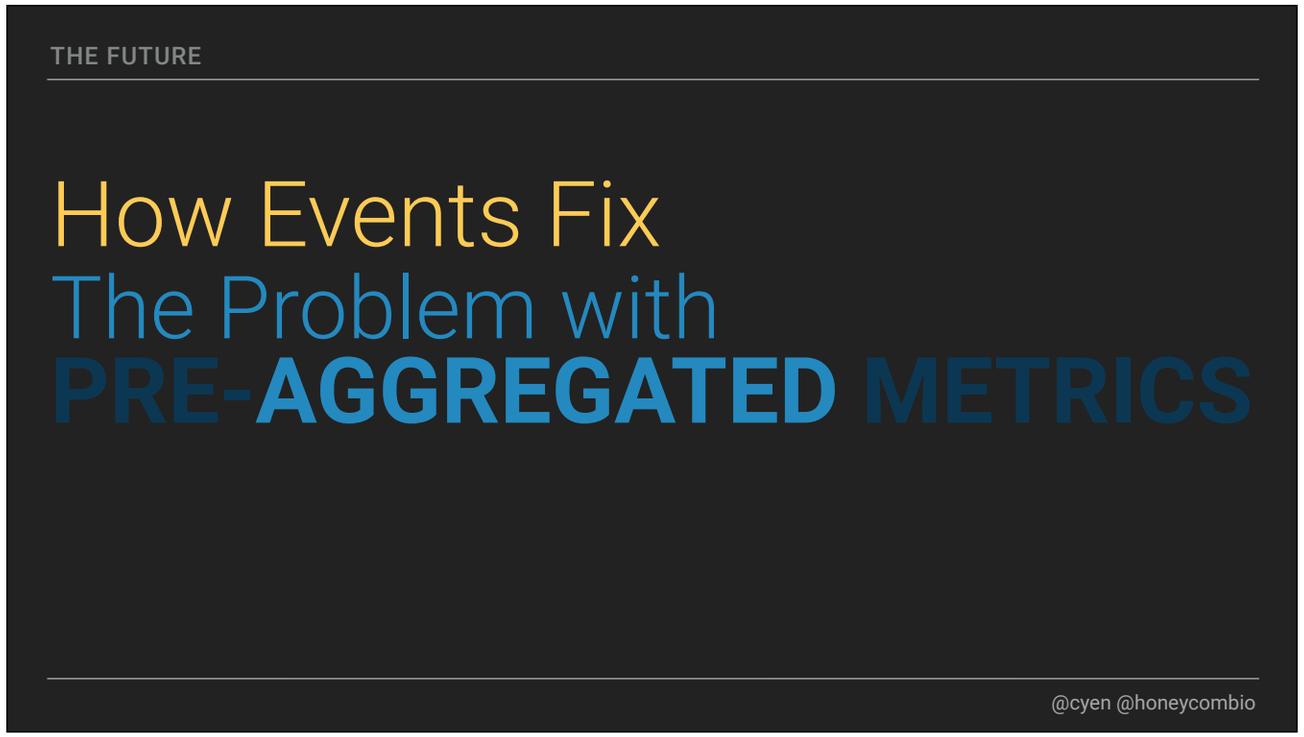
OBSERVABILITY AND NEXT-GEN TOOLS

- Event-based systems
- Approximation algorithms
- Intelligent sampling techniques

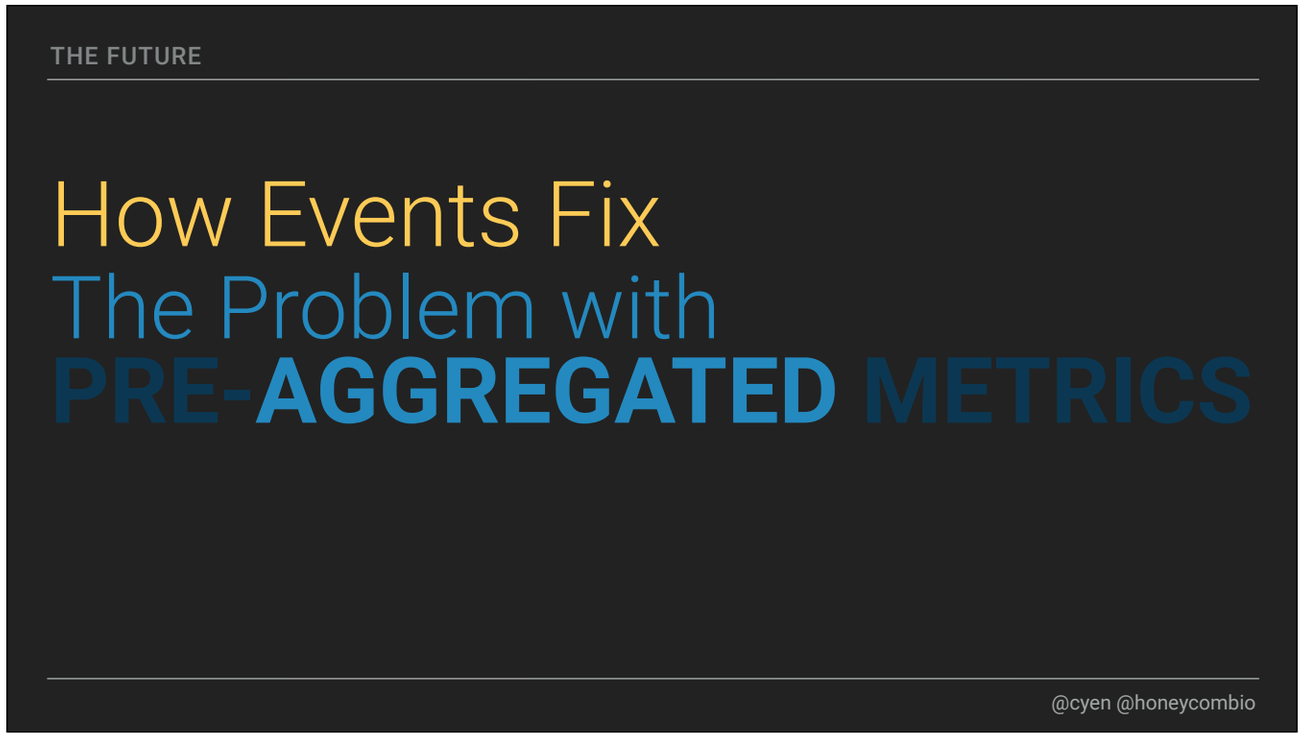
These techniques free us from the problems that exist in the pre-aggregated metrics world, and give us the freedom to hunt down problems in our systems, and surface the unknown-unknowns.
So how do they address the various problems we've talked about so far?

- Event-based systems and read-time aggregation let you ask questions of whatever tiny cross-section of your traffic that you're interested in, even if you never thought you’d care about how a particular combination of - build ID and app version and user agent - happen to interact.

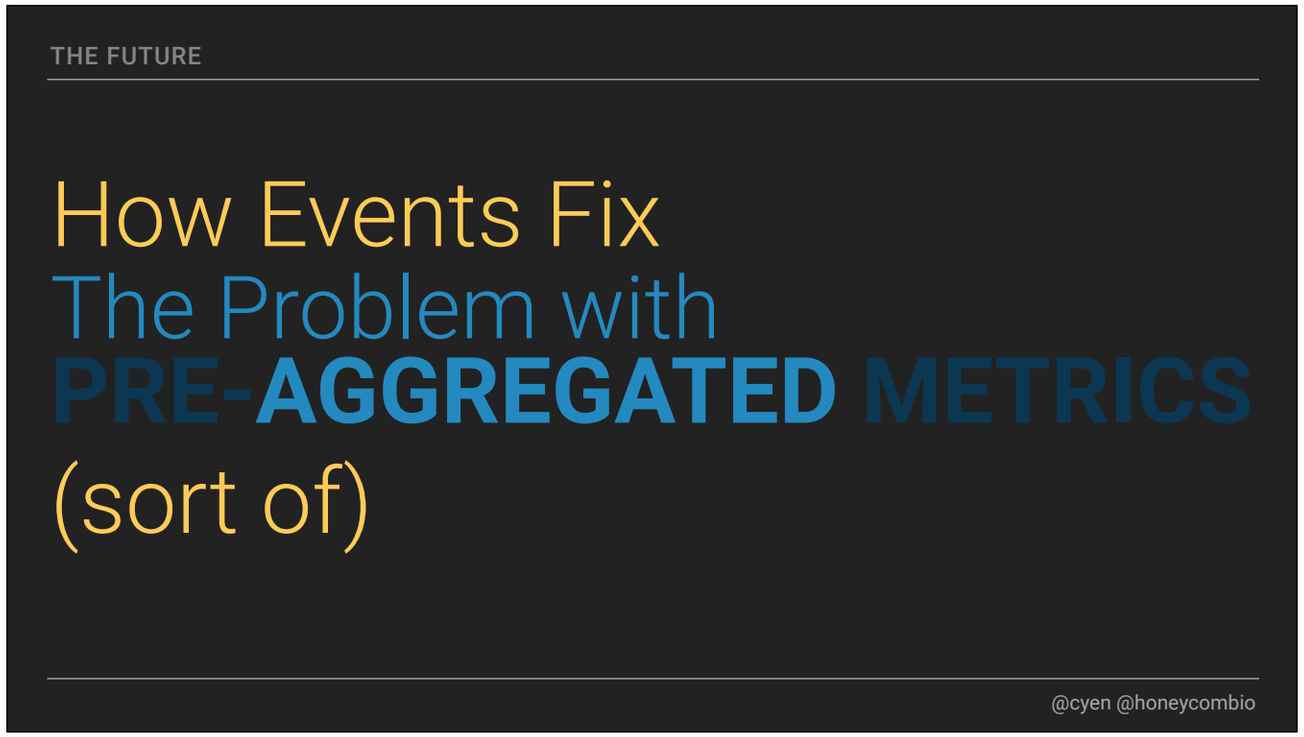
- If good engineers know a problem exists for a specific use case, they do everything they can to fix or prevnt the problem -- which means that history doesn't repeat, it rhymes, and you've got to make sure your instrumentation and monitoring systems are flexible enough that they don’t just help you find past known problems… but help you identify unknowns, quickly.



- Event-based systems and read-time aggregation sidestep the cardinality issue entirely, because storage scales based on some sampling of your system volume, rather than the number of unique values used to describe your system's traffic.



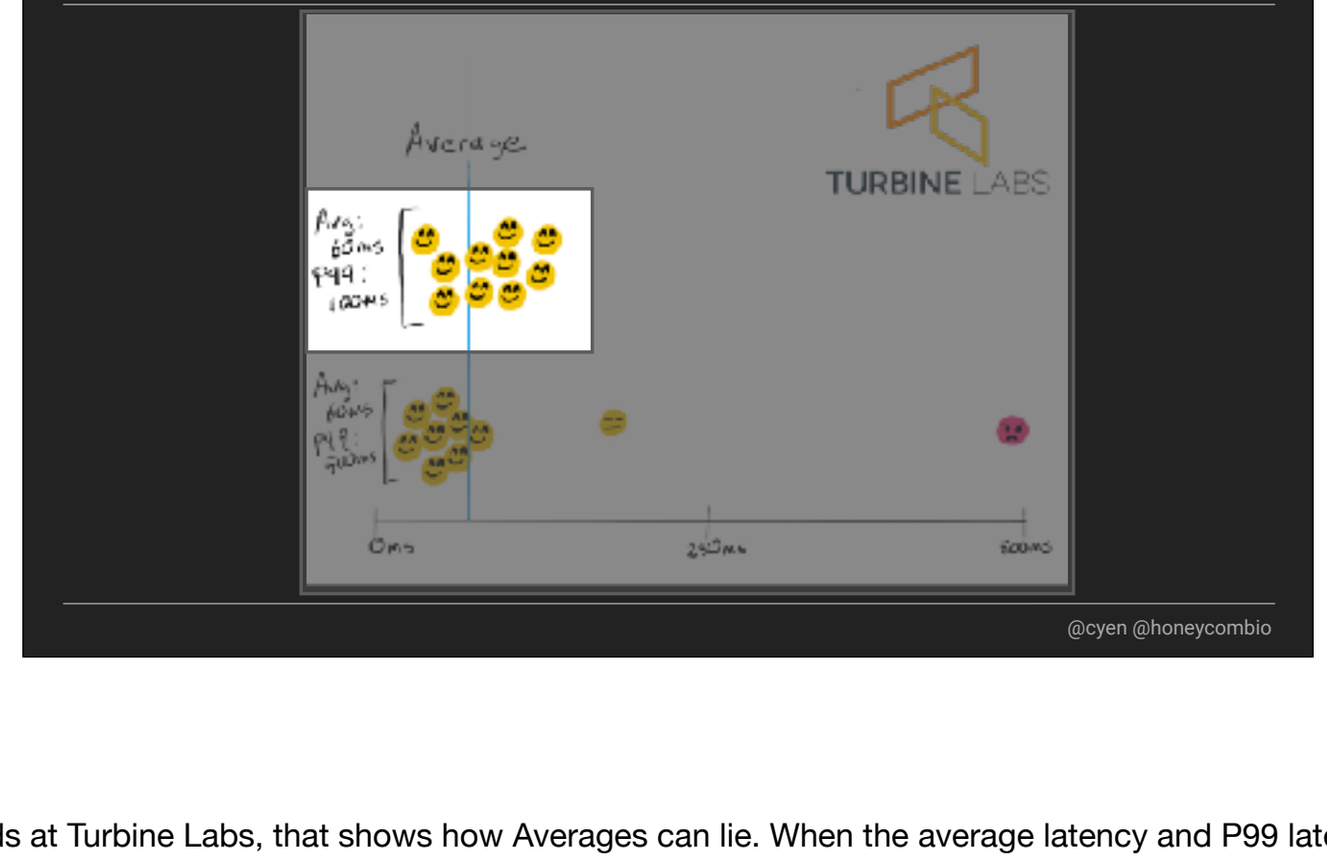
- You can store arbitrarily high-cardinality fields (customer_id! user agent! or even... session / request_id!) and calculate speedy statistics for each group. Finding the user agents correlated with the highest-latency requests? Now easy with a system that doesn't rely on artificially constraining the number of unique metric names to stay stable.



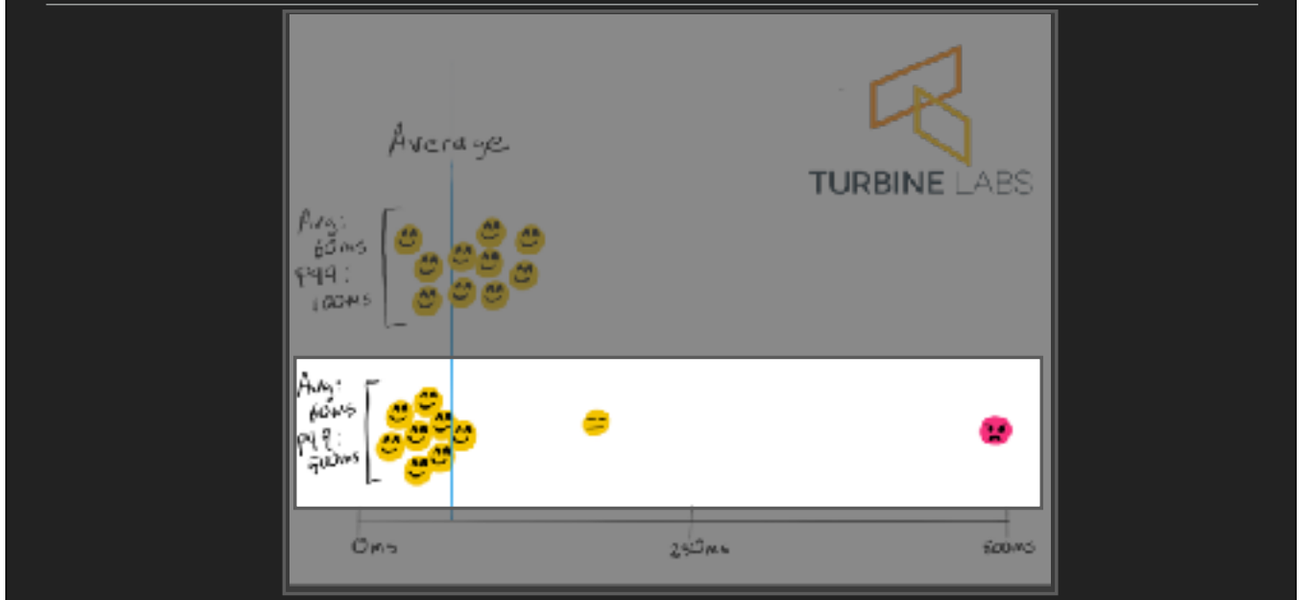
- And, just because I like disclaimers and caveats — note that computing statistics across a really high-cardinality field is still hard. it'll still impact you. if you're trying to calculate average latency or error rates across thousands of groups at the same time, it's **going** to be more expensive to calculate than tracking a single measurement no matter what… but if your storage system is smart about it, it only gets expensive when you demand it. you have the power to **control** that pain by adjusting your queries, rather than it just being an impossibility (in order to protect your system).

- You can always find out more. In event-based systems, you can always find out **what actually happened** underneath a given dot.

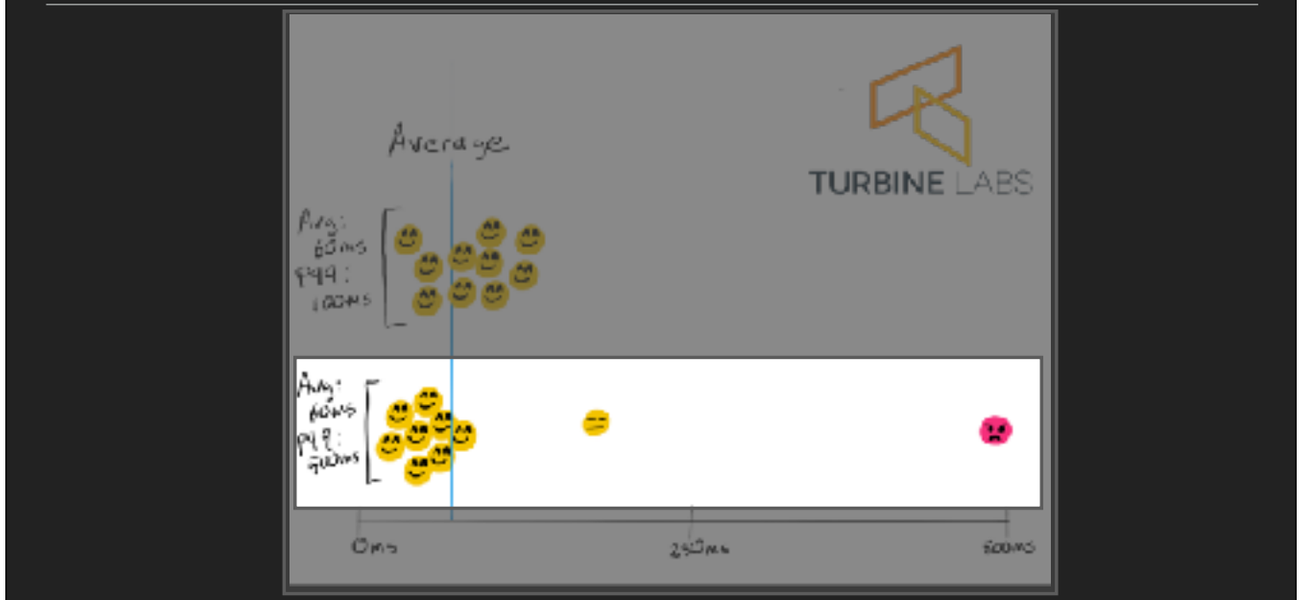
- I want to run through a example scenario where we dig deeper to understand some outliers. if you're a savvy monitoring person, you know that Averages are fine to talk about but easily obfuscate reality. [27:00]



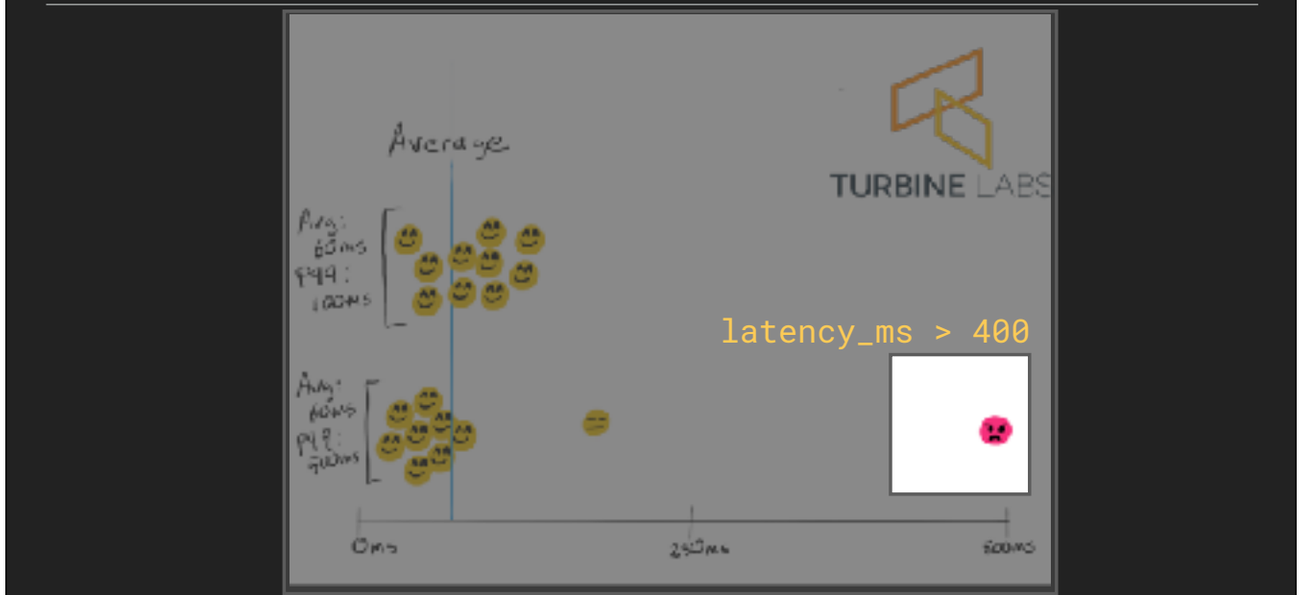
This is a great illustration , by our friends at Turbine Labs, that shows how Averages can lie. When the average latency and P99 latency, or the average and slowest 1% of requests, are pretty close together, your system doesn't have a ton of outliers and is delivering a pretty consistent experience to your clients.



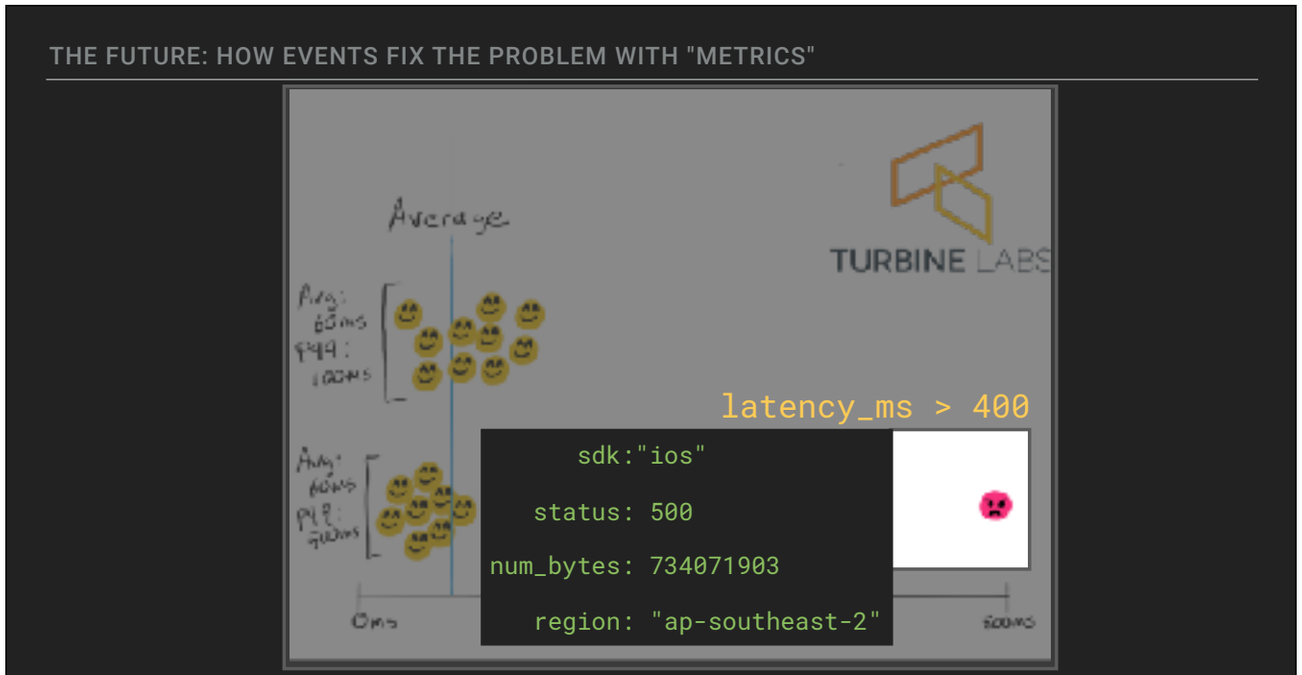
When they're far apart... you may have a bunch of generally happy clients and a small handful of really furious ones.



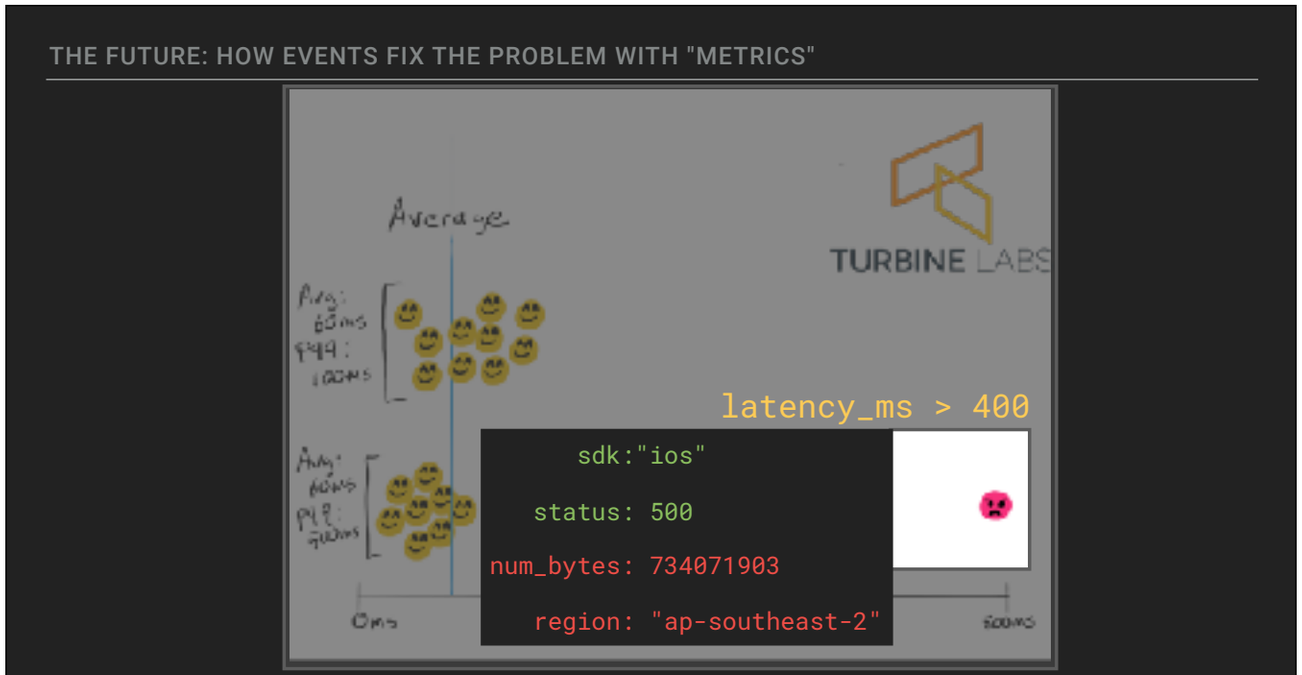
In a pre-aggregated metrics system, you'd be able to know that the slowest 1% of requests were really slow... but would have to go look at a bunch of other graphs to see if you could figure out other attributes that correlate with the slowness, in order to track it down and figure out if it needs fixing.



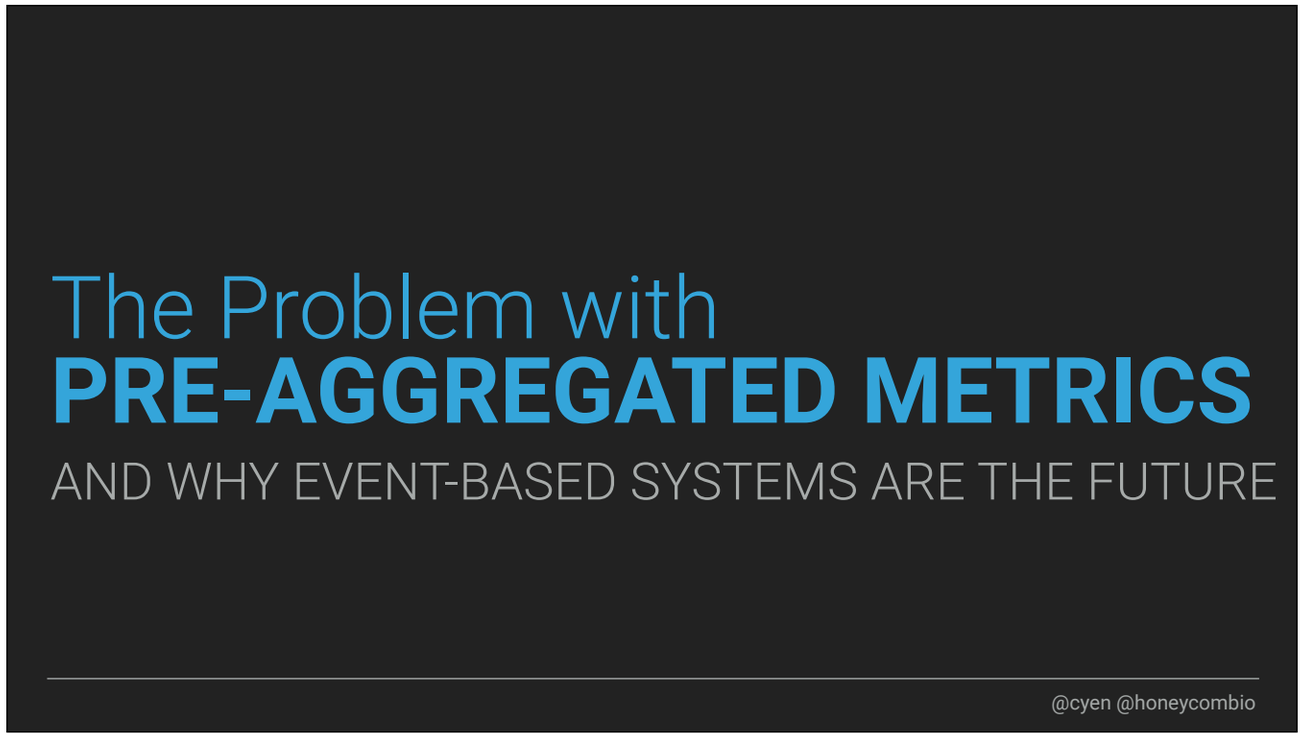
- Instead, in an event-based world, you can just dive in.
    - Throw in a filter to isolate the requests you care about



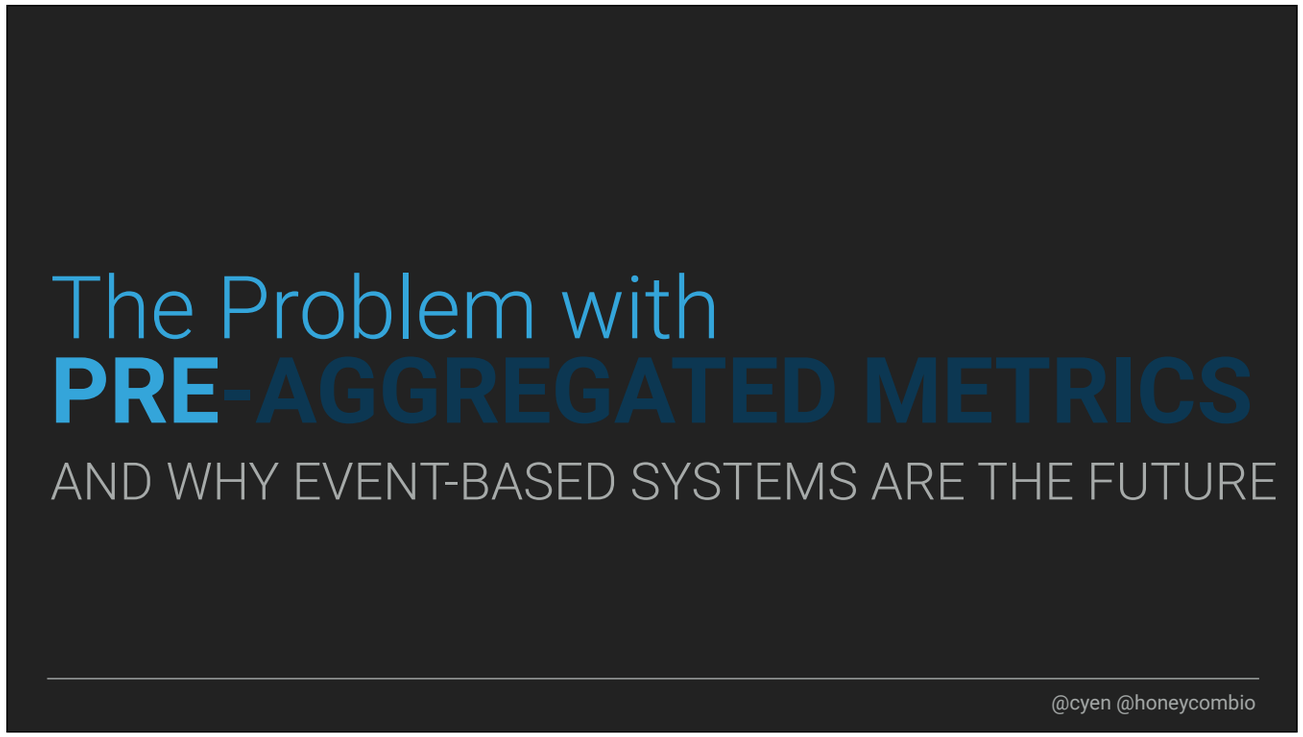
And then dip into the raw data to see if you can learn something interesting about the angry event



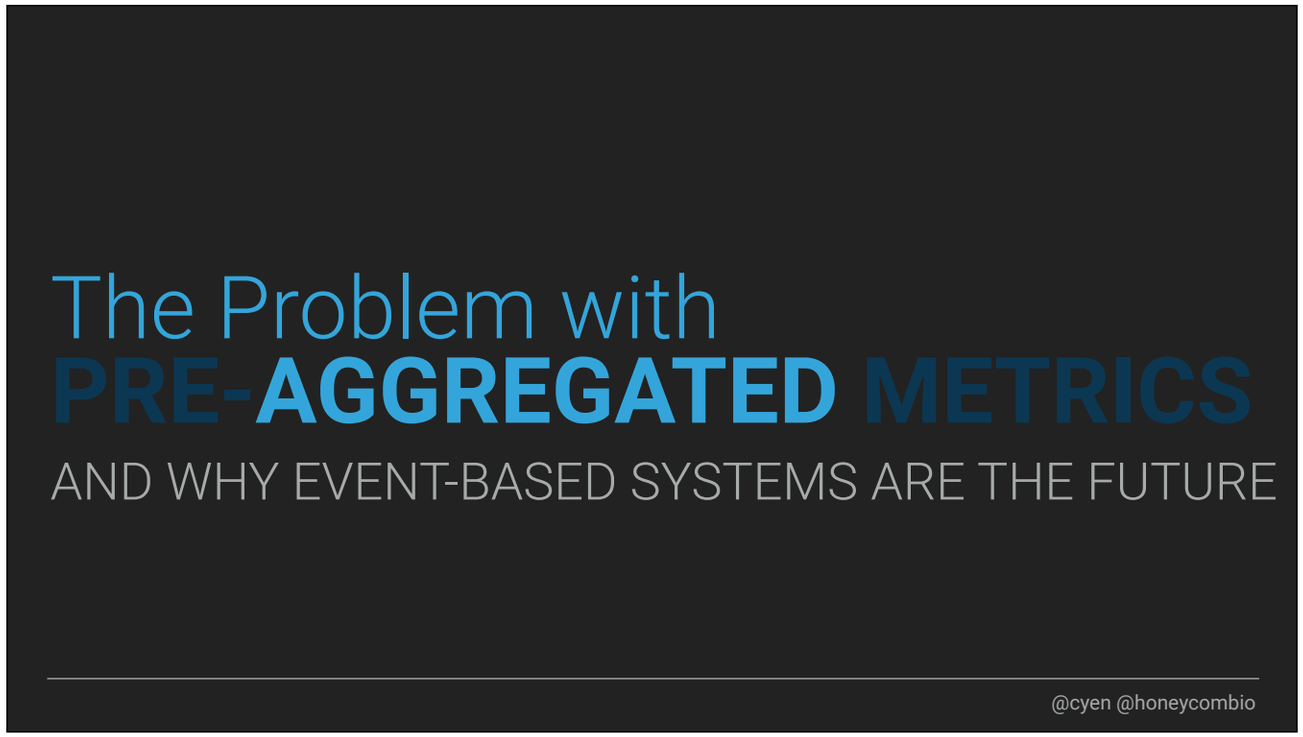
By being able to actually, immediately, **find out** about particular outliers, we can unlearn the correlation dance that we used to do, and just get to the answers we want.



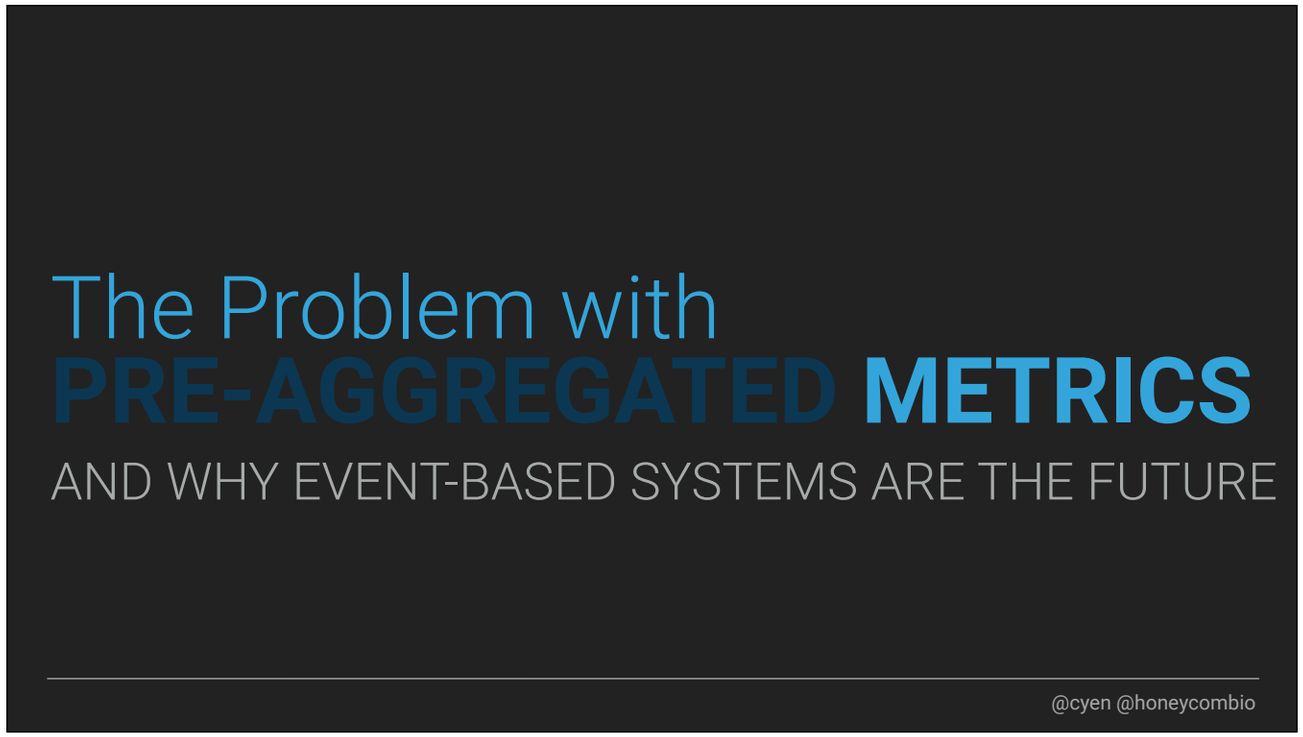
Let's wrap up: why Pre-aggregated metrics are a problem, and why event-based systems are the future.



- [prob w/PRE:] couldn't anticipate this particular thing as a trouble spot, so weren't prepared



- […AGGREGATED:] even if we had, we might not have the storage to be able to track a whole bunch of cross products anyway



- […METRICS:] and now that we've identified something interesting here, there's no way to poke any deeper, to try and identify the context of **what actually happened**.

- If any of these problems have resonated with you, give your friendly neighborhood events-based system a whirl and see what bad habits you can unlearn in your systems.