

Hi I'm Dietrich

[d2fn.com](https://d2fn.com)

[github.com/d2fn](https://github.com/d2fn)

[twitter.com/d2fn](https://twitter.com/d2fn)

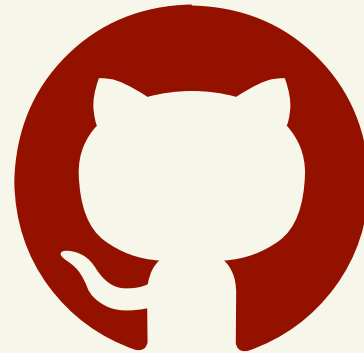
Previously I built Boundary's  
historical database



I've spent most of my career thinking about time series data and building tools to help answer questions about that kind of data. Previously I built the historical database powering Boundary which I named after a competitive eater.



## Now I build analytics infrastructure at GitHub



- github now
- didn't have a strong analytics presence when i joined
- team had to figure out how to structure itself
- add analytical rigor to an already high-functioning company

Now I work at GitHub building analytics infrastructure. But when I joined GitHub we didn't have a strong analytics presence so part of the challenge was for the team to figure out how to structure itself and how to connect with the rest of the company. This was about two years ago so GitHub was already a strong company. We knew we wanted to me more data driven in our operation but it's not intuitively obvious how to add this kind of rigor to an already high functioning company. So much of the teams' responsibilities came down to figuring out how to structure the team and how to be effective.

**Analytics Services Team**

**&**

**Analytics Infrastructure Team**

Presently, we've arrived at this dual mode of function. We are part services team, part infrastructure team.

# Question-driven Development

- question-driven development
- company externalizes questions to us by way of a repo + issues

This philosophy of question-driven development came early on in the formation of the team. The services part of the team largely acts as a place for other teams—sales, marketing, product, even system engineering—to externalize a lot of their ad-hoc analytical functions. And our interaction there is pretty simple. We have a GitHub repo where others create issues, then we start a dialog to get more information until eventually we can either answer their question.

# Answer enough questions and patterns start to emerge

- answer enough questions, patterns emerge
- those patterns become infrastructure

And in answering these questions and thinking about our supporting infrastructure, there are some patterns that have stood out. So we're going to spend the rest of our time together today talking about some of these patterns, and work through how to think about building some really solid analytics infrastructure.

And this talk is more about things to take into consideration and how to approach the problem than it is about what technologies to use. I've also done my best to put something together that is very practical without basing too much of the talk on problems specific only to GitHub.

# Anecdotes



So let's start with a couple of anecdotes and come back to some of these issues along the way.

Rowers with LEDs attached to their paddles. Timed exposure.





**James Golick**  
@jamesgolick



Following

the cool thing about tweeting about music  
instead of computers is that people get  
happy and sharing instead of mad and  
insulting.



RETWEETS

50

FAVORITES

71



2:25 PM - 7 Oct 2014

And if we're going to do that, let's start with something music related. People like music and it makes them happy so why not right?


what do musical scales being  
played on a violin look like?

What do musical scales being played on a violin look like?



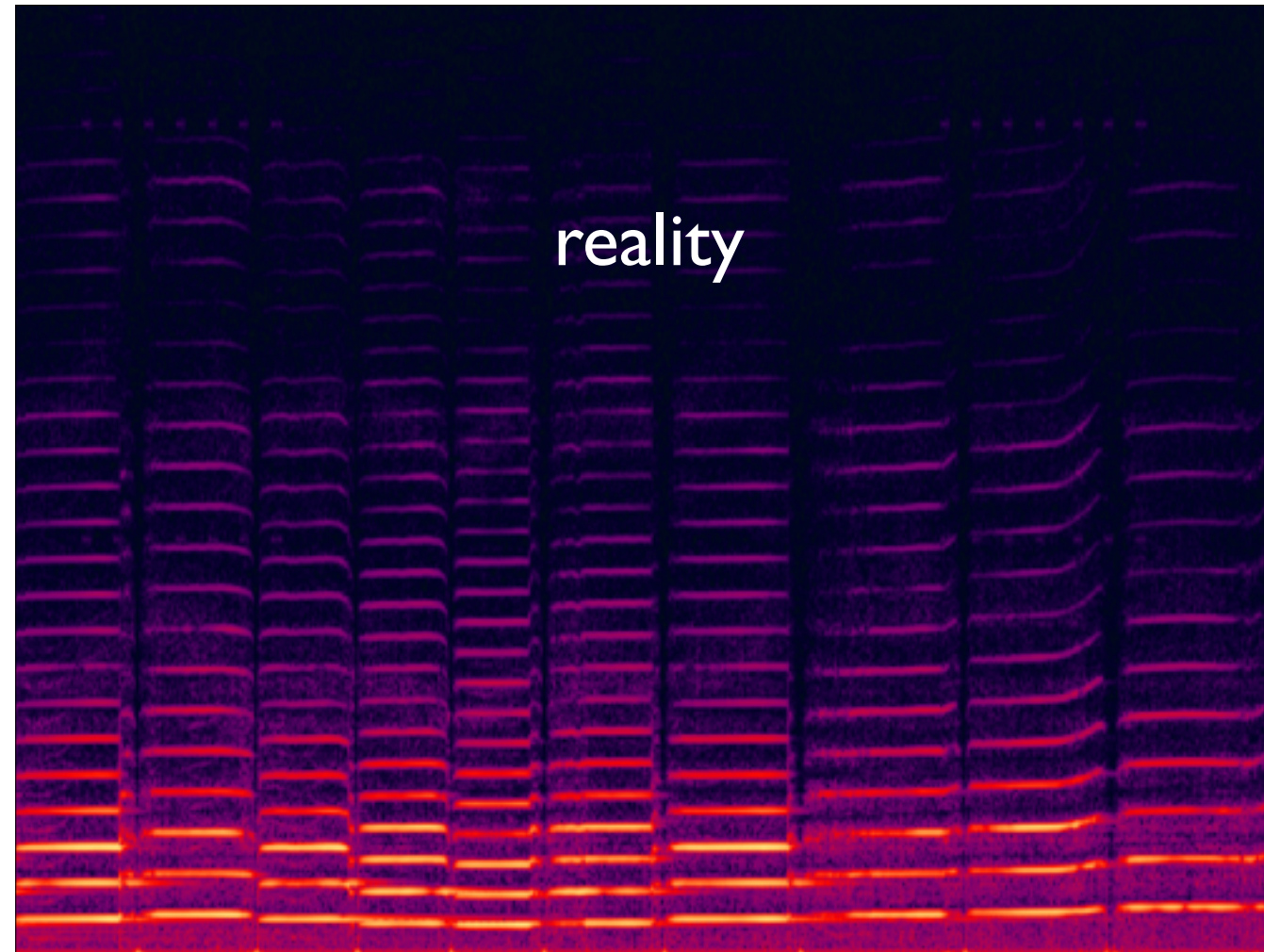
Well, I'll give you a hint. It doesn't look like this.

nope

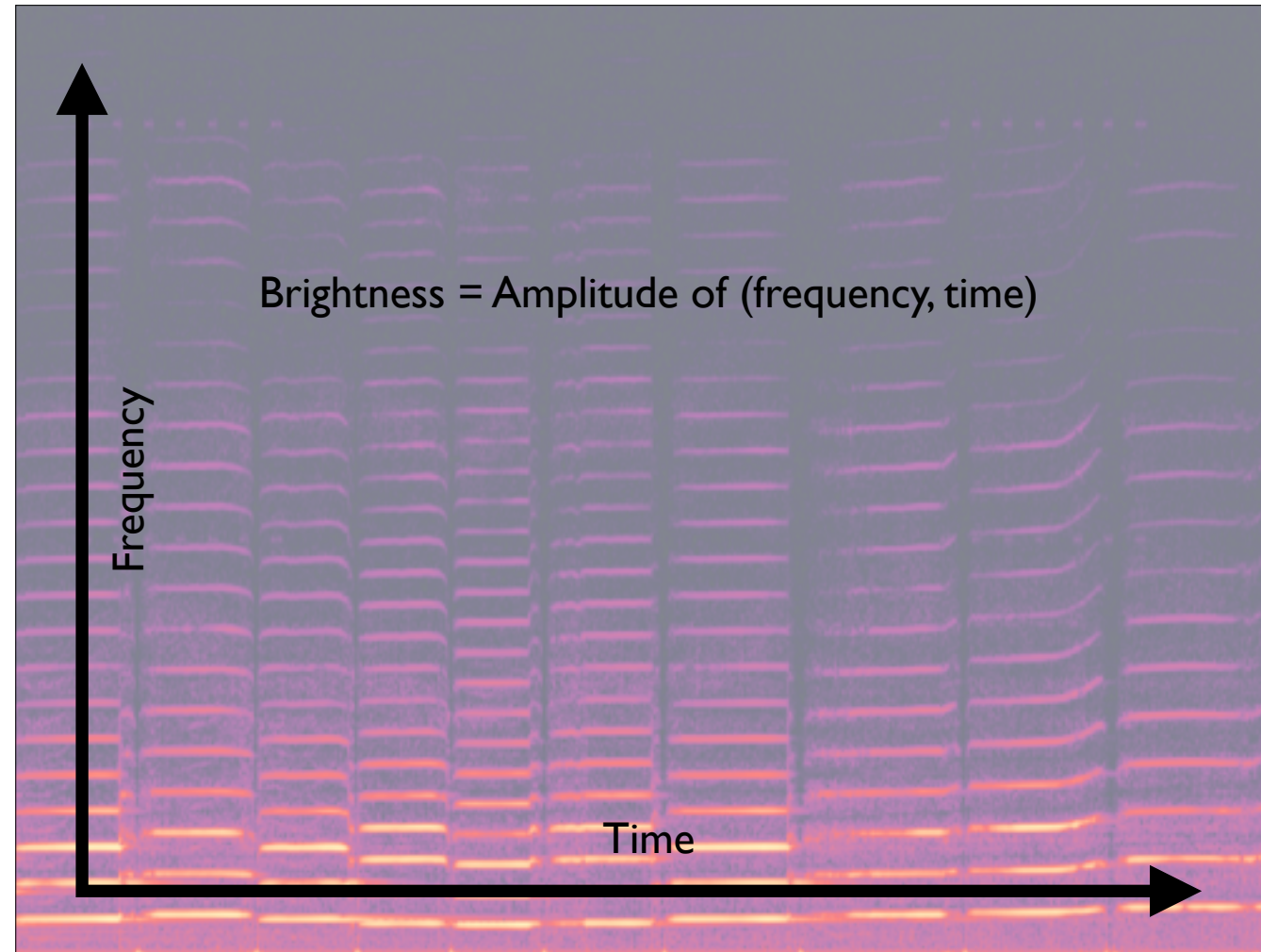


A musical score for the word "nope" in G-flat major (two flats) and 4/4 time. The melody is written on a single staff with a treble clef. It consists of three measures: the first measure contains four quarter notes (G-flat, A-flat, B-flat, C); the second measure contains four quarter notes (D, E, F, G-flat); the third measure contains four quarter notes (A, B, C, D). The notes are all on a half-note value.

This is the ideal we think of. This is what someone with a violin would read in order to know how to play the music. But if we measure the actual sound over time, and the strength of the sound at various pitches, it looks quite different.

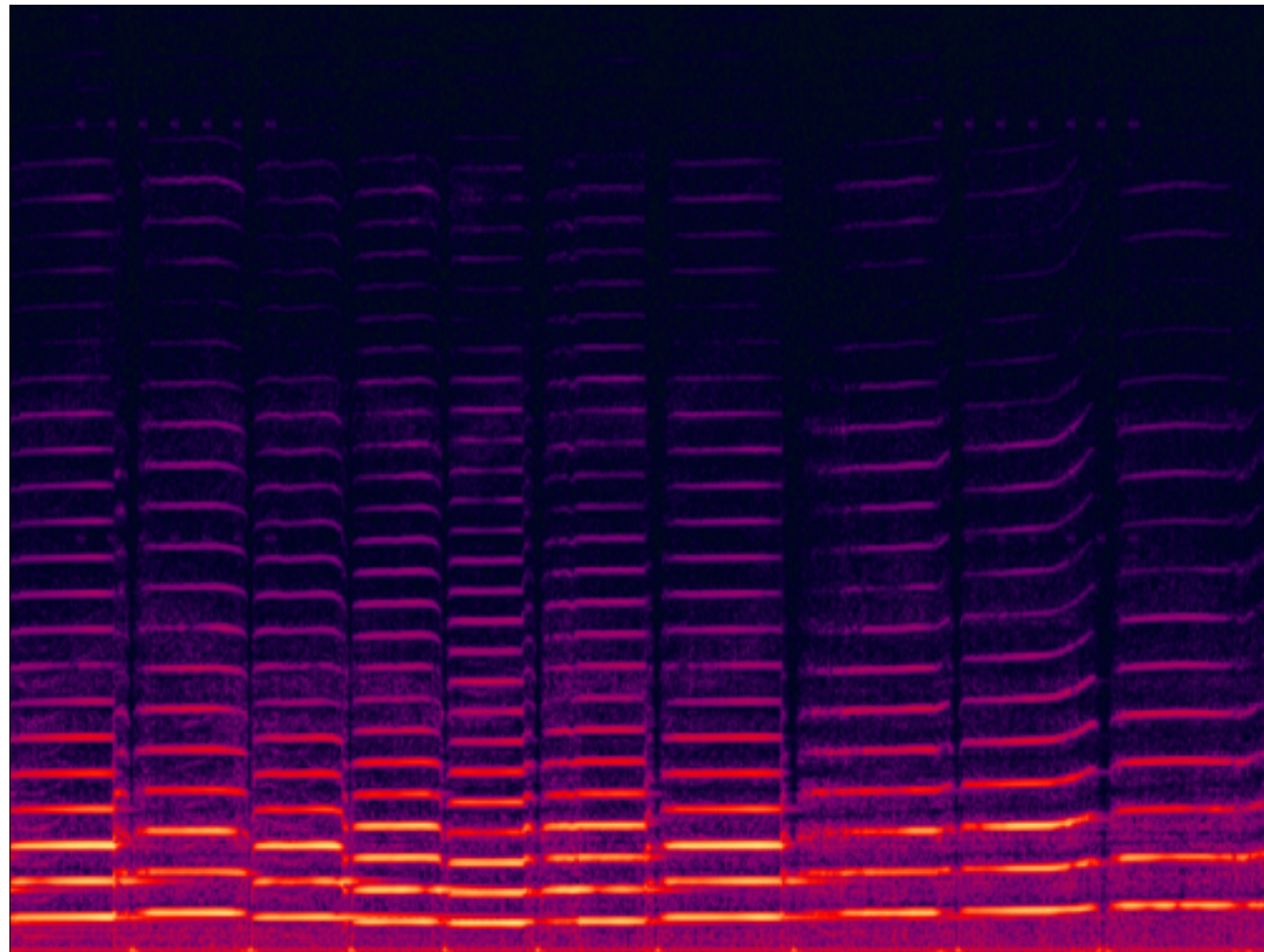


We get something like this. This is a spectrogram. Does anyone know what a spectrogram is? Well a spectrogram is a way to visualize sound over time.

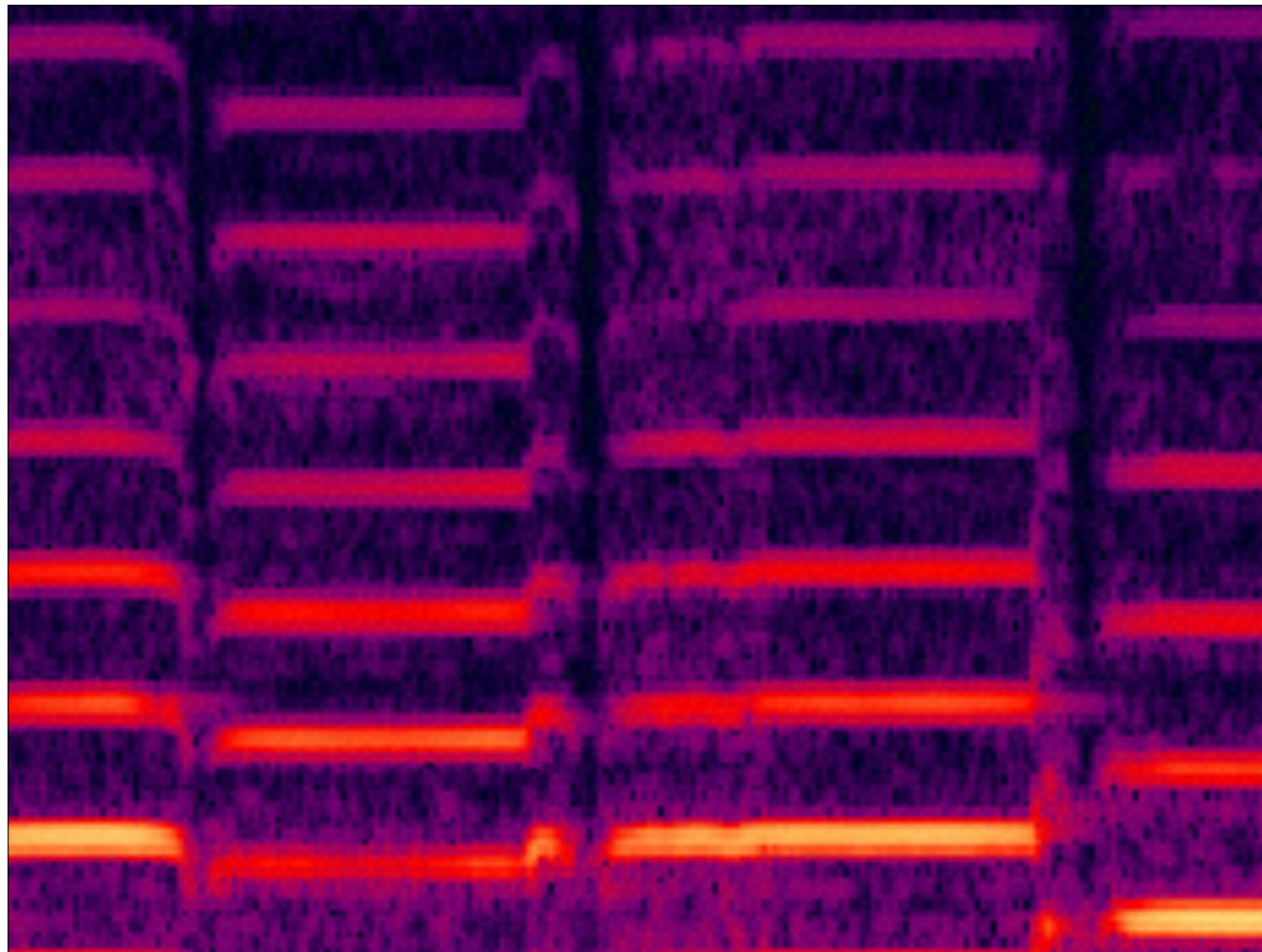


And specifically what the strength of the sound is at the various frequencies that it gives off. And the way to read a spectrogram is, time runs left to right, and frequency goes from low to hi, bottom to top. And at the intersection of frequency and time, the intensity of color tells us the volume of that sound.





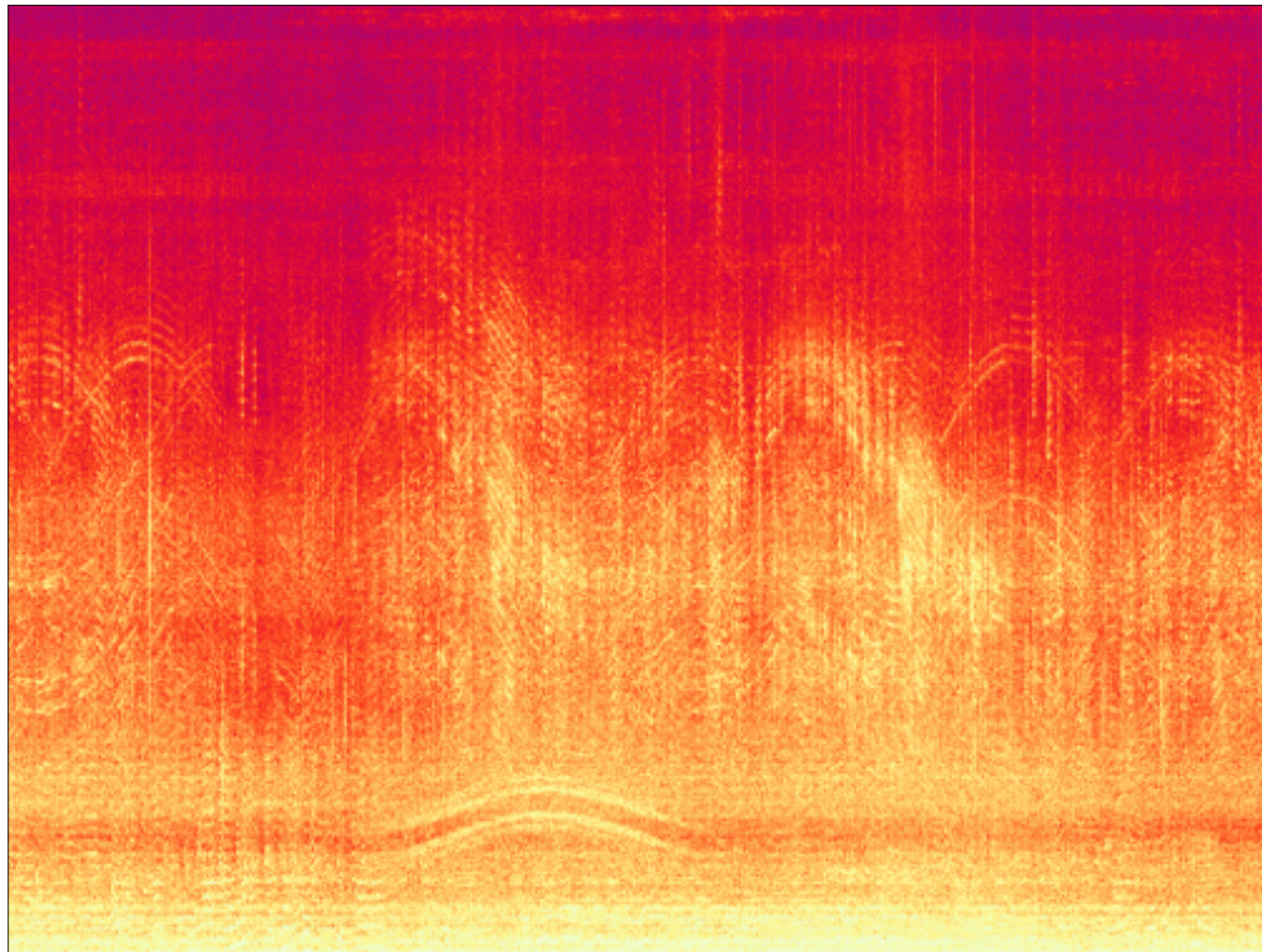
So what do we see here. Well, the individual notes are pretty clear in the vertical banding. But the first thing that jumps out are these horizontal lines. What are these? Well, the lowest one is the base note. This is the note being intentionally played by the violinist when she reads sheet music and plays a note. But the horizontal lines above that are the harmonic frequencies. These are what give the note character, and depth, and complexity. Without these, you would be left with a pure tone. One of those pure notes that might come from an old computer. This is what is known as timbre, and it is largely how we are able to distinguish a middle C played on a violin, from a cello, or an oboe, or a piano.



And if we zoom in, there is even more going on here. See this little jitter. This is likely where the violinists fingers moved in preparation for moving to the next note. And the various noise we see between the harmonics might be explained by hands moving over the grain of the strings themselves.

Whatever the case, there is a story to be told here. And we could probably discuss this image for the rest of the talk trying to parse out all of its structure.

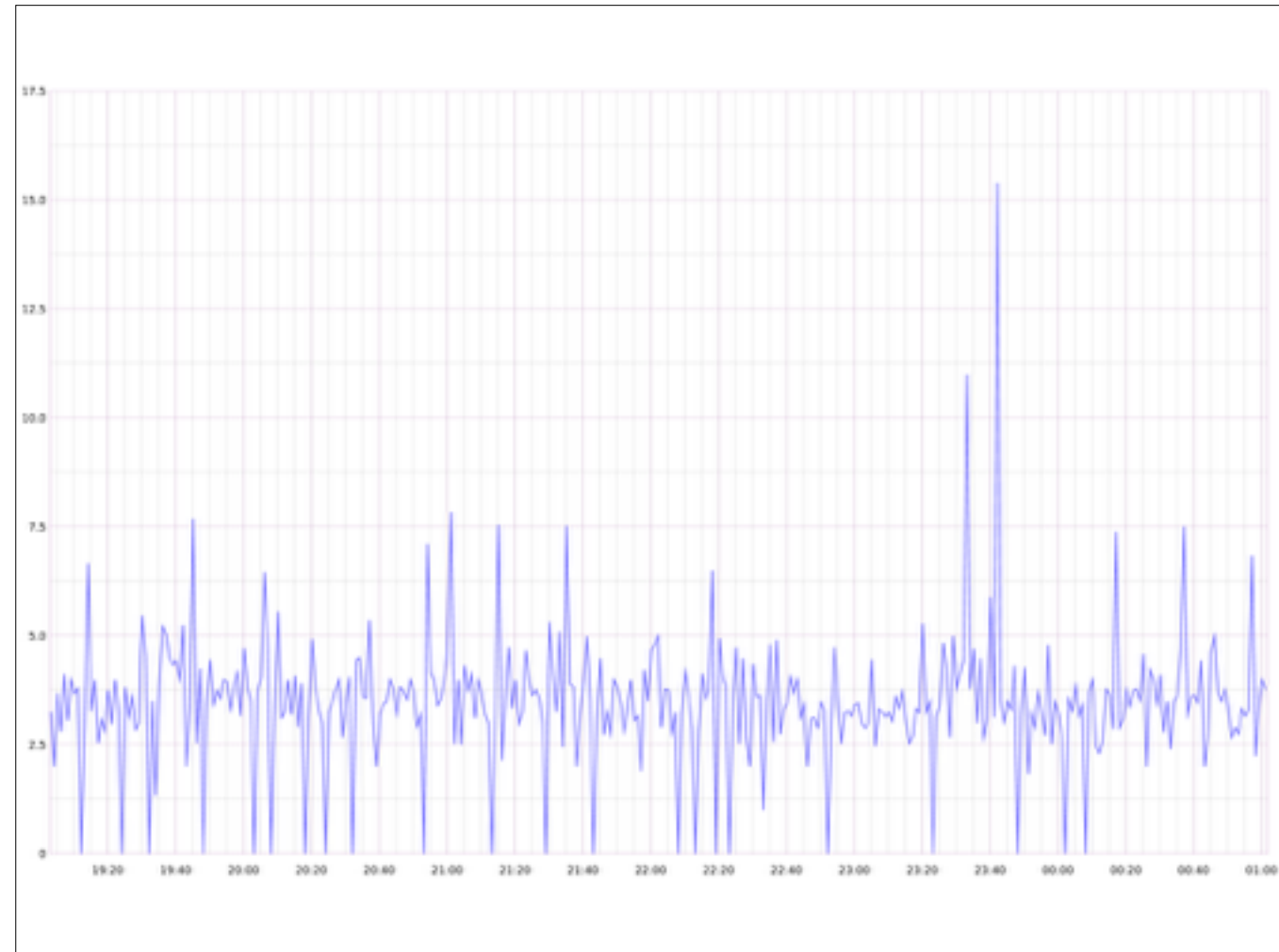




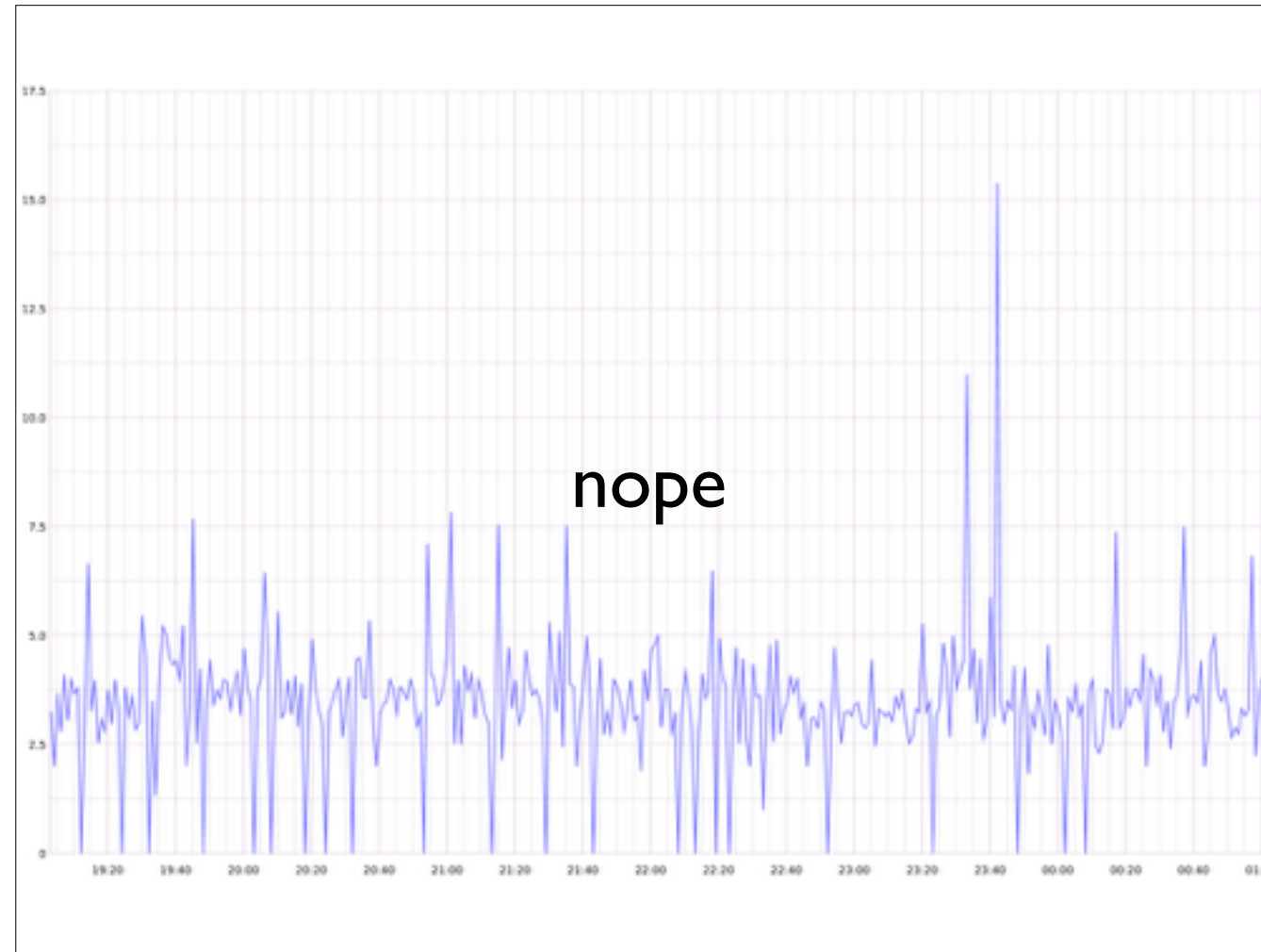
Now that's all fine and good for a single instrument, but what happens if we mic a whole orchestra and and plot the spectrogram for a whole orchestra, we definitely see a lot of structure but it's a lot harder to pick out what's what. All of the instruments are being aggregated together here in one view meaning there is loss of dimensionality--specifically the dimensionality of which instrument is making which sounds. We'll come back to this in a second.

what do service  
latencies look like?

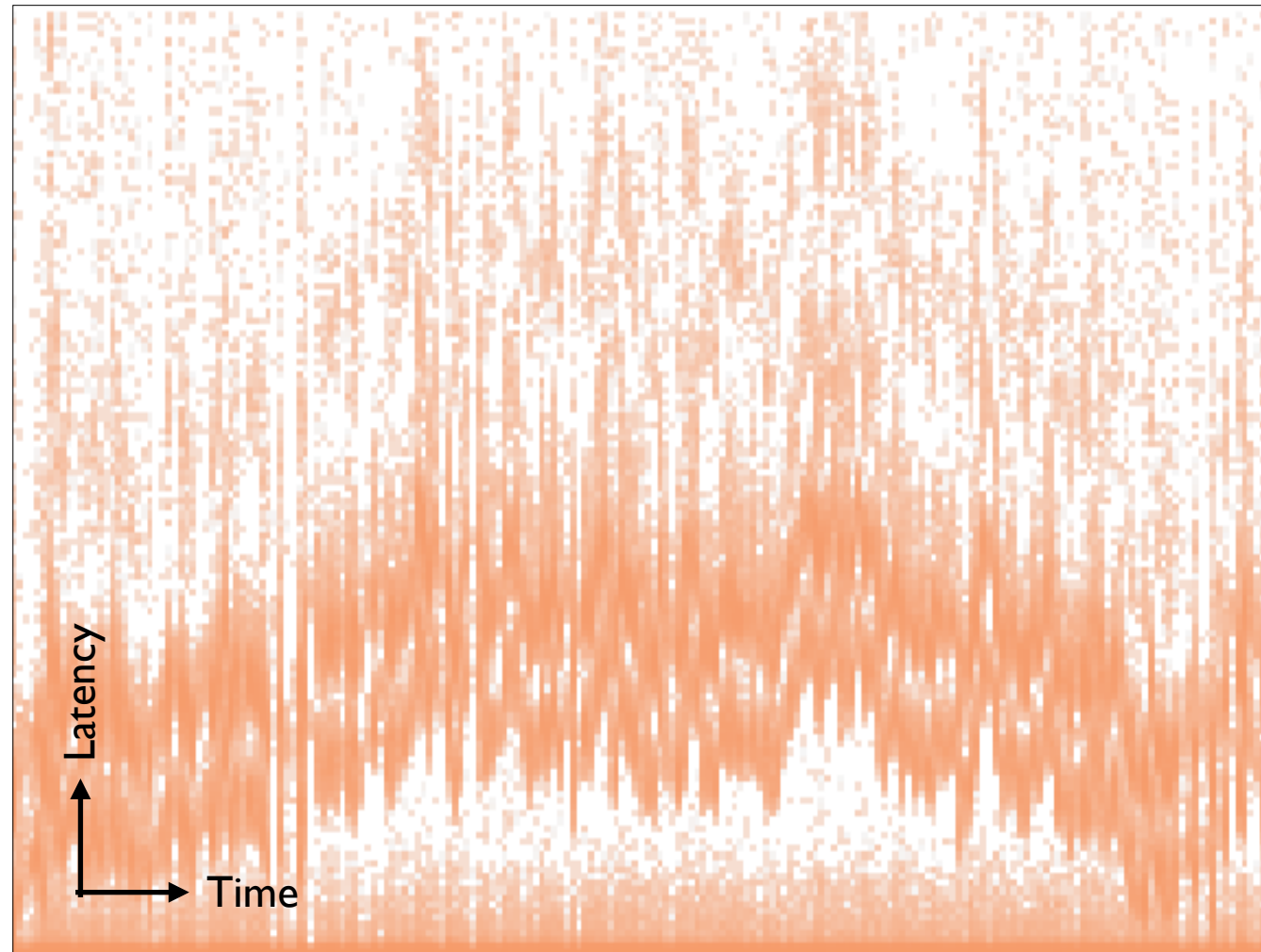
So, with this in mind. Let's say you have a service in production. What do its response times look like?



Like this?



Well no. This might be how you represent the mean or the 99th percentile latencies, but latencies are not point in time measurements. They are a distribution of values over a window of time. And if we measure that distribution of small enough windows of time, we see something like this.



- Same structure as audio spectrogram
- Low latency at bottom, arc of higher response times.
  - Bottom is reads, arcs are writes?
  - Why two arcs for the writes? Coming from two different hosts?
  - What are the hidden variables that explain bimodal structure of this chart?
- Explaining this structure improves our mental model of systems in production and helps us to explain what their dynamic behavior looks like. And when it changes, it gives us something to explain.

And low and behold some of the same kinds of structure we saw in the violin spectrogram, we see in this latency heat map. Now there aren't harmonics, because these aren't audio signals, but there is structure.

We see lots of low latency data at the bottom. That's good. Then we see a few bands of latency forming an arc. And there seem to be two bands. How might we explain what's going on here? Well, the low band might be reads with the bands up top being writes? That might make sense. But why are there two bands? And why do they track so closely to one another. Perhaps if this data is coming from two different hosts which back a service, this could be explained by slightly different write performance on the two hosts. Looking at the distribution this way bakes bimodality jump out. And when you see this kind of pattern, your curiosity leads you to find any hidden variables. And this whole process can be a tremendous boon for your mental model the real behavior of your infrastructure. Not just what your ideal is. Because your ideal is almost certainly wrong, or at least has holes if you don't measure it.

**Moral:**

**Aggregation is information loss**

So to wrap up this anecdote, there are a few morals to take away. The first is that aggregation is information loss. It can be useful, but it does destroy some information. Just like it was difficult to make out the individual instruments in the sound coming out of the orchestra, it can be difficult to find the hidden variables in our data when it's all aggregated together.

**Moral:**

**Latencies are distributions not  
point-in-time measurements**

And that it is important to think of latency information as a distribution that changes over time, not just a value that changes over time. And tooling that makes it easier to us to get visibility into this kind of thing would be a really wonderful thing for the community to have.

# Tail latencies are still good



Also to be clear I'm not proposing this view as an alternative to showing tail latencies. They are an important thing to understand and control.

I'm simply using this example to show that if we simply look at the data, a lot of previously not seen structure can show itself.



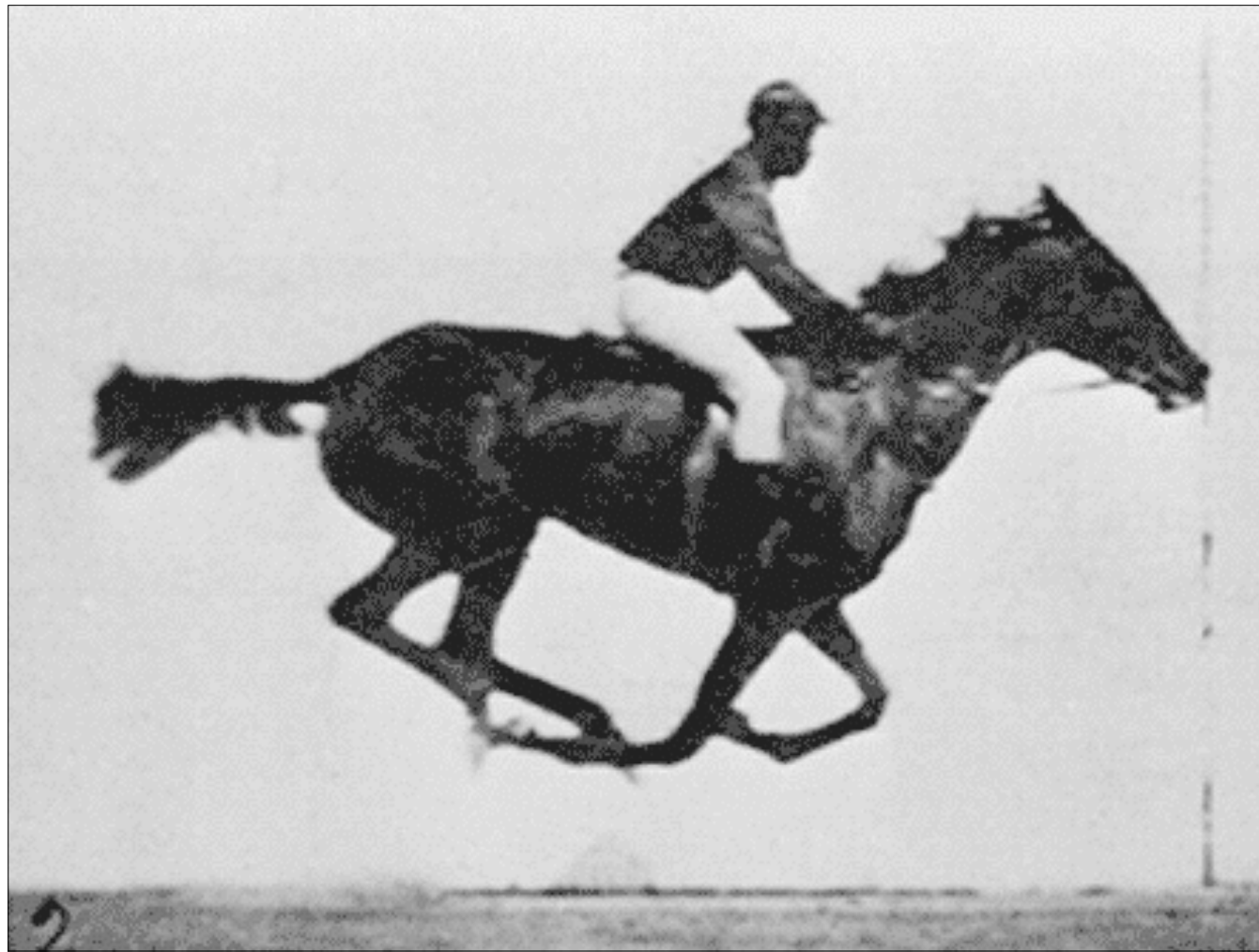
“

In 1872, the former governor of California, Leland Stanford, a businessman and race-horse owner, hired Muybridge for some photographic studies. He had taken a position on a popularly debated question of the day — whether all four feet of a horse were off the ground at the same time while trotting.

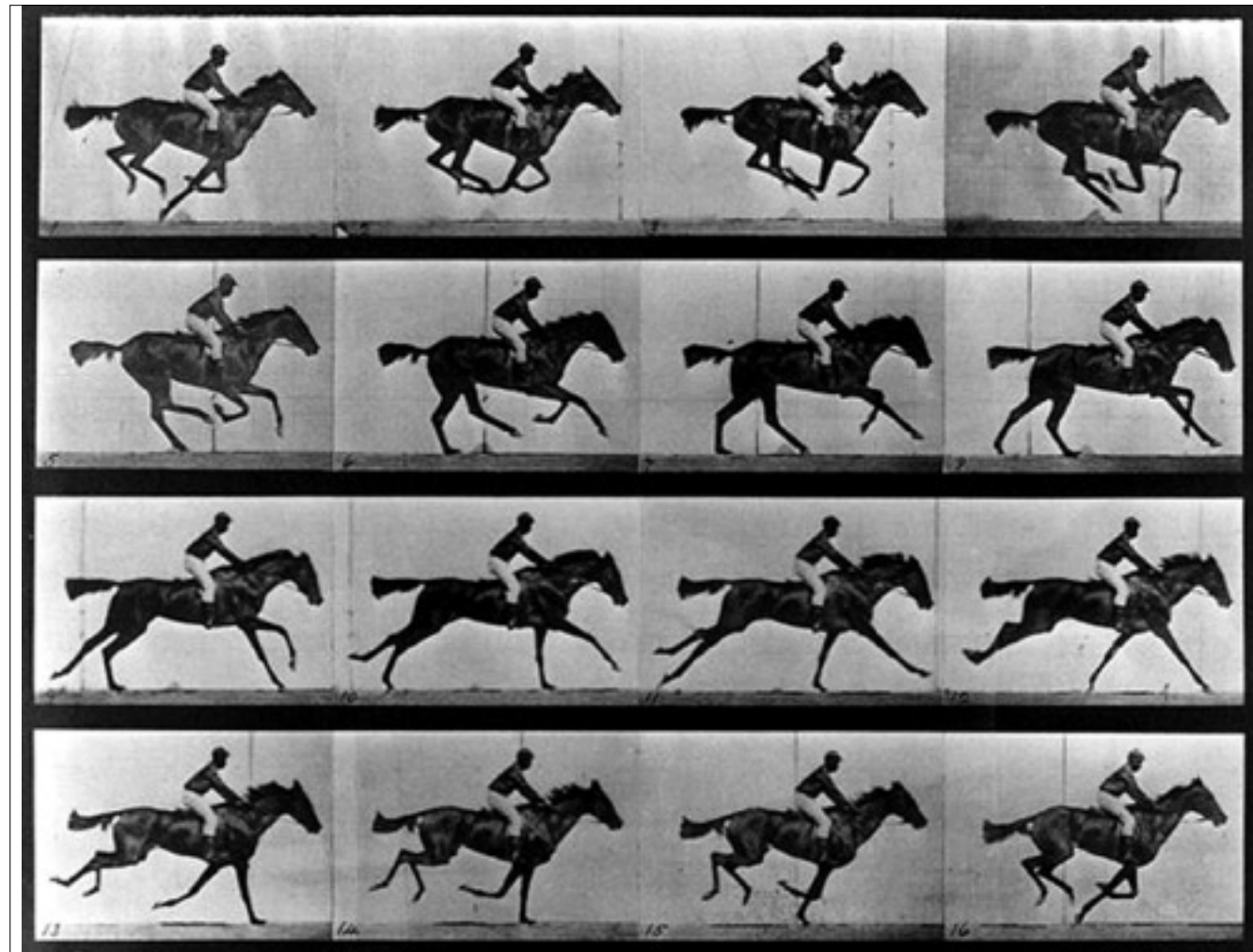
So let's look at one more anecdote. Back in 1872 Leland Stanford, yes the founder of Stanford University and former California governor, took a position on a popular question of the day--whether or not all four feet of a horse are off the ground during full trot. He employed Edward Muybridge to help answer this question.

Happens so fast that it's not  
obvious by watching

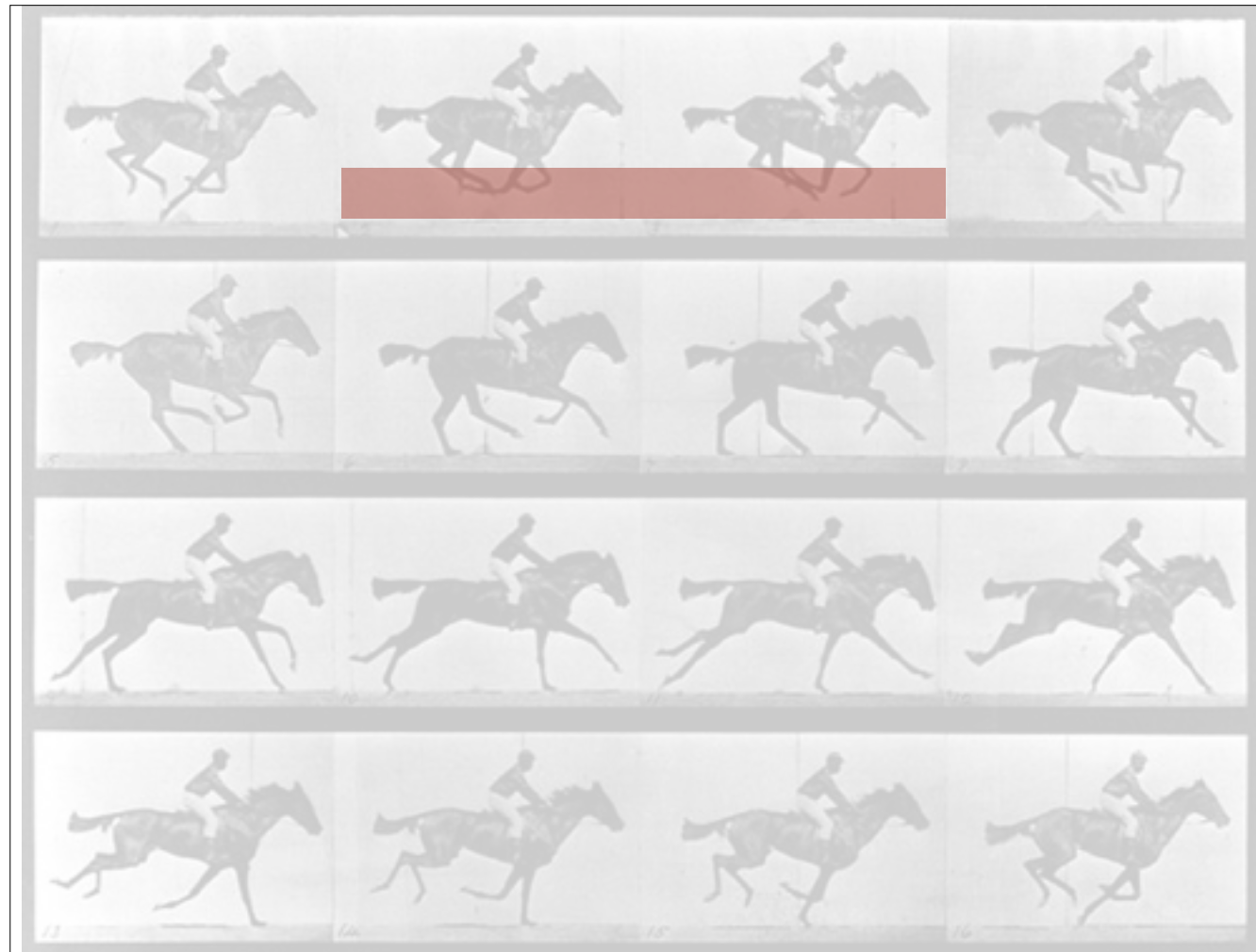
Camera shutters and human  
reaction are too slow to prove



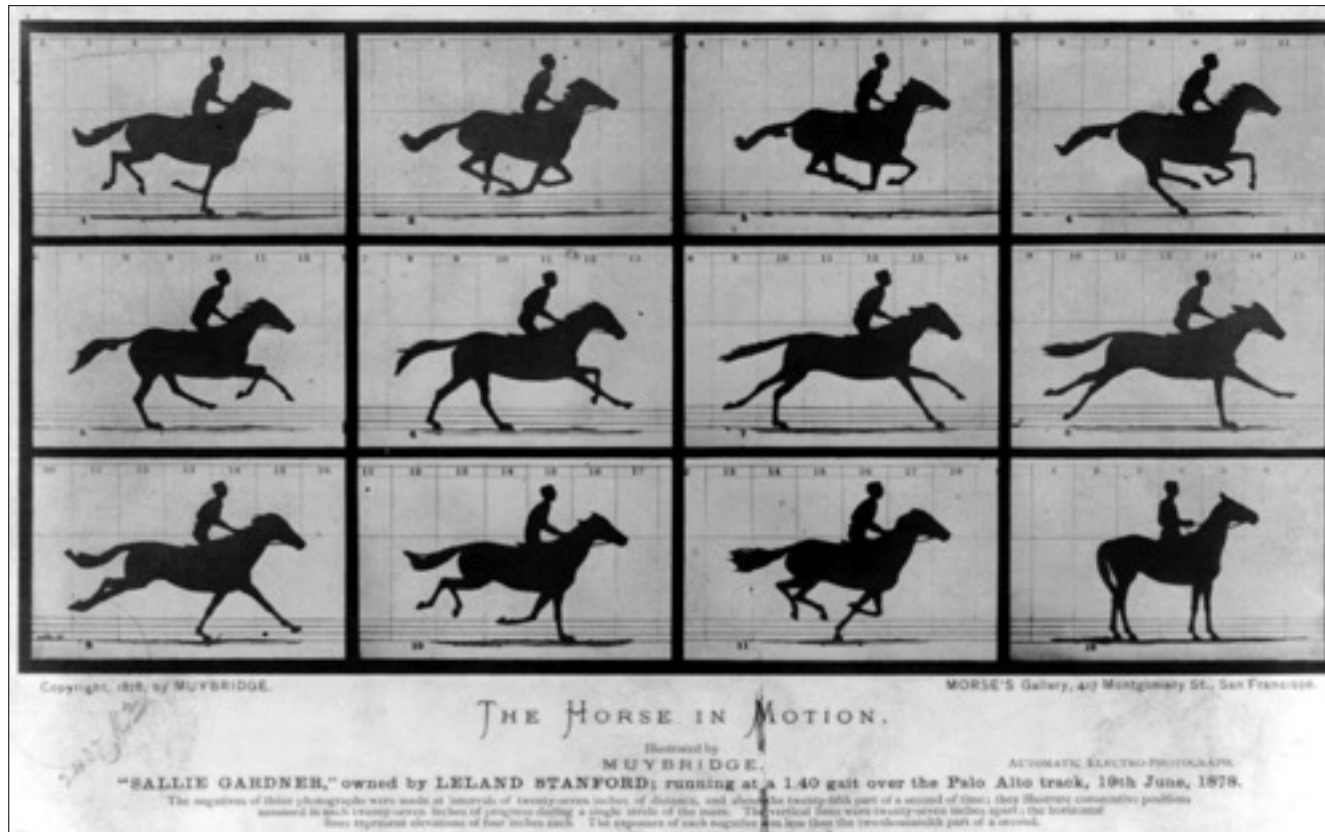
And Muybridge answered this question by tracking a horse during full gallop, taking lots and lots of pictures over time.



Here we see the individual frames from the last video.



And you can clearly see in red here where all four legs leave the ground.



*“The negatives of these photographs were made at intervals of [...] about twenty-fifth part of a second of time”*

And what he found was yes, all four feet do leave the ground. But no one had observed this before. It simply happened too quickly for a person to see it or, much less, reliably prove that this was happening. Even artists before this time, when they would paint a horse in trot, would show one or more of the legs touching the ground. And Muybridge didn't invent a fundamentally new type of measuring device. He just took something we had already, photography, and made it go really really fast and product lots of time series information in an effort to catch what our eyes couldn't.

Q:

I've been transferring 5Gbps  
over a 10 GigE for a full minute.  
Is switch capacity being  
exceeded?

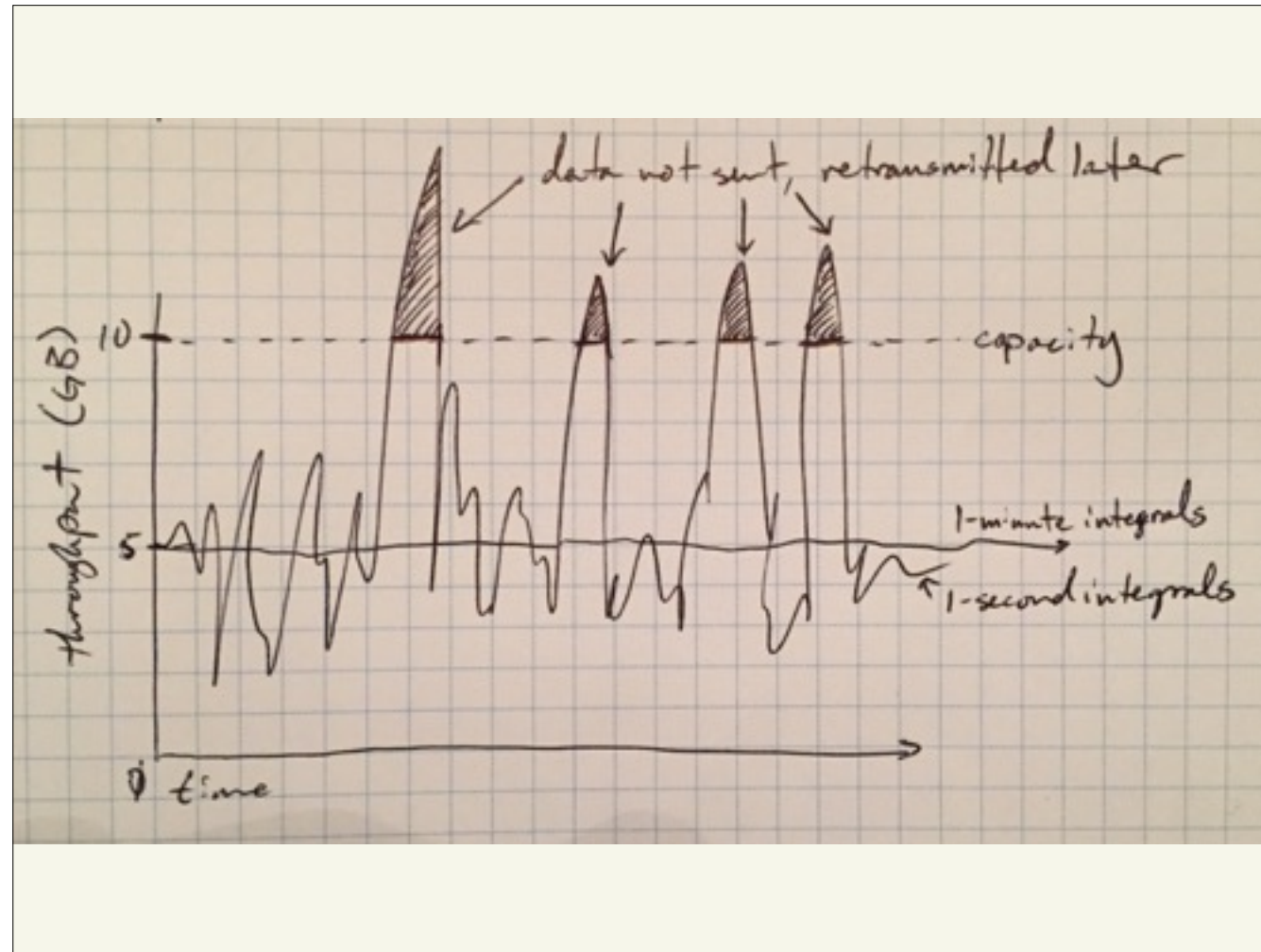
So you maybe know where this is going now. If you've been transferring data at 5 gigabits per second over a 10GigE connection for a minute, am I saturating the link?

A:

??  
??  
??  
??  
??  
??

Well, it might not even cross your mind that this is a possibility. Especially if you are looking at one minute aggregations in something like Graphite. You're nowhere near the limit of a 10GigE device.





But if you actually measure and look at what is happening second by second, or even millisecond by millisecond--in other words, take snapshots with ever-increasing frequency--you might see something quite different. You might see throughput topping out around 10 gigabits, thus inducing retransmits or lost datagrams.

**Moral:**

**Measure as often as possible**

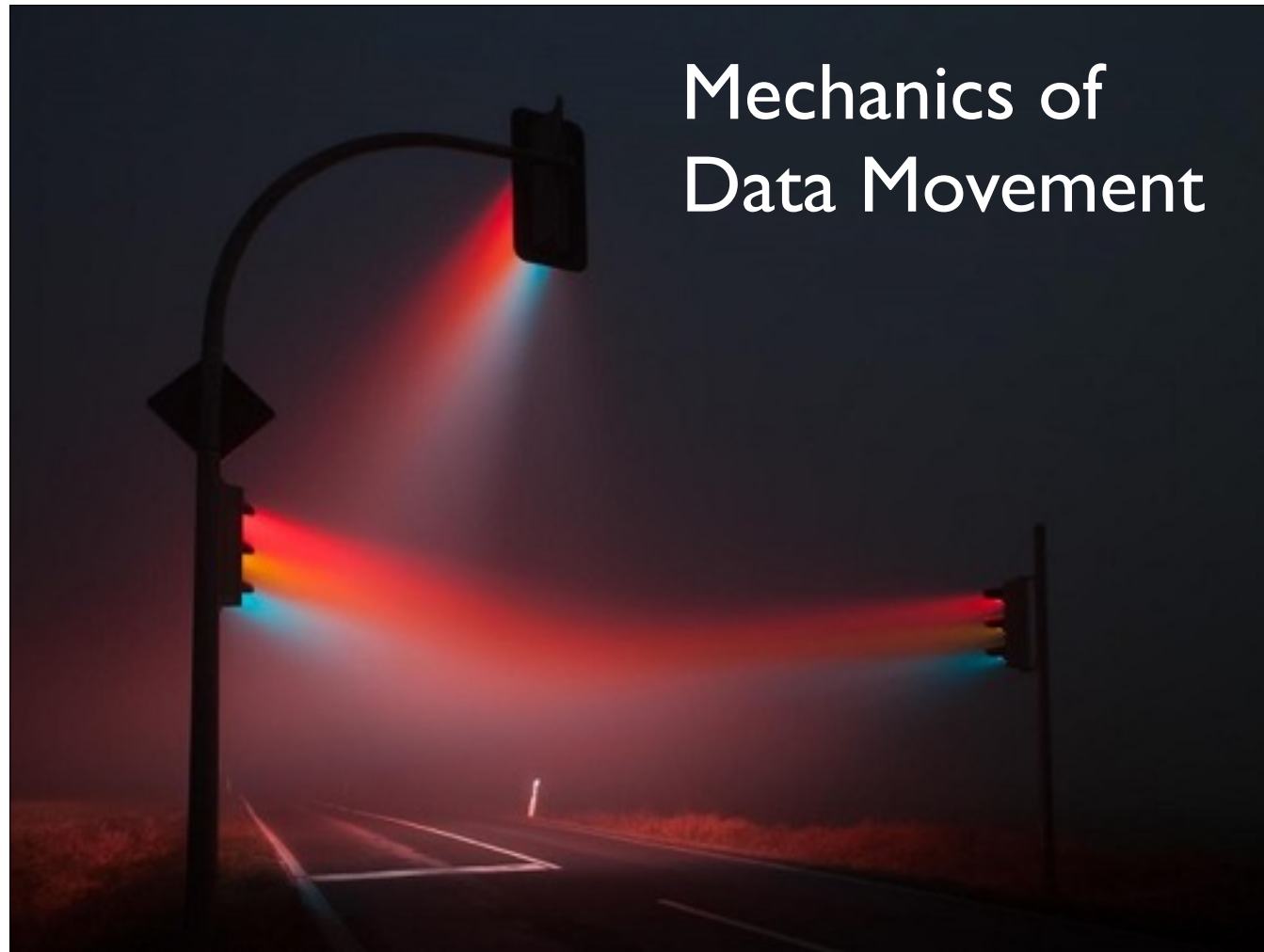
So what is the moral here? Well, measure as often as possible. Computers do things really quickly so measuring at the rate of human thought and you may miss a lot of what's really happening. Kind of like trying to get situational awareness under a strobe light on a very slow cycle.

Be particularly conscious of  
information loss in visuals

Especially when  
pixel resolution  $<$  data resolution

And we want to be particularly careful using charts and graphs to infer something that lies beyond their level of precision.

# Mechanics of Data Movement



So now that we've looked at some entertaining anecdotes, let's put some of this to work. Let's start to consider more of the mechanics of how to collect data and move it about in a way that let's us get a better understanding of the stuff we are trying to measure.

“

Automatically collects 80% of  
interesting user events on the  
web

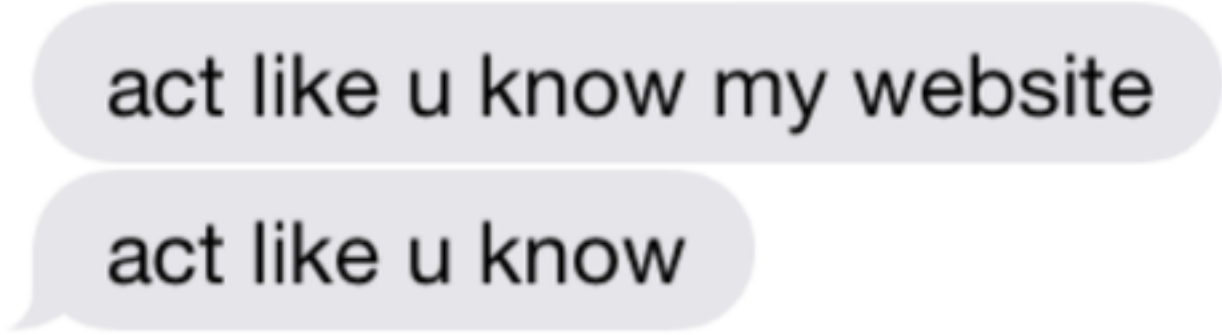
First some motivation. There are quite a few vendors and vertically integrated systems that will collect lots of useful stuff for you. And you should definitely use them. There is a lot of value there.

“

Automatically collects 80% of  
interesting user events on the  
web

**CAVEAT EMPTOR**

But be careful. Because they don't know your product like you do. They don't have a mental model of how your users behave. They don't know your systems and your infrastructure like you do. There is all of this context they are missing.

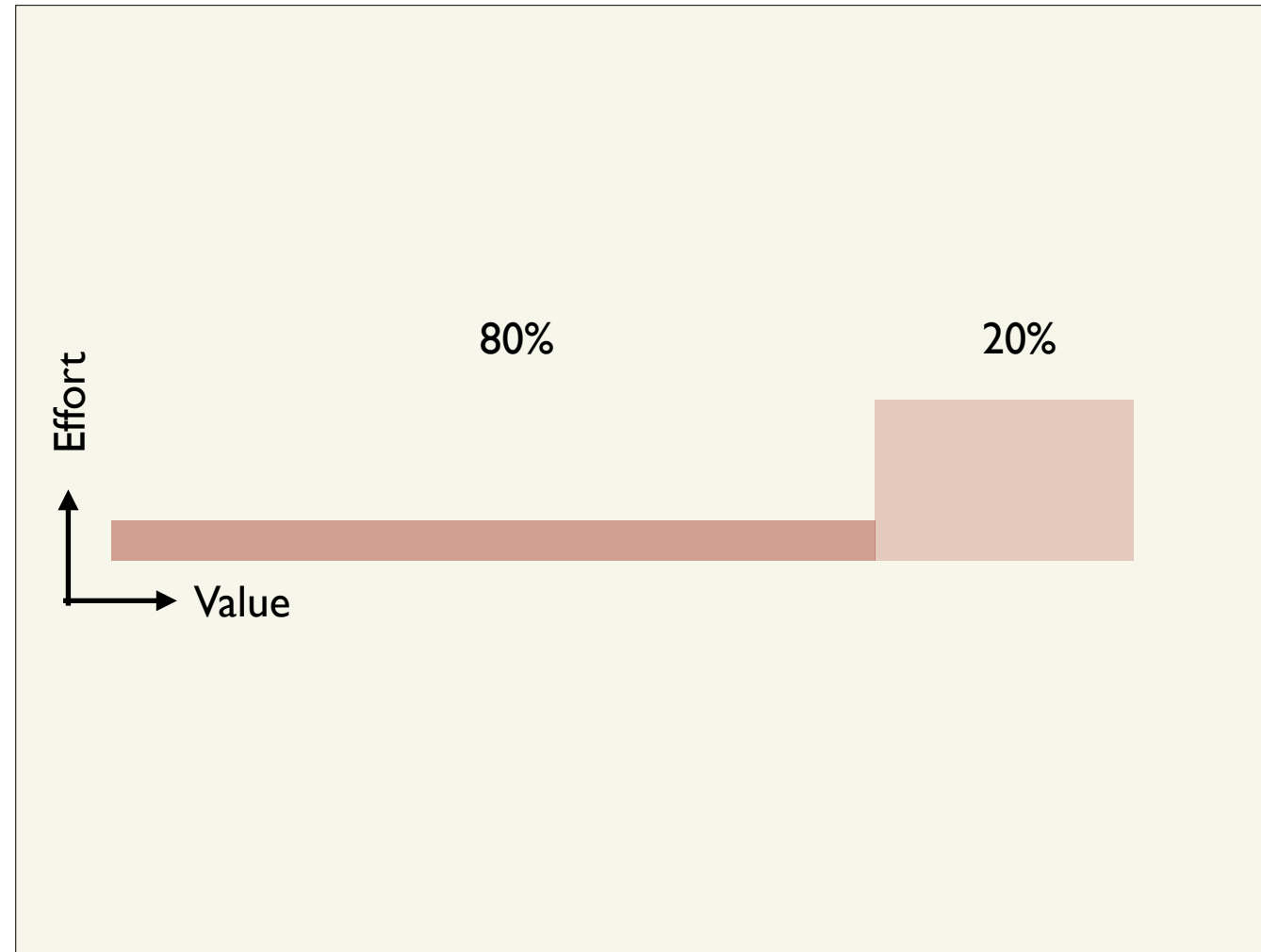


act like u know my website

act like u know

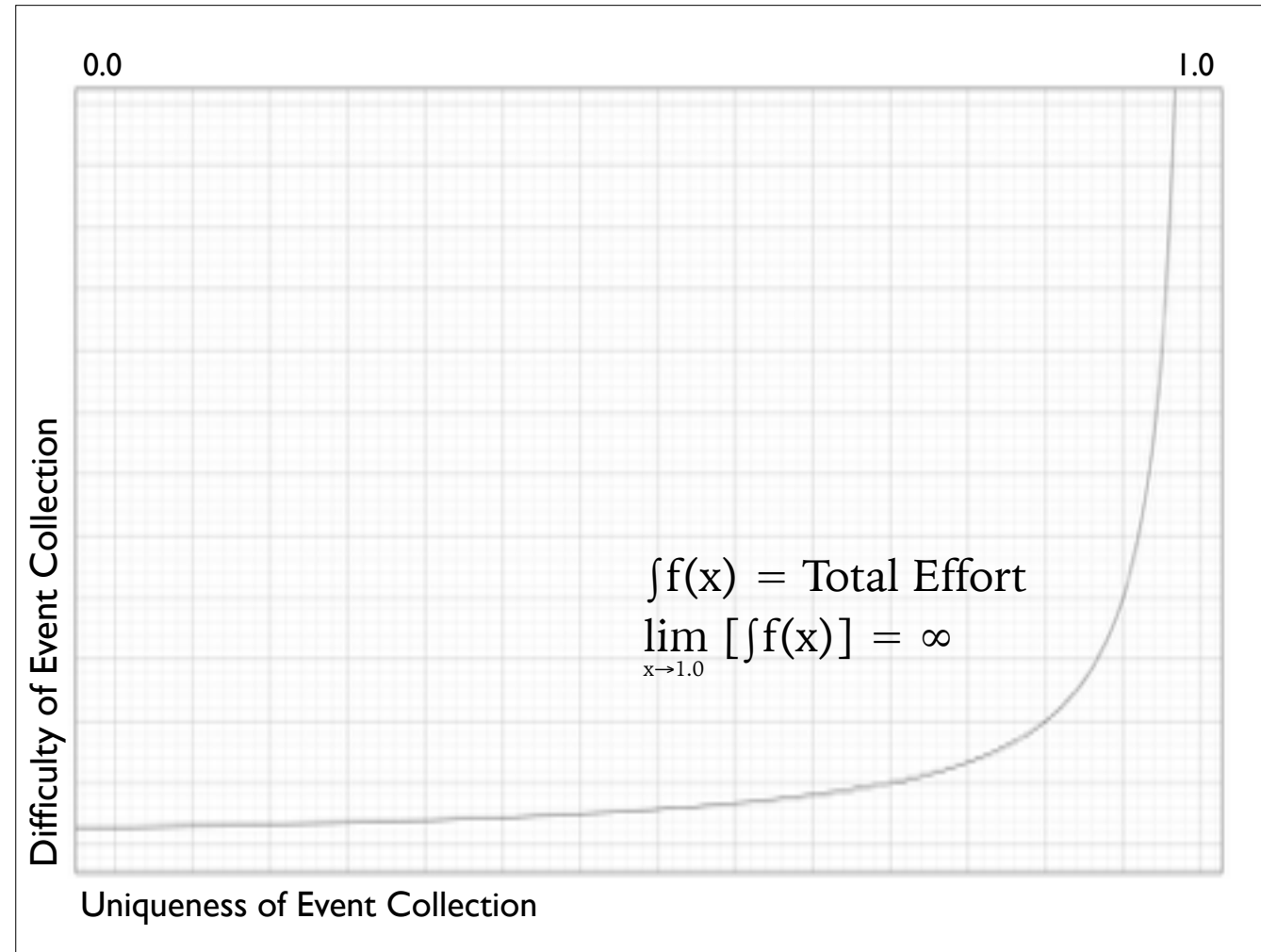
- @blinsay

I think my friend Ben Linsay summed it up pretty well -- "Act like you know my website!".



And while it's tempting to think of our data collection problems following this kind of ill defined 80/20 rule - where we get 80% of the value for 20% of the effort, that's not really the whole truth.





The questions that are most specific to your situation are going to require special attention. They are more likely to require data collection highly specific to your situation simply because it's your situation. And that the more specific our questions become to our specific situation, the more difficulty we should expect to encounter collecting that data.

the universe is under no  
obligation to be observable

Because it's a sad sad world. The universe is under no obligation to be observable by any of us.

and it gets worse

And it gets worse,

our tools don't help

because our tools don't helps us in this regard do they?

the database backing your  
product holds “live”  
transactional data

*...but*

it doesn't help you understand  
user behavior over time

We have this rails app and use active record to save user data to MySQL, but that doesn't really help us understand how that data has changed over time does it?

routers intelligently move data  
from place to place

*...but*

they don't help you understand  
communication patterns in a  
distributed system

And our network card helps us communicate with the outside world, but we don't have pervasive tools at our disposal for really understanding communication patterns in a distributed system.

So these tools do their job, but we have to bring other tools to bear to understand how they are doing their job and how their behavior changes over time. We have to do this if we hope to really understand what's happening.

an aircraft moves humans and  
cargo from place to place

*(in a giant metal bullet traveling at half the  
speed of sound which is pretty cool)*

And at this point I'd like to run with another example. The aircraft. It moves people and cargo from place to place, but what if something goes wrong. The propulsion system isn't going to help us figure that out. It's job is to make the plane go. The lavatory isn't going to help. No, we have to build other systems to help us understand the precipitating state that led to a failure.

*but...*

an aircraft doesn't tell us why it  
crashed and the conditions that  
precipitated failure

The propulsion system isn't going to help us figure that out. It's job is to make the plane go. The lavatory isn't going to help. No, we have to build other systems to help us understand the precipitating state that led to a failure.





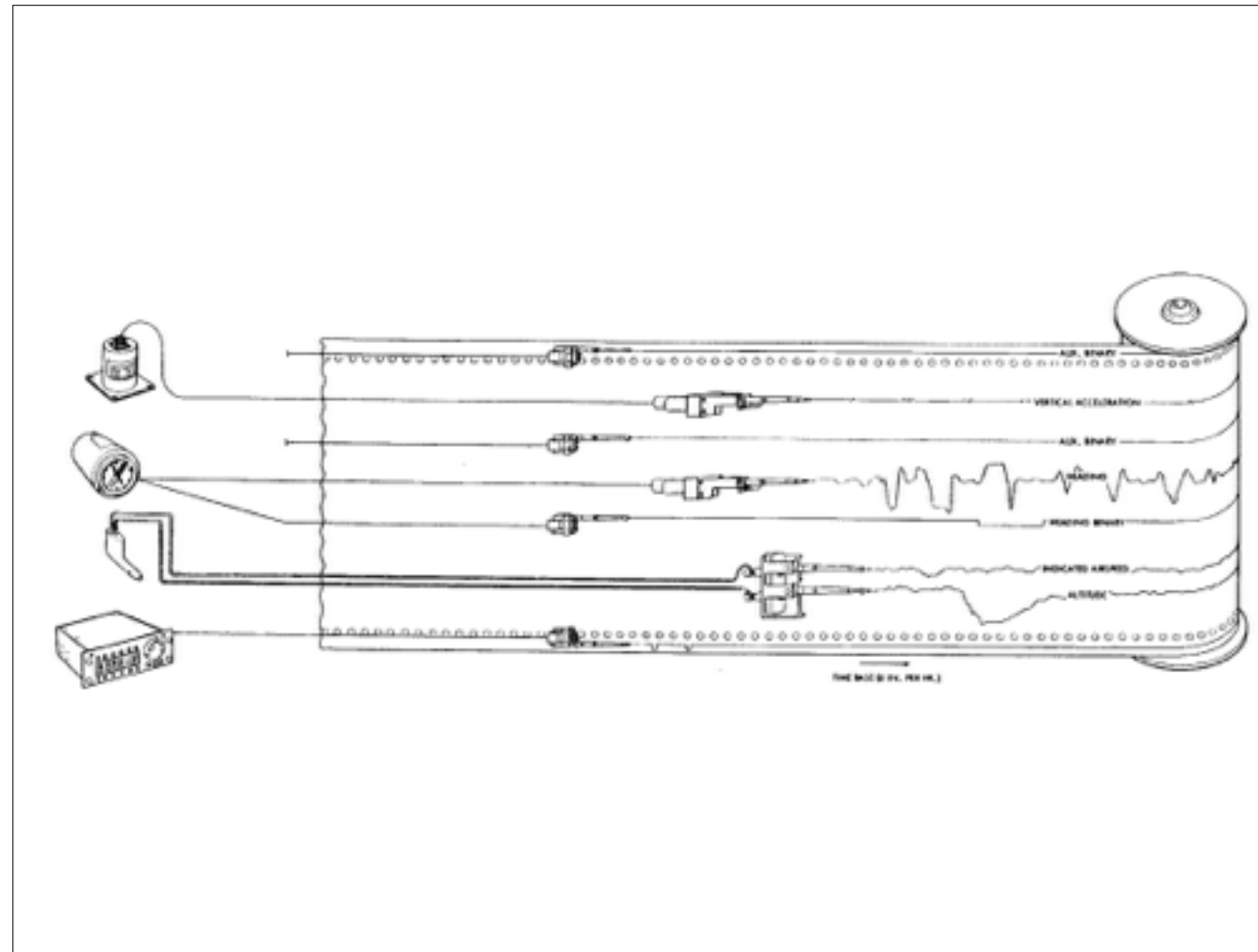
The need for a crash-survivable recording device became apparent following a series of airline crashes in the early 1940s. This spurred the Civil Aeronautics Board (CAB) to draft the first Civil Aviation Regulations calling for a flight recording device for accident investigation purposes. However, recorder development was delayed by shortages brought about by World War II. As a result, such a device was not available, and after extending the compliance date three times, the CAB rescinded the requirement in 1944. The CAB issued a similar flight recorder regulation in 1947, after the war, but a suitable recorder was still not available and the regulation was rescinded the following year.

And we understood this back in the 40s. After a series of airline crashes the CAB (Civil Aeronautics Board) drafted the first regulations calling for a flight recording device. But, we had a little thing happen around this time called World War II and we didn't have the resources to develop this piece of technology. So again in 1947, issued another regulation was issued, but we still didn't have the technology.

“

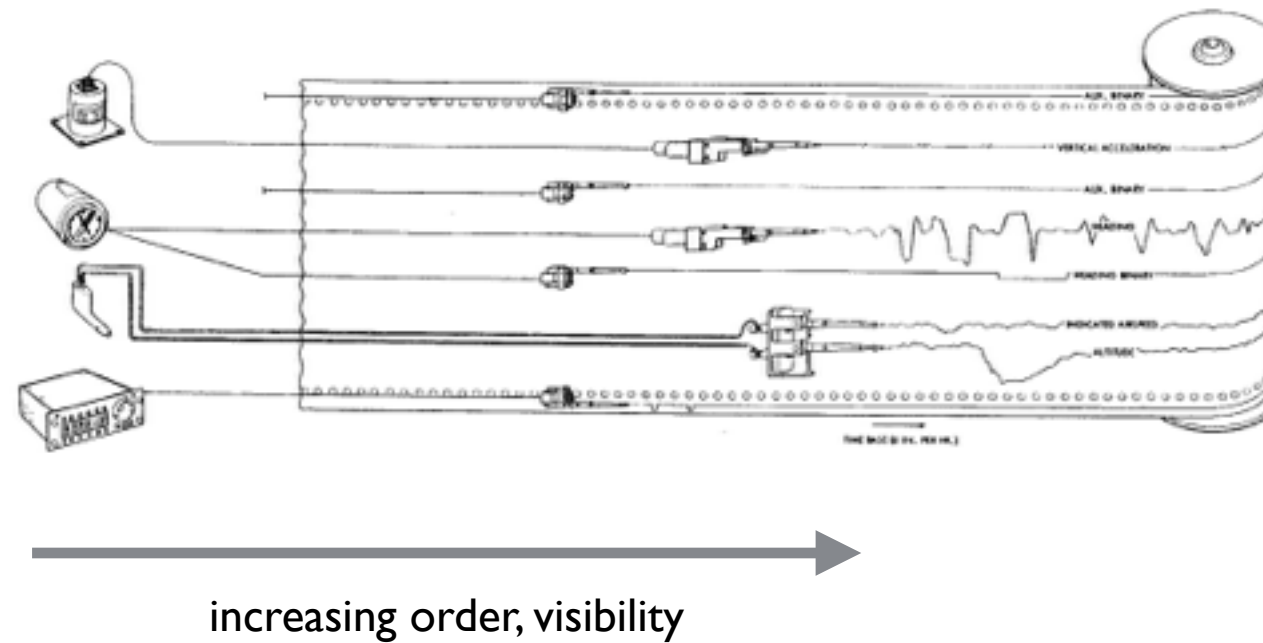
Finally in 1957, after determining that suitable recording devices were available, the CAA issued a third round of flight recorder regulations. These regulations called for all air carrier airplanes ... to be fitted with a crash-protected flight recorder by July 1, 1958, that records altitude, airspeed, heading, and vertical accelerations as a function of time. This marked the introduction of the, first true crash-protected flight data recorder.

In fact, it was a full 10 years later in 1957 that the technology was available that a third round of regulation could be put in place. And they were pretty simple. They recorded altitude, airspeed, heading, and vertical acceleration.



And they did it using this. A Foil Oscillographic Recorder.

a triumph of monitoring and system visibility



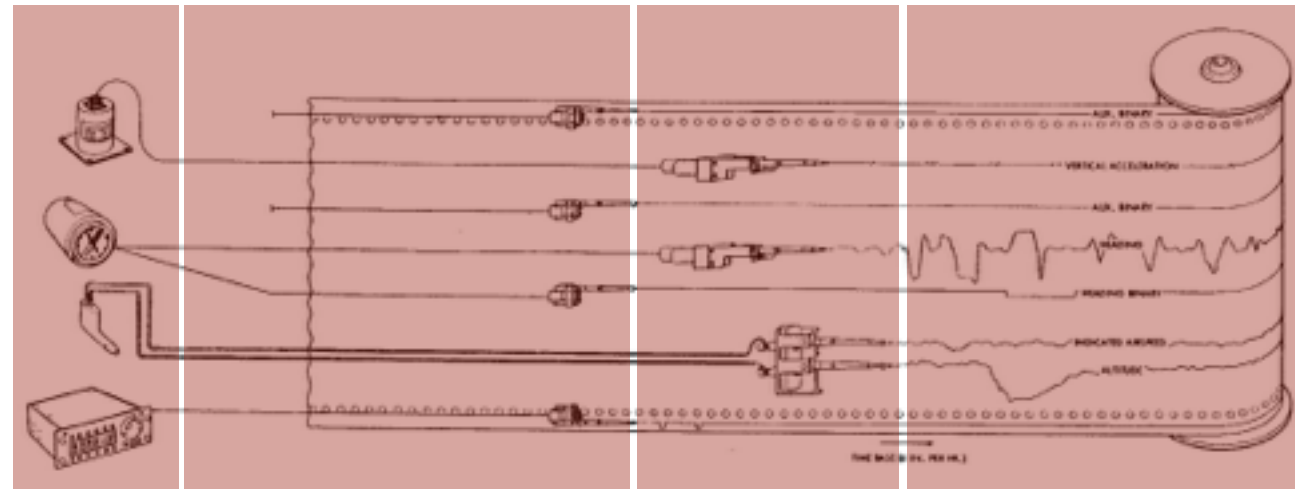
And this thing is truly a triumph of visibility. So let's work through it from beginning to end (or flatten the list and work it from head to tail, there's a functional programming joke in here somewhere).

Collect

Normalize

Record

Analyze



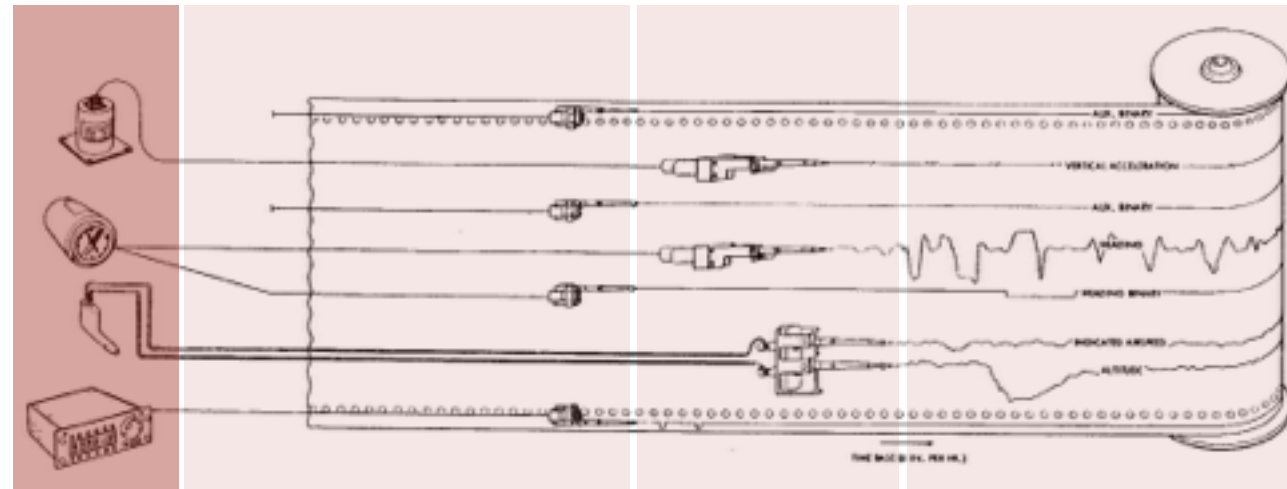
And for purposes of analogy, we're going to think of the job of the foil oscillographic recorder in this way, from start to finish. And we'll see what we can learn from it.

Collect

Normalize

Record

Analyze



The universe is indifferent to our pain.  
An unbounded amount of work happens here.

First the responsibility is to collect the data we care about in the event of a crash. This is where an unbounded amount of work happens because, as we all know, the universe is indifferent to our pain and it doesn't care how hard we have to work to understand it.



*whatever it takes*

made with sparkles.com

So if you remember nothing else, it would be to do \_whatever it takes\_ in the data collection phase. But specifically, here are some ways to construct good events.

# Event Collection

- Collect as much context as feasible. Esp context that is
  - Expensive to retrieve later
  - Mutable
  - PII Sensitivity
- Collect as frequently as feasible (cc Muybridge)
- Don't wait until you have a perfect, or even working, analytical system. Collect for the future.

Events should contain as much context as possible. Imagine that you are processing this event at some time in the future, and it should contain everything you need without joining with some other data source. Because that's probably going to be really costly and add lots of complexity and latency to this hypothetical thing you'll be building.

This is especially important when the context is mutable and may change over time. For example, one thing we do at github is gauge language popularity over time. So if we look at a stream of pushes, we should be able to look at this event and directly unpack what languages are contained in the diff without actually looking back at the code or, worse, reaching back into git storage to find that out.

Another thing is to collect as frequently as possible. Sometimes, you collect events representing some outside stimulus, like a page view. But at times when you're polling some other metric like disk utilization, network throughput, the value of a stock, etc,

And most importantly, start collecting `_something_`. Because before you do your message loss rate is 100%. And once you do start building out more and more analytics infrastructure and want to start training your models, you'll be glad to have all of that historical context to use.



# Helpers

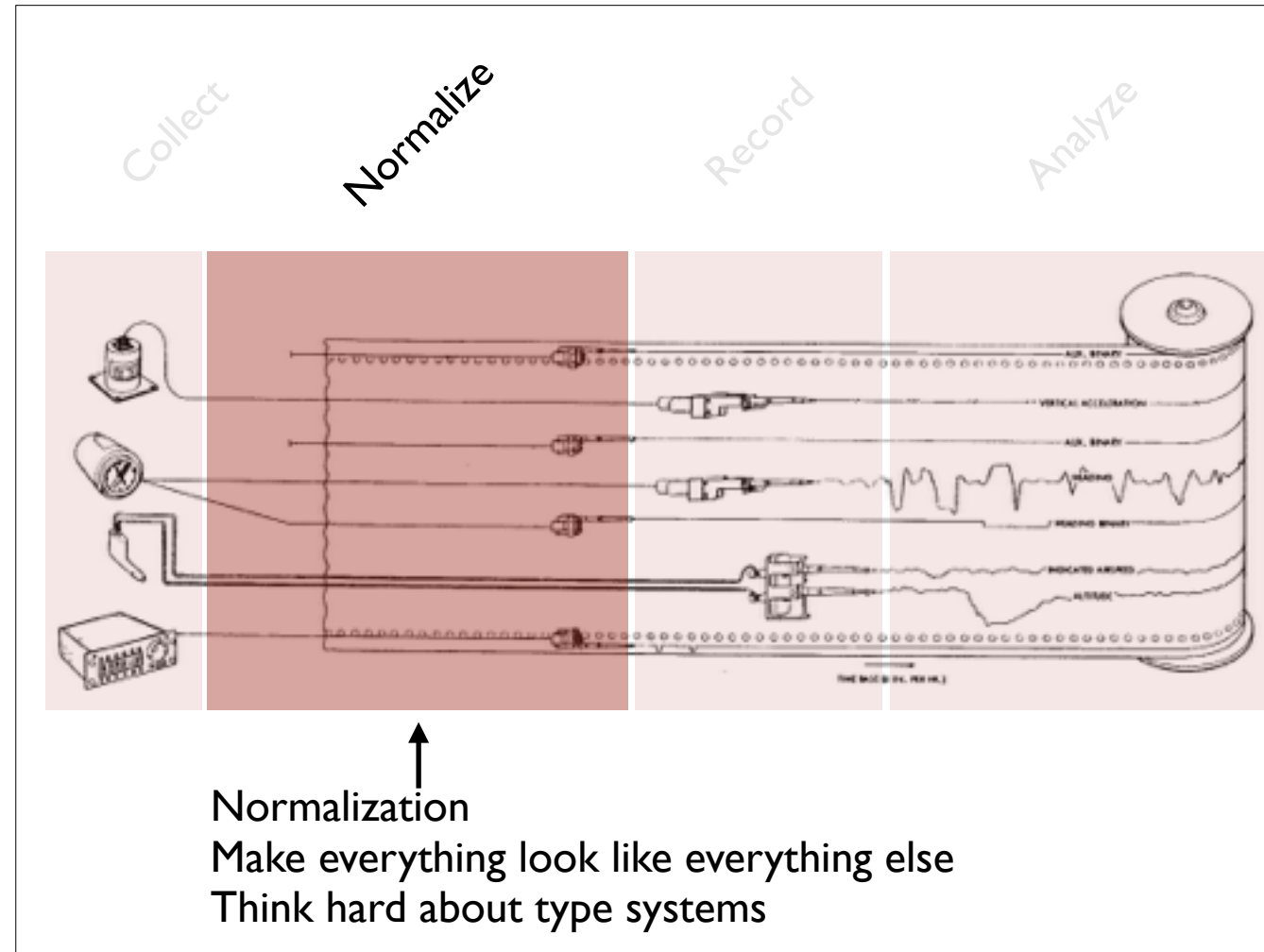
- libpcap
  - Pull stats from packet headers
- Instrument ActiveRecord models, but careful not to block unicorns
  - Context collection accrues latency
  - Write events to separate process which batches and submits to event collection API
- Local daemons to poll iostat, iotop, jstat, etc and report to collector

And I've mentioned the database and networking examples a couple of times, so I thought I'd mention some ways of dealing with that here.

If you're interested in measuring how a computer is communicating over the network, libpcap is a good choice. It is a nice C library that lets you inspect packet headers for port/protocol information and payload sizes. It's a nice primitive upon which to build network visibility tools.

As far as instrumenting changes to a user database, the approach we've taken at GitHub, since we use ActiveRecord, is to instrument the models themselves to capture any change events. The thing to look out for here is that this work happens in the context of a web request and we'd like to keep those fast. We don't want our event collection to start slowing down the site and creating a frustrating experience for users. So we either trigger a background job at this point, or write these events to a local queue, then we have a process which does nothing but pull events from that queue and write them to our event collection service.

Another approach we take is to simply write little daemons that poll various system stats and push them up to a collection service. And remember the lesson earlier about polling to do it as frequently as feasible to avoid the strobe light effect.



With event collection out of the way, we can start to discuss this next step. Once the measurement devices get their reading, they have to turn those signals into something that normalizes the way their etchings take place. They need to stay in their lane and not corrupt data written by others. The analogy here isn't direct, but one way to think of this is in terms of the types you use to represent your events. We want a predictable structure for these events so that they can be recorded uniformly for interpretation later.

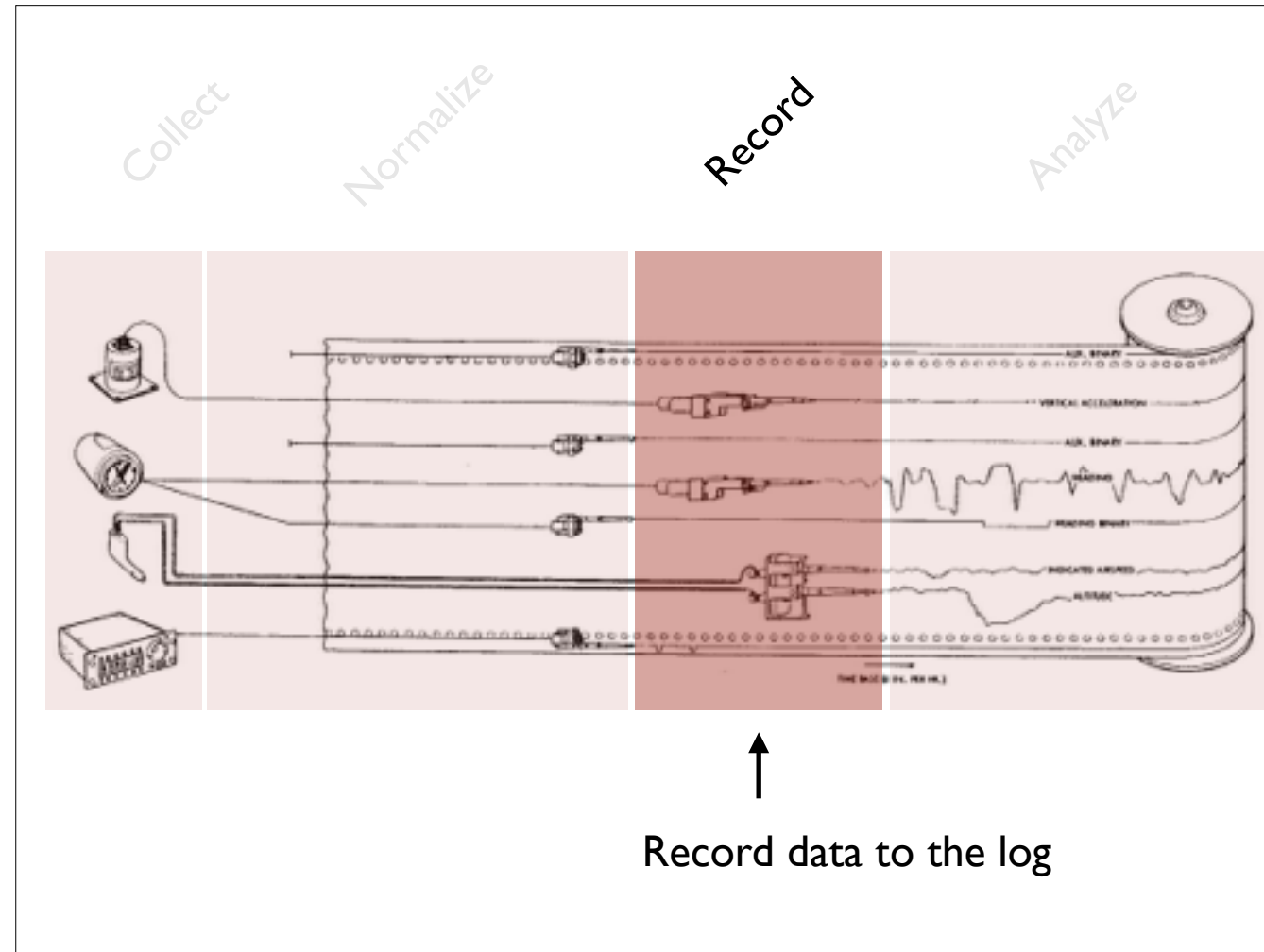
# Predictable Event Structure

- !!! Dimensions + Metrics !!!
- Common data represented uniformly
  - Timestamps, logged in user ids, etc
- Differentiate between dimensions and measurements
  - Esp important in dynamically typed languages, but important for static as well

And I know I mentioned type systems, but don't necessarily mean that in the context of any particular language so we don't have to get into a debate over static versus dynamic typing. But there is some predictable structure with which we want to imbue our events.

First, it's important to have a distinction between dimensions and measures. Dimensions are the things like when something happened, the user that performed an action, the type of event, if the event is, say, a commit to a source code repository, the languages used in that commit are another example of a dimension. These are the parts of a that categorize the metrics. And the metrics are the numbers you're collecting. Things like latency, number of unique users seen, or lines of code contained in a particular commit to a source code repository.

Now, if you're using static typing to represent events, it's still a good idea to know which of your properties are dimensions and which are measures. But the distinction is especially important in dynamic languages to help you know where to look for your categorical data, and where to look for your numeric data.



So on to recording. This is the point at which, in the recorder we see here, the data is actually etched onto the scrolling foil. It's done here, iirc at a rate of about 6 inches per hour.

# Event Recording

- API
  - Simple HTTP API (Post JSON to `collector.yourproduct.co/events`)
- Storage:
  - Prefer data logs over queues
    - Think Kafka instead of Kestrel, etc
    - Multiple consumers with independently managed offsets
    - Consumer groups keep data flowing in the event of a failed consumer
  - Dedicated consumer for writing blocks of data permanently to S3
    - Balance compactness with utility. Make data consumable by Redshift if possible
    - EMR over raw data using HIVE

But luckily we don't have to etch our data onto a foil sheet. We get to build a handy service with a well-defined API. Nothing novel here. We just accept as a POST, a collection of events.

What we do next isn't particularly difficult but it is easy to get wrong. We write the events to a data log. And the distinction here between a data log and a queue is very important. As you might recall, queues allow one consumer to get one event. With a data log, each consumer can, if it so wishes, consume the whole stream. Each consumer, or group of consumers, can consumer a data stream fully. This means that multiple workers, in parallel, can focus on computing different things on the data without interfering with or blocking one another. I'm really a big fan of using Kafka for this.

But the important thing here is that using a data log versus a queue allows lots of different systems to process data as it is received rather than having everyone block on one service to doll that data out and figure out how to multiplex that out. But if you want continuous visibility, use a data log.

And because data doesn't stick around in your data log forever, one nice thing to do if you can, is have a dedicated consumer which does nothing but batch up events in blocks of, say, 10 minutes or an hour, and upload them to S3 for permanent storage. It's pretty cheap and opens up some really nice analytical capabilities like RedShift queries and running HIVE queries over your raw data.

# Event Recording “Helpers”

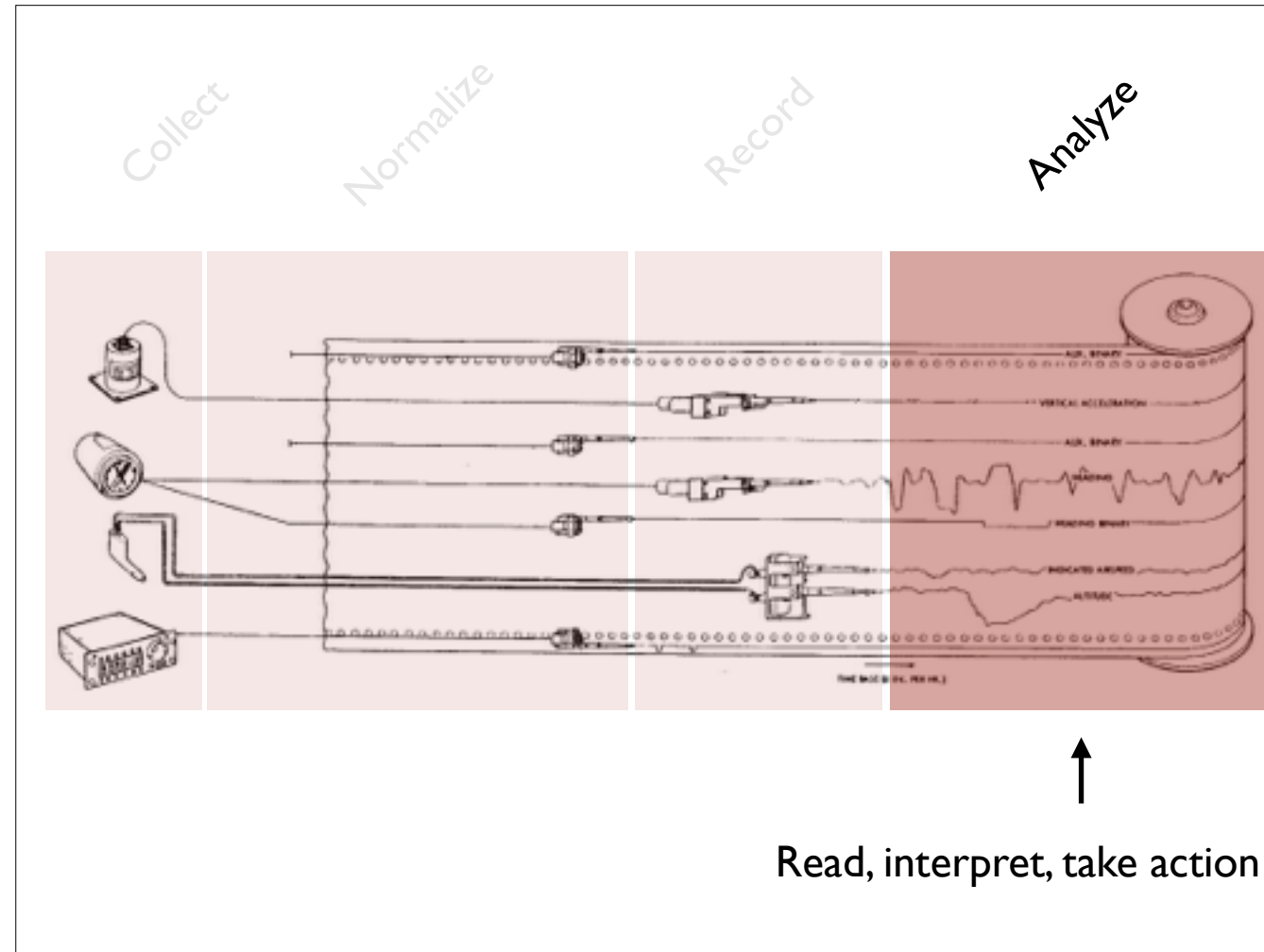
- Geocode IP addresses
  - cc Normalization
- Attach other valuable context to incoming events

Now thinking about the API that is accepting events and writing them to the log, this service can actually add value to the data it receives. This is one of the most important things you can do to localize complexity rather than distribute it out to all of the event collection sites. For example, if your events contain IP addresses, you might want to geocode those to know what region or country they are coming from. And there are probably lots of others specific to your own use cases. It's important enough that I wanted to call attention to it here. This is also something that, if you've done well at normalizing event structure, it becomes simpler to augment events with this additional categorization.

# Feedback

- Send events back to yourself
  - The system we are building needs to, itself, be observable.
  - Can dog-food our own API
  - Breathe consciousness into the universe by making it self-aware

And lastly, the system can send its own significant events back to itself once you've reached a certain point of sophistication. For example, if you have a consumer reading from the data log and uploading blocks of events to S3, you might send one more events into the stream which describe that upload event - how long it took, how big the file was, when it was uploaded, etc.



Now, we get to the last part. Analyzing, interpreting, and taking action on the stuff you've been collecting. The key observation to make here is that we have a streaming system. And that streaming system has lots of normalize, well structured, context-rich events coming from all corners of your infrastructure. From how users use your product, to what is happening in your infrastructure, to how various systems are performing. And, again, because we are representing this data in a data log, we have continuous visibility into this stuff. And our ability to see how the world around us is changing, we're limited only by the latency from collection to being picked up by one of the consumers.



# Streaming System = Continuous Visibility

The key observation to make here is that we have a streaming system. And that streaming system has lots of normalize, well structured, context-rich events coming from all corners of your infrastructure. From how users use your product, to what is happening in your infrastructure, to how various systems are performing.

And, again, because we are representing this data in a data log, we have continuous visibility into this stuff. And our ability to see how the world around us is changing, we're limited only by the latency from collection to being picked up by one of the consumers.

# Analysis

- Direct use
  - Graphite is good but is best for data of low dimensionality
  - Aggregate interesting data into PostgreSQL
    - Let the database do the work for you!
- Roll your own visuals on top of that

The first is direct use. That is, directly taking that information, putting it in a database, storing it in hadoop for bulk analysis, and even building visualizations directly atop some of that data. Graphite is a good, but remember that these well constructed events have a lot of context and a lot of dimensions--and graphite isn't great at handling data with dimensionality, and especially not high cardinality dimensions. But it is a good tool to use and it's super hackable and can save you from needing to write your own visualization tooling.

# Analysis

- Perform aggregations and joins on one or more streams
- Write that out as another stream
- Repeat to form higher and higher levels of analysis

The other camp of types of consumers I wanted to talk about real quick is the kind that simply perform some kind of aggregation and feed it back into the event stream. Say you have several consumers that want breakdowns of page views per day by country. Rather than having everyone who wants that kind of data compute it from the raw stream, you can have one consumer do that aggregation work, write it back to kafka as a new topic, then have the consumers feed off of that. And for this type of work specifically tools like Samza can help you to manage that lifecycle dealing with a consumer dying and coming back up with the right state.



# fin?

And this is where I really hate to end the talk, because at this point is where all of the really interesting stuff happens. And we're starting to build some demonstrations internally that help us see GitHub data in ways we haven't seen it before. It's really exciting and I hope I've helped you to think about your own problems in a different light so that we can share in this excitement together.

References, slides, and  
other talk material

[d2fn.com/phillyete2015](http://d2fn.com/phillyete2015)

thanks  
-@d2fn