Observability 👀

Know what's happening in production

Max Edmands // honeycomb.io // Nov 2020

Hi, I'm Max, it's nice to meet everyone.

I'm a professional software engineer, this slide has a bit about my history on it.

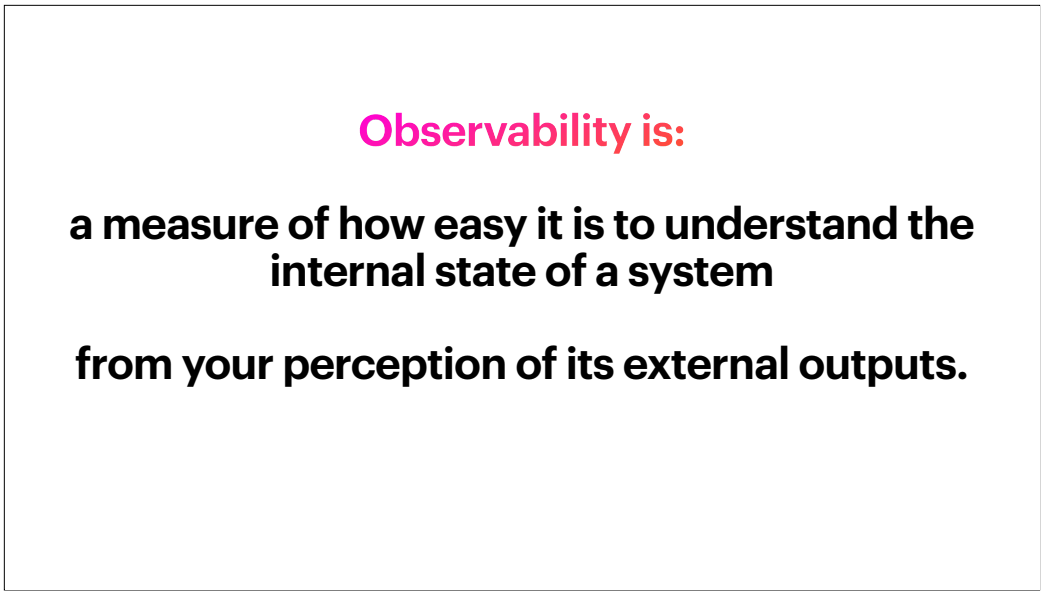
Before I get started: I haven't given a presentation to a group this large over Zoom. I'm going to do my best to regularly look at the chat, but if there's ever a moment where you have a question and it seems like I'm moving on without addressing it, feel free to unmute yourself and yell at me or something, I want to do my best to make sure that your question gets answered.

So, all of the companies on this slide were building internet services of one sort or another.

> Almost everything I'm going to say in this lecture I've learned on the job, more or less
>
> I work at Honeycomb now, we make the observability tool you'll be using in this class. I think that it's pretty great, and I'm gonna be walking you through some of the things you can do with it during this lecture.
>
> Lastly: I think I'm obligated to say here that my opinions are my own and don't represent my employer in any way.

Observability is:

So: What is observability?



Here's one definition:

Observability is a measure of how easy it is to understand the internal state of a system from your perception of its external outputs.

As software developers, we have a superpower -- we can read the code. This means we have the ability to see inside a system, all of the messy guts and complex logic, and try and understand it that way.

But we're not omniscient. Underneath the code we wrote is more code that someone else wrote, and underneath that is more code, and if you go deep enough eventually you get to hardware that was manufactured by a company somewhere, plugged into an electricity grid that you have no control over. And so ultimately, if we want to know **what's going on** with our software, we have to look at things that it's doing on the outside.

Observability is:

a measure of how easy it is to understand the internal state of a system

from your perception of its external outputs.

What kinds of external outputs might a system have?

Can anyone give an example of an external output of a system?

* logs
* data files or records
* server responses
* user experience (you can see the app responding to input!)
* core dumps / stack traces
* error messages
* blinky lights or sounds
* emails, text messages, or push notifications
* graphs and charts

Observability is:

a measure of how easy it is to understand the internal state of a system

from your perception of its external outputs.

Why is it important to understand the state of your system?

Another question - why is it important to understand the state of your system?

Because you want to know if it's working!

* You wrote a thing, now it's running - but is it actually doing the thing you expected it should be?
* There's no way to tell unless you can observe it.

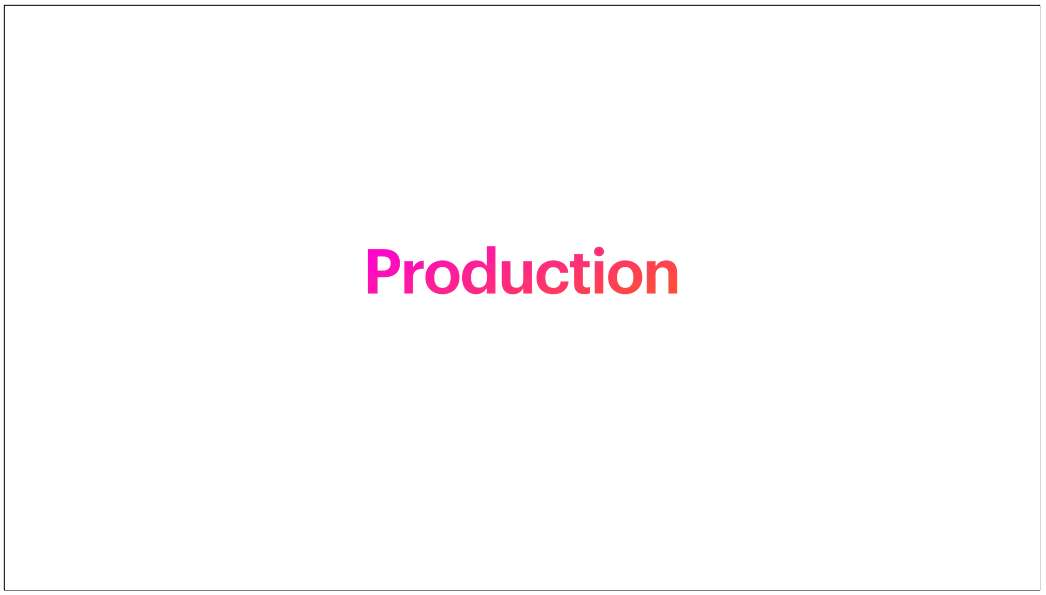
Because things go wrong!

* And they go wrong in lots of exciting ways that you might not be able to anticipate when you're writing your code
* Not just software bugs: user error, or assumptions you made when writing the code turning out to not be true 100% of the time
* When they go wrong, you need to know as much about the problem as possible if you want any hope of fixing it.

Because you need to make changes!

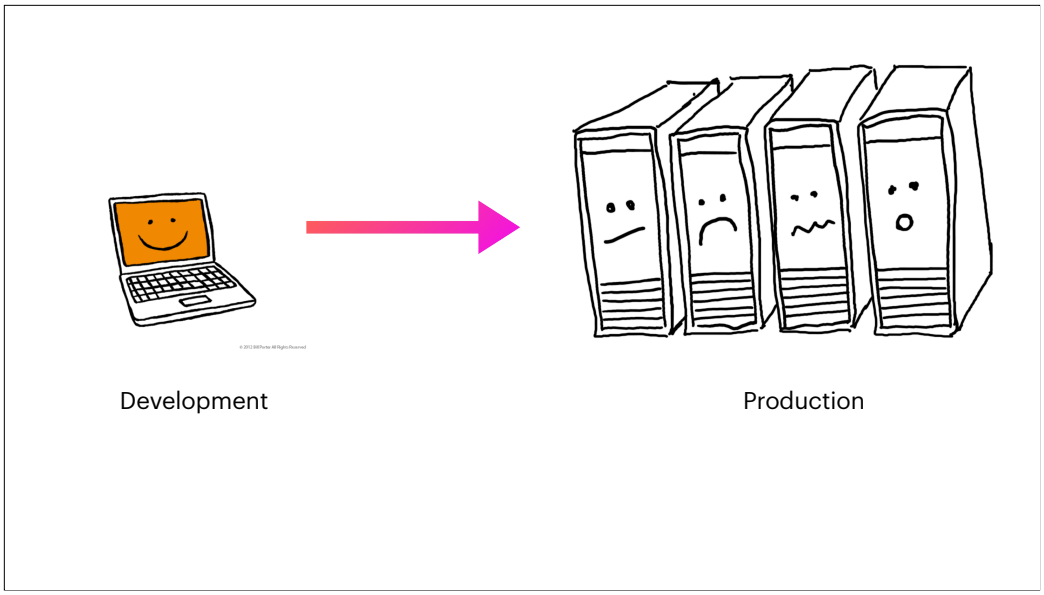
* It's very, very rare that I sit down to make some software, write all the code in a single take, hit "run", and then walk away, never to touch the code again.
* Even if it does things perfectly and there are no bugs, maybe my needs change, or I want to add a new feature.
* If so, I need to be able to understand what's going on in my system now in order to figure out exactly what I want to change

later



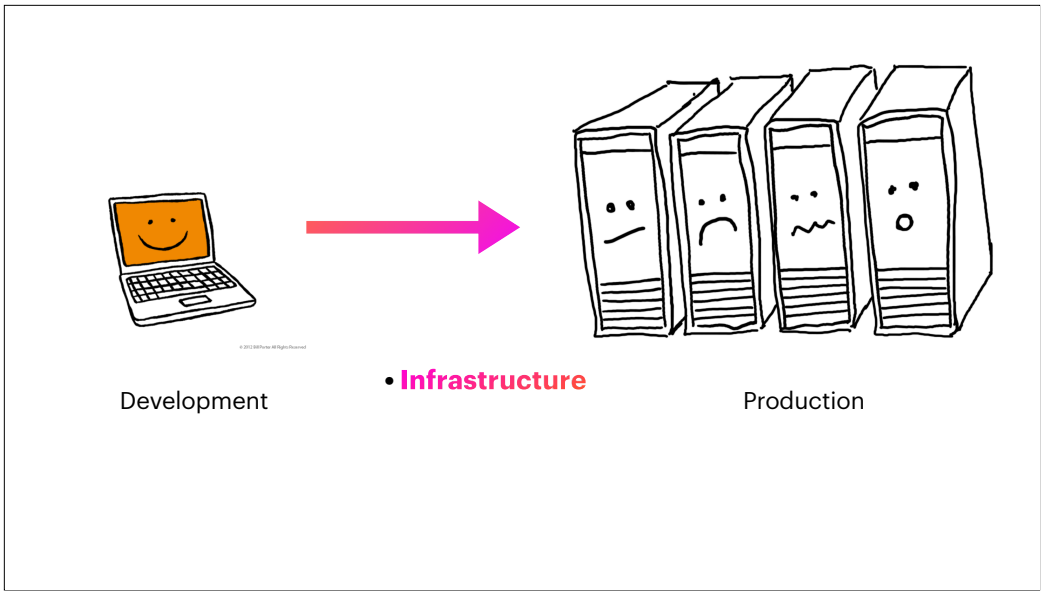
Observability is even more important because of a couple of factors.

One factor is that **code runs in production**.



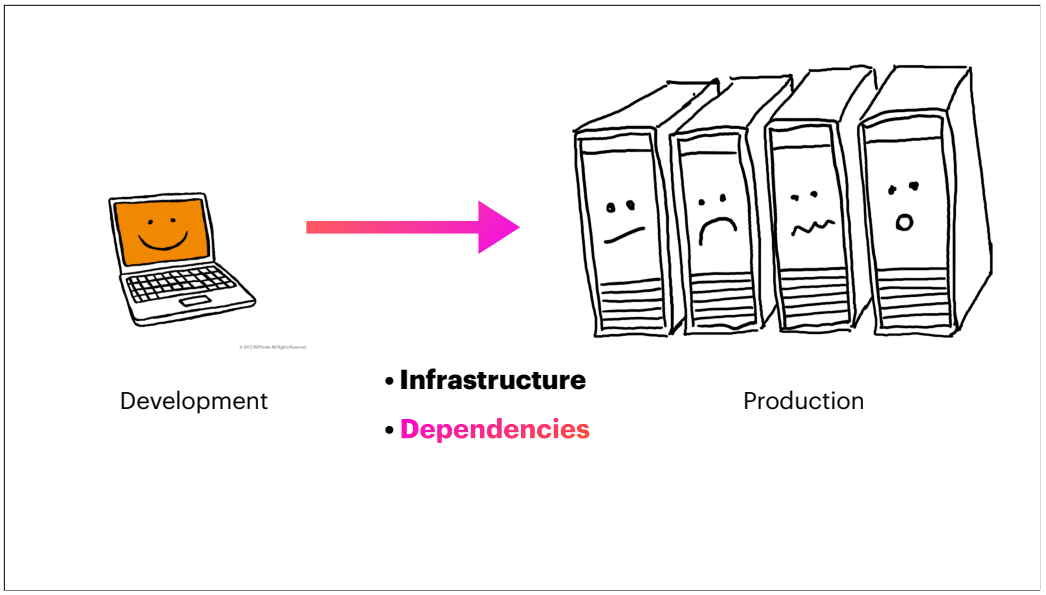
In order to ship something useful, you have to get it off of your developer environment on your own personal machine and put it in a place where others can use it.

Unfortunately, there are always significant differences between the way things play out in development and the way that they play out in production.



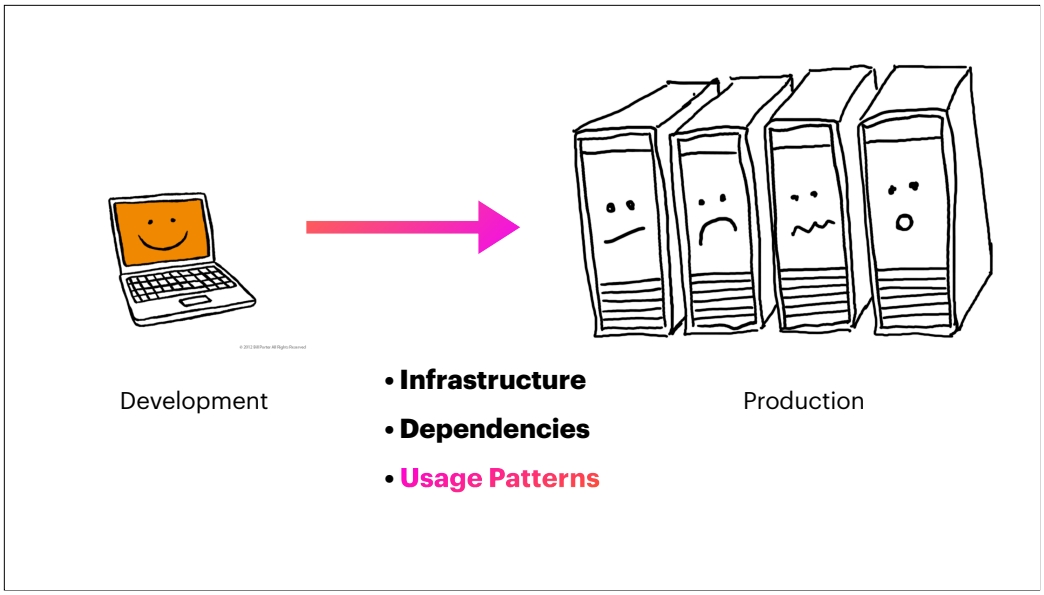
Infrastructure

- Everything's on one machine in your local, but in production your software is distributed across many machines with a network in between.
- There's a whole host of potential unreliability you have to contend with:
    - latency
    - outages
    - machines you have no control over shutting down



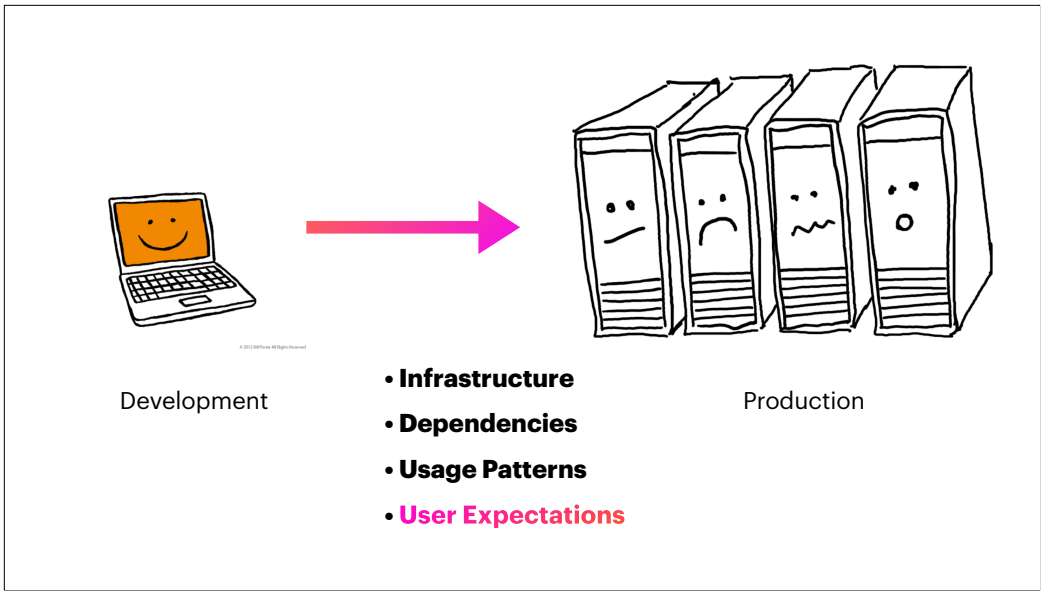
Dependencies

* Maybe you're working locally on a Mac or Windows machine, but in production things are running on some version of Linux
* Maybe your users are on Safari while you did all of your testing in Chrome
* Maybe the version of MySQL that you have running in RDS is different from your local version



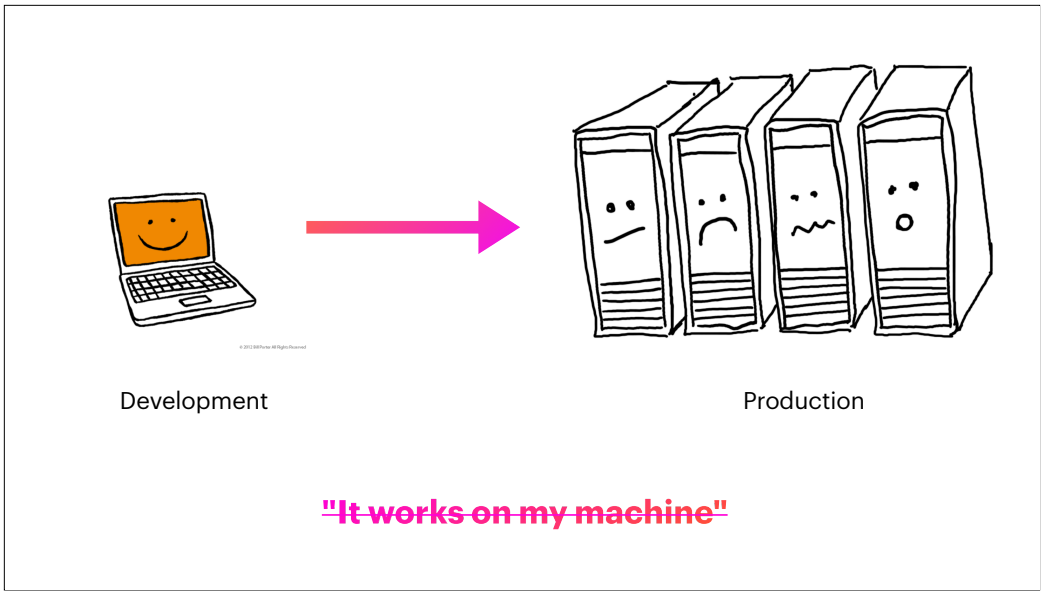
Usage patterns

* You may have written scripts to do load testing, but almost certainly the traffic you'll actually receive is different from your load test
* Perhaps you're the subject of a DDOS attack (distributed denial of service)



User expectations

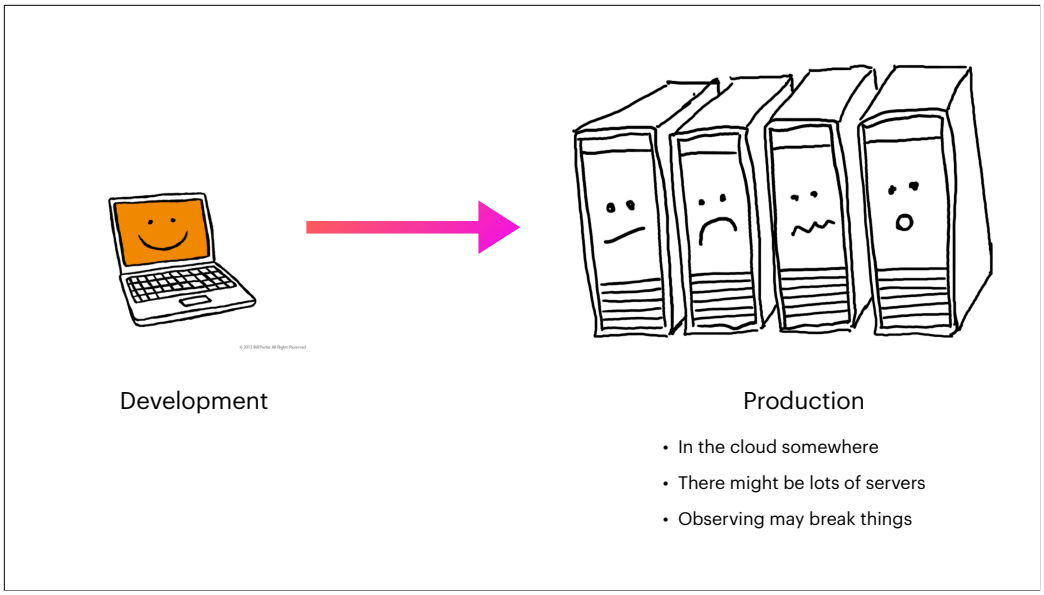
* Maybe you wrote a tool that you intended for one purpose, but it turns out people are using it for something completely different
    * One of my favorite examples of this: Pinboard, a tool that was written to store and share bookmarks to web pages people wanted to save around the net, ended up getting heavily adopted by authors of fanfiction to share things they had written



"It works on my machine" isn't good enough, it needs to work in production or you might as well not have done the work in the first place.

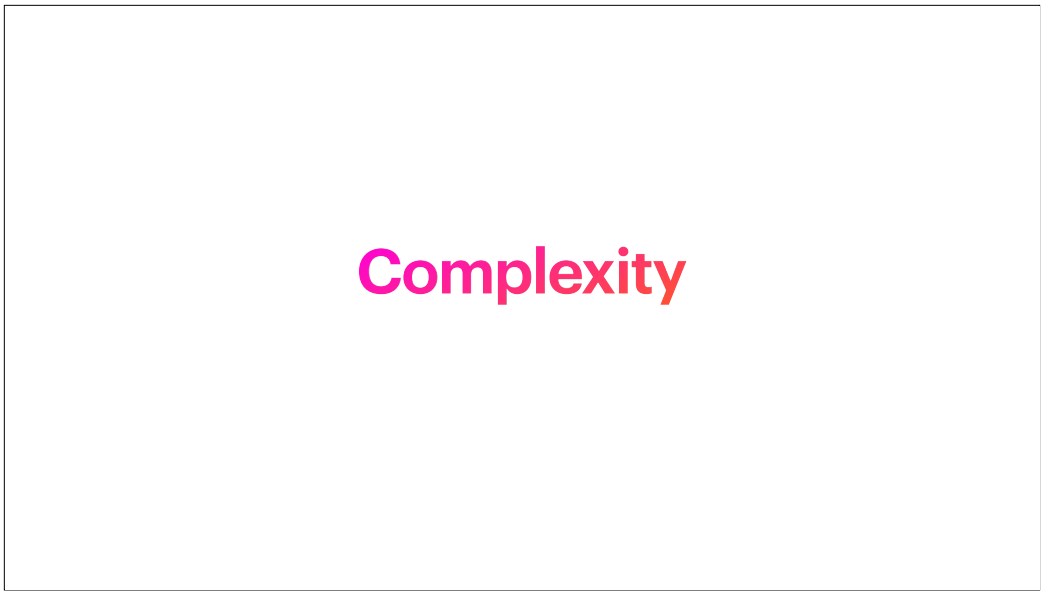
Observability is a measure of how well you can understand what's going with your system in both of these places, despite all of these differences.

And it's more challenging in production.

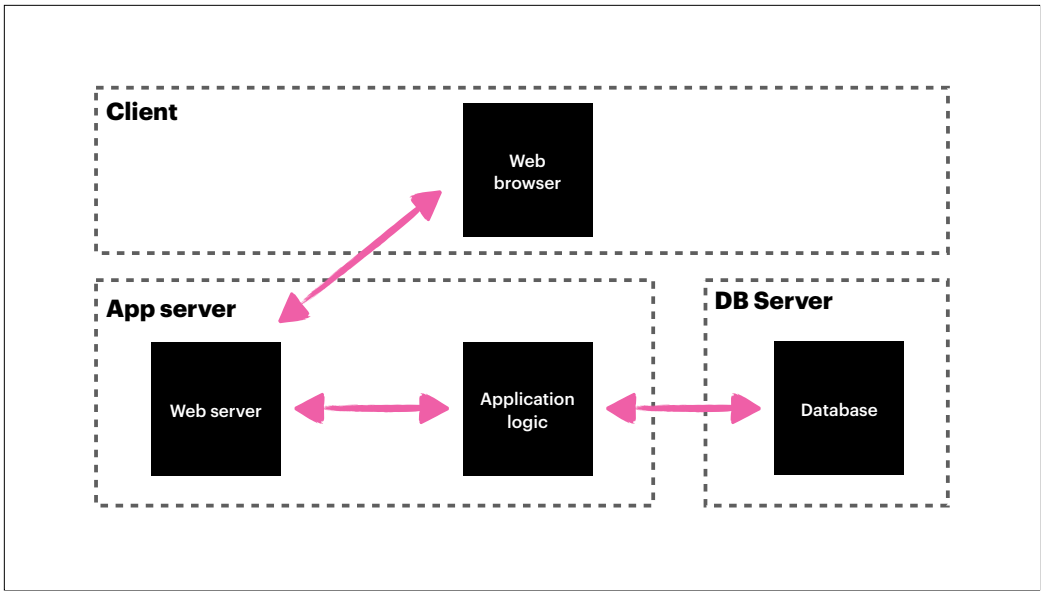


* It's far away from you, running in the cloud somewhere, so by default a lot of those outputs we talked about earlier are going to be harder to access

* There might be many, many services running -- which means to understand what's going on you have to look at them all
* The act of logging into production to look at logs could potentially be interfering with the server's ability to actually do the thing it's supposed to do by consuming resources, or letting attackers in



Another reason observability is important is because systems are complex.

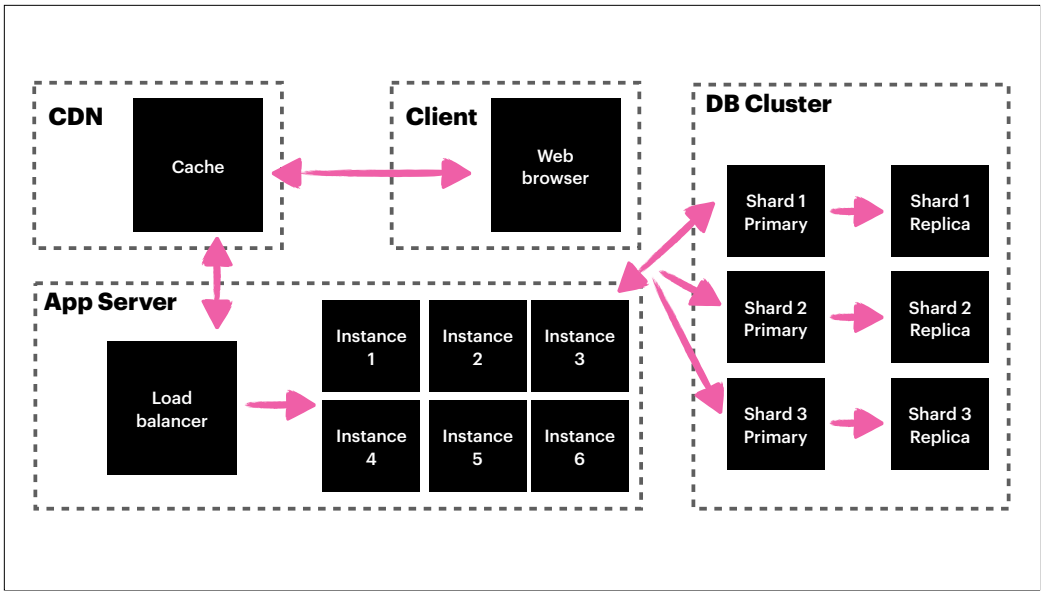


Here's an example of a relatively simple system that serves up web pages.

When I started my career, a lot of the internet services I worked on had this general shape. They were built on stacks like LAMP (Linux, Apache, MySQL, PHP).

Even a system like this one, you have 3 or 4 different entities (depending on how you do your math) that all have to coordinate with each other in order to get things done.

And this is a simple case. Usually, things are more complicated than this



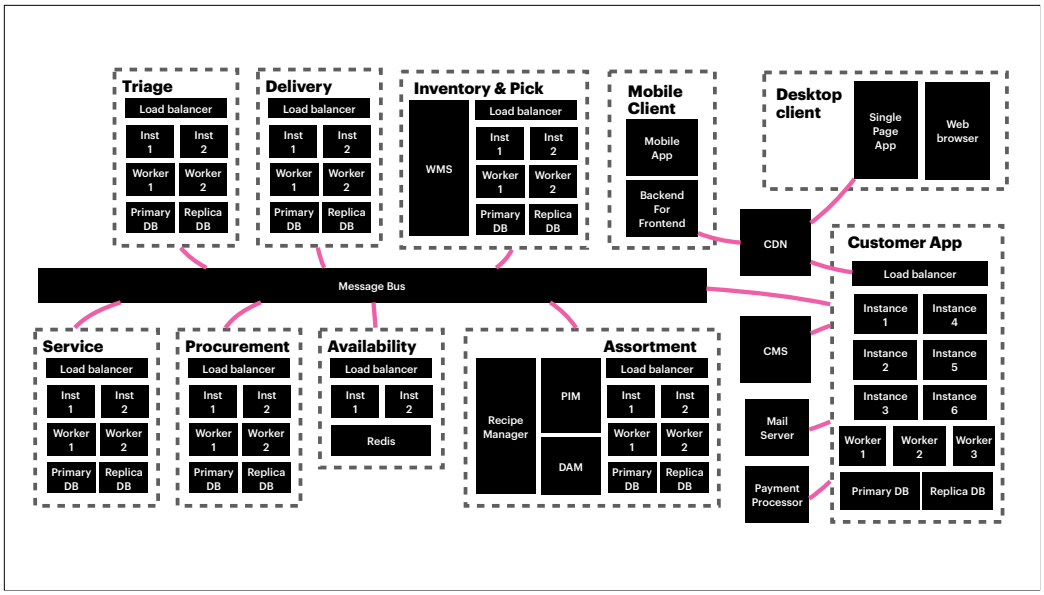
I wouldn't be surprised if by the end of this class you'll have created systems that look a bit more like this

We still have the same basic moving parts as before, but to handle scale, we're running multiple instances of the same app server, and there's a load balancer in front of them to route requests.

Maybe you also have a cache sitting in front of all that, so that repeated requests don't actually hit your servers at all.

And maybe you've split your database load between shards, and given each shard a read-replica so that if the primary goes down the replica can take over.

In this example, we've gone from 4 different individual but cooperating subsystems to 15, in order for the system as a whole to function. And this architecture is pretty standard.



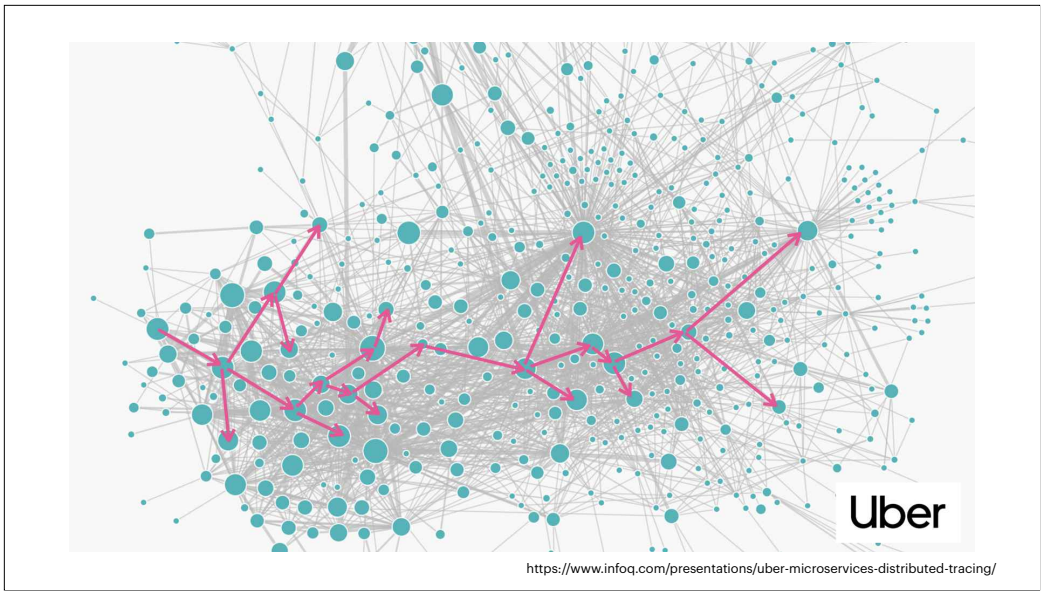
Here's a (somewhat simplified) example of the system that I worked on while I was working for an online grocery store.

I'm not going to spend too long on this, but broad strokes:

* The architecture was split into front-of-house and back-of-house (infrastructure responsible for allowing customers to buy things, and then infrastructure responsible for getting things shipped to customers)
* The online shopping experience was backed by an API, and fronted by a web app and a mobile app.
* Orders would get published onto a message bus, and get picked up by various back-of-house tools that managed things like warehouse inventory, delivery routing, and so on.

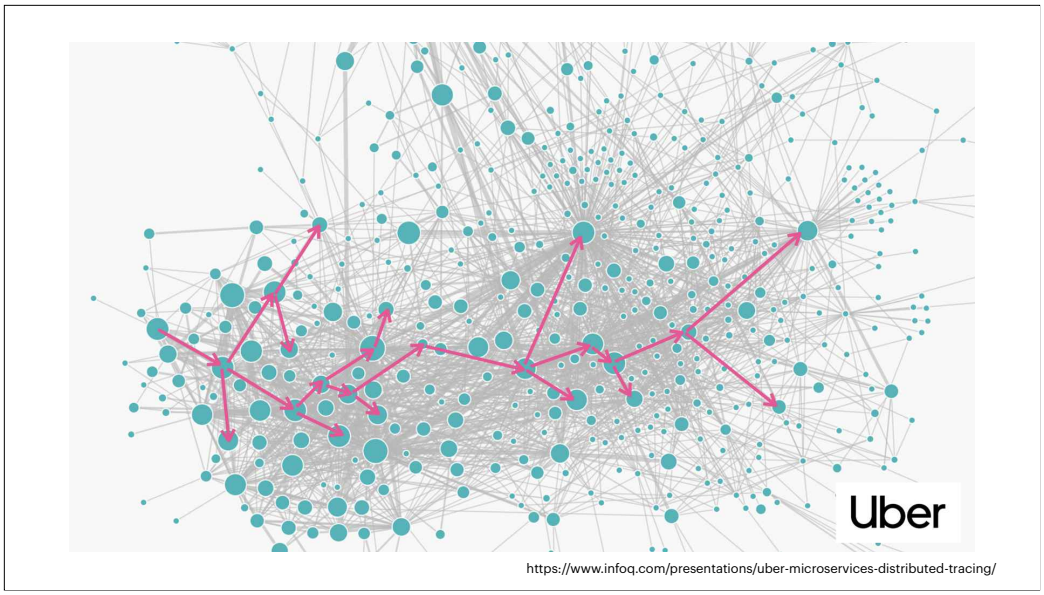
All told, this system was made up of ~100 independent moving parts, that all needed to communicate and coordinate with each other to get things working.

Imagine trying to answer the question: "Is the system working?" when the system looks like this.



Here's a graph generated at Uber, representing a subset (!!) of the microservices they had deployed at a particular point, along with the lines of communication between them.

(Side note: That pink tree embedded in the graph represents a single customer request making its way through the system. I'm going to be talking more about that in a bit.)

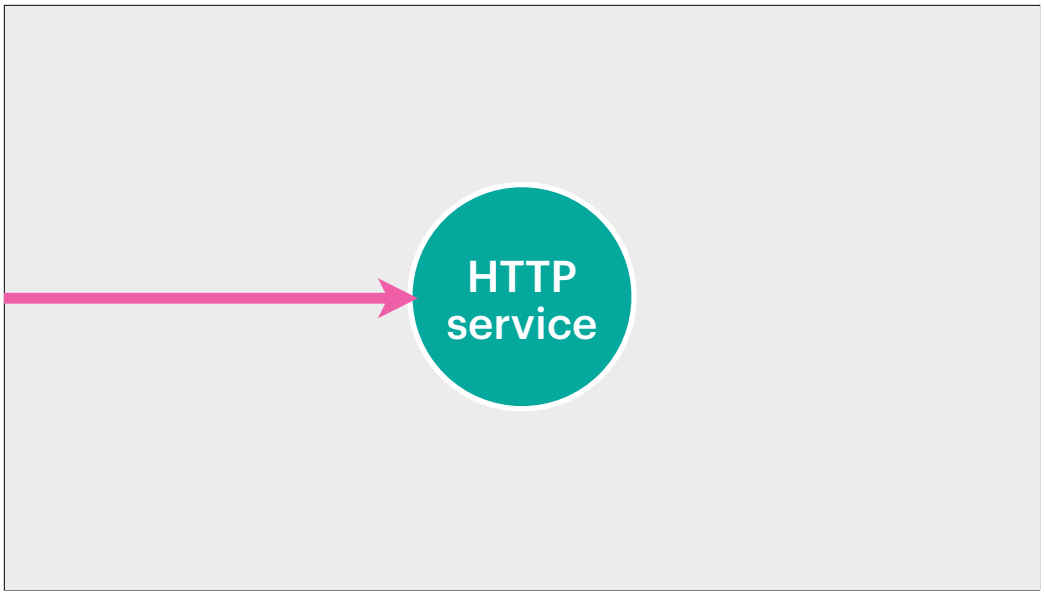


So if you have a really complicated system, that complexity shouldn't stop you from being able to figure out what's going on.

It's super important to be able to know what's happening when things are going wrong,
but it's also important to know when new and unusual things are happening that might be OK.

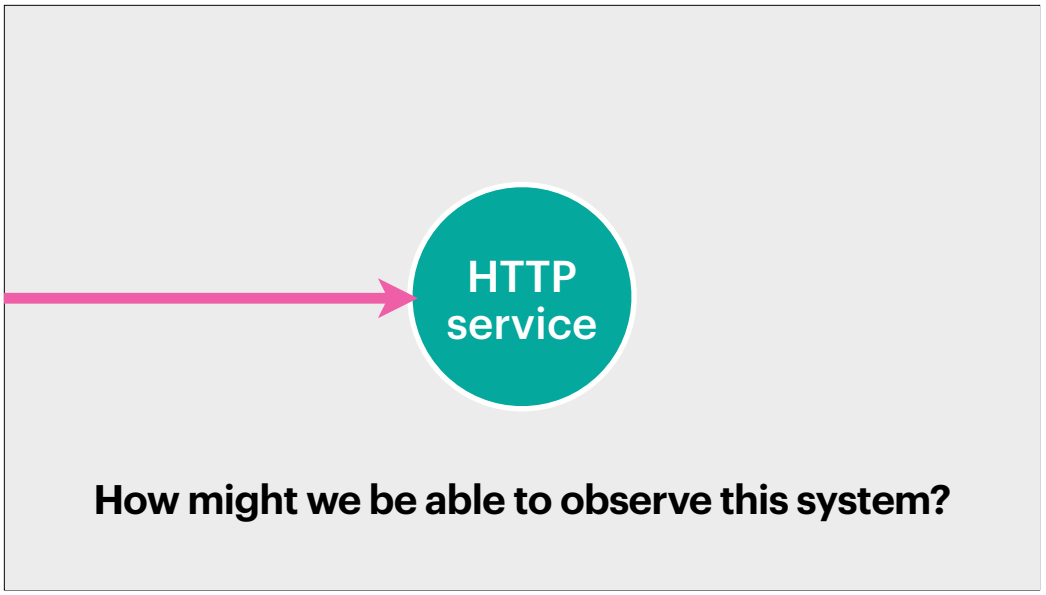
- e.g. a subset of your system's load suddenly dropped by 50%. is that bad? or does it mean that the new version of xyz that you just deployed is really efficient?
- e.g. distribution of queries drastically shifted. did something break or did one of your customers just make a really big announcement that's causing a ton of traffic that your system is handling just fine?

It's also important to be able to answer questions that come up during day to day development:
* e.g. you're thinking of making a change to the system, how will customers be affected?

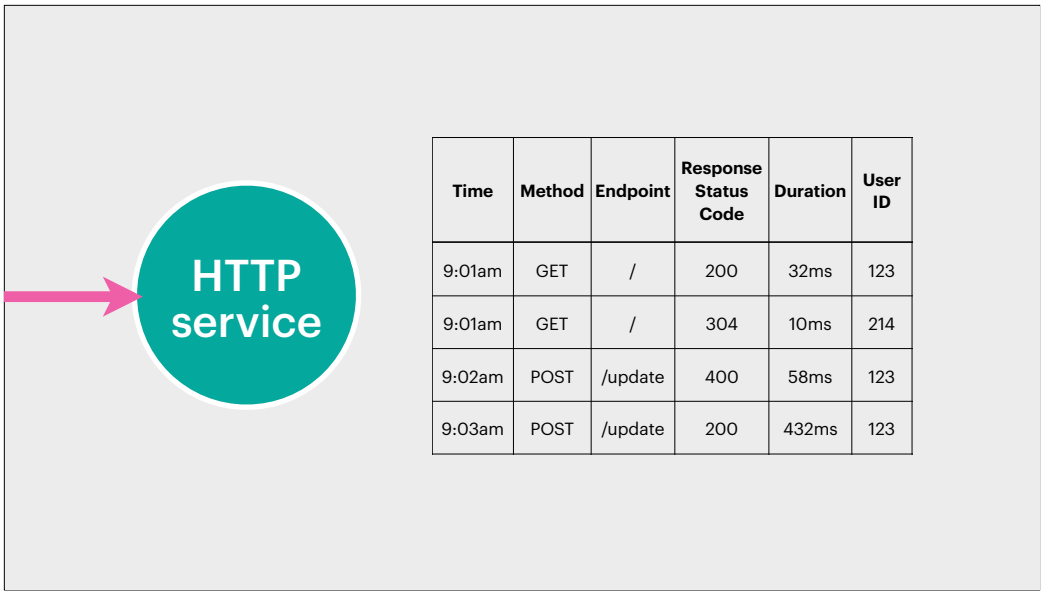


Let's put the super complicated Uber example aside for a bit and reduce the complexity.

Here is a much simpler system that's made up of just one service, that is capable of reacting to outside events: when it gets a request it does processing of some kind and sends back a response.



How might we observe this system?



How might we observe this system?
By recording whenever events happen, and giving the observer access to these records.

Here, any time a request comes in to the service, we're noting it in a table that we can access when needed.

* It's helpful to see events come in in near-realtime
* But also seeing the history of events allows us to see trends and outliers

Since the beginning of software, it's been customary for processes to publish a stream of logs, either to a console or to a text file.

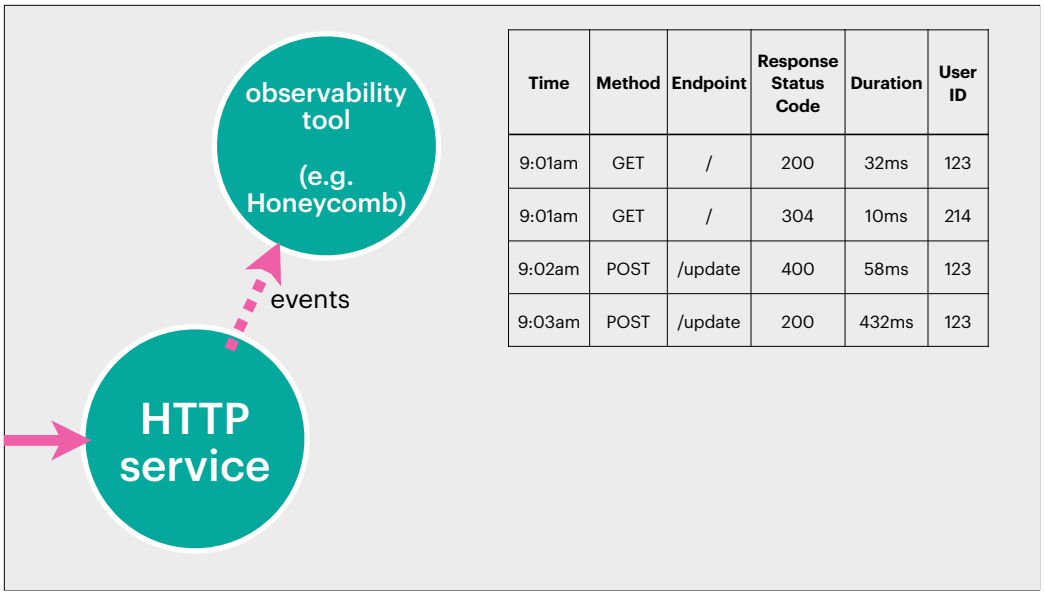
Operators could then directly connect to the system to read those logs.
But there are some problems with this approach:

- It requires you to be proactive
- It requires the system to be up, and connection between you and the system to be working (and it's usually the case that you

need this information the most when these things *aren't* working)
- Textual logs are generally not structured data which means seeing the trends and outliers requires manual math and a lot of squinting
- Any tooling you might install on the system in order to understand the data better takes resources away from the system itself

So, what's the fix here?

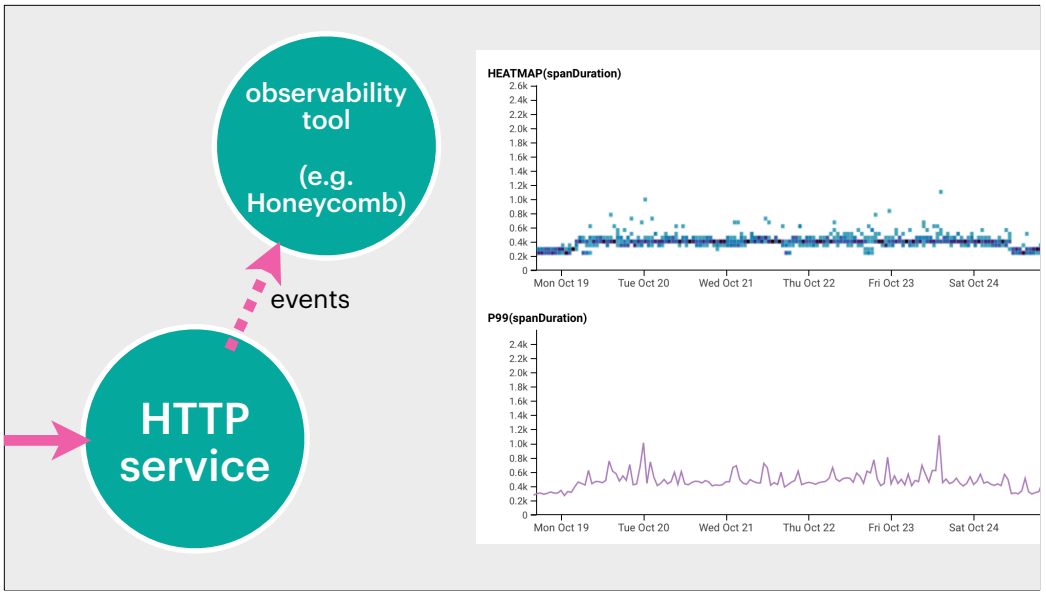
* Send events to a separate system that's responsible for observability



Time	Method	Endpoint	Response Status Code	Duration	User ID
9:01am	GET	/	200	32ms	123
9:01am	GET	/	304	10ms	214
9:02am	POST	/update	400	58ms	123
9:03am	POST	/update	200	432ms	123

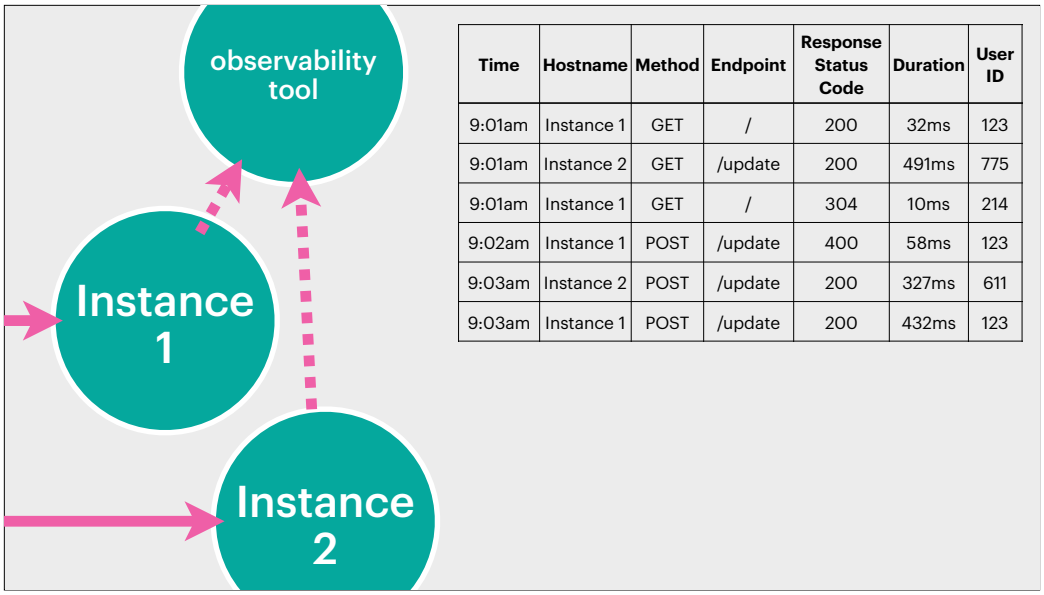
Honeycomb is an example of this variety of system.

It also happens to be the system you're using in the apps you're building for this class, and later I'll give you a demo of how it works.



Benefits of these kinds of tools:

* They visualize event data with graphs & charts
* They let you aggregate the data to understand trends
* They let you dig into outliers to find out what makes them different
* They will alert you proactively when things go wrong, so you don't have to rely on users telling you

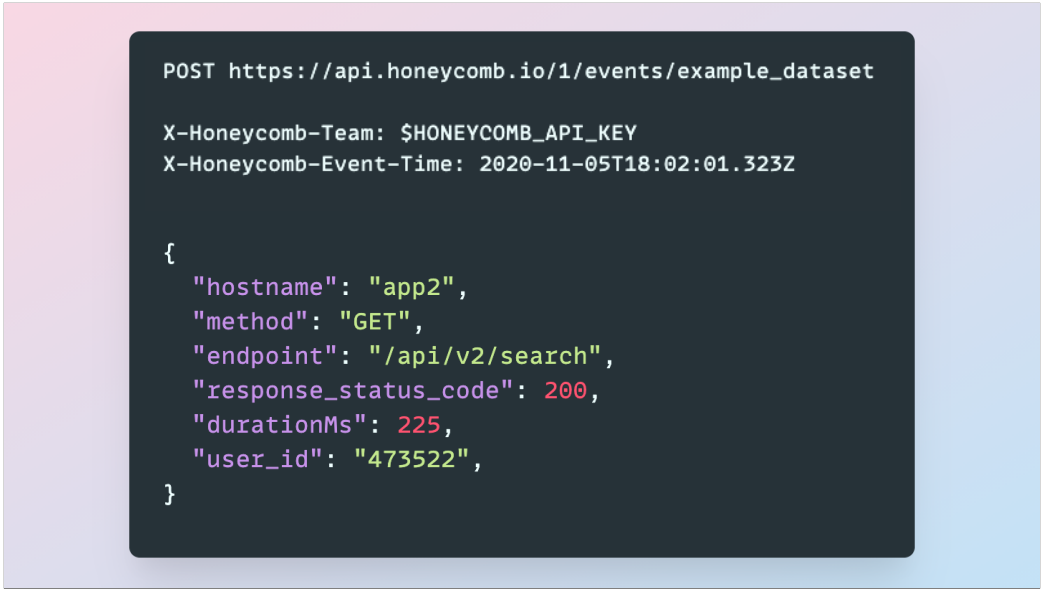


Time	Hostname	Method	Endpoint	Response Status Code	Duration	User ID
9:01am	Instance 1	GET	/	200	32ms	123
9:01am	Instance 2	GET	/update	200	491ms	775
9:01am	Instance 1	GET	/	304	10ms	214
9:02am	Instance 1	POST	/update	400	58ms	123
9:03am	Instance 2	POST	/update	200	327ms	611
9:03am	Instance 1	POST	/update	200	432ms	123

Say your service scales horizontally and you have two instances of them in production at the same time.

Another benefit to your observability tool running in a separate service is that you can see trends across all of the instances you have running.

Notice here we're recording the hostname in every event we send up so we can tell where the event came from.

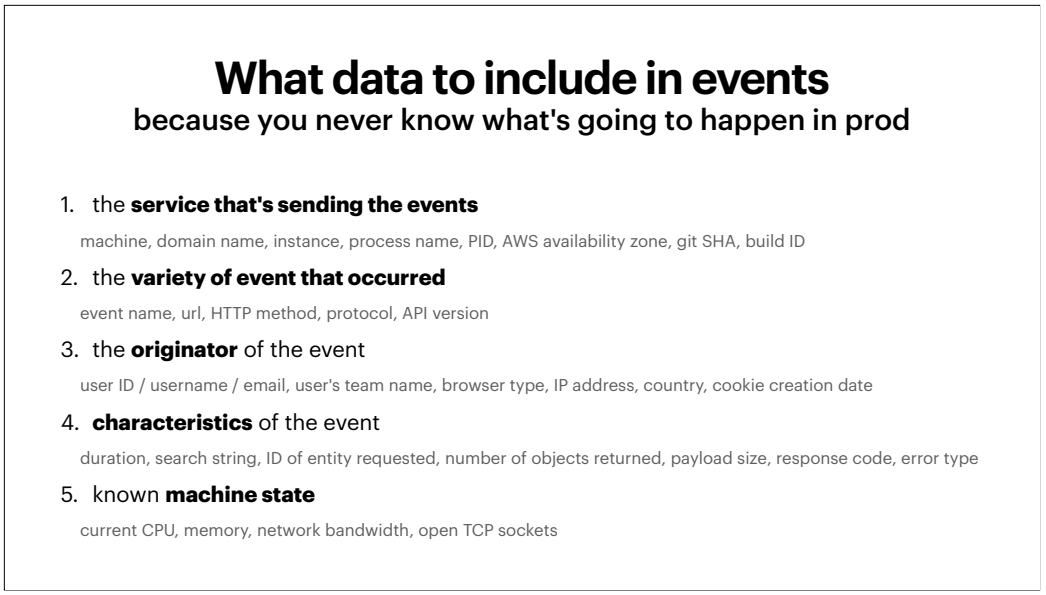


In Honeycomb, events get sent via HTTP request. The API looks roughly like this.

All the events end up in a **dataset**
All events have an event time which you can specify via HTTP header

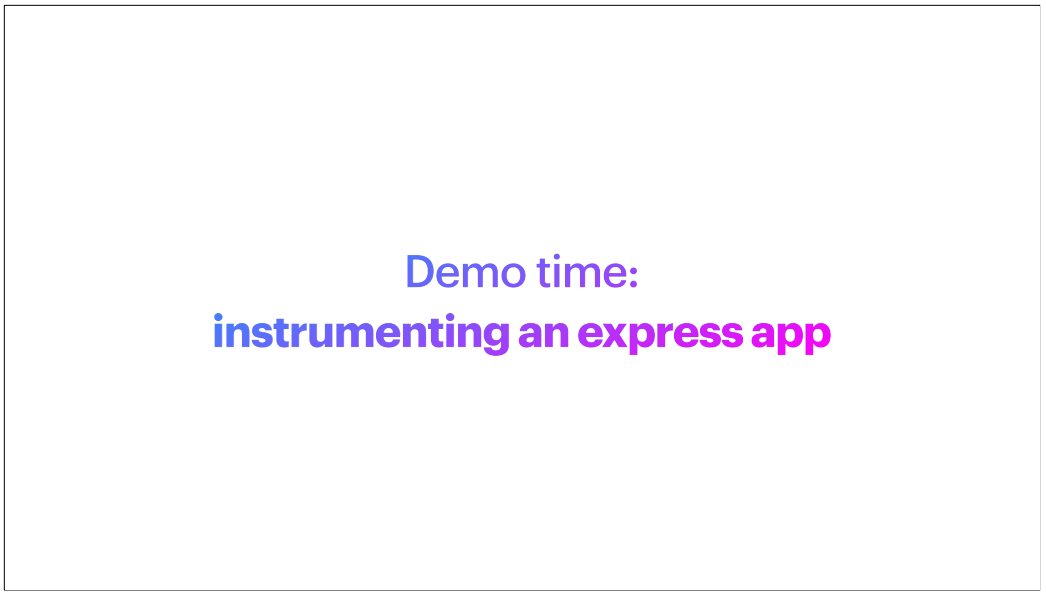
All other information that's in the event gets sent in JSON, as part of the request body, and it's entirely up to you what it contains.

This event is really small with only 6 keys.
You can make events arbitrarily large, and the more information you provide, the more information you'll have access to when you're debugging, so it's very worthwhile to include as much data as you can.



It's good to make your events as wide as possible and at the bare minimum include information from all of these categories.

This list is pretty generic, ideally you want to include as much domain information as is relevant to the service you're instrumenting



* look at server/package.json to show that honeycomb-beelines is installed

* go to server/src/server.ts and look at the setup code at the top

- Run the server, make a request
- Look at the data in honeycomb
    - https://ui.honeycomb.io/cs188/home/bespin
    - Look at the "total requests" graph, there should be a tiny little spike
    - Open the "recent events" tab and click on an event to see the JSON
    - Go to "dataset settings", click on "schema" and "unique fields" to see a list of all the fields that have ever been sent for this dataset

Adding
- Use beeline.addContext() in the bespin app to add user data

- (maybe line 44)

Instrumentation

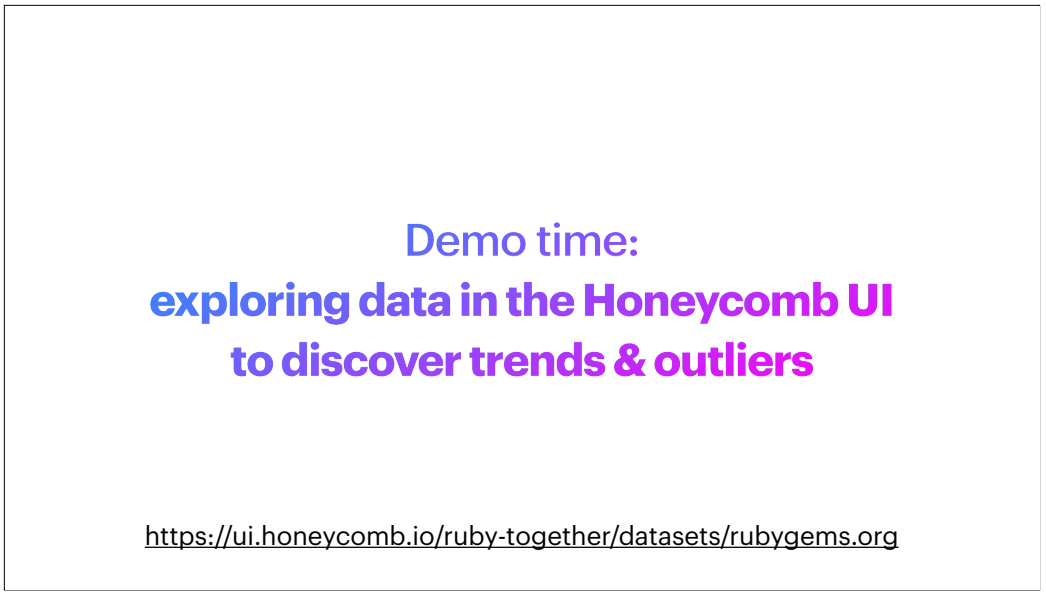
Publishing events from your app

- Use the **honeycomb-beeline** library
  - Automatically sends events at key moments in your app, for example:
    - HTTP request was handled
    - DB call was made
  - You can add data to events by using `beeline.addContext()`

Some thoughts about event schemas

- Events can be **any shape supported by JSON**
- It's OK if data is **present** in one request but **absent** in another
  - (e.g., error messages only exist when there are errors)
- Recommendation: **be consistent** across requests and across services
  - Don't publish `user_name` from one service and `user.name` from another service

If you're publishing "userName" in one service and "user_name" in another service, you will lose the ability to graph data across both services.



https://ui.honeycomb.io/ruby-together/datasets/rubygems.org

This is a publicly-accessible dataset, populated by the CDN for rubygems.org -- the package manager for the Ruby programming language.

1. Create a new query: P95(time_elapsed)
    1. RubyGems.org is this speed or faster for 95% of requests
2. Show AVG(time_elapsed) and compare to P95
    1. Percentiles are better than averages for this sort of data, because the outliers have such a strong effect on the average
3. Show AVG(time_elapsed) grouped by

Higher-order concept: Traces

A **trace** is a collection of data representing a single request or transaction in your system, from beginning to end

Examples:

1. A browser makes an HTTP request to the CS188 website and receives a 200 OK response containing data stored in a MySQL database
2. A Twilio-enabled phone number receives an SMS message, and receives an automated reply as the original message is forwarded to 2 recipients
3. Spotify kicks off an update to personalized "Discover Weekly" playlists for a subset of accounts

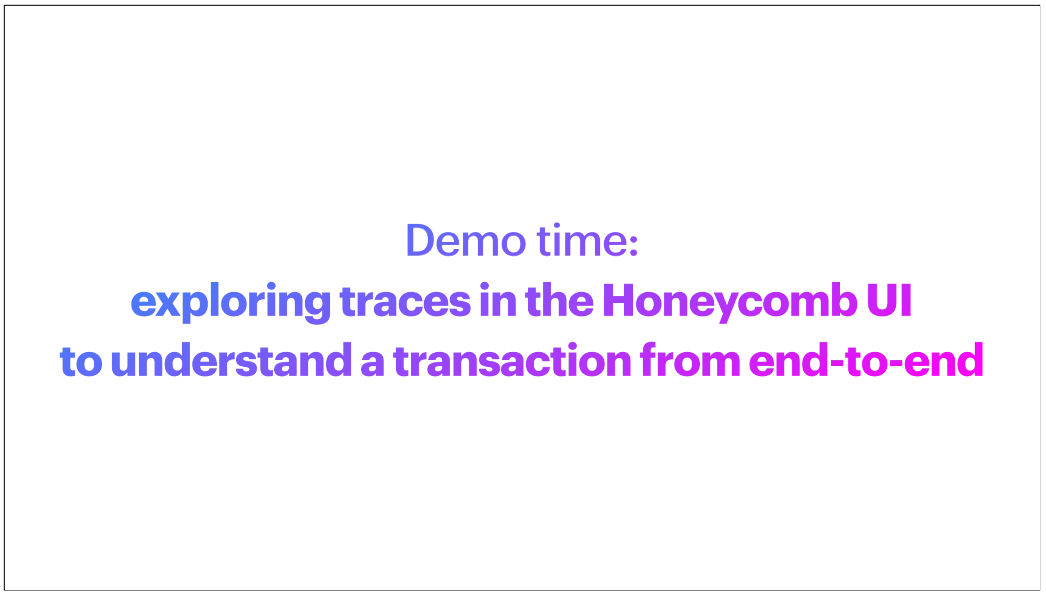
Questions -
In these examples, what might be useful data to store as part of the trace?

Spans

A trace is a collection of **spans**. Each span represents a self-contained *part* of the request or transaction.

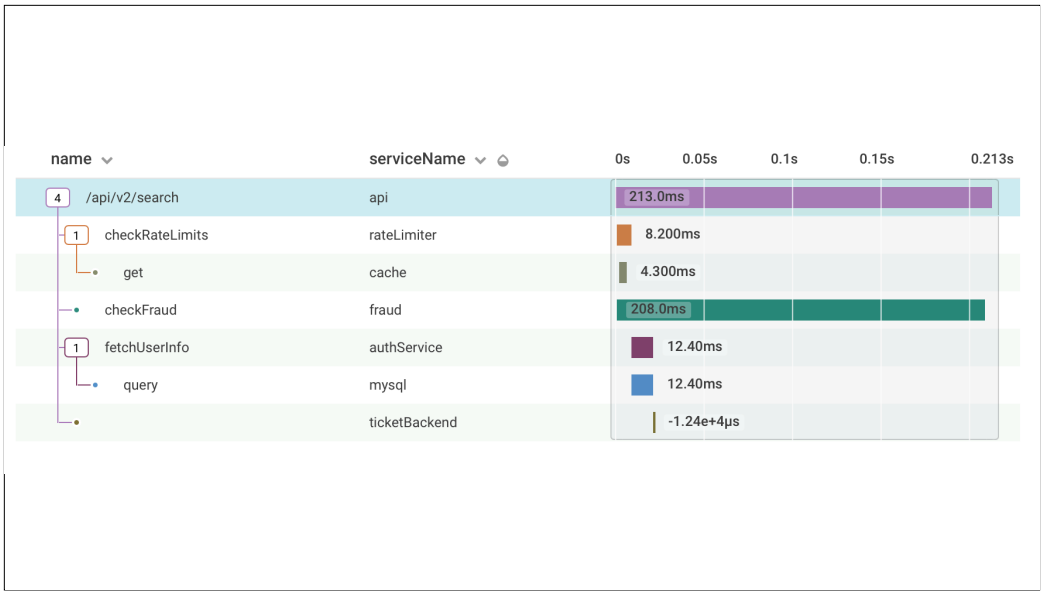
Examples:

1. a Node.js server receives a request, processes it, and responds
2. a MySQL instance runs a query and returns results
3. a server running Varnish receives an HTTP request and forwards it to another server
4. React renders a server-side template



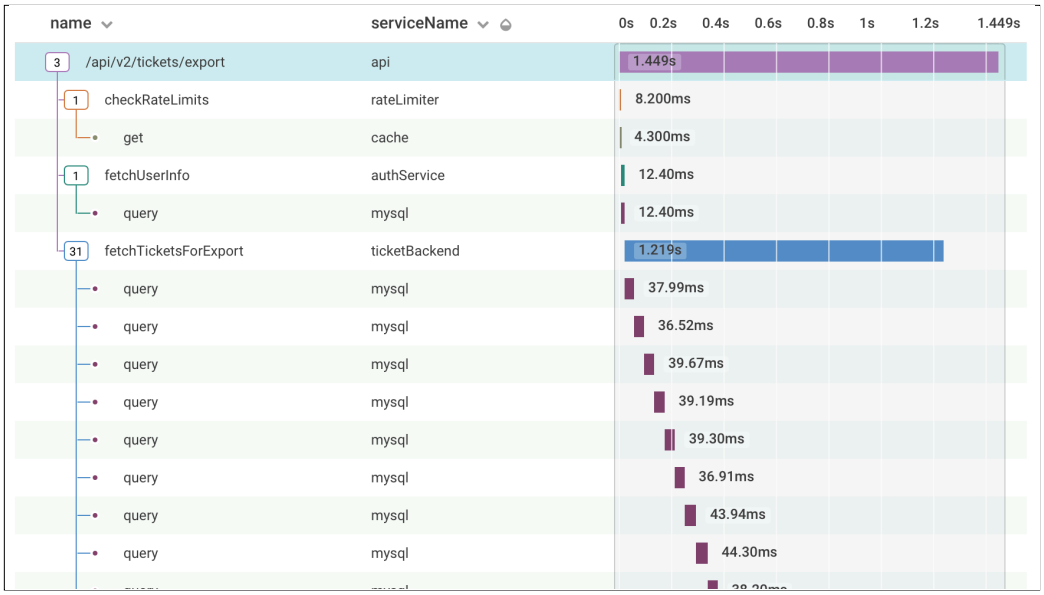
start with the bespin dataset

look at some interesting traces on https://play.honeycomb.io/quickstart/datasets/tracing-tour
Look at HEATMAP(duration_ms)



Here's a visualization of a single trace with a few spans.
What can we learn from this?

- Server took 213ms (0.2 seconds) to respond, that's pretty respectable
- A trace is a tree data structure.
- Every span represents work done on a different service.
- Every span has a duration (measured in milliseconds on this graph)
- Every span contains data (name, serviceName, duration in this example)
- **Root spans** delegate work to child spans.
    - in this example, all the root spans take longer than their slowest child. this is not necessarily a requirement.



Here's a visualization of a different trace -- one that shows a request that took significantly longer than the first one.

Anyone have an idea of why that might be?

What would you look into to try and fix it?

---

answer: fetchTicketsForExport made 31 sequential database calls.
this could be an **n+1 query** (initial query loaded 30 tickets, and then ticketBackend iterated through the list and made a different database query for each ticket)

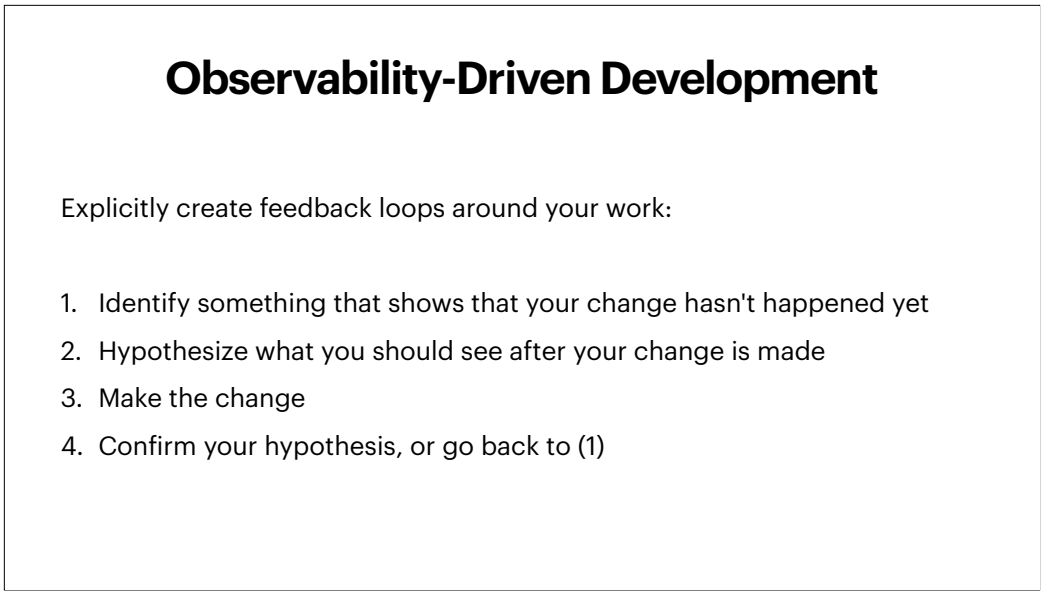
this could *potentially* be fixed by changing the first db query to load up all the needed data.

A span can be represented as an event with some special stuff attached:

durationMs
id
traceId

https://www.snippetshot.com/
#eyJnaXN0Ijoie1xuICBcImR1cmF0aW9uTXNcIjogMjI1LMYYY3VzdG9tZXJfaWTEGVwiMTcxMVwixx5lbmRwb2ludMYbL2FwaS92Mi9zZWFyY2jJJWJ1aWxkyUAzMTUwyRvIFTlhMTdiNjY4M2RiODgwMTPJIXVzy3s0NzM1MjLJHGZyYXVkX2R1csQeOTjHFnRyYWNlSd9YImlzX2Vycm/
FOzDHFHBsYXRmb3JtxjthbmRyb8R6xx5ob3N0bmFtZccecHDqAInoAQZfc2hhcMch9wEMYXZhaWxhYmlsaXR5X3pvbscudXMtd2VzdC3qAVrKa9dKc2VydmljZU7JKGFwacoddGF0dXNfY29kxR0yMDBcbn0iLCJsYW5nIjoiSlNPTiIsImNvbG9ycyI6WyJyZ2IoMjU0LCAyMTUsIDIyNikiLMUVMTkwxBA3LCAyNDgpIl19

Now that we've done all this work to instrument a system, let's talk about some ways you can use these tools in your day-to-day.



One big one is using observability to set up feedback loops for yourself while you're developing software. Observability-driven development works like this:

1. Identify something that shows that your change hasn't happened yet
2. Hypothesize what you should see after your change is made
3. Make the change
4. Confirm your hypothesis, or go back to (1)

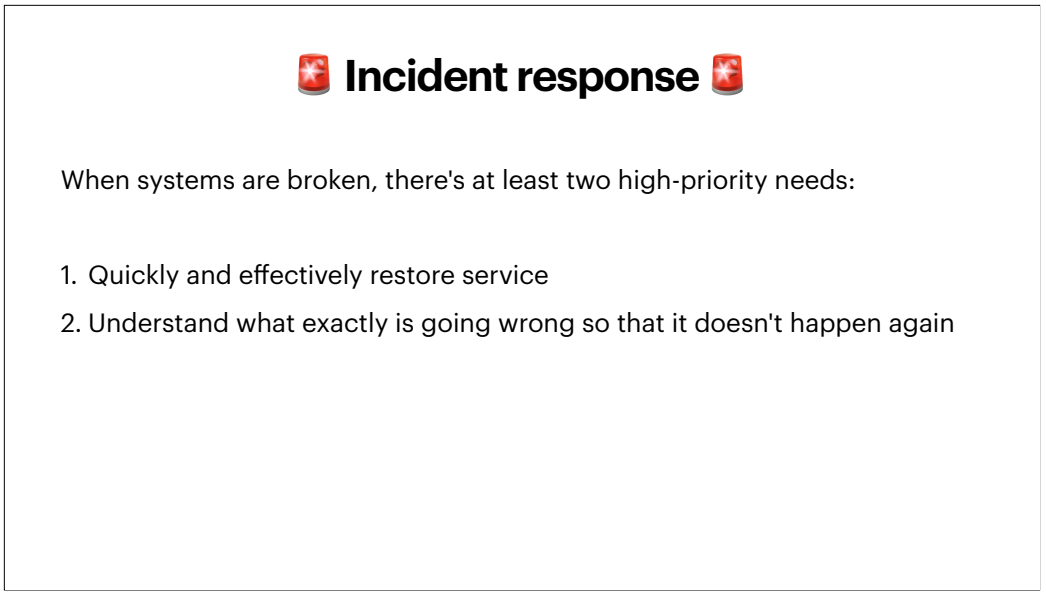
If you're at step 1 and you can't find something to observe, then stop right there and make a thing that lets you observe that part of your system first. And use this process to build *that*.

Example:
* Find a graph that shows your response times are slower than you'd like them to be
* Make a change
* See if response times increase

Example:

* See that there are less signups with a particular homepage design
* Make a change
* See signups increase

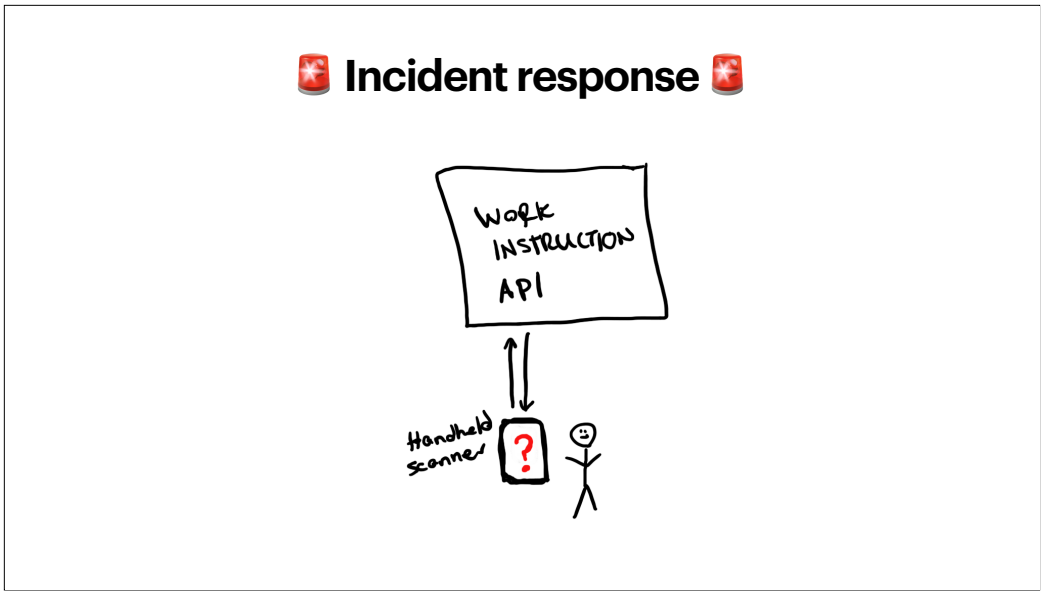
Benefits: you'll really know that your system is working, and you'll end up with a more observable system when things break.



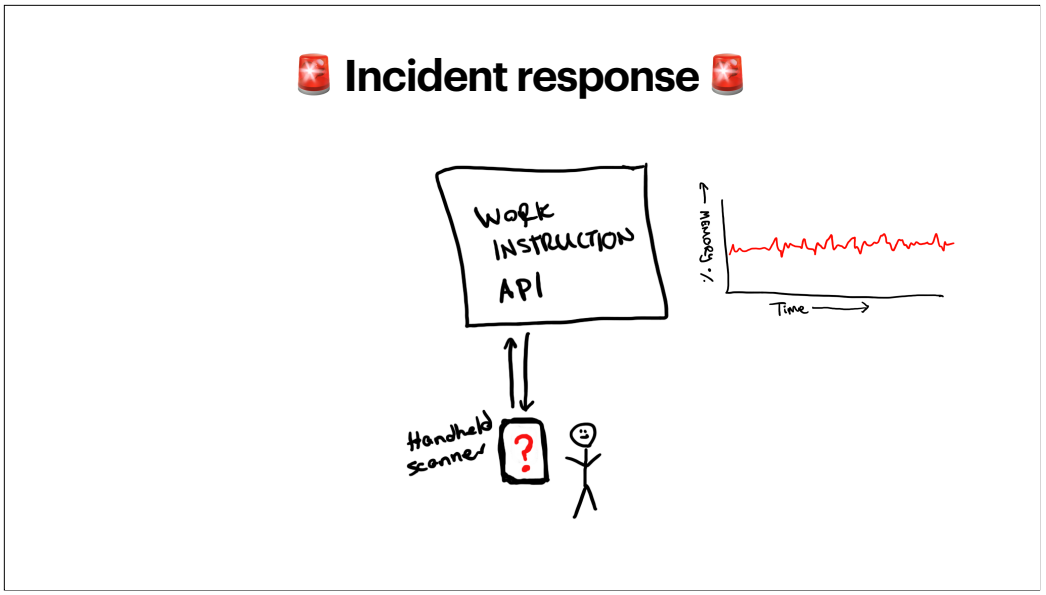
Let's say your system is on fire.

Maybe customers are complaining because the web site won't load.

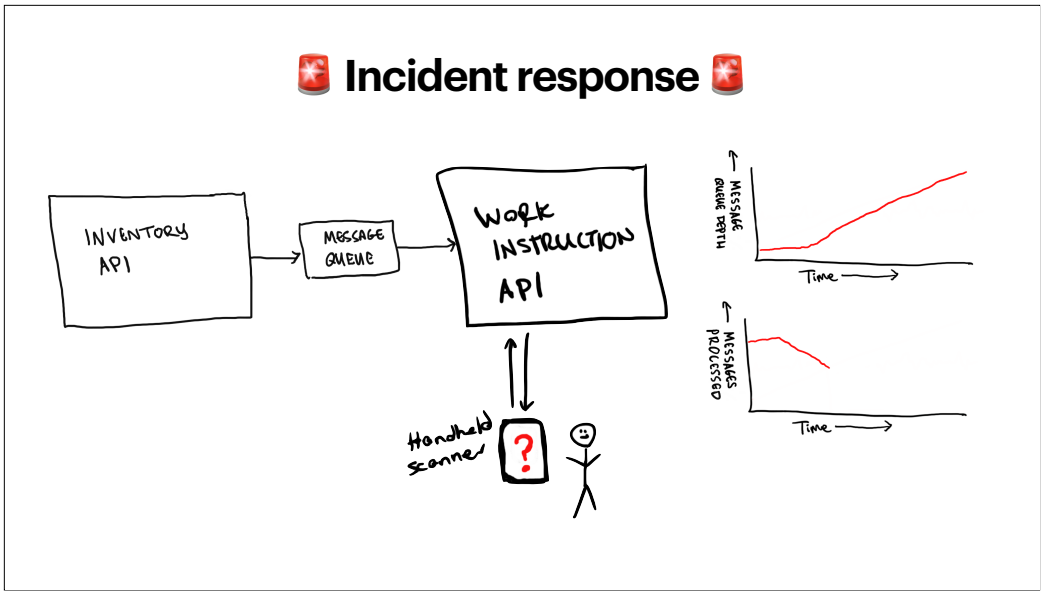
Or, to pull an example from my past - my team at Good Eggs was responsible for maintaining software that allowed the warehouse distribution process to run smoothly.



After a bit of digging, I found a nugget of information that explained the issue.



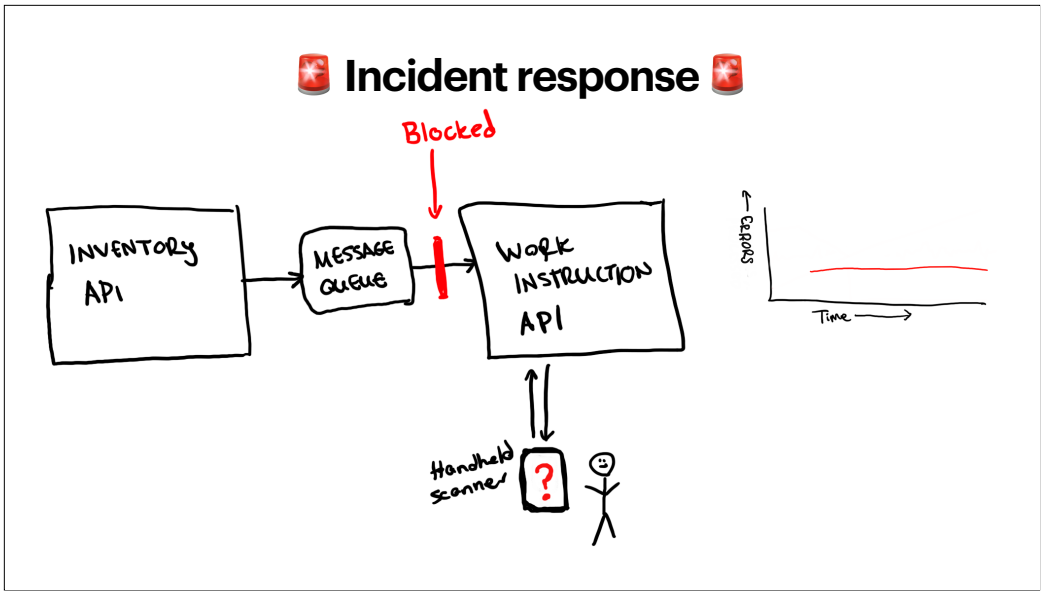
First thing I checked was memory utilization over time, and the graph disproved my hypothesis. Memory usage over time seemed to be fairly stable.



So I poked around a bit more, and found a few interesting things:

There was another API we maintained to manage our warehouse inventory. It would send updates to the work instruction API whenever items in inventory were added, removed, or moved location.

And I noticed that the amount of messages that were waiting in the queue had recently started increasing steadily, while the rate at which the work instruction API was processing these messages had dropped to zero.



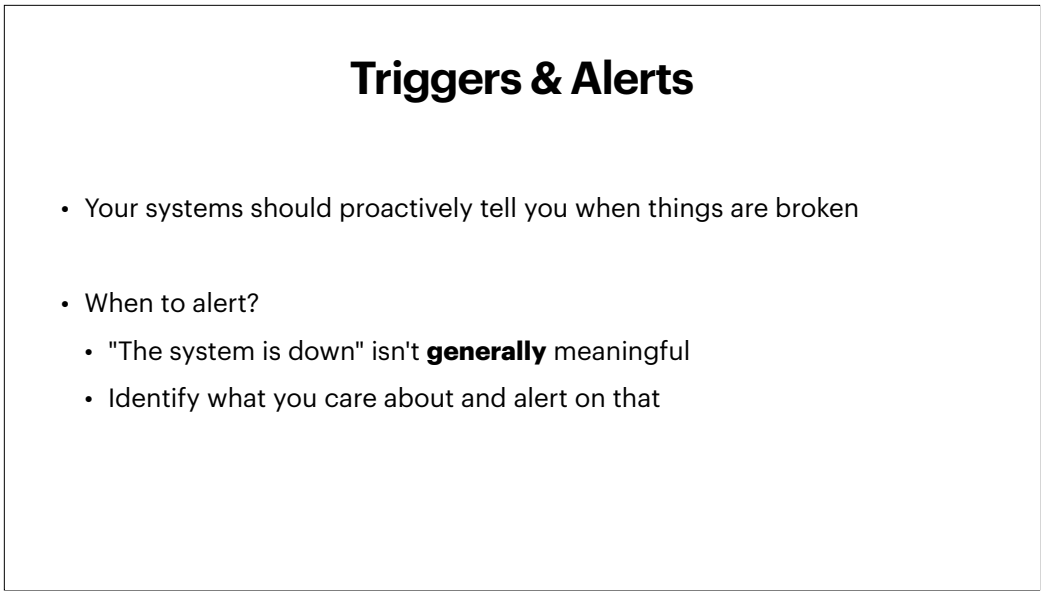
Additionally, around the same time we started seeing a lot of errors in the queue consumer. Looking at those errors, the story became clear: The message queue consumer for our work instruction API was consistently failing every time it was trying to process an incoming message from our inventory system.

My original plan of restarting the work instruction API would have had no effect - it would have started up and continued attempting to consume messages and failing.

So, then I took a look at the messages at the front of the queue and found the culprit: many copies of the same message, referring to inventory at a location that didn't exist.

At this point, it was clear what steps I had to take to unblock the warehouse: I removed the offending messages from the queue, and instructions started flowing again. Then I went back to bed. When I came into work the next day, I wrote code to better handle invalid inventory locations.

I hope it's clear from this example that the more observable your system, the easier time you'll have trying to fix things that go wrong with it.



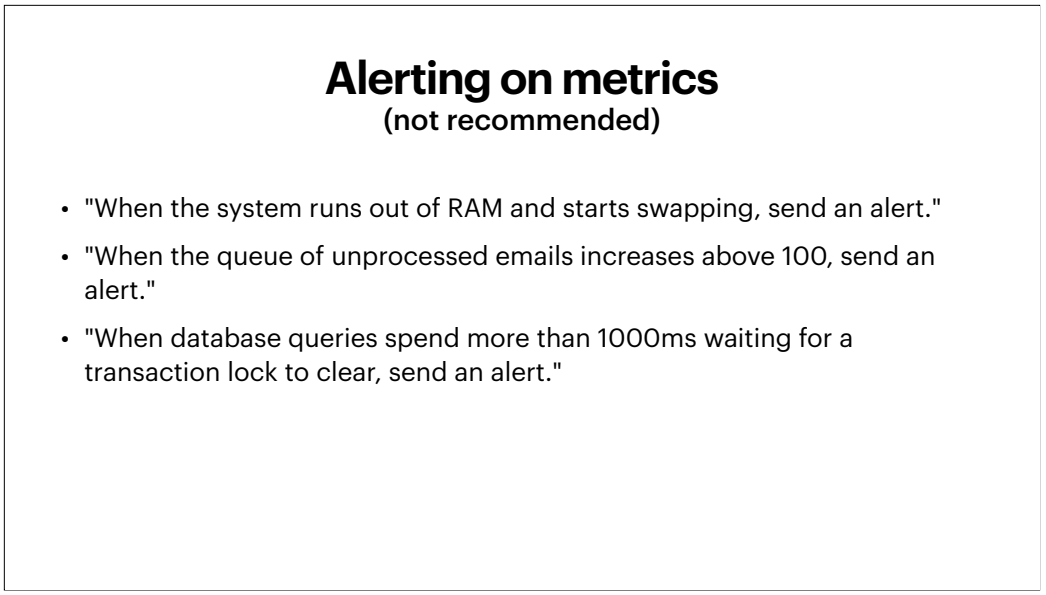
Your systems should proactively tell you when things are broken
Much better they do this than your users tell you. Some users might not actually tell you things are wrong, they might just leave. Also, you have the benefit of saying, "yep, we're working on it!" when your users do eventually tell you.

Tools like Honeycomb give you the ability to do this - you can set up triggers to alert you by email, text message, slack notification, etc when specific conditions are met.

"The system is down" isn't *generally* meaningful
* It could be that things are working as intended, but they are painfully slow
* It could be that a small percentage of users are experiencing catastrophic errors and things are totally fine for everyone else
    * Anecdote:
        * I encountered a bug once, on an e-commerce web site.
        * Things had been working fine, until all of a sudden we started getting reports that the system was completely down.
        * I took a look and it worked fine for me.
        * But we were displaying "recommendations" on the home page, and it turned out that all the users for whom

the site was "down" were getting recommended this one specific product that had an emoji in the name.
* It turned out that we were using a text parsing library on the homepage that couldn't handle emoji, and when it broke, the entire page wouldn't load, causing a broken site for a seemingly random subset of users.

So, "system is down" could mean a variety of things, it's helpful to have your own custom definition of "working".

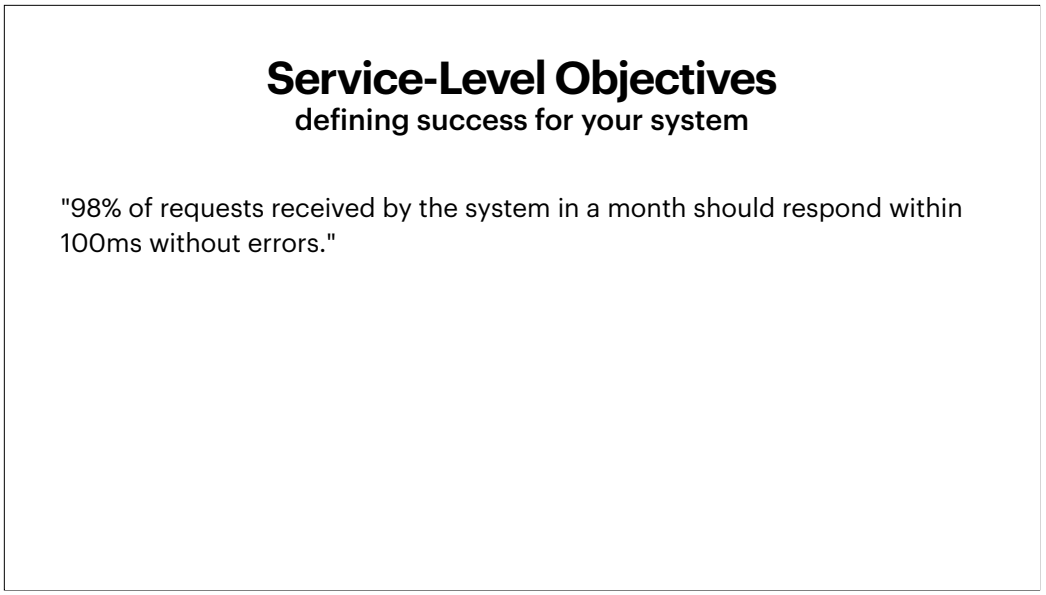


Historically, it was common for people to set up alerts on very specific things. (When the system runs out of RAM and starts swapping, send an alert.)

There are problems with this approach:

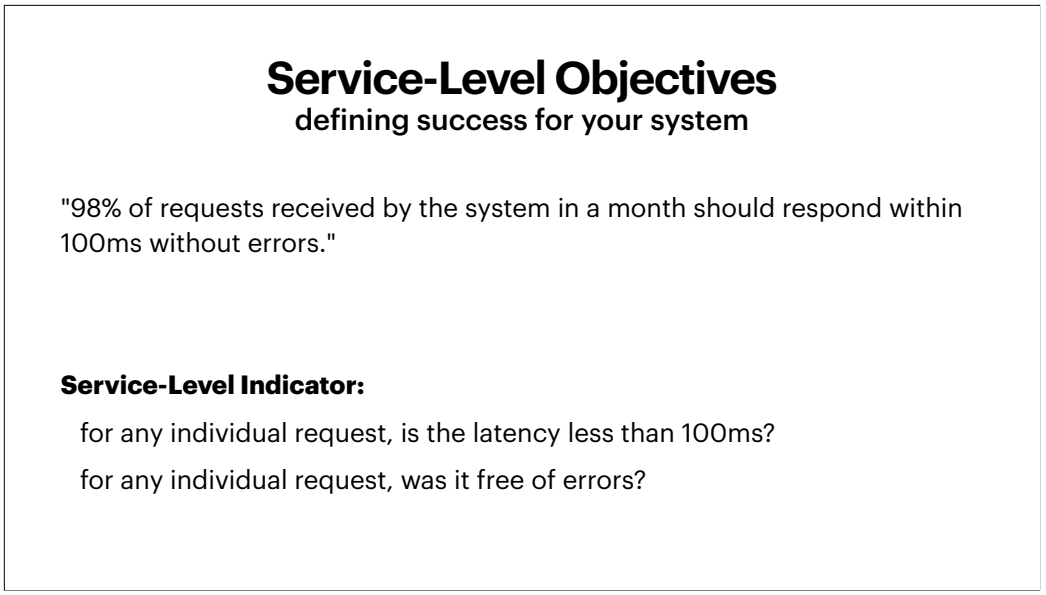
1 - it's too specific. if you try and enumerate all of the things that could go wrong this way, you'll have to set up tons and tons of alerts, and odds are when things are breaking many of the alerts will fire at the same time, which is incredibly stressful.

2 - it's not specific enough. it's hard to justify deciding to wake yourself up in the middle of the night to respond to a problem like these because you don't know anything about how your actual objectives are being affected by this problem.



Instead, set "service level objectives" -- high level goals for your system to meet. Alert when it seems like you won't be able to meet these objectives without making a change to your system.

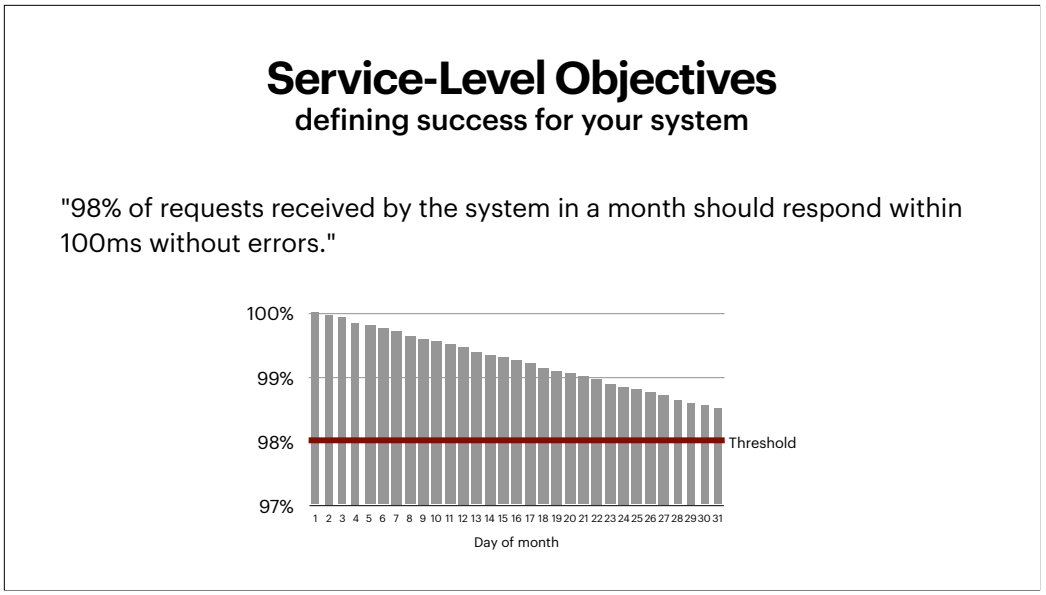
For example, here we have a good service-level objective: 98% of requests received by the system in a month should respond within 100ms without errors.



In order to alert on this, we need a service-level indicator (i.e., a specific metric to track)

It has to be an observable characteristic of the system.

In this case, we can use request latency: the amount of time between when a user makes a request and when the response arrives. We can say that an individual request **succeeds** if latency is less than 100ms and that it didn't throw any errors.

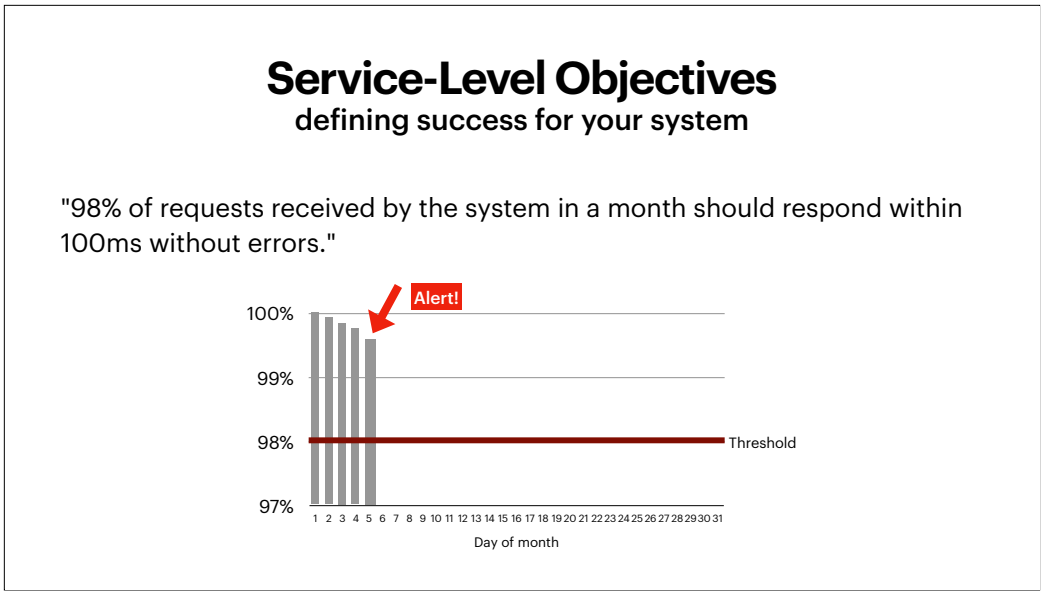


Then, we can set up our alerting such that it only triggers if we're on track to fail at meeting an objective. At that point, there is a very clear reason to do something.

Here's a graph of this service-level objective over the course of a month.
You can see that we aren't responding to every request successfully, but that's OK -- by the end of the month, 98.5% of our requests have met the criteria, so we still have some budget left over. All's good.

Successful systems don't have 0 errors. Rather, they're the ones that solve user needs -- be outcome oriented, don't micromanage all the risk. Risk is unavoidable, no matter how hard you try.

Embrace failure. Build systems that can tolerate it.



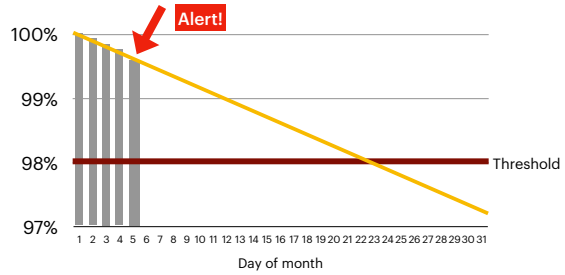
Here's an example of when it might make sense to trigger an alert.

We're only on the 5th day of the month, and, assuming that the amount of incoming requests stays consistent, we've already seen enough errors that if we don't make any changes, we'll blow through our budget.

Service-Level Objectives

defining success for your system

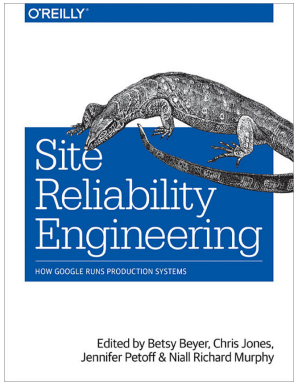
"98% of requests received by the system in a month should respond within 100ms without errors."



But the nice thing about alerting this early is that we still have an opportunity to fix things!

Service-Level Objectives

defining success for your system



If you're interested in knowing more about this philosophy, I recommend reading a copy of this book.

Questions?

Sampling

How to deal with scale

- When your system gets big enough, aggregating events & sending them off to a 3rd party system will still take a toll
- Solution: Send a "representative sample" of events to your system instead of all of them
- Don't do this prematurely!

Instrumenting 3rd party systems

What about open source programs that support your app?

- e.g., MySQL? Load balancers, CDNs, etc?
- OpenTelemetry

Other production observability tools

- Error reporting - Bugsnag, Sentry, Airbrake
- Log aggregation - ELK, Graylog, CloudWatch, SumoLogic
- Data warehousing - Metabase, Mode Analytics

DWH -- thinking about Facts & Dimensions
OLTP vs OLAP databases. Don't query your primary with OLAP queries.

Cardinality

The number of unique values that a field can have

Some examples, in increasing order of cardinality:

- service name: only one of these per service
- build id: changes every time you deploy
- user name: scales with the number of users
- session id: different every time someone logs in
- request id: different every time a browser makes an HTTP request