Monitoring software solutions

UA.DETI.IES

Recap

- Software development process
  - Different models (sequential, incremental, evolutionary, …)
- Agile development methods
  - Agile principles and project management
- DevOps Technical benefits
  - Continuous Integration and Continuous Delivery (CI/CD)

Team manager

Product owner

Developer

Architect

DevOps master

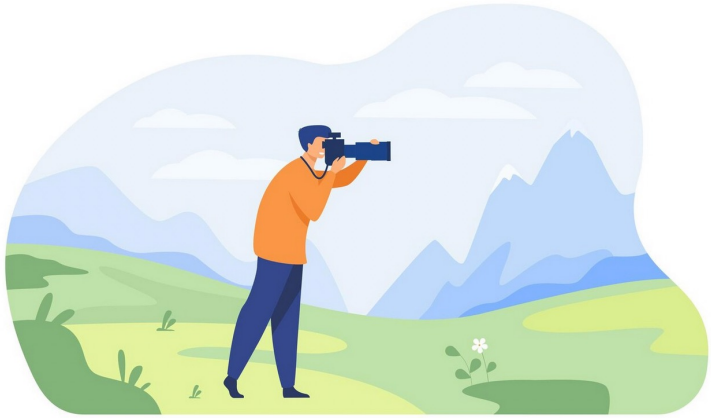
Recap

- Architectures
  - Different models (but a great percentage had a frontend and backend
- Microservices
  - To enhance specific components of the solution
- Event-driven
  - When several services need to exchange information
- AppSec
  - Some controls to enhance security


NOW WHAT?

Observability

- For a software application to have observability, you must be able to do the following:
  - Understand the **inner workings** of your application
  - Understand **any system state** your application may have gotten itself into
    - even new ones you have never seen before and couldn't have predicted
  - Understand the internal state **without shipping any new custom code to handle it**
    - because that implies you needed prior knowledge to explain it



Observability

- Definition
  - For software systems is a **measure** of how well you can understand and explain any state your system can get into, no matter how novel or bizarre

- If you can understand any bizarre or novel state without needing to ship new code, **you have observability**

Importance of observability

- Examples of questions to think to yourself:
  - Can you understand what **any particular user** of your software may be experiencing at **any given time**?
  - Can you **compare any arbitrary groups of user requests** in ways that let you correctly identify **which attributes are commonly shared by all users** who are experiencing unexpected behavior in your application?
  - Once you do **find suspicious attributes** within one individual user request, can you search across all user requests to **identify similar behavioral patterns** to confirm or rule out your suspicions?
  - … and more
    - See Chapter 1 of Observability Engineering (Charity Majors, Liz Fong-Jones, George Miranda)

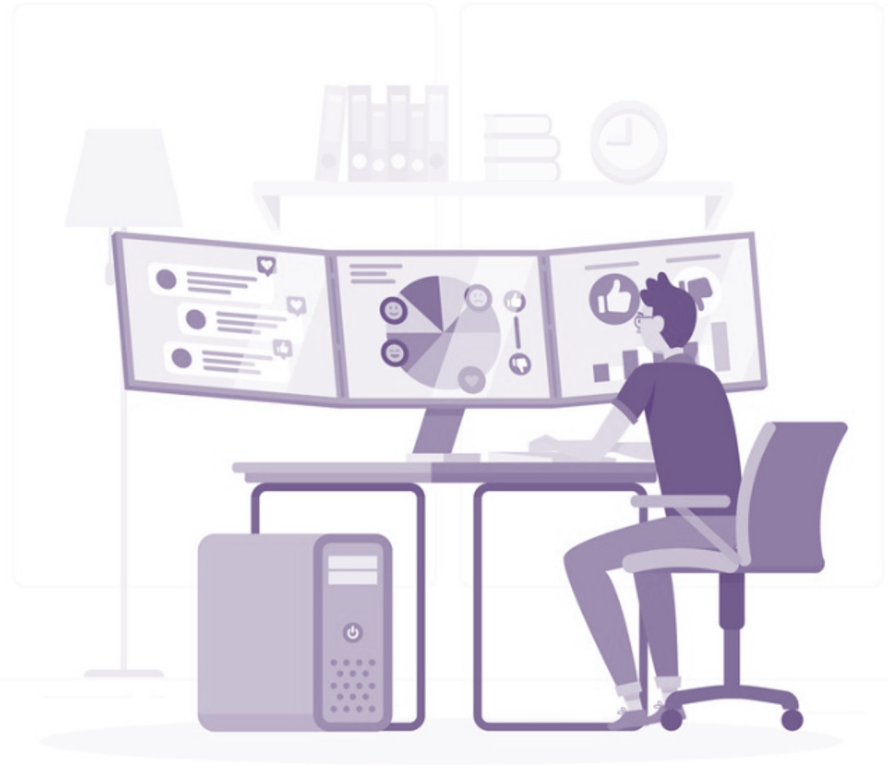
Applying observability to software

- How does one **gather that data** and **assemble it for inspection**?
- What are the **technical requirements** for processing that data?
- What **team capabilities** are necessary to benefit from that data?

Monitoring software

- Definition
  - **Observing** and **checking** the progress or quality of (something) over a period, to keep under systematic review

- Goals
  - Alert for possible issues
    - Broken features
  - Support future planning
  - Understand business performance
  - Detect anomalies
    - That can be malicious users



Monitoring Anti-Patterns

- *"An anti-pattern is **something that looks like a good idea,** but which **backfires badly when applied.**"*

  – Jim Coplien

Anti-Pattern #1: Tool Obsession

- Many monitoring efforts start out the same way
  - *"We need better monitoring!"* someone says
  - Someone else blames the current monitoring toolset
    - for the troubles they're experiencing
    - suggests evaluating new ones
- Hoping to find a single tool that will do all of that for you is simply delusional
- Monitoring is multiple complex problems under one name
  - Monitor your applications at: I) the code level; II) performance of infrastructure; and III) spanning tree topology changes

Anti-Pattern #1: Tool Obsession

- Avoid cargo-culturing tools
  - adopting tools and procedures of more successful teams and companies in the **misguided notion** that the tools and procedures are what made those teams successful

Anti-Pattern #1: Tool Obsession

- Sometimes, you really do have to build it!
  - Some scripting to help automatizing some specific processes
- The single pane of glass is a myth
  - It's great eye candy for visitors
  - But… it do not work

Anti-Pattern #2: Monitoring-as-a-Job

- At first glance, it makes sense
  - Create specialized roles so people can focus on doing that function perfectly
- However, when it comes to monitoring, there is a problem
  - **How can you build monitoring for a thing you don't understand?**
- The anti-pattern:
  - Shirk the responsibility of monitoring at all by resting it solely on the shoulders of a single person

Anti-Pattern #3: Checkbox Monitoring

- Set up the simplest and easiest things and check it off the to-do list
  - Since person setting up monitoring doesn't completely understand how the system works
- What does "working" actually mean?
  - Monitor that
- For example, monitoring a webapp
  - Check the setup of an HTTP GET /
  - Record the **HTTP response code**, expect an HTTP 200 OK response, **specific text** to be on the page, and the **request latency**

Anti-Pattern #3: Checkbox Monitoring

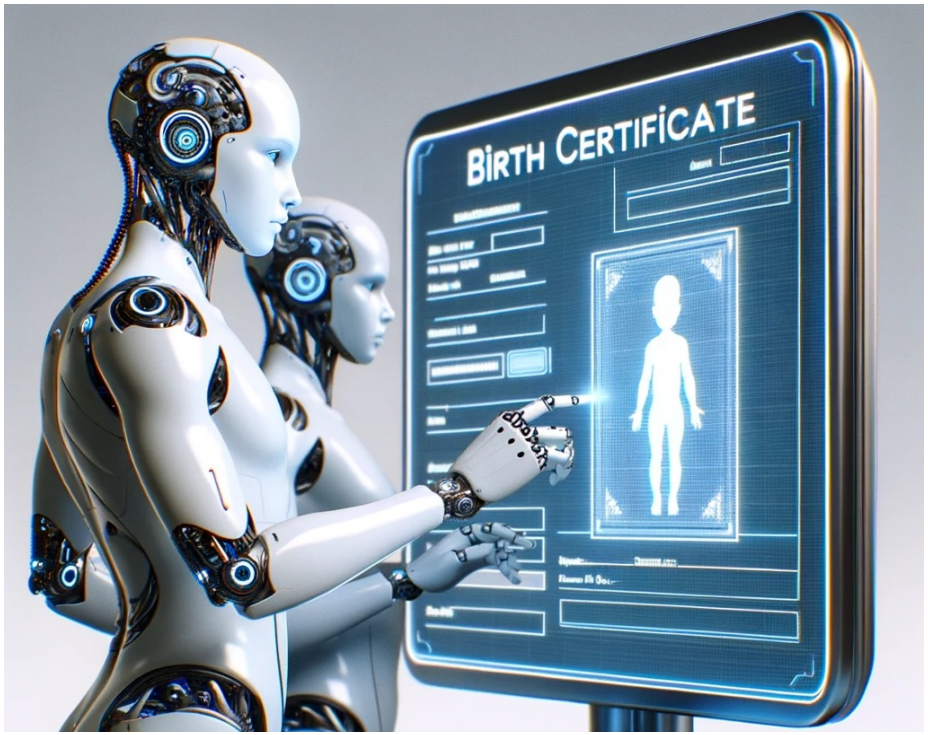
- You are recording metrics like system load, CPU usage, and memory utilization, but the service still **goes down without your knowing why**
- You find yourself **consistently ignoring alerts**, as they are false alarms often
- You are **checking systems for metrics every five minutes** or even less often
- You are **not storing historical metric data**

Anti-Pattern #4: Using Monitoring as a Crutch

- Monitoring is great for alerting to problems
  - but don't forget the next step: **fixing the problems**
- If constantly adding more monitoring to a service
  - stop and invest some effort into making the **service more stable and resilient** instead

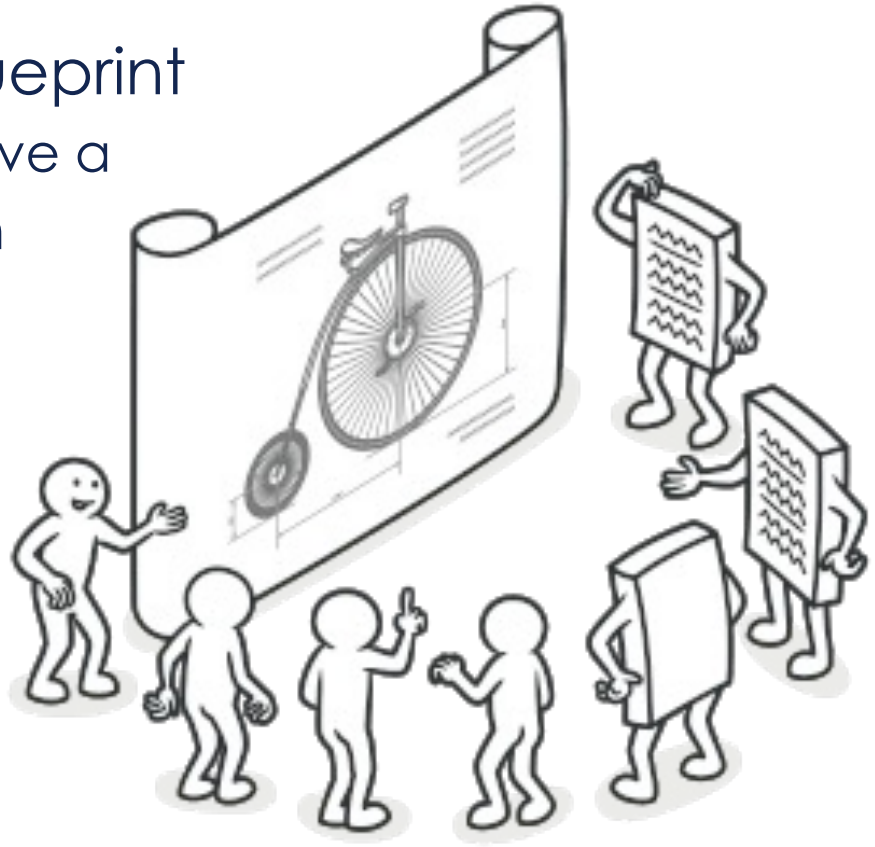
Anti-Pattern #5: Manual Configuration

- Monitoring should be 100% automated
- Services should **self-register** instead of someone having to add them



Monitoring Design Patterns

- Design patterns are typical solutions to common problems in software design
- Each pattern is like a blueprint
  - Can be customized to solve a particular design problem
- The same principles also apply to monitoring



Pattern #1: Composable Monitoring

- Use multiple specialized tools and couple them loosely together, forming a **monitoring "platform"**

- If one tool no longer suits the business needs
  - It can be replaced by another
  - Instead of replacing your entire platform

Pattern #1: Composable Monitoring

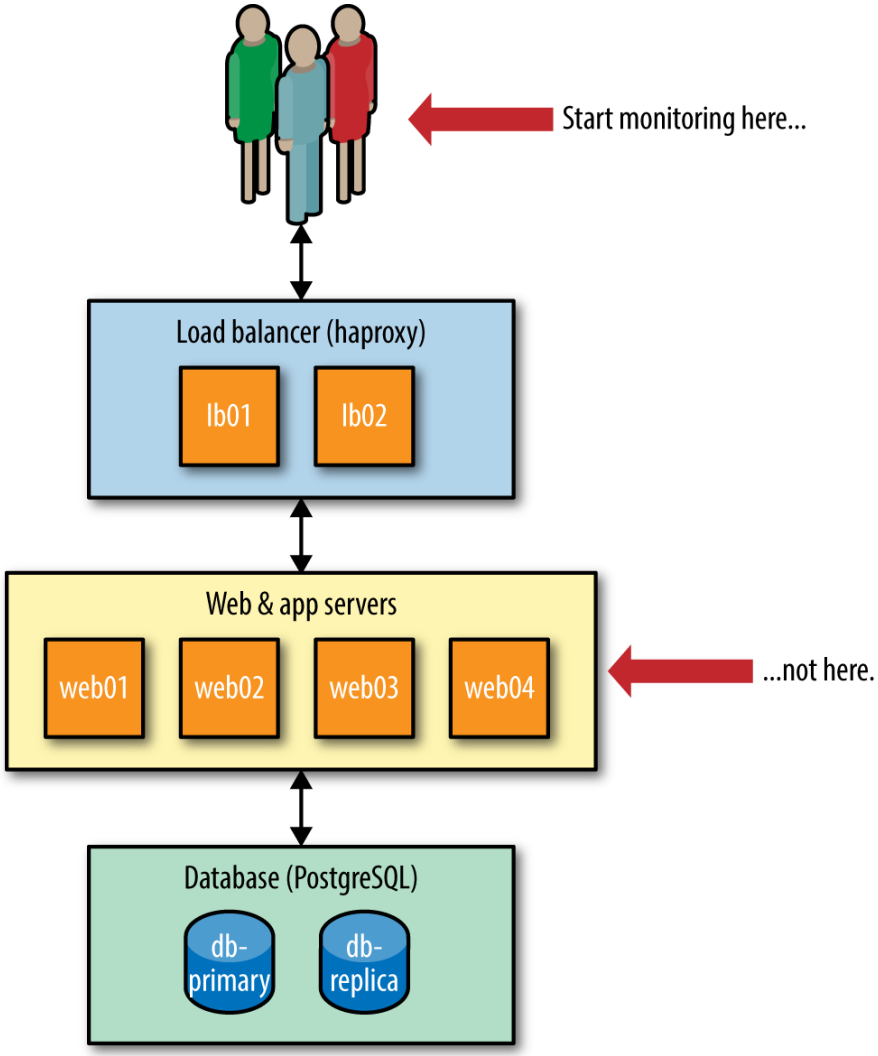
- A monitoring service has five primary facets:
  - Data collection
  - Data storage
  - Visualization
  - Analytics and reporting
  - Alerting

- Monitoring does not exist to generate alerts
  - Alerts are just one **possible outcome**

- Every collected metric and graph does not need to have a corresponding alert

Pattern #2: Monitor from the User Perspective

- A user does not care about the implementation details
- They care about whether the **application works**
- As such, you want **visibility from their perspective first**



Pattern #3: Buy, Not Build

- You are probably **not an expert at building and maintaining a high-throughput**, mission-critical monitoring services

- SaaS allows the team to focus on the company's product
  - Easier and quicker to get up and running with
  - Sometimes, those are free
    - Uptime Robot
    - Google Analytics

Pattern #4: Continual Improvement

- Building a world-class monitoring service take years
  - Several monitoring tools
  - Some of them discontinued
- Google, Facebook, Twitter, Netflix, ...
  - Started small and increased their monitoring services over time
- It is common to rearchitecting the monitoring service
  - Every two or three years as business needs change and the industry evolves

Monitoring the business

- Software is a piece of the business
  - It can be a core or a support piece
  - But… business owners are often asking different questions
    - When compared with software engineers

- Using monitoring tools, we can answer some executives' questions
  - Creating more value

Business KPIs

- Key Performance Indicators
  - Metric used to measure companies' health as a whole

- From an executive or founder's perspective
  - Are customers able to use the app/service?
  - Are we growing, shrinking, or stagnant?
  - How profitable are we? Is profitability increasing, decreasing, or stagnant?
  - Are we making money?
  - Are our customers happy?

Real-world example: Reddit

- Reddit is monetized via ads and Reddit Gold

- Measuring the core functionalities
  - User logins
  - Users currently on the site
  - Comments posted
  - Threads submitted
  - Votes cast
  - Private messages sent
  - Gold purchased
  - Ads purchased

Tying business KPIs to technical metrics

- 1st Reddit KPI: User logins
- Tracking user logins would give both successful and failed logins
- But… if the **backend service** responsible for **handling user logins** were having issues
  - This metric would not show us
- Tracking success and failure **separately is even better**

Tying business KPIs to technical metrics

Business KPI	Technical metrics
User logins	User login failures, login latency
Users currently on the site	Users currently on the site
Comments submitted	Comment submission failures, submission latency
Threads submitted	Thread submission failures, submission latency
Votes cast	Vote failures, vote latency
Private messages sent	Private message failures, submission latency
Gold purchased	Purchase failures, purchase latency
Ads purchased	Purchase failures, purchase latency

Tying business KPIs to technical metrics

Business KPI	Technical metrics
User logins	User login failures, login latency
Users currently on the site	Users currently on the site
Comments submitted	Com ubmission failures, subr tency
Threads submitted	Threa mission failures, submi ncy
Votes cast	Vot latency
Private messages	Priv ilures, sub
Gold purchased	Pu urchase latency
Ads purchased	urchase latency

My app does not have those metrics!

- Well…
  - To get visibility into the performance of the application and infrastructure, **we need to have a design for it**
- Since we can tie business KPIs to technical metrics
  - We just need to learn how to find them
- Maybe the **product owner** can help…
  - This role is focused on understand what the **customers want** and **work with engineering to get it built**

Frontend monitoring

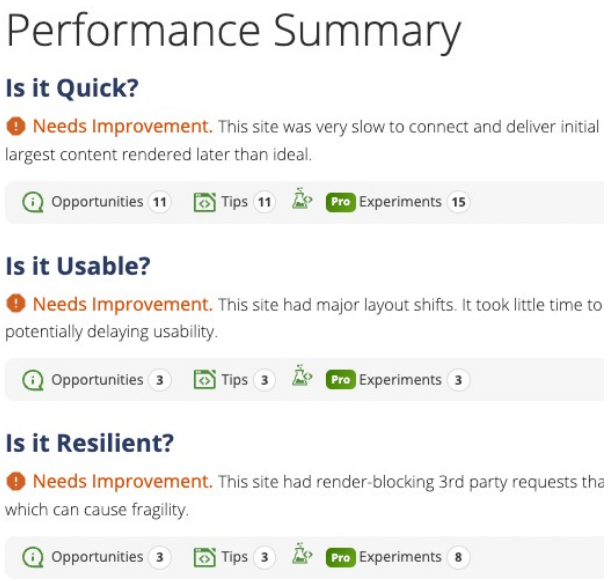
- Shopzilla's page load time dropped from 6 seconds to 1.2 seconds
  - Resulting in a **12% increase in revenue** and a **25% increase in page views**
- Amazon found that revenue increased by **1% for every 100 ms** of load time improvement
- How they know that?
  - By monitoring frontend
  - And **correlating data**

Frontend monitoring

- Two main approaches to frontend monitoring:
  - Real User Monitoring (RUM)
    - For instance, Google Analytics
  - Synthetic

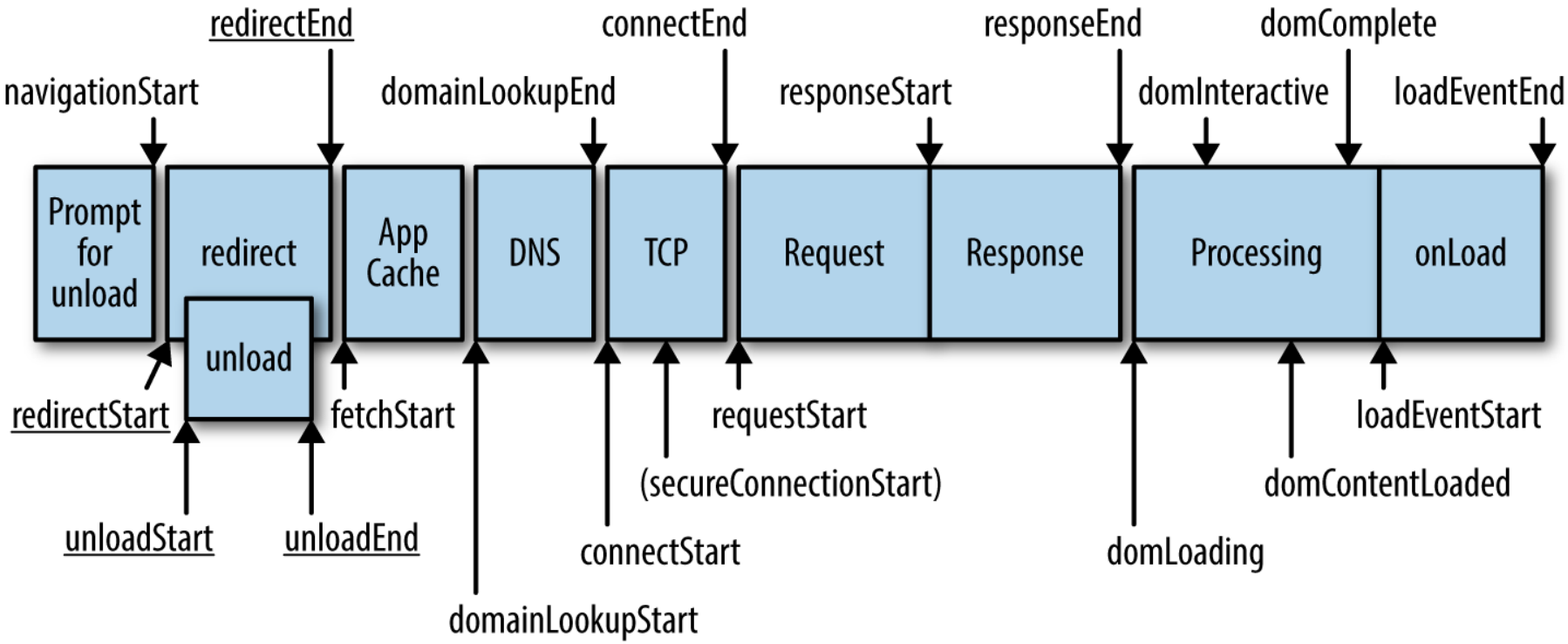
- RUM is the core of frontend monitoring
  - It is monitoring performance experienced by real users under real conditions

- Synthetic creates fake requests to generate data
  - Tools like WebpageTest.org



Frontend monitoring

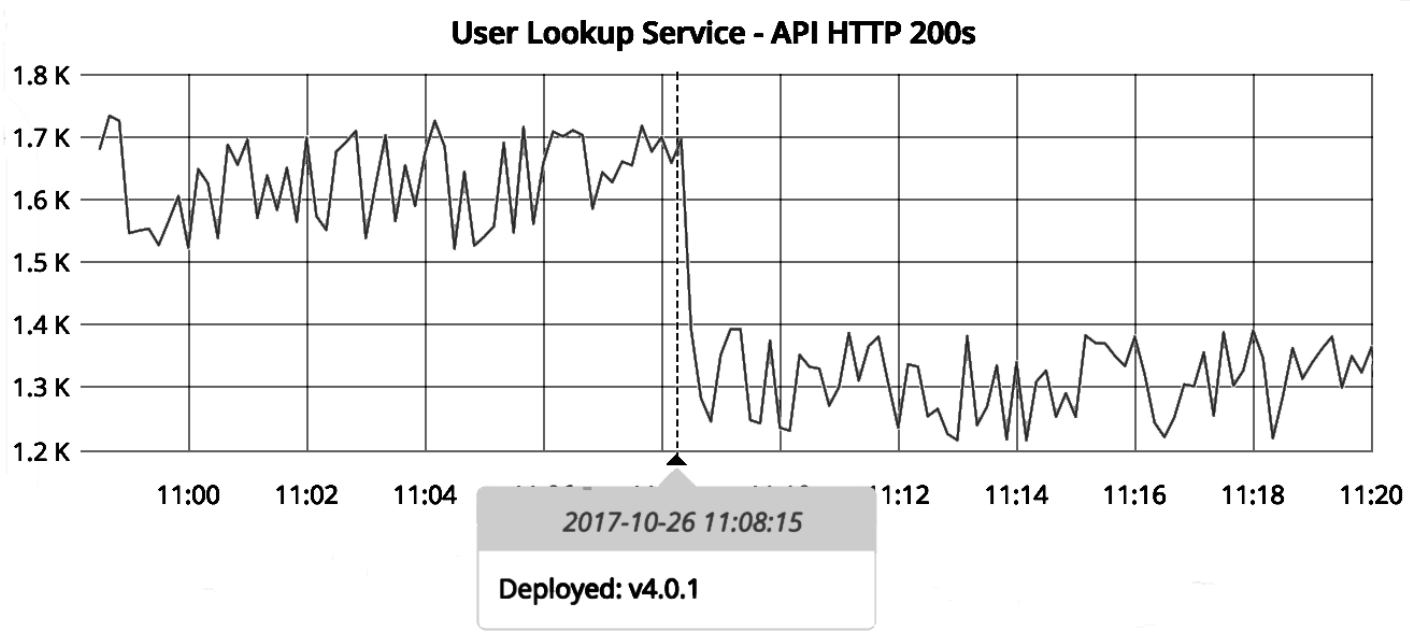
- Navigation timing API metrics



Application monitoring

- Instrumenting the application with metrics and logs
  - It increases the ability to understand and troubleshoot the performance of the applications

- Tracking releases and correlating with performance in apps and infrastructure



Application monitoring

- The **/health endpoint pattern** is very useful
  - Though only useful for certain architectural designs

```
def health():
    try:
        # Connect to a SQL database and select one row
        with sql_connection.cursor() as cursor:
            cursor.execute('SELECT 1 FROM table_name')
            cursor.fetchone()
        return JsonResponse({'status': 200}, status=200)
    except Exception, e:
        return JsonResponse({'status': 503, 'error': e}, status=503)
```

- A very simple query to return a single row
  - If the connection is successful, an HTTP 200 is returned
  - While an HTTP 503 is returned if it fails

Python can be one of the scripting languages for this.

Application monitoring

- **Microservice monitoring** is not that different from any other application
- Then **distributed tracing** is probably where we want to begin investing time and effort
- Application code runs on servers somewhere
  - The performance of those servers has huge impact on the performance of the application

Server monitoring

- The number of users grows, and we do not monitor the server
  - For some reason, we end up without space on the disk
  - What happens?

- Standard OS metrics
  - CPU
  - Memory
  - Network
  - Disk
  - Load
    - Measure how many processes are waiting to be served by the CPU

Server monitoring

- The number of users grows, and we do not monitor the server
  - For some reason, we end up without space on the disk
  - What happens?

- Standard OS metrics
  - CPU
  - Memory
  - Network
  - Disk
  - Load
    - Measure how many processes are waiting to be served by the CPU

Server monitoring

- SSL Certificates have an expiration date
  - Monitoring it is simple
  - We just want to know how long we have until they expire
    - And do something to let us know before that happens
- Web servers produce logs
  - How many response codes 5** our application is getting?

Groups	Meaning	Common codes
1**	Informational	100 Continue
2**	Success	200 OK, 204 No Content
3**	Redirection	301 Moved Permanently
4**	Client errors	400 Bad Request, 401 Unauthorized, 404 Not Found
5**	Server errors	500 Internal Server Error, 503 Service Unavailable

Server monitoring

- The number of connections is a good indicator of overall traffic levels
  - But… it isn't necessarily indicative of how busy the database actually is
- Slow queries are the **bane of high-performance** database infrastructures
  - A slow query will often manifest as a **slow user experience**
- Database servers is a special topic
  - Several works have been done on **database monitoring and tuning**

Other topics that can be monitored

- More topics about server monitoring
  - Load balancers, message queues, caching, DNS, logs, …
- Network monitoring
  - SMNP, traffic flows and other things
  - More about this on Networking and Services classes
- Security monitoring
  - Vulnerability detection and management
  - Intrusion Detection Systems (IDS)
  - More about this on Information And Organisational Security classes, or Master degree

Let's put this into practice – use case

Tater.ly's mission is to help french-fry aficionados find the best french fries in all the land. Users come to Tater.ly to look up restaurants and read reviews about their french fries, as well as post their own reviews. The french fries are also rated on a scale of one to five, with five being the best. Restaurants can create their own pages or users can create them. Restaurants can "claim" their pages if the page already exists. Tater.ly makes money through advertising by placing a Featured Fry at the top of search results, with restaurants paying an advertising fee for the slot. The ad fees are based on number of impressions—that is, the number of people that see the ad (as opposed to "clicks," that is, the number of people who click on the ad). Because the ad price is based on impressions, restaurant owners can choose how much to spend and whether to show their ad at peak times or non-peak times. It also allows us to run multiple ads.

Let's put this into practice – use case

Tater.ly's mission is to help french-fry aficionados find the best french fries in all the land. Users come to Tater.ly to look up restaurants and ***read reviews about their french fries****, as well as* ***post their own reviews****. The french fries are also* ***rated on a scale of one to five****, with five being the best.* ***Restaurants can create their own pages*** *or users can create them.* ***Restaurants can "claim" their pages*** *if the page already exists. Tater.ly* ***makes money through advertising*** *by placing a Featured Fry at the top of search results, with restaurants paying an advertising fee for the slot. The* ***ad fees are based on number of impressions****—that is, the number of people that see the ad (as opposed to "clicks," that is, the number of people who click on the ad). Because the ad price is based on impressions,* ***restaurant owners can choose how much to spend*** *and whether to show their ad at peak times or non-peak times. It also allows us to run multiple ads.*

Let's put this into practice – metrics

- To start off, there are some basic metrics that tell us the state of the venture:
  - The number of restaurants reviewed
  - The number of active restaurants (that is, restaurant page owners logging in)
  - The number of users
  - The number of active users
  - Searches performed
  - Reviews placed
  - Ads purchased
  - The direction and rate of change for all of the above

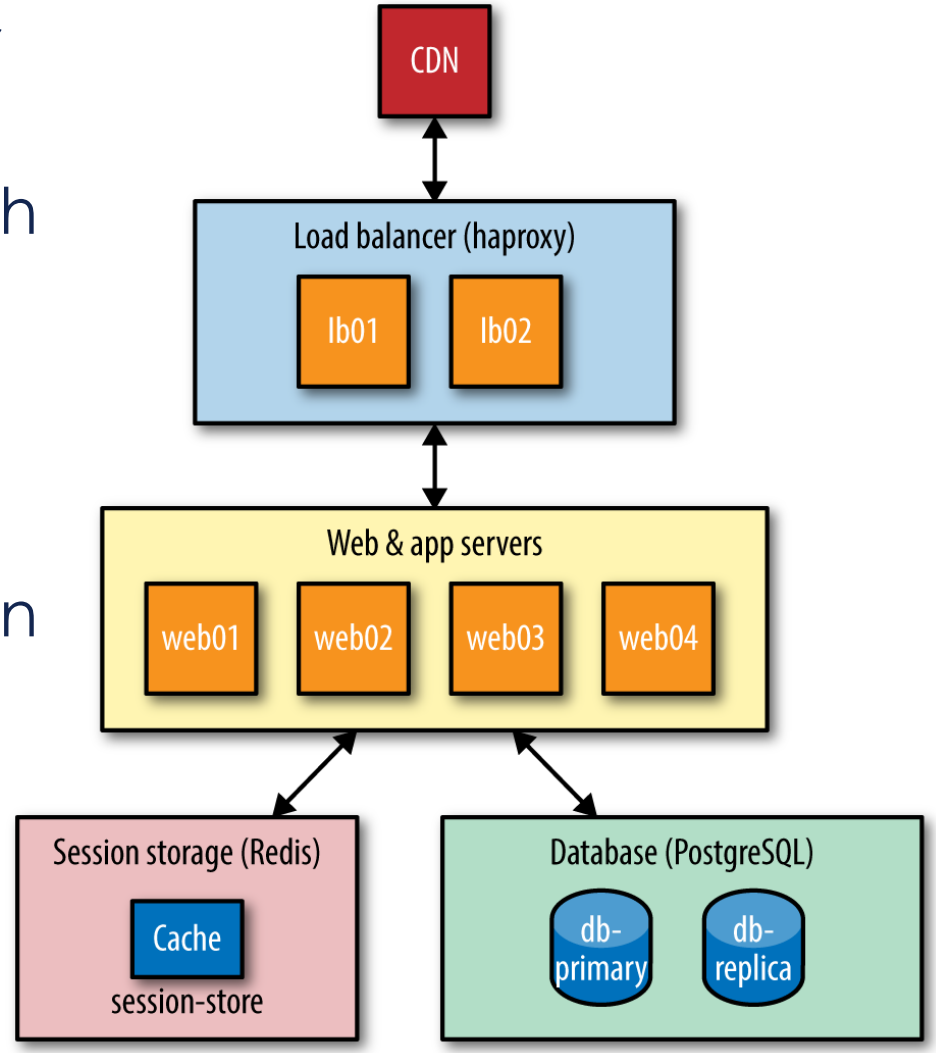
Let's put this into practice – monitoring

- Frontend
  - There is only one big thing we need to make sure we've got: RUM metrics
    - Using a frontend monitoring tool

- Application and Server
  - First, we need to check the architecture of the solution
  - It uses a web hosting provider rather than their own datacenters
    - managing hardware and network is of low concern to them

Let's put this into practice – architecture

- A a standard three-tier architecture
- Traffic comes in through a CDN
  - with the origin set to our load balancers (2x)
- 4x web servers
- PostgreSQL database in primary-replica configuration
- Single Redis server for session storage



Let's put this into practice – metrics

- User actions
  - Page load time
  - User logins: successes, failures, length of time taken, daily active users, weekly active users
  - Searches: number performed, latency
  - Reviews: reviews submitted, latency

- Load balancer
  - Haproxy
    - requests per second
    - healthy/unhealthy backends
    - HTTP response codes at frontend and backend

Let's put this into practice – metrics

- Database
  - PostgreSQL (inside the app)
    - query latency
  - PostgreSQL (at the database server)
    - transactions per second
- Session storage
  - Redis (inside the app)
    - query latency
  - Redis (at the Redis server)
    - transactions per second
    - hit/miss ratio
    - cache eviction rate

Let's put this into practice – metrics

- Web & app servers
  - Apache
    - requests per second
    - HTTP response codes
  - Standard OS metrics
    - CPU utilization
    - Memory
    - Network throughput
    - Disk I/O per second (IOPS) and space

Let's put this into practice – logs

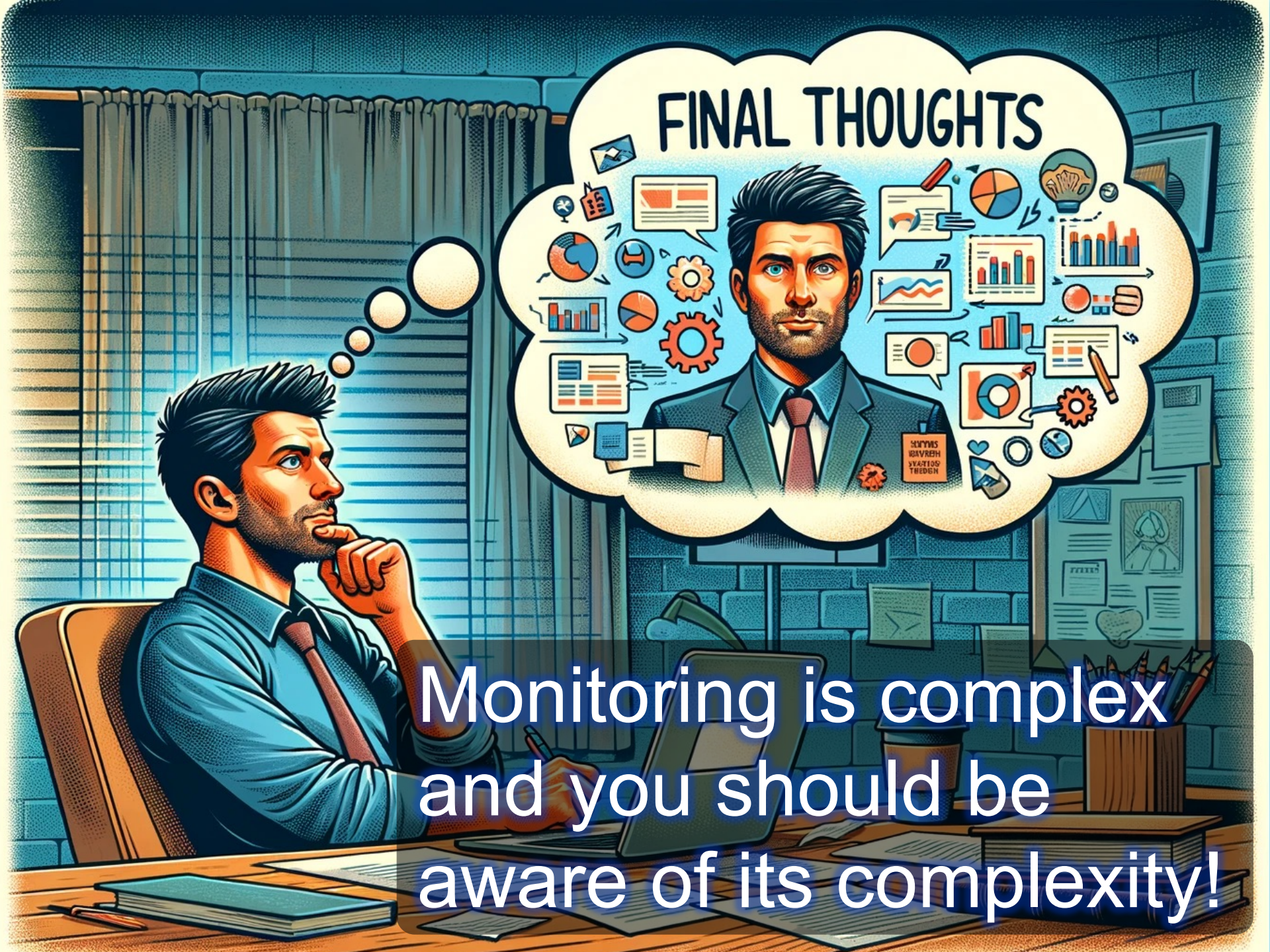
- User logins
  - User ID
  - Context (success? failure? reason for failure?)
- App
  - Exceptions/tracebacks
- The service logs for all the server-side daemons we're using
  - Apache, PostgreSQL, Redis log, and haproxy

Let's put this into practice – security

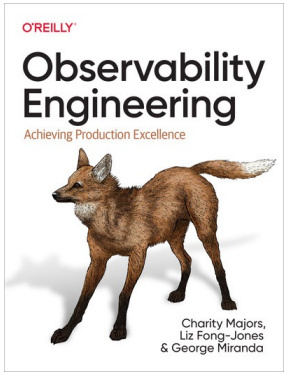
- Since isn't subject to any compliance or regulatory requirements, security monitoring is straightforward:
  - SSH
    - login attempts and failures
  - syslog logs
  - auditd logs

Let's put this into practice – alerting

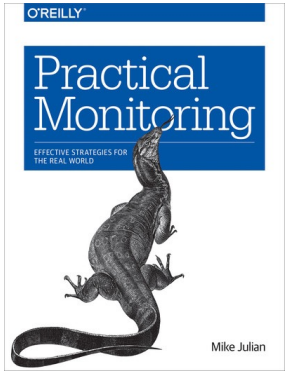
- Looking at the metrics and logs we've identified; It would be expected these alerts to be in place:
  - Page load time increasing
  - Increasing error rates and latency on Redis, Apache, and haproxy
  - Increasing error rates and/or latency for certain application actions
    - searches, review submissions, user logins
  - Increasing latency on PostgreSQL queries


FINAL THOUGHTS
Monitoring is complex and you should be aware of its complexity!

Resources & Credits



- Charity Majors, Liz Fong-Jones, George Miranda, Observability Engineering, O'Reilly Media, Inc., 2022
  chapters 1 - 4



- Mike Julian, Practical Monitoring, O'Reilly Media, Inc., 2017