In development we are testing to make sure the application works as designed. As you approach your deployment, you want to make sure your application works when used in ways you didn’t intend. TL;DR: Users are terrible. The world is filled with evil forces (and idiots). Use the CI tooling we have learned about (and will learn about) today to help us.

Recall: CI, CD and more

CI rigorously tests every integration in production-like environment
- Prevent development-production mismatch
  - Test multiple browsers, etc.
  - “Stress test” code for performance, fault-tolerance, etc.

Make what was rare and fraught commonplace!
Recall: DevOps principles

- Involve operations in each phase of a system’s design and development,
- Heavy reliance on automation versus human effort,
- The application of engineering practices and tools to operations tasks

In addition to Heroku, what are some other “DevOps”ey things you have done this semester? One example is using the “scripts” in the package.json files to automate complex operations.

As we you move up levels of abstractions, increasingly someone else takes care of installing Linux, Nginx, etc., patching security vulnerabilities, library hell (incompatible libraries, etc.), automating scaling, etc.
What is the trade-off? You are paying for the PaaS to handle all of those tasks.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
What about upgrades?

- Can’t or don’t want to rollout new feature simultaneously to all servers
  Version \( n \) and \( n+1 \) will co-exist
- Naïve solution: Downtime
- Alternative: Feature flags
  1. Do non-destructive migration
  2. Deploy code protected by feature flag
  3. Flip feature flag on; if disaster, flip it back
  4. Once all records moved, deploy entirely new code
  5. Apply migration to remove old columns
- Other FF uses: A/B testing, ...

Can’t (takes time to update potentially many servers), won’t (if bad, want to expose minimum number of users before rollback)

Preview of role for monitoring. Best case automatically detect problems and rollback change. An example of automation (at a deep level) and engineering tools applied to operations tasks.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Kinds of monitoring

“If you haven’t tried monitored it, assume it’s broken.*”

• At development time (profiling)
  Identify possible performance/stability problems before they get to production

• In production
  Internal: Instrumentation embedded in application and/or framework
  External: Active probing by other site(s)/tools.

“The sources of potential complexity are never-ending. Like all software systems, monitoring can become so complex that it’s fragile, complicated to change, and a maintenance burden. Therefore, design your monitoring system with an eye toward simplicity. In choosing what to monitor, keep the following guidelines in mind: The rules that catch real incidents most often should be as simple, predictable, and reliable as possible....”

*Google SRE Book
Performance and security metrics

<table>
<thead>
<tr>
<th>Performance &amp; Stability</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability or Uptime</td>
<td></td>
</tr>
<tr>
<td><em>What % of time is site up and accessible?</em></td>
<td></td>
</tr>
<tr>
<td>Responsiveness</td>
<td></td>
</tr>
<tr>
<td><em>How long after a click does user get response?</em></td>
<td></td>
</tr>
<tr>
<td>Scalability</td>
<td></td>
</tr>
<tr>
<td><em>As number users increases, can you maintain responsiveness without increasing cost/user?</em></td>
<td></td>
</tr>
<tr>
<td>Authorization (Privacy)</td>
<td></td>
</tr>
<tr>
<td><em>Is data access limited to the appropriate users?</em></td>
<td></td>
</tr>
<tr>
<td>Authentication</td>
<td></td>
</tr>
<tr>
<td><em>Can we trust that user is who s/he claims to be?</em></td>
<td></td>
</tr>
<tr>
<td>Data integrity</td>
<td></td>
</tr>
<tr>
<td><em>Is users’ sensitive data tamper-evident?</em></td>
<td></td>
</tr>
</tbody>
</table>

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Google’s 4 “golden” signals

- **Latency**  
  Can be confounded by errors. How?  
  *Time to service a request*

- **Traffic**  
  Application specific metric: requests/s, I/O rate, ...  
  *How much demand is being place on your system*

- **Errors**  
  *Rate of requests that fail*

- **Saturation**  
  *How “full” your system is (when will you hit ceiling?)*

“Fast” errors will reduce your overall latency resulting in misleading metrics. Even worse though are “slow errors”.
Let:
F = Film Explorer’s (FE) availability
H = Heroku’s availability
C = Internet connection availability
P = Your perception of FE availability

Which relationship among these quantities holds?

A. $P \leq C \leq H \leq F$
B. $P \geq \min(F, H, C)$
C. $P \leq C \leq \min(H, F)$
D. Insufficient information to answer

Answer: D

What information is missing? How someone is using your app? Are they on it constantly or just once a week? If the latter happens when the app is down... perception will be worse than reality. What we really care about is availability for a specific user population. More generally though, our instinct is $P \leq \min(F, H, C)$. A reminder that there are a lot of moving parts to successfully delivering your application.
“Premature optimization is the root of all evil”*

• Users expect speed!
  99 percentile matters, not just “average”
• There are lots of reasons for “too slow”
• Don’t assume, measure!
  Monitoring is your friend: measure twice, cut once!

*Variously attributed to Hoare, Knuth, Dijkstra, ...
Simplified (& false) view of response time

For normal distribution of response times:
±2 standard deviations around mean is 95% CI

Average response time $T$ means:
95%ile users are getting $T+2\sigma$
99.7%ile users get $T+3\sigma$

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
The mean (and 95% are very misleading!) You are likely not satisfied with mean performance. Instead need to have a threshold for "satisfactory".

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Service Level Objective (SLO): Target value for your service

Instead of worst case or average metric, specify a percentile, target and window

99% of requests complete in < 1 second, averaged over a 5 min. window

SLOs set customer expectations

Make sure you have a safety margin

Overachieving can be problematic too! How?

Service Level Agreements (SLAs) attach contractual obligations to SLOs

How could overachieving possibly be problematic? Creates over-dependency (i.e. consumer assumes that service never goes down, but then it does...) Google actually introduces planned outages to prevent over dependency.

Film Explorer's target uptime is 99.9% (three nines...). Yesterday there was a one hour outage. Which of the following is true?

A. Because of the outage, Film Explorer can't meet it's uptime goal this year
B. Film Explorer can still meet it's uptime goal for the year only if there are no more outages
C. Film Explorer can still meet it's uptime goal for the year even if there are more outages
D. Depends on users. If no users tried to access FE during window, then uptime wasn't impacted

Answer: C

Three nines corresponds to 8h45m57s of downtime per year, so the yearly goal is OK. However if it is a monthly goal, only 43m50s of downtime permitted per month. See https://uptime.is/99.9.
How can you fix “slow”? 

- **Add more resources, i.e. over-provision**
  Easy to scale presentation and logic tiers for small sites (readily automated in the “cloud”)
  More expensive for larger sites (10% of 10,000 machines is a big number!)
- **Make your application more efficient**
  Most effective when one bottleneck

What this means that are real advantages to staying small... And by that I mean small in terms of resources (not in customers, etc.) Conversely when you are really big even small optimizations have big payoffs, i.e. small relative improvements have big absolute impacts.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
The fastest computation is the one you don’t do

- Don’t forget big-O and CS fundamentals, e.g.
  
  `Array.include vs. Set for unique`
  
  Smart use of DB indexes

- Caching (and memoization more generally)

- Avoid “toxic” queries, e.g.
  
  “n+1” query for associations

---

DB indexes are at their heart a data structures problem, i.e. how do you turn a linear scan into a sub-linear lookup.

Outgrowing single-machine database requires big investment in sharding, replication, etc. As an alternative, find ways, like those above to relieve pressure on DB (so that you can stay in a PaaS friendly tier).

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Indexes: $O(< n)$ queries

Index is a tree, hash-table or other data structure optimized for efficient queries

<table>
<thead>
<tr>
<th># of reviews:</th>
<th>2000</th>
<th>20,000</th>
<th>200,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read 100, no indices</td>
<td>0.94</td>
<td>1.33</td>
<td>5.28</td>
</tr>
<tr>
<td>Read 100, FK indices</td>
<td>0.57</td>
<td>0.63</td>
<td>0.65</td>
</tr>
<tr>
<td>Performance</td>
<td>166%</td>
<td>212%</td>
<td>808%</td>
</tr>
</tbody>
</table>

Sub-linear scaling!

Why not use an index for every field?
- Requires additional storage space for each index
- Slows down insert/edit (need to update the index)

Read 100 reviews out of table via foreign key, i.e. Review.movie_id

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
There is another version of this joke: “There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors.”

Need to be thoughtful about what can actually be cached (do you have to be logged in?) and handling expiration (did something change). For example, imagine Film Explorer won’t show listings for NC-17 movies to user’s under 17. How would that impact caching? Can’t use client side or page caching in the web server.

https://martinfowler.com/bliki/TwoHardThings.html

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
n+1 queries (or leaky abstractions)

Recall in the Film Explorer a user “has many” movies “through” ratings
User.query().where('zip', '05753').then((fans) => {
  fans.forEach((fan) => {
    fan.$relatedQuery('movies')
  });
});

1 query for each user (i.e. n+1 queries for n users)
More subtle for other ORMs, e.g. fan.movies() is really a query

User.query()
  .where('zip', '05753')
  .eager('movies')
  .then((fans) => {
    fans.forEach((fan) => {
      fan.movies ...
    });
  });

DB “leaking” is more relevant to ORMs like Active Record (where queries aren’t so obvious). Can’t just do the natural thing ... need to take DB into account.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Suppose Movie has many Users through Ratings. Which foreign-key index would most speedup the query:
```
movie.$relatedQuery('raters')
```
which obtains the users who rated that movie.

<table>
<thead>
<tr>
<th>Movie</th>
<th>Rating</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>movield</td>
<td>userid</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>53</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

A. Movies.ratingId
B. Ratings.movield
C. Ratings.userId
D. Users.reviewId

Answer: B

Recall the foreign keys are not present in the Users or Movies table, so those are not relevant answers. To identify the users who rated a movie we will need to query the Ratings table for all the ratings for that movie, i.e. the WHERE criteria will involve Ratings.movield. That query will benefit from an index on Ratings.movield. The corresponding userId values will be used to query into the User table (either as a join or separate query). We won't benefit then as much from an index on Ratings.userId (and there is an automatic index on the primary key Users.id).