Recall: Deployment is closing the loop

*Programs that are never deployed have not fulfilled their purpose. We must deploy!*  
To do so we must answer:  
• Is our application in a working state?  
• Do we have the necessary HW/SW resources?  
• How do we actually deploy?

Presumably we are building applications to solve problems for users (not just for us). An application that is never deployed for use by those users has not fulfilled its purpose.

You have done all these things! We use testing and Travis-CI for the first, Heroku for the second and Heroku’s git push for the third...
Recall that Contiguous Integration (CI) – what we have been practicing emphasizes frequent small integrations (hence the name). Some of the key principles:

- Maintain a single source repository (with an always deployable) branch
- Automate the build
- Build should be self testing
- Everyone integrates with the master frequently
- Automate deployment

A key element of CI is rigorously testing every integration. We use Travis-CI for that purpose. Once those tests past we should be ready to integrate and deploy.

https://martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration

Once we are confident our application is deployable, there are two related concepts:

* **Continuous Deployment:** Every change automatically gets put into production, and thus there are many production deployments each day.
* **Continuous Delivery:** An extension of CI in which SW is deployable throughout its lifecycle, the team prioritizes keeping SW deployable, and it is possible to automatically deploy SW on demand.

https://martinfowler.com/bliki/ContinuousDelivery.html
In our projects we are aiming for a Continuous Delivery-like workflow in which our applications start and stay deployable throughout the development process. As with CI, this reduces the complexity (and risk) of deployment by enabling us to do so in small increments. And Continuous Delivery facilitates getting user feedback by frequently getting working SW in front of real users. Although to mitigate risk companies will often first deploy for a small subset of users.
True or False? The development team's goal of launching new features is in conflict with the operations team's goal of ensuring services stay live and usable.

A. True
B. False

Answer: True

From the Google SRE handbook: "At their core, the development teams want to launch new features and see them adopted by users. At their core, the ops teams want to make sure the service doesn’t break while they are holding the pager. Because most outages are caused by some kind of change—a new configuration, a new feature launch, or a new type of user traffic—the two teams’ goals are fundamentally in tension."
As a practical matter, the trend towards DevOps means that as the application developer you are responsible for more of the traditional “operations” tasks (provisioning machines, deploying, etc.) while “operations” teams are increasingly automating operational tasks to support frequent deployment, fault tolerance, and more.

Definition sourced from: https://landing.google.com/sre/book
The operational work involved in supporting a service can realistically scale how as the service grows by 10X?

A. \( O(1) \): Just one-time efforts to add resources
B. Sublinear: There will be additional work required as a function of service size
C. \( O(n) \): The effort will have to grow linearly with demand
D. Greater than \( O(n) \): Increasing scale means increasing complexity

Answer: A

Again from the Google SRE handbook: "An ideally managed and designed service can grow by at least one order of magnitude with zero additional work, other than some one-time efforts to add resources." If we don't achieve at least sub-linear scaling we have real trouble as any growth will consume a corresponding amount of resources (that is our growth will consume all the new resources, e.g. revenue, that growth brings).

To do so, one needs highly automatic systems. Automation goes beyond just provisioning resources, it is techniques like automatically rolling out changes to a small fraction of users, detecting errors (through monitoring) and then automatically rolling back the changes!

This lecture extensively references Google’s SRE handbook. The role of site reliability engineer (SRE) is the closest to DevOps as a job. Popularized by Google, SREs are engineers who focus on running products and "create systems to accomplish the work that would otherwise be performed, often manually, by sysadmins."
Heroku is an example of PaaS. All of you have needed to do is deploy! No or minimal configuration required. Amazon AWS and other cloud providers would be an example of IaaS. You don’t ever interact with the physical HW (and can (de-)provision automatically and on-demand), but you are responsible the installation and configuration of software, configuring networking, etc. At the the lowest-level you could buy and setup the physical HW sometimes in your own data centers or in rented datacenter space.

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.

*aaS: __________ as code

<table>
<thead>
<tr>
<th>Platform-as-a-Service</th>
<th>Three-tier architecture as code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Deploy (that’s it!)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Infrastructure-as-a-Service</th>
<th>“Infrastructure as code”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Configure (with tools like Ansible, etc.)</td>
</tr>
<tr>
<td></td>
<td>2. Deploy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bare Metal</th>
<th>Just infrastructure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Rack</td>
</tr>
<tr>
<td></td>
<td>2. Configure</td>
</tr>
<tr>
<td></td>
<td>3. Deploy</td>
</tr>
</tbody>
</table>
What is the trade-off? You are paying for the PaaS to handle all of those tasks and at (above) a certain scale you could do it for less.

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

- 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), and don’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....”
Performance and security metrics

Availability or Uptime
What % of time is site up and accessible?

Responsiveness
How long after a click does user get response?

Scalability
As number users increases, can you maintain responsiveness without increasing cost/user?

Authorization (Privacy)
Is data access limited to the appropriate users?

Authentication
Can we trust that user is who s/he claims to be?

Data integrity
Is users’ sensitive data tamper-evident?

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
“Fast” errors will reduce your overall latency resulting in misleading metrics. Even worse though are “slow errors”.

---

**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?)*
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 <= C <= H <= F
B. P >= min(F, H, C)
C. P <= C <= 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 <= 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, ....

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Simplified (& false) view of response time

For *normal distribution* of response times:
\[ \pm 2 \text{ standard deviations around mean is 95\% CI} \]

*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.
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.


---

**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
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 its uptime goal this year
B. Film Explorer can still meet its uptime goal for the year only if there are no more outages
C. Film Explorer can still meet its 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 is one of the hardest components to scale, aim to be kind to your database.*

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” films “through” ratings

```javascript
User.query().where('zip', '05753').then((fans) => {
  fans.forEach((fan) => {
    fan.$relatedQuery('films')
  });
});
```

1 query for each user (i.e. n+1 queries for n users)

More subtle for other ORMs, e.g. `fan.films()` is really a query

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

Just 1 or 2 queries, but DB “leaking” through ORM abstraction

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.

This is reminder that while libraries like Objection.js make our life easier and help us be more productive, it is still critical understand what the tools are doing behind the scenes. And thus what is fast or not.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Suppose Film has many Users through Ratings. Which foreign-key index would most speedup the query:

```js
film.$relatedQuery('raters')
```

which obtains the users who rated that movie.

A. Films.ratingId
B. Ratings.filmId
C. Ratings.userId
D. Users.reviewId

Answer: B

Recall the foreign keys are not present in the Users or Films 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.filmId. That query will benefit from an index on Ratings.filmId. 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).