Recall: Deployment is closing the loop

Programs that are never deployed have not fulfilled their purpose. We must deploy!

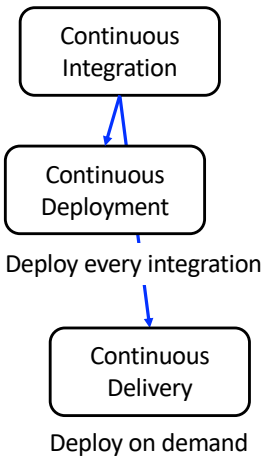
To do so we must answer:

- 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 GitHub actions, for the first, csci312.dev for the second and csci312.dev's git push for the third…

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.

Then we deploy!

By deploying frequently, we make what was rare and fraught common and unremarkable!

CI/CD was a mechanism to ensure our application is in working state, and possibly directly and automatically deploy each new change.

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 GitHub actions 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.

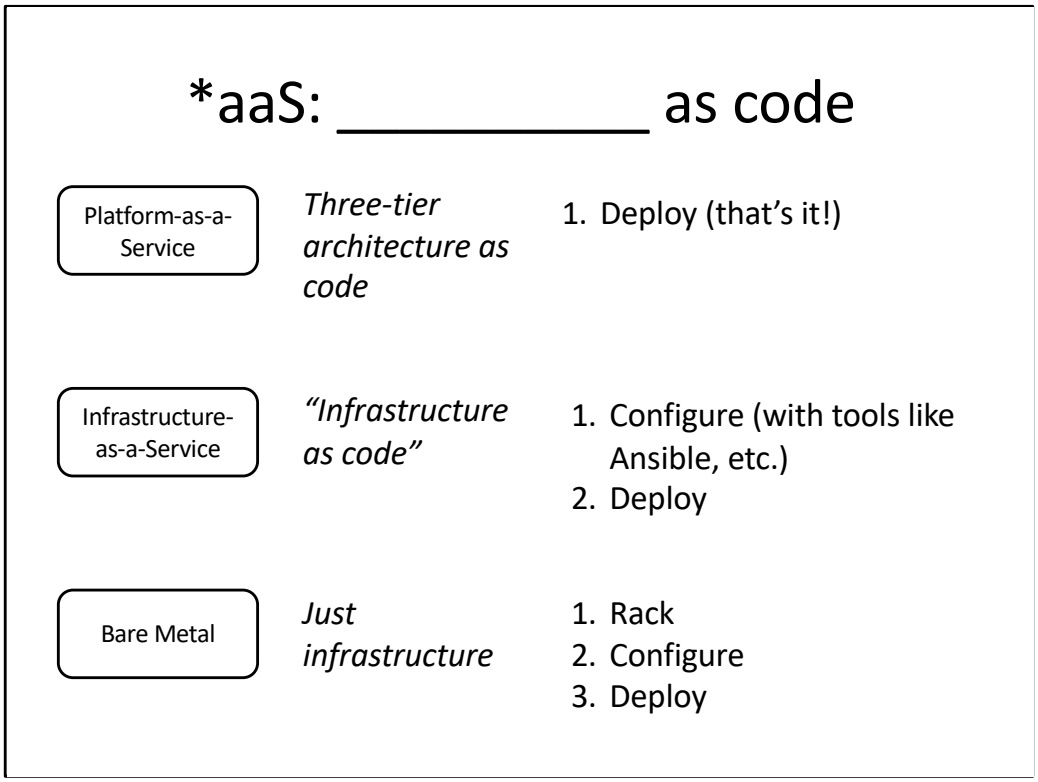
Recall: DevOps principles

- 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

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. That is, they are creating tools that turn previously physical tasks, e.g., obtaining server hardware, into programmatic tasks.

In addition to csci312.dev, 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.

Definition sourced from: https://landing.google.com/sre/book



Increasingly we obtain the necessary hardware, not by buying servers, but by programmatically provisioning cloud hardware and various levels of abstraction, described as "something as a service"

We will often partition *aaS (something-as-a-service) into 3 levels:…

csci312.dev, fly.io, Heroku are examples of PaaS. All of you must do is push code to 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. csci312.dev is implemented on top of an IaaS provider (Digital Ocean), that is I programmatically provisioned a server for us (somewhere) and configured it for our needs. At the the lowest-level you could buy and setup the physical HW sometimes in your own data centers or in rented datacenter space.

As we you move up levels of abstractions, increasingly someone else takes care of installing Linux, Nginx, etc., patching security vulnerabilities, library (in)compatability, automating scaling, etc.

The *aaS division of labor

PaaS handles…	You handle…
"Easy" tiers of horizontal scaling	Minimize load on database
Component-level performance tuning	Application-level performance tuning (e.g., caching)
Infrastructure-level security	Application-level security

For example, with a PaaS, the platform typically handles … while you handle … By component level performance tuning we mean tuning the front-end web server, load balancer, etc., i.e. , the components around your application (not the React components in your application). Examples of infrastructure-level security would be making sure the web server is up-to-date (fully patched), SSL (https) is configured correctly, etc.

What is the trade-off? You are paying for the PaaS to handle those tasks and at (above) a certain scale you could do it cheaper yourself.

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 there are bugs, we want to expose minimum number of users before rollback).

Preview of role for monitoring. The best case automatically detect problems and rollback change. Doing so is an example of automation (at a deep level), and engineering tools applied to operations tasks, i.e., DevOps and SRE fully realized.

What is A/B testing? That is where we want to test a particular design, content, etc. by performing a randomized experiment, i.e., some users see "A" and some see "B", and we test which performs better, e.g., leads more people to buy.

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

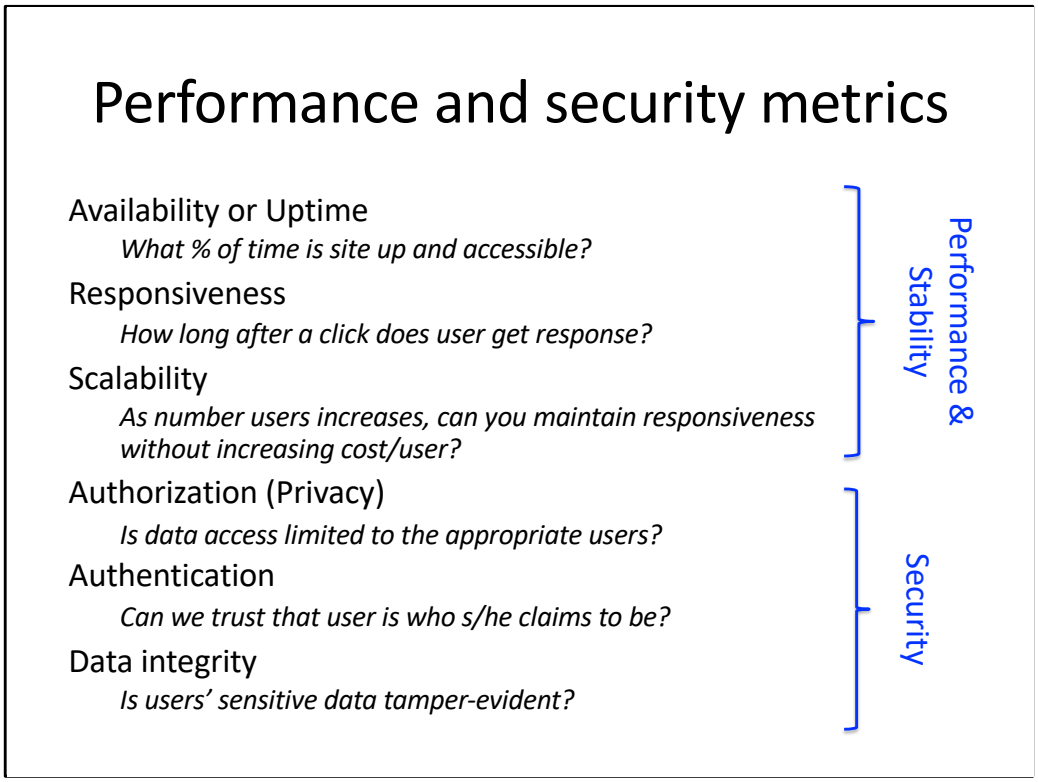
  External: Active probing by other site(s)/tools

*Google SRE Book

What do we mean by internal instrumentation? An example, we don't really do anything when a fetch failed (except maybe print the error message). Instead, we could send those errors to a monitoring service that helps us spot patterns, e.g., a sudden spike in failures for a particular endpoint, etc.

"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.

https://landing.google.com/sre/book/chapters/monitoring-distributed-systems.html#xref_monitoring_golden-signals



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?)

<click> Why can latency be confounded by errors? "Fast" errors will reduce your overall latency resulting in misleading metrics. Even worse though are "slow errors".

Let:
S = Simplepedia's availability
D = csci312.dev's (PaaS) availability
C = Internet connection availability
P = Your perception of Simplepedia's availability
Which relationship among these quantities holds?

A. P <= C <= D <= S
B. P >= min(S, D, C) P <= min(S, D, C)
C. P <= C <= min(D, S)
D. Insufficient information to answer

Answer: D

What information is missing? How is someone 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(S, 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

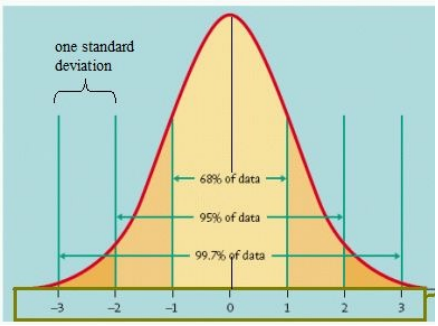
For *normal distribution* of response times:

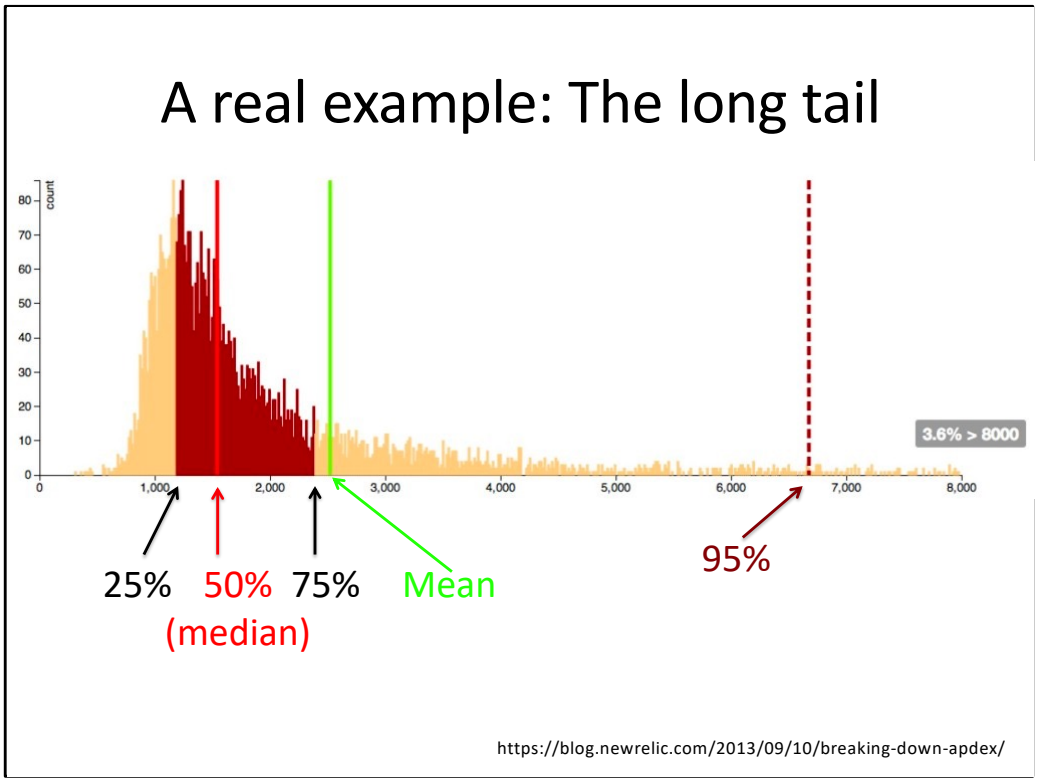
±2 standard deviations around mean is 95% CI

Average response time *T* means:

95%ile users are getting T+2σ

99.7%ile users get T+3σ





The mean (and standard deviation are very misleading!) You are likely not satisfied with mean performance. Instead need to have a threshold for ”satisfactory”.

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

<click 2x> How could overachieving possibly be problematic? Creates over-dependency (i.e., consumer assumes that service never goes down, but then it does…) Google introduces planned outages to prevent over dependency. And there are multiple tools that have been developed to simulate upstream service failures to make testing more robust.

https://landing.google.com/sre/book/chapters/service-level-objectives.html

Simplepedia'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, Simplepedia can't meet its uptime goal this year
B. Simplepedia can still meet its uptime goal for the year only if there are no more outages
C. Simplepedia 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 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, so we could meet our goal this month. See https://uptime.is/99.9.

Note that question isn't about perception, as we discussed before, but a specific metric uptime, that is independent of the users.

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 there is 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 big even small optimizations have big payoffs, i.e., small relative improvements have big absolute impacts.

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 investment in sharding, replication, etc. As an alternative, find ways, like those above, to relieve pressure on the DB.

Indexes: O(< n) queries

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

# of reviews:	2000	20,000	200,000
Read 100, no indices	0.94	1.33	5.28
Read 100, FK indices	0.57	0.63	0.65
Performance	166%	212%	808%

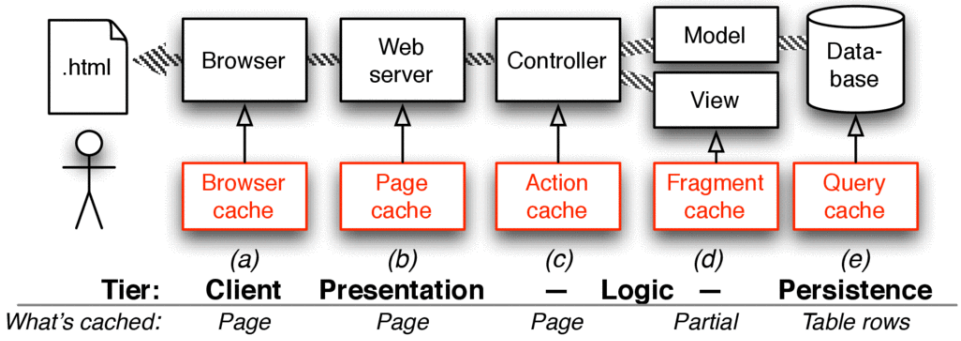
Sub-linear scaling!

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

Cache what hasn't changed



"There are only two hard things in Computer Science: cache invalidation and naming things." –Phil Karlton

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 be cached (e.g., do you have to be logged in?) and handling expiration (e.g., did something change). For example, imagine Film Explorer won't show listings for NC-17 movies to users under 17. How would that impact caching? Caching would have to be user aware, which would likely prohibit caching at the web server level (i.e., before the request even got to our application).

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

n+1 queries (or leaky abstractions)

Recall in the Film Explorer a user "has many" films "through" ratings

```
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., when `fan.films()` is really a query

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

Just 1 or 2 queries, but DB may "leak" through ORM abstraction

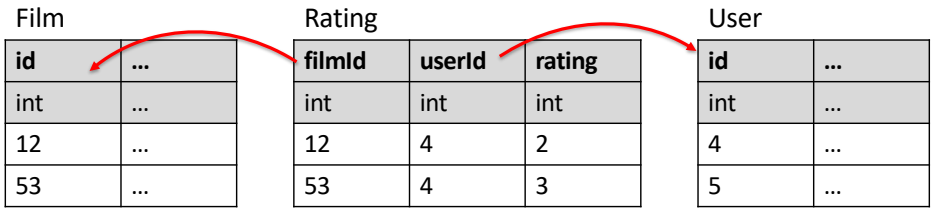
DB "leaking" is more relevant to ORMs like Active Record (where queries aren't so obvious). Can't just do the natural thing … need to take DB into account.

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.

Suppose Film *has many* Users *through* Ratings. Which of these foreign-key indexes would most speedup:

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

which obtains the users who rated that specific film.



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 A and D are not relevant/possible answers. To identify the users who rated a movie we will need to query the Ratings table for all the ratings for that movie. 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 from an index on Rating.userId, because we are pulling those values from the rows fetched from Rating.