Outline of topics

Continuous integration & continuous deployment

Upgrades & feature flags

Availability & responsiveness

Monitoring

Relieving pressure on the database

Defending customer data

Development vs. Deployment

Development:

- Testing to make sure your app works as designed

Deployment:

- Testing to make sure your app works when used in ways it was *not* designed to be used

Bad News

"Users are a terrible thing"

some bugs only appear under stress

production environment != development environment

the world is full of evil forces

and idiots

Good News: Deployment is much easier

- Before:
- get Virtual Private Server (VPS), maybe in cloud
- install & configure Linux, Rails, Apache, *mysqld, openssl, sshd, ipchains, squid, qmail, logrotate…*
- fix almost-weekly security vulnerabilities and find yourself in Library Hell
- tune all moving parts to get most bang for buck
- figure out how to automate horizontal scaling
- Today: ***PaaS*** *(platform as a service)*
- Heroku, CloudFoundry, Microsoft Azure, …
- Basic tuning, security, etc. professionally managed (“curated”)

Our goal: *stick with PaaS!*

PaaS handles…	We 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

Is this really feasible?

- Pivotal Tracker & Basecamp each run on a single DB (128GB commodity box <$10K)
- Many SaaS apps are not world-facing (internal or otherwise limited interest)

"Performance & security" defined

- Availability or Uptime

What % of time is site up & accessible?

- Responsiveness
- How long after a click does user get response?
- Scalability
- As # users increases, can you maintain responsiveness without increasing cost/user?
- 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?

Let R = RottenPotatoes app's availability
H = Heroku's availability
C = Comcast Internet availability
P = Armando's perception of RP availability

Which statement(s) are TRUE?

- $P \sim \max(C, H, R)$
- $P \leq \min(C, H, R)$
- $P \leq C \leq H \leq R$
- All of the expressions are true

Availability and Response time

- Gold standard: US public phone system, 99.999% uptime ("five nines")
- Rule of thumb: 5 nines ~ 5 minutes/year
- Since each nine is an order of magnitude, 4 nines ~ 50 minutes/year, etc.
- Good Internet services get 3-4 nines
- Response time: how long after I interact with site do I perceive response?
- For small content on fast network, dominated by latency (not bandwidth)

How important is response time?

- How important is response time?*
- Amazon: +100ms => 1% drop in sales
- Yahoo!: +400ms => 5-9% drop in traffic
- Google: +500ms => 20% fewer searches
- Classic studies (Miller 1968, Bhatti 2000)

<100 ms is “instantaneous”

>7 sec is abandonment time

Jeff Dean,
Google Fellow

“Speed is a feature”

- http://code.google.com/speed

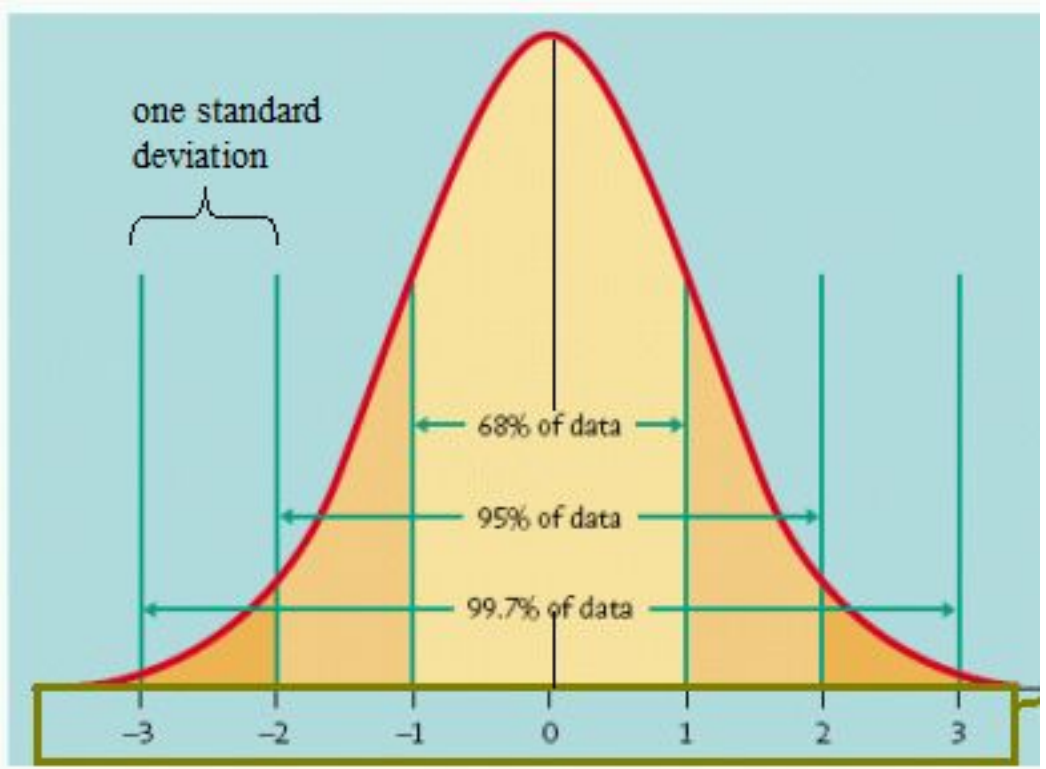
Source: Nicole Sullivan (Yahoo! Inc.), *Design Fast Websites*, http://www.slideshare.net/stubbornella/designing-fast-websites-presentation

*

Simplified (& false) view of performance

•For *standard normal distribution* of response times around mean: ±2 standard deviations around mean is 95% confidence interval

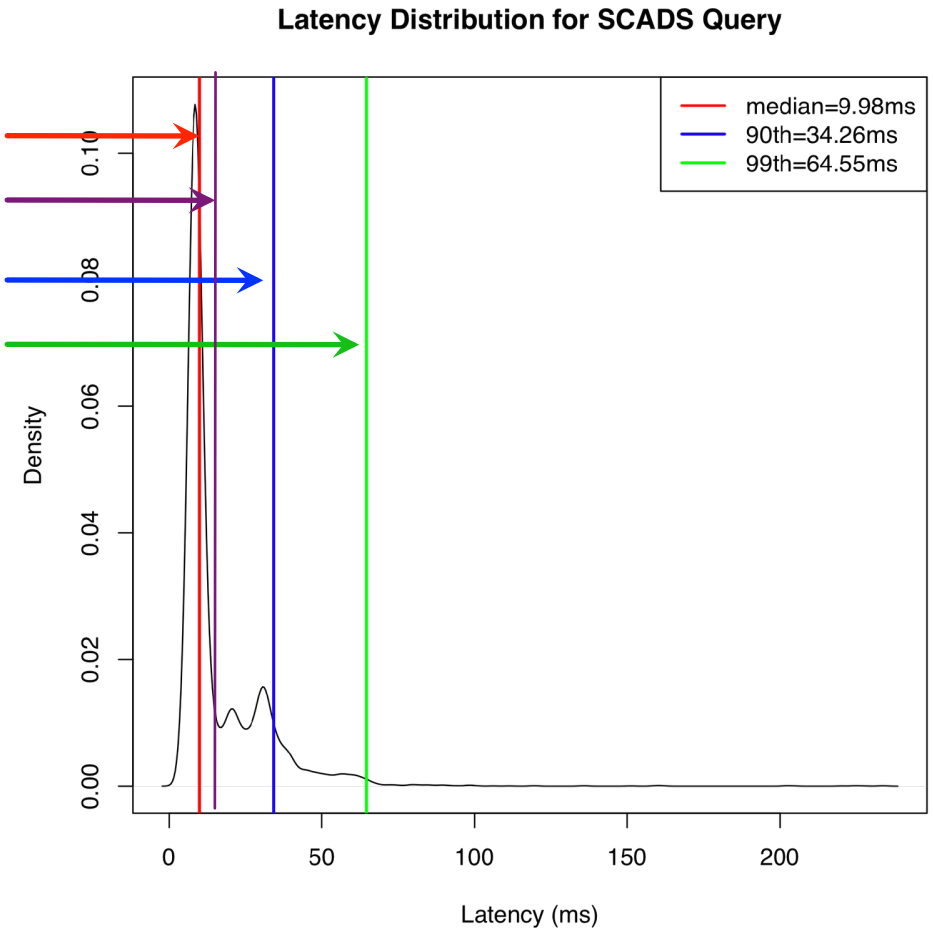
- *Average* response time T means:
- 95%ile users are getting T+2σ
- 99.7% users get T+3σ



Mean & Median vs. 99th %ile

Median 10ms
Mean 17ms
90% 34ms
99% 65ms

99%ile is ~6x worse than median!



*

Service Level Objective (SLO)

- Time to satisfy user request
("latency" or "response time")
- SLO: Instead of worst case or average: what % of users get acceptable performance
- Specify %ile, target response time, time window
  - e.g., 99% < 1 sec, over a 5 minute window
- why is time window important?
- Service level *agreement* (SLA) is an SLO to which provider is contractually obligated

Apdex: simplified SLO

- Given a threshold latency *T* for user satisfaction:
- *Satisfactory* requests take t≤*T*
- *Tolerable* requests take T≤ t ≤ 4T
- Apdex = (#satisfactory + 0.5(#tolerable)) / #reqs

What to do if site is slow?

- Small site: overprovision
- applies to presentation & logic tier
- before cloud computing, this was painful
- today, it's largely automatic (e.g. Rightscale)
- Large site: worry
- Provision 1,000-computer site by 10% = 100 idle computers
- *Insight: same problems that push us out of PaaS-friendly tier are the ones that will dog us when larger!*

RottenPotatoes.com's target uptime is 99.9%. Yesterday there was a one hour outage. Which statement is true:

- Because of the outage, RottenPotatoes has no hope of meeting its uptime goal
- RottenPotatoes can still meet its uptime goal if there's no more outages this year
- If no users actually tried to get to the site during the outage, uptime wasn't hurt
- There isn't enough information to determine whether RottenPotatoes can meet its uptime goal

*

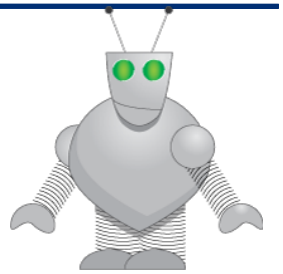
Releases Then and Now: Windows 95 Launch Party

Releases Then and Now

- Facebook: master branch pushed once a week, aiming for once a day (Bobby Johnson, Dir. of Eng., in late 2011)
- Amazon: several deploys per week
- StackOverflow: multiple deploys per day (Jeff Atwood, co-founder)
- *Rationale:* **risk** == # of engineer-hours invested in product since last deploy!

*Like development and feature check-in, deployment should be a **non-event** that happens all the time*

Successful Deployment



- **Automation:** consistent deploy process
- PaaS sites like Heroku, CloudFoundry already do this
- Tools like Capistrano support it for self-hosted sites
- **Continuous integration:** integration-testing the app beyond what each developer does
- Pre-release code checkin triggers CI
- Since frequent checkins, CI always running
- Common strategy: integrate with GitHub

https://github.com/saasbook/hw2_rottenpotatoes/

Why CI?

- Differences between dev & production envs
- Cross-browser or cross-version testing
- Testing SOA integration when remote services act wonky
- Hardening: protection against attacks
- Stress testing/longevity testing of new features/code paths
- Example: Salesforce CI runs 150K+ tests and automatically opens bug report when test fails

Continuous Deployment

- Check in => CI => deploy *several times per day*
- deploy may be auto-integrated with CI runs
- So are releases meaningless?
- Still useful as customer-visible milestones
- "Tag" specific commits with release names

```
git tag 'happy-hippo' HEAD
git push --tags
```

- Or just use Git commit ID to identify release

RottenPotatoes just got some new AJAX features. Where does it make sense to test these features?

- Using *autotest* with RSpec+Cucumber
- In CI
- In the staging environment
- All of these

*

The trouble with upgrades

- What if upgraded code is rolled out to many servers?
- During rollout, some will have version *n* and others version *n*+1…will that work?
- What if upgraded code goes with schema migration?
- Schema version *n*+1 breaks current code
- New code won't work with current schema

Naïve update

1. Take service offline
2. Apply destructive migration, including data copying
3. Deploy new code
4. Bring service back online

http://pastebin.com/5dj9k1cj

- May result in unacceptable downtime

Incremental Upgrades with Feature Flags

1. Do nondestructive migration

http://pastebin.com/TYx5qaSB

2. Deploy method protected by feature flag

http://pastebin.com/qqrLfuQh

3. Flip feature flag on; if disaster, flip it back
4. Once all records moved, deploy new code without feature flag
5. Apply migration to remove old columns

“Undoing” an upgrade

- Disaster strikes…use downmigration?
- is it thoroughly tested?
- is migration reversible?
- are you sure someone else didn’t apply an irreversible migration?
- Use feature flags instead
- downmigrations are primarily for *development*

Other uses for feature flags

- Preflight checking: gradual rollout of feature to increasing numbers of users
  - to scope for performance problems, e.g.
- A/B testing
- Complex feature whose code spans multiple deploys
- *rollout* gem (on GitHub) covers these cases and more

Which one, if any, is a POOR place to store the value (eg true/false) of a feature flag?

- A YAML file in config/ directory of app
- A column in an existing database table
- A separate database table
- These are all good places to store feature-flag values

Kinds of monitoring

- "If you're not monitoring it, it's probably broken"
- At development time (*profiling*)
- Identify possible performance/stability problems *before* they get to production
- In production
- Internal: instrumentation embedded in app and/or framework (Rails, Rack, etc.)
- External: active probing by other site(s).

Why use external monitoring?

Detect if site is down

Detect if site is slow for reasons outside measurement boundary of internal monitoring

Get user's view from many different places on the Internet

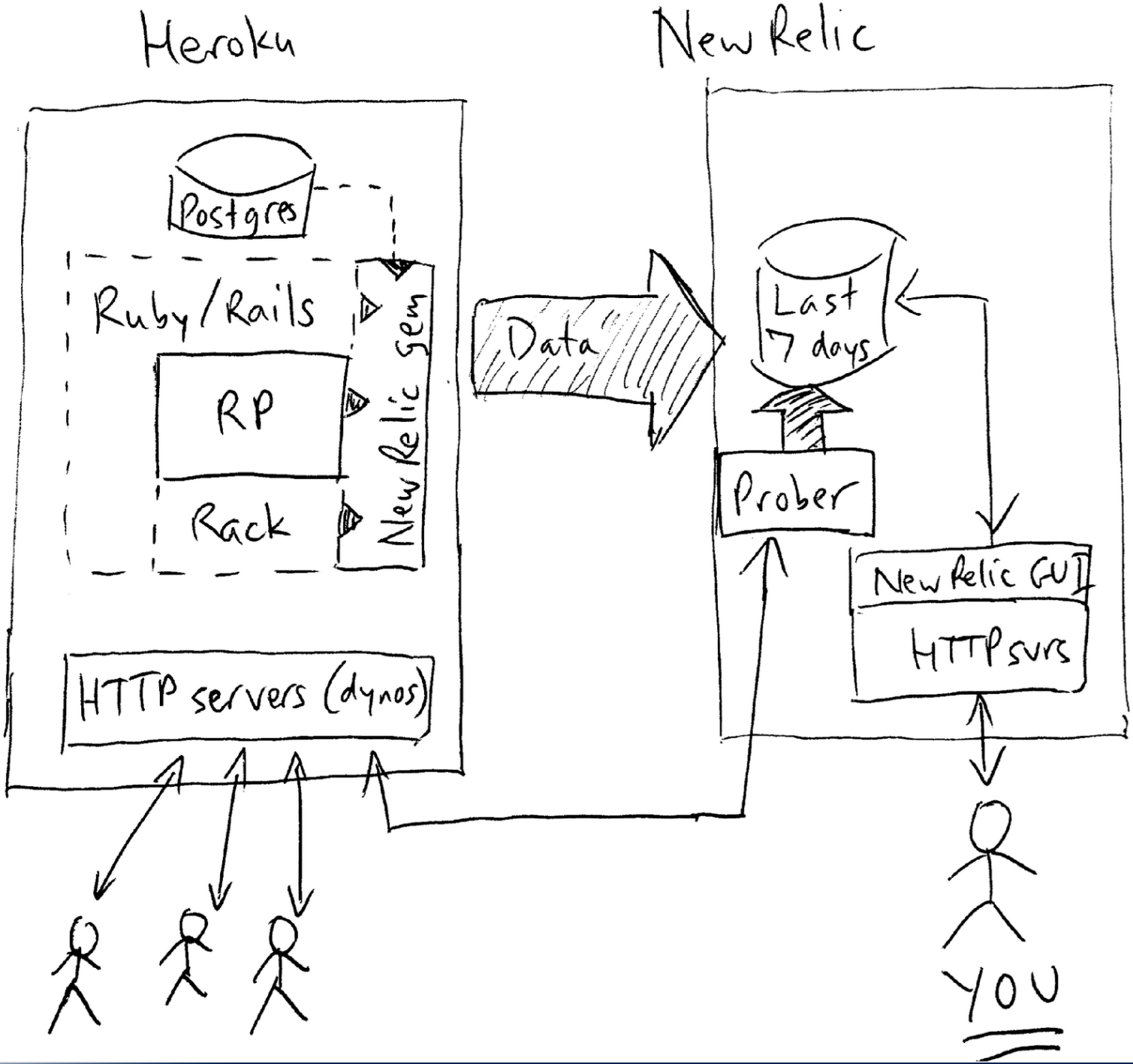
Example: Pingdom

Internal monitoring

- pre-SaaS/PaaS: *local*

http:/
monitor.millennium.berkeley.edu/

- Info collected & stored locally, eg Nagios
- Today: *hosted*
- Info collected in your app but stored centrally
- Info available even when app is down
- Example: New Relic
- conveniently, has both a development mode and production mode
- basic level of service is free for Heroku apps


Heroku
New Relic
Postgres
Ruby/Rails
RP
Rack
New Relic gem
HTTP servers (dynos)
Data
Last 7 days
Prober
New Relic GUI
HTTP svrs
YOU

Sampling of monitoring tools

What is monitored	Level	Example tool	Hosted
Availability	site	pingdom.com	Yes
Unhandled exceptions	site	airbrake.com	Yes
Slow controller actions or DB queries	app	newrelic.com (also has dev mode)	Yes
Clicks, think times	app	Google Analytics	Yes
Process health & telemetry (MySQL server, Apache, etc.)	process	god, monit, nagios	No

- Interesting: Customer-readable monitoring features with cucumber-newrelic

http://pastebin.com/TaecHfND