COPENHAGEN
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2014

goto; conference

SCALING DURING HYPER-GROWTH

Oliver Nicholas
Uber Technologies, Inc

**AS IT TURNS OUT,
SCALING DURING HYPER-GROWTH IS MUCH
EASIER IF YOU MAKE SMART DECISIONS
BEFORE THE HYPER-GROWTH PART.**

WHO AM I?

Qualifications

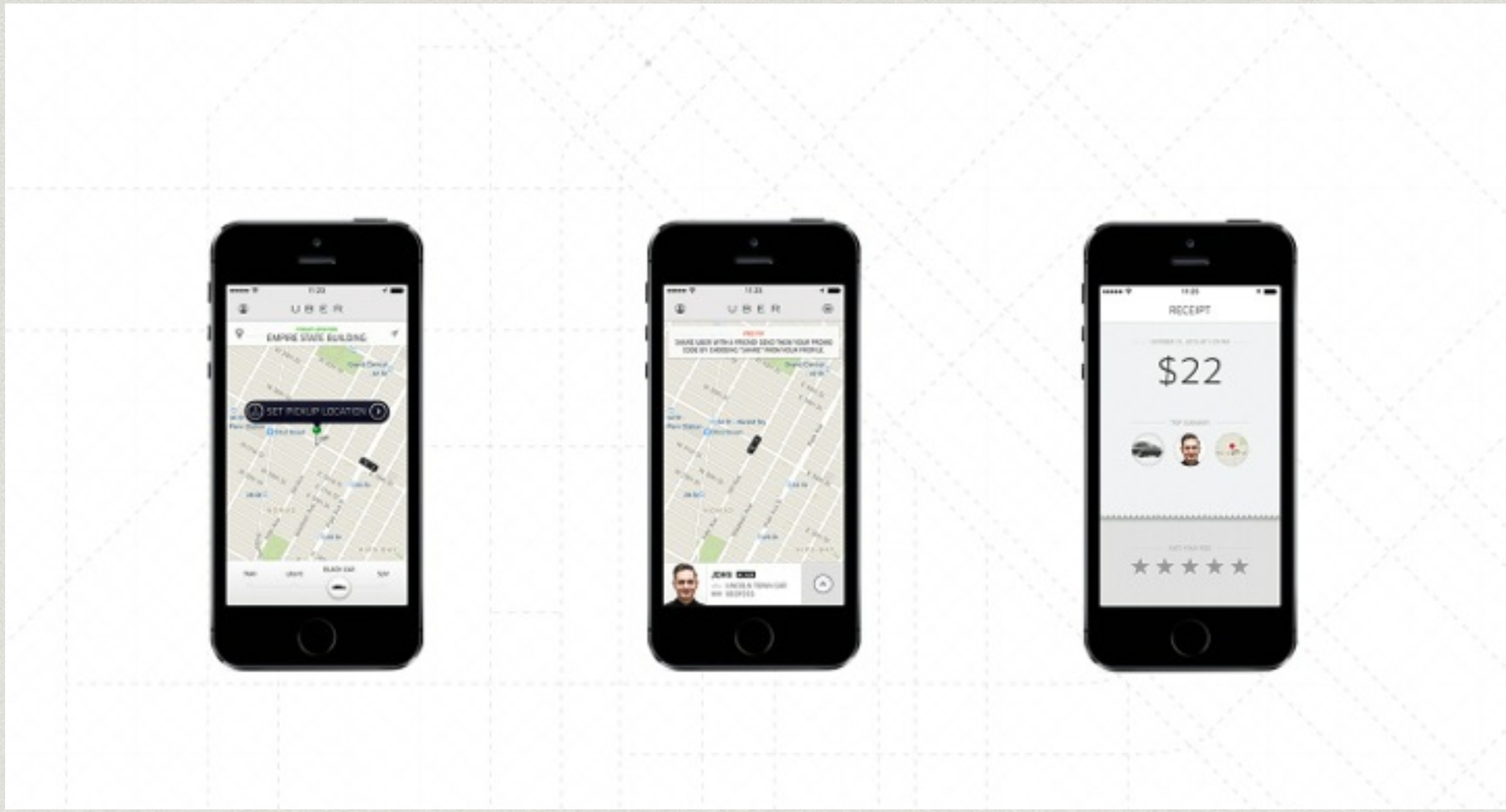
Engineering & Engineering Management positions at:

2006-2008: Ooga Labs
4x employee growth

2008-2012 Yelp
10x employee growth
10x traffic growth

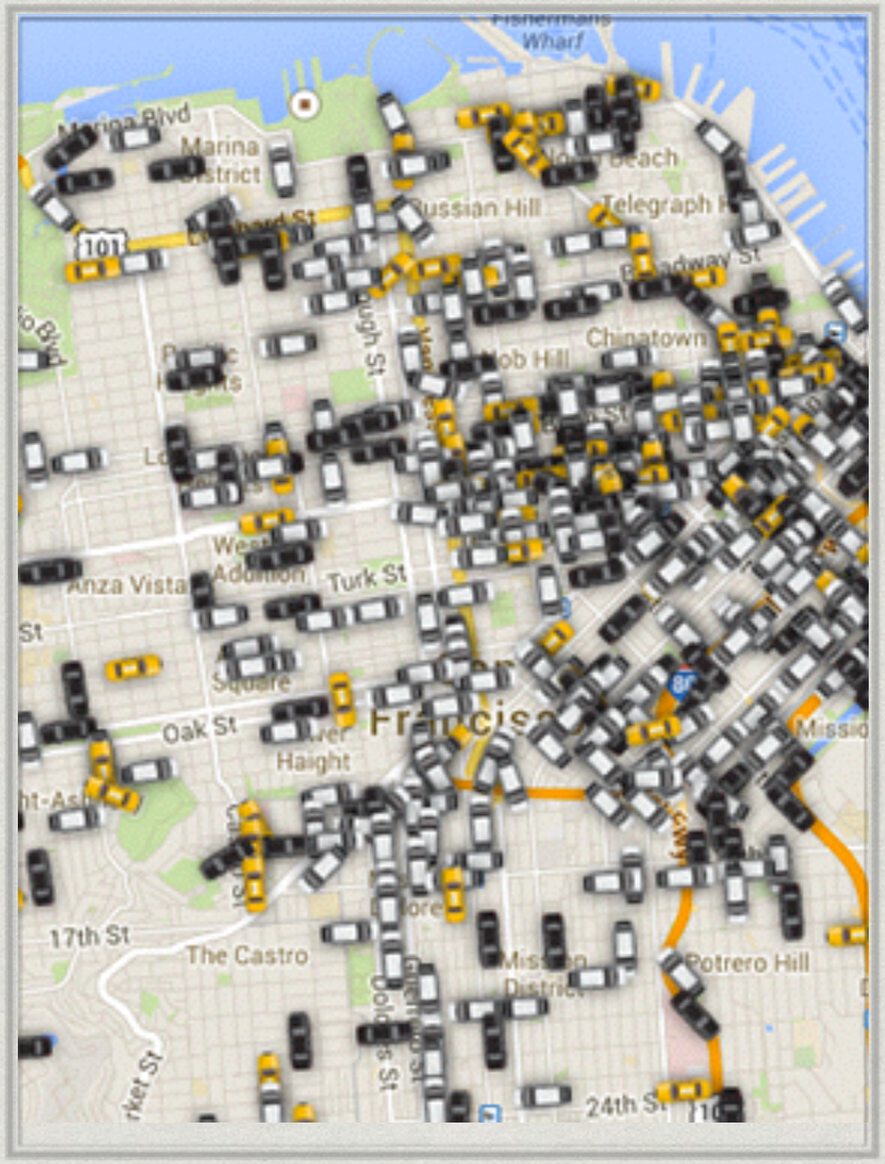
2012-Present Uber
15x employee growth
"a lot of" traffic growth

The Uber Experience



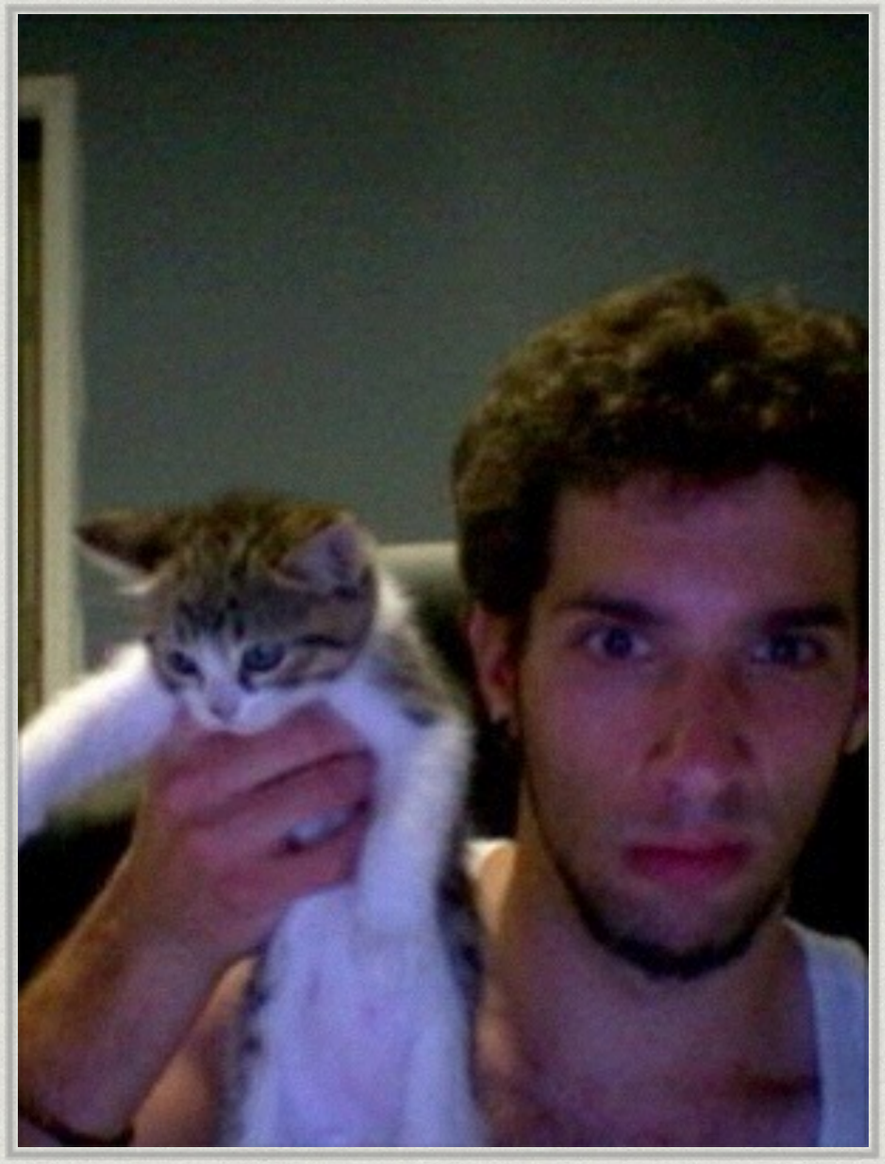
Uber?

- Founded in 2009 in San Francisco
- First cars on the roads in Summer 2010
- $1.4B+ funding
- Majority of employees are working locally within their city
- Aarhus R&D site opened January 2014 - Hiring Software Engineers!!



Anti-Qualifications

- 2.6 GPA in High School
- Only 8 years of post-university professional experience.
- Only 3 years management experience
- Haven't read a full book this calendar year



YOU NEED A WIKI

- Doesn't matter where it's hosted, but you may as well run it yourself.
- Make sure you take backups.
- Make sure it's accessible even when your main infrastructure is down.
- Document processes that should be followed and lessons learned

PEOPLE

SUPER IMPORTANT

People

- At many points, *people will be your primary constraint.*
- Roughly speaking, you want nerdy, intelligent, humble "A Players" who don't mind rolling up their sleeves. You want *grit*.

Hire Generalists

- Generalists are dynamic; they can solve any problem that arises without the need to make yet another hire.
- Of course, you shouldn't hire a generalist to design a nuclear reactor.

Hire "A Players"

- A Players are those who will get involved and be passionate about the project; they are dedicated.
- "A Players hire A Players. B Players hire C and D Players."

Have a People Person

- Not a People Person? Get one.

DESIGN FOR FAILURE

YOU'LL BE A HAPPIER PERSON.

Design for Failure

- Your datacenter will fail you.
- Your servers will fail you.
- Your database will fail you.
- Your own code will fail you.
- Your people will fail you.
- This is all becoming *more* true over time.

Expect Failure

"*If you want to make God laugh, tell him about your plans.*"

- In the old days, hardware was supposedly stable. And it *still* failed sometimes.
- We've been trying to internalize this for over 25 years - RAID was named in 1987.
- We now have permission to *expect* failure, and that makes all the difference.

Tests won't save you

- A good testing and staging regime will significantly reduce development time. They will also catch some bugs.
- They will not prevent failures.

FAILURE-RESILIENT ARCHITECTURE

EXPECT THE UNEXPECTED

Design for Failure

At the datacenter level:

- Redundant power feeds.
- Redundant Internet connections.
- Redundant datacenters (or AZ's)! Far enough apart that a small meteor strike won't disable all of them.

Design for Failure

At the server level:

- Disks must be RAIDed.
- Master databases must have slaves and you must practice quick promotion.
- Master/slave pairs or entire groups of app servers must be rack-distributed to withstand top-of-rack switch or power failure.

Design for Failure

At the service level:

- Processes should be stateless where possible.
- Calls to unreliable systems *and all 3rd parties* should have reasonable timeouts.
- Functionality should gracefully degrade if a particular sub-service is inaccessible.

Service Oriented Architecture

- Smaller codebases.
- Smaller deployable units.
- Better-defined areas of responsibility.
- But don't forget a good code search tool!

Service Isolation

- Where possible, isolate your services from failures in each other.
- Be willing treat the service boundary the same as you would any other network boundary - expect failure.
- Separate critical functions into their own services, and accept different deploy schedules or processes for them.

SITUATIONAL AWARENESS

YOU CAN'T REACT IF YOU DON'T KNOW WHAT'S FAILED

What Failure?

- Even with good design decisions, you will experience failures and regressions.
- You need to know when something has failed.
- Not all failures are obvious! An endpoint going from 100ms to 10,000ms is as good as a failure in most circumstances, but HTTP 200 status codes might mask it.

Track Everything

- Requires infrastructure investment since this is by nature a high-volume activity.
- Done correctly, helps you move faster.

Log Everything

- Log Everything*.
  - *Don't ever log HTTP response bodies or POST request bodies.
- Aggregate your logs, don't make people intrude on sensitive production infrastructure for them.
- Make recent logs easily searchable!
- Put them all on S3 and run MapReduce jobs on them!

Graph Everything

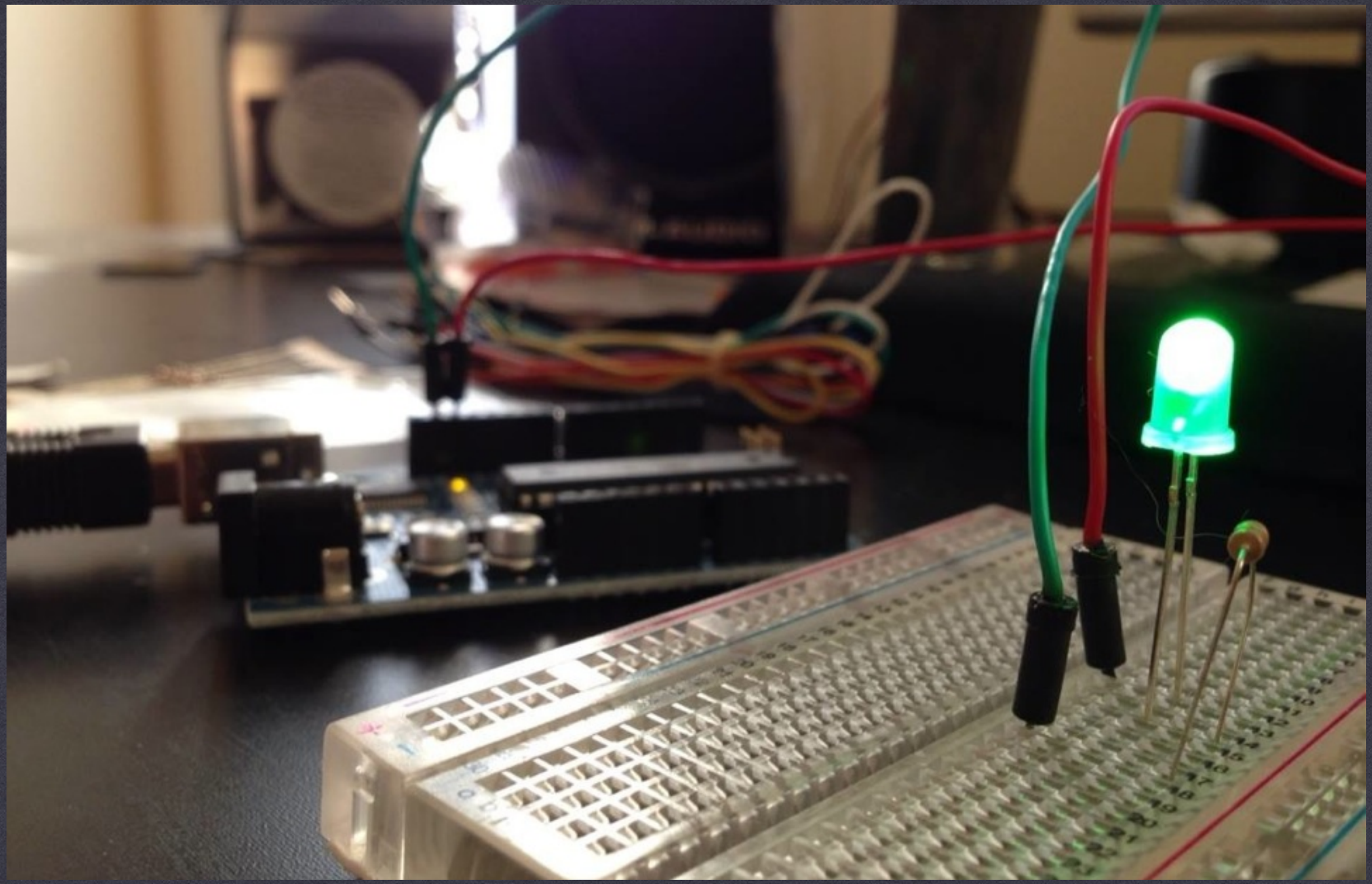
- Emit stats about your application into your time-series / graphing system.
- Do this in a separate thread of execution, or from your logs.
- Link your useful dashboards on your wiki.

Monitor Everything

- The foundation of rapid iteration is:
  - Automated, comprehensive testing.
  - Knowing if something has gone wrong.
- Relentlessly add monitoring and tune your thresholds.
- Use PagerDuty.
- Monitor PagerDuty.
- Write post-mortems. Put them on your wiki. Use them to add more monitoring.

Track Everything

- The beauty of Track Everything is that once you take the time to implement it, it lets you move faster.
- Less afraid to deploy.
- Less need for dedicated QA.



FOUNDATIONAL TECHNOLOGY CHOICES

TAKE RISKS ONLY WHERE IT MAKES SENSE

Use Boring Technology

- Old [active] software might be boring, but:
  - The bugs have already been worked out.
  - There are lots of experienced people to hire.

All Boring Software?

- A good rule of thumb: Don't take risks where you don't intend to innovate.
- If you have to get experimental, ensure what you choose has a very active developer community.
- Just don't take risks with your primary datastore.

Right tool for the job

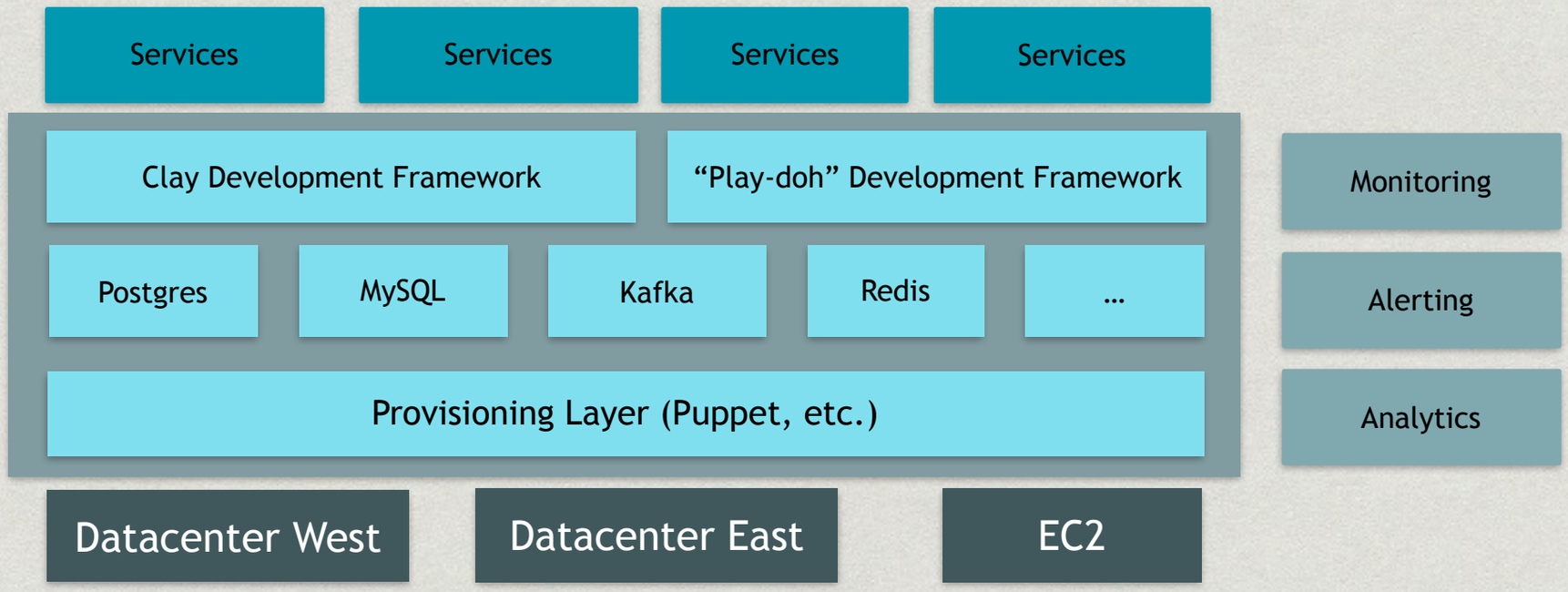
- As early as possible, separate OLTP from batch/ offline workloads.
- If you don't need transactions, you may not need a relational database at all.
- If you can afford to lose some data, consider storing it in-memory.

Be Secure

- Security is *hard*, dealing with lapses is harder; wastes political capital and lots of time.
- Encrypt your offsite database backups.
- Encrypt/obfuscate PKs shown to the front-end.
- Have a proper RBAC framework, even if the first implementation is naive.
- Centralize auth so it's harder to mess up.
- Don't roll your own crypto.

Create frameworks

- Create frameworks that encapsulate your best practices.
- "Strongly suggest" that engineers build new projects on these frameworks.



What are we doing?

- You need a wiki.
- Hire gritty polymaths.
- Pick boring technology where you don't *really* require innovation; resist the urge to try everything new.
- Design systems with the expectation that nearly every downstream system might fail or be slow.
- Reduce statefulness, and ensure good redundancy and failover for stateful systems.
- Track everything.
- Stay secure.
- Encode all these best practices into a framework for your developers.