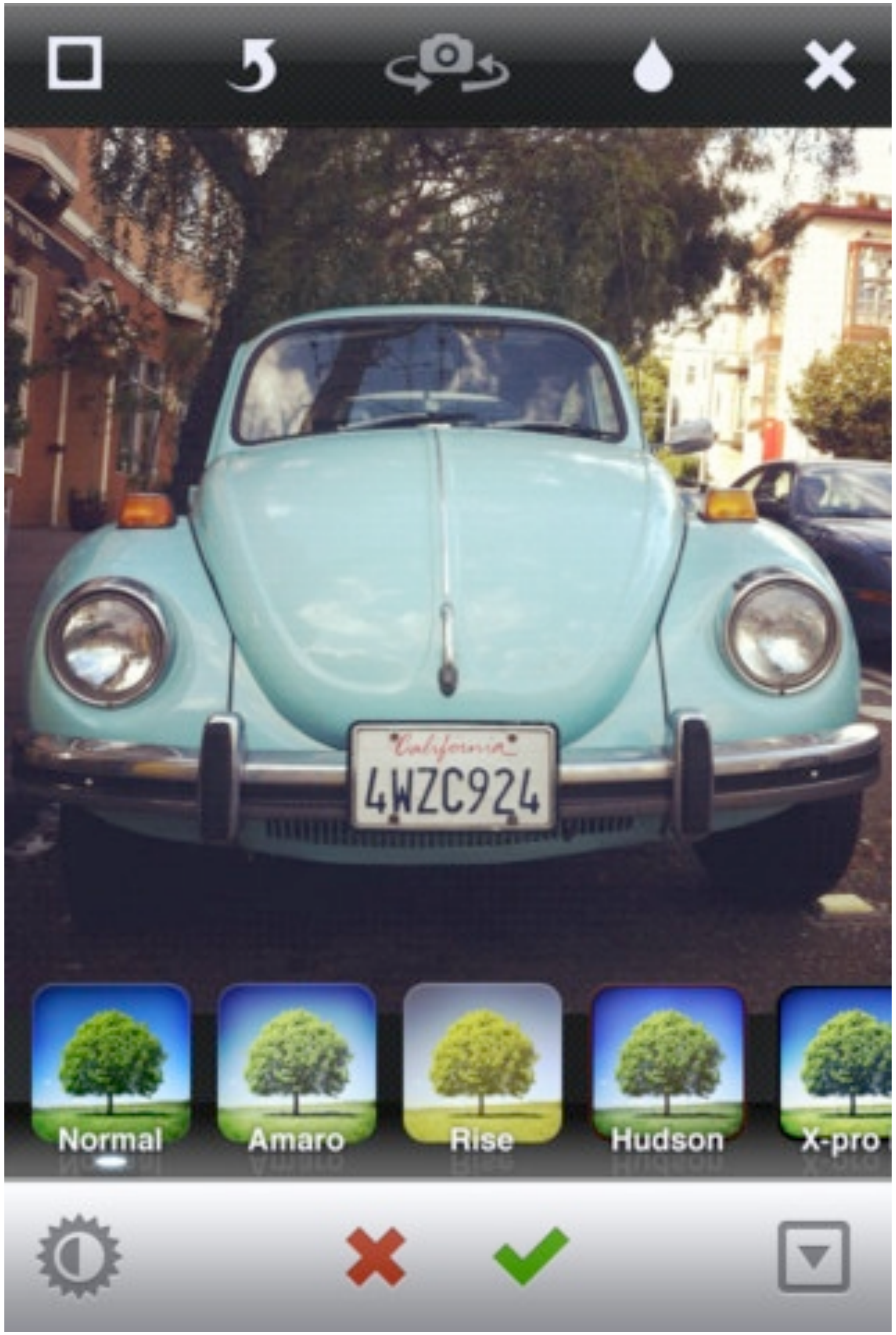
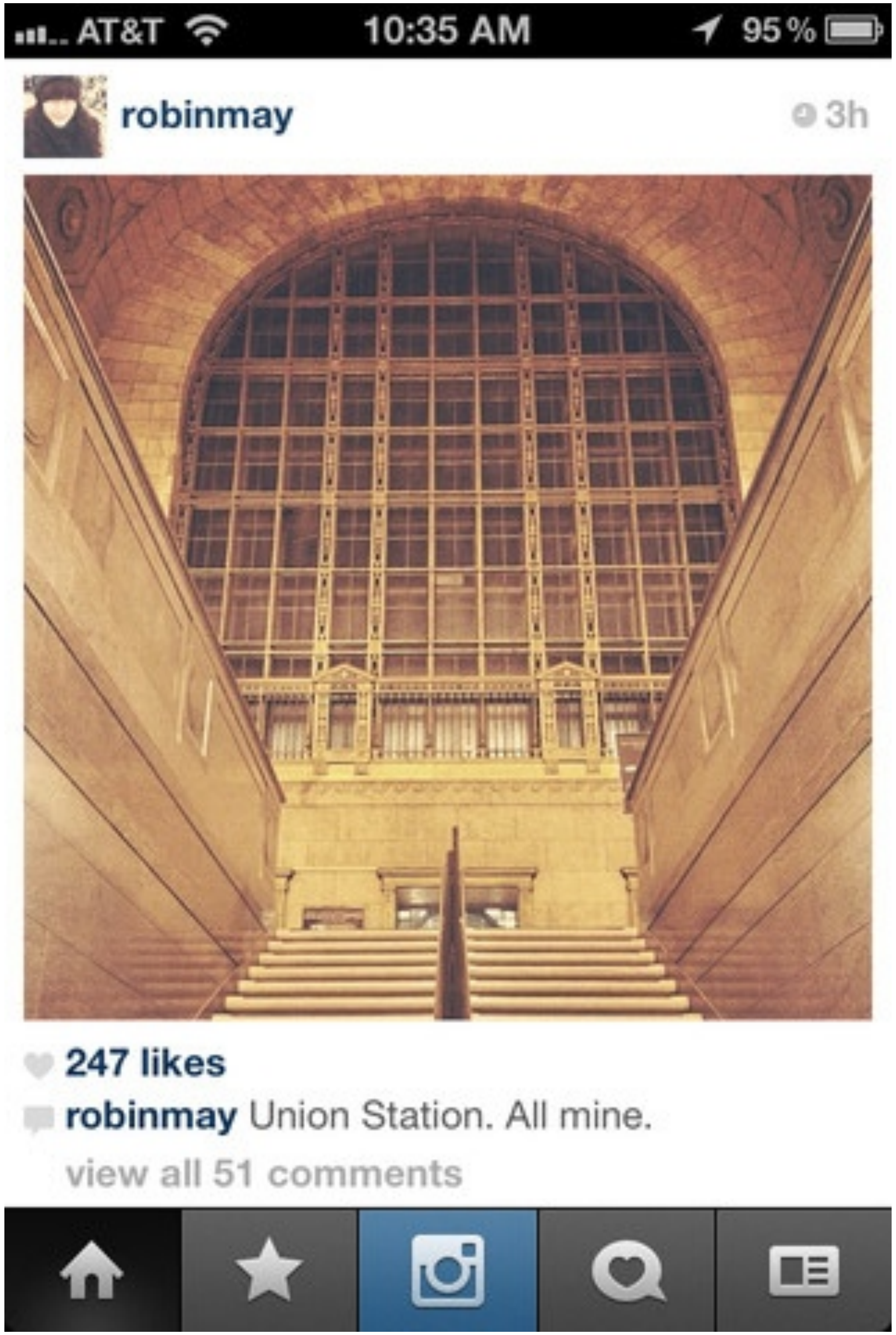
Scaling Instagram

AirBnB Tech Talk 2012
Mike Krieger
Instagram

me

- Co-founder, Instagram
- Previously: UX & Front-end @ Meebo
- Stanford HCI BS/MS
- @mikeyk on everything


California
4WZC924
Normal
Amaro
Rise
Hudson
X-pro

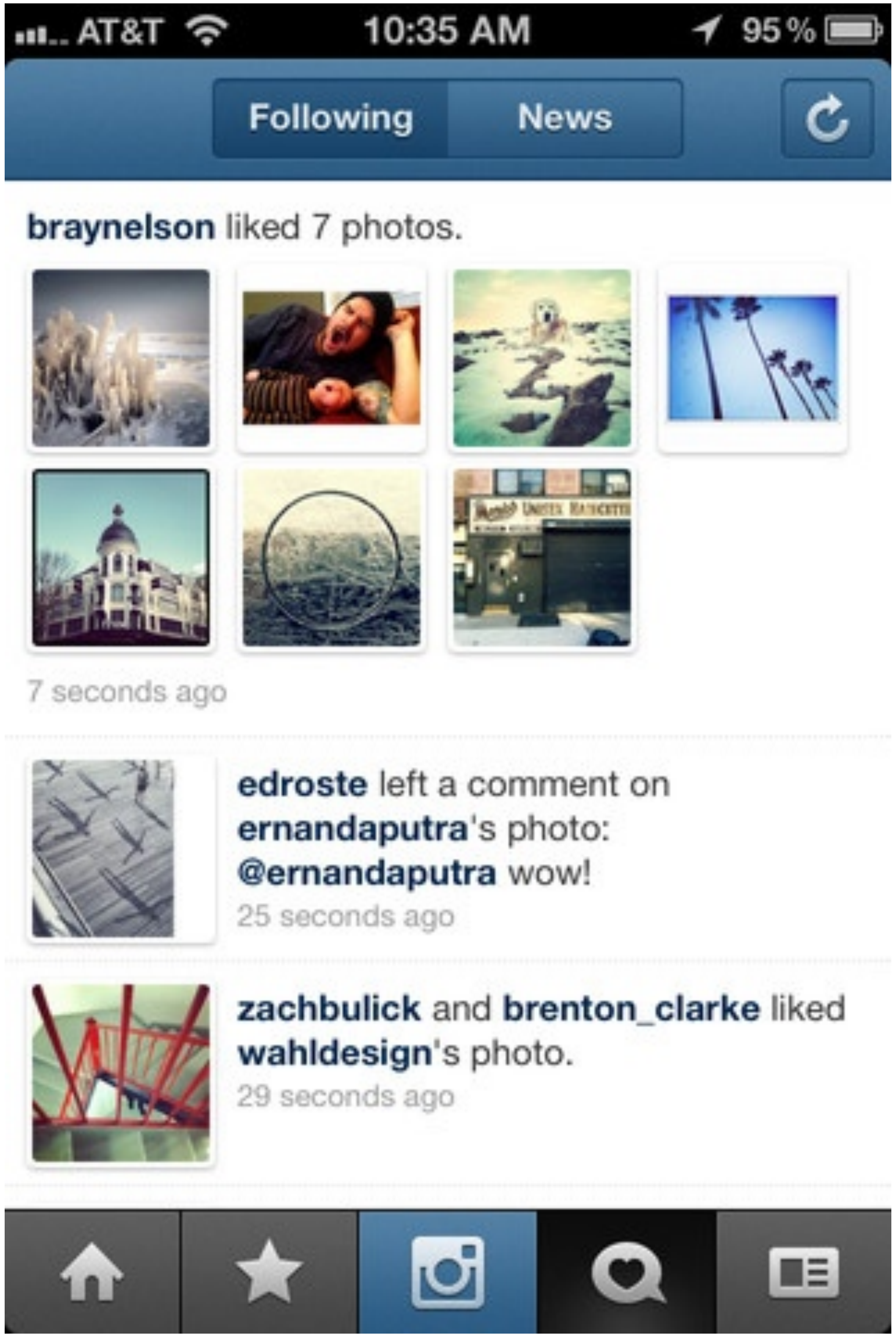


robinmay

247 likes

robinmay Union Station. All mine.

view all 51 comments



braynelson liked 7 photos.

7 seconds ago

edroste left a comment on **ernandaputra**'s photo: **@ernandaputra** wow!

25 seconds ago

zachbulick and **brenton_clarke** liked **wahldesign**'s photo.

29 seconds ago

communicating and
sharing in the real world

30+ million users in less than 2 years

the story of how we scaled it

a brief tangent

the beginning

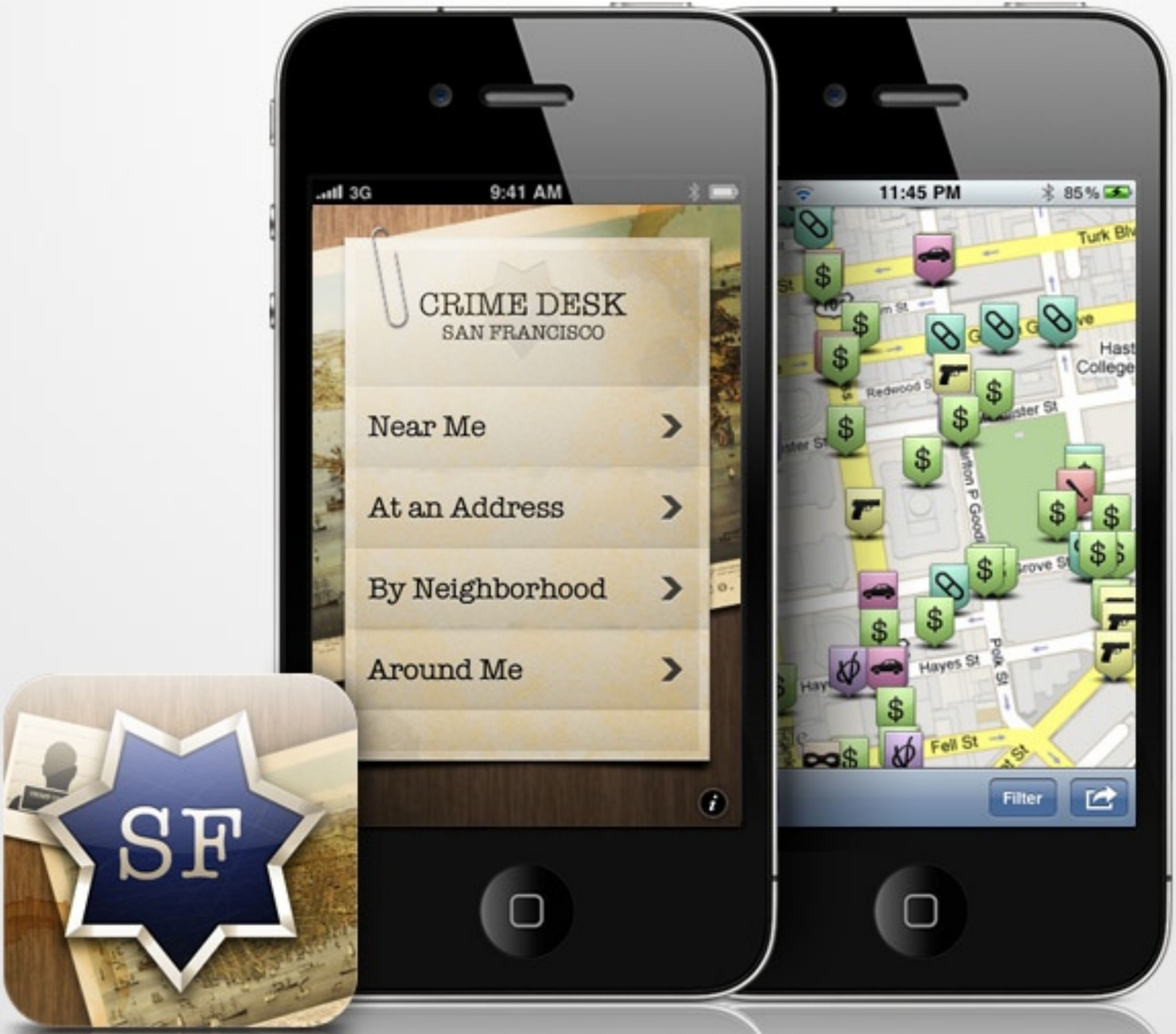
2 product guys

no real back-end experience

analytics & python @ meebo

CouchDB

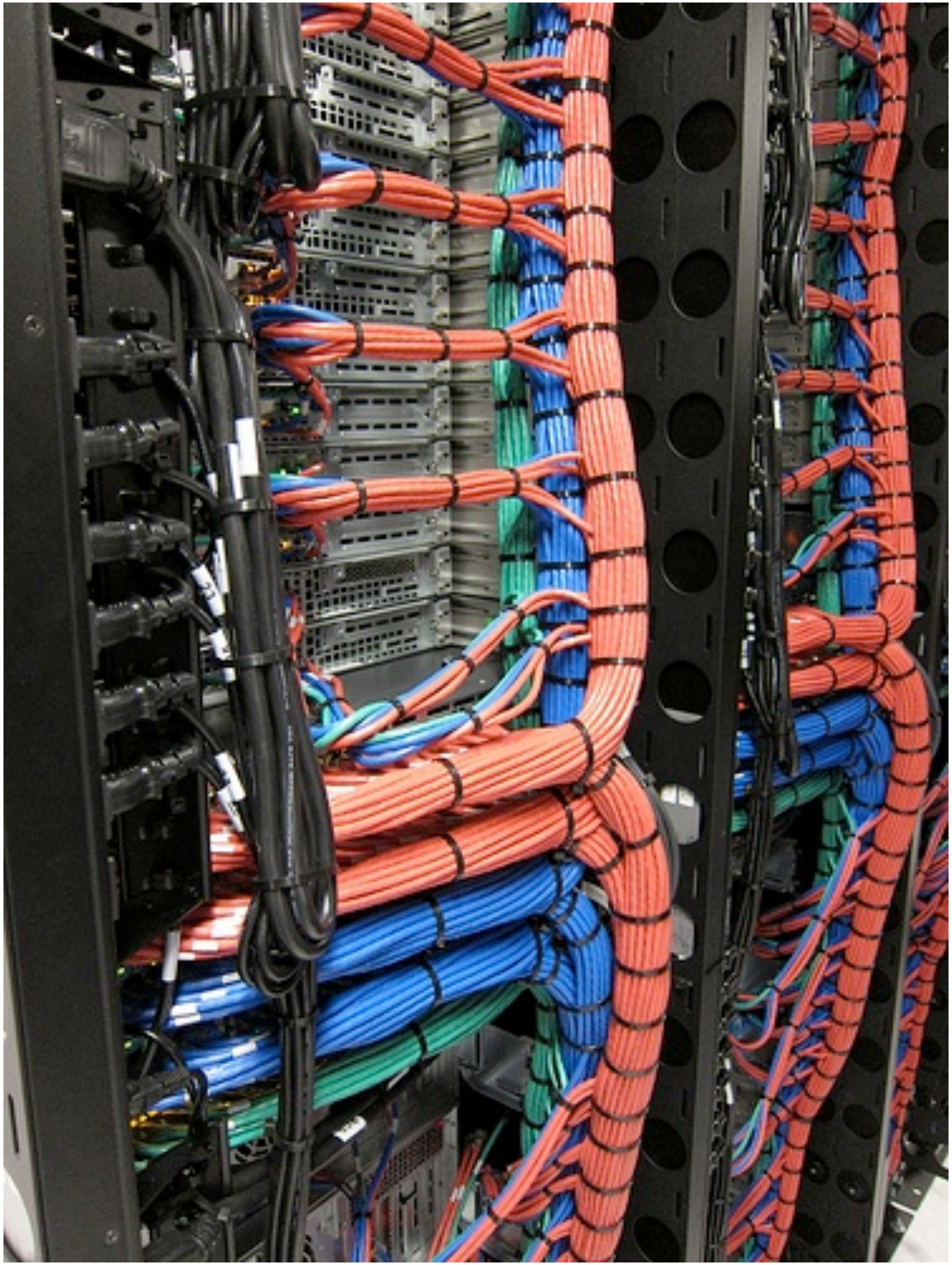
CrimeDesk SF


3G
9:41 AM
CRIME DESK
SAN FRANCISCO
Near Me
At an Address
By Neighborhood
Around Me
SF
11:45 PM
85 %
Turk Blv
Hast
College
Redwood St
Hayes St
Polk St
Fell St
Grove St
Filter

let’s get hacking

good components in place early on

…but were hosted on a single machine somewhere in LA



less powerful than my MacBook Pro

okay, we launched.
now what?

25k signups in the first day

everything is on fire!

best & worst day of our
lives so far

load was through the roof

first culprit?

favicon.ico

404-ing on Django,
causing tons of errors

lesson #1 : don't forget your favicon

real lesson #1: most of your initial scaling problems won't be glamorous

favicon

```
ulimit -n
```

```
memcached -t 4
```

prefork/postfork

friday rolls around

not slowing down

let’s move to EC2.

Taurin- und koffeinhaltiges Spezialgetränk. Kohlensäurehaltig.
Boisson à base de taurine et caféine. Boisson gazéifiée.
Red Bull
ENERGY DRINK

scaling = replacing all components of a car while driving it at 100mph since…

“"canonical [architecture] of an early stage startup in this era."

(HighScalability.com)

Nginx &
Redis &
Postgres &
Django.

Nginx & HAProxy & Redis & Memcached & Postgres & Gearman & Django.

24h Ops

Airbnb.com
Verified Photo

Airbnb.com
Verified Photo

our philosophy

1 simplicity

2 optimize for minimal operational burden

3 instrument *everything*

walkthrough:

1 scaling the database
2 choosing technology
3 staying nimble
4 scaling for android

1 scaling the db

early days

django ORM, postgresql

why pg? postgis.

moved db to its own machine

but photos kept growing
and growing…

…and only 68GB of RAM on biggest machine in EC2 so what now?

vertical partitioning

django db routers make it pretty easy

```
def db_for_read(self, model):
  if app_label == 'photos':
    return 'photodb'
```

...once you untangle all your foreign key relationships a few months later…

photosdb > 60GB

what now?

horizontal partitioning!

aka: sharding

“surely we’ll have hired someone experienced before we actually need to shard” you don’t get to choose when scaling challenges come up evaluated solutions

at the time, none were up to task of being our primary DB did in Postgres itself

what's painful about sharding?

1 data retrieval

hard to know what your primary access patterns will be w/out any usage in most cases, user ID

2 what happens if one of your shards gets too big?

in range-based schemes (like MongoDB), you split

```
A-H: shard0
I-Z: shard1
```

```
A-D: shard0
E-H: shard2
I-P: shard1
Q-Z: shard2
```

downsides (especially on EC2): disk IO

instead, we pre-split

many many many (thousands) of logical shards that map to fewer physical ones

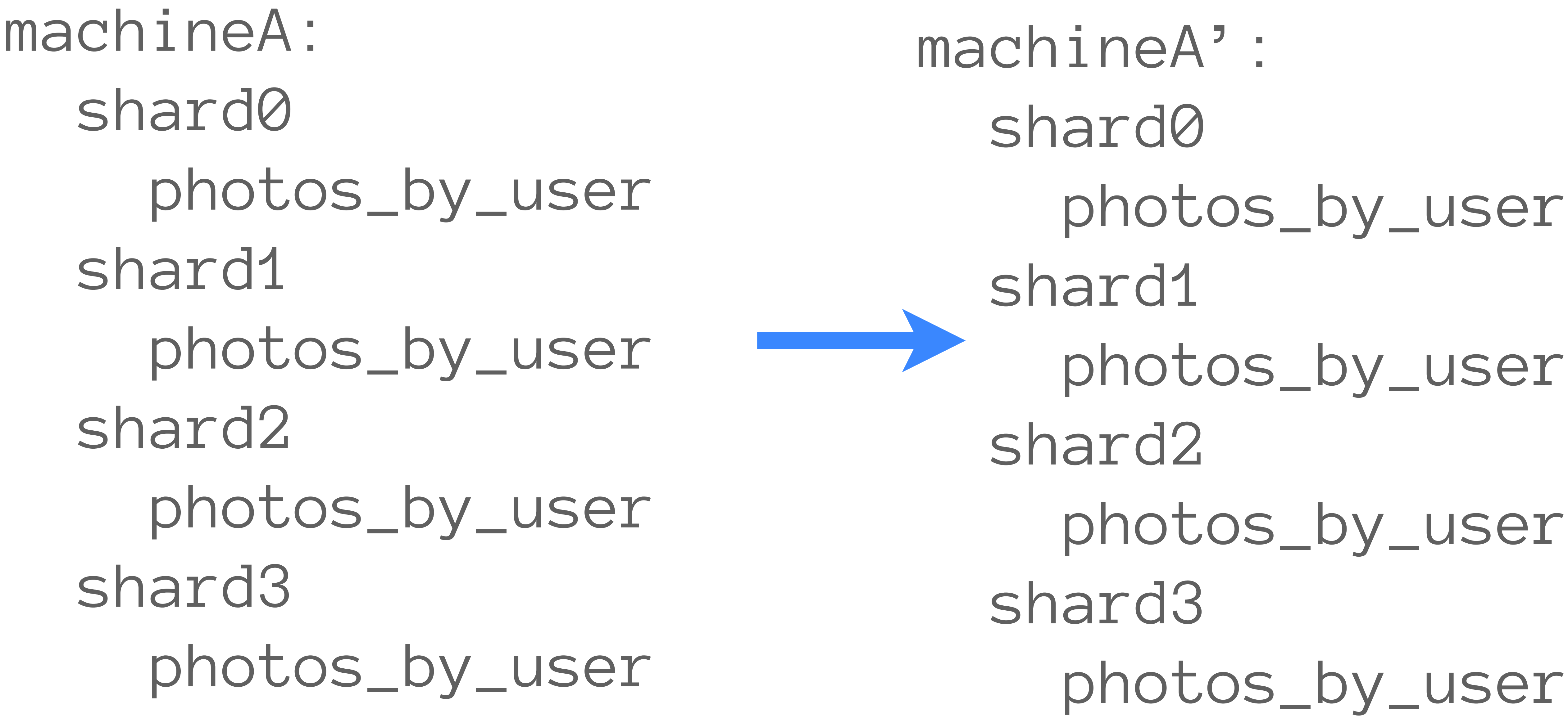
```
// 8 logical shards on 2 machines

user_id % 8 = logical shard

logical shards -> physical shard map

{
  0: A, 1: A,
  2: A, 3: A,
  4: B, 5: B,
  6: B, 7: B
}
```

```
// 8 logical shards on ~~2~~ 4 machines

user_id % 8 = logical shard

logical shards -> physical shard map

{
  0: A, 1: A,
  2: C, 3: C,
  4: B, 5: B,
  6: D, 7: D
}
```

little known but awesome PG feature: schemas not “columns” schema

```
- database:
  - schema:
    - table:
      - columns
```

```
machineA:
  shard0
    photos_by_user
  shard1
    photos_by_user
  shard2
    photos_by_user
  shard3
    photos_by_user
```



```
machineA:
  shard0
    photos_by_user
  shard1
    photos_by_user
  shard2
    photos_by_user
  shard3
    photos_by_user
```

```
machineA’:
  shard0
    photos_by_user
  shard1
    photos_by_user
  shard2
    photos_by_user
  shard3
    photos_by_user
```

```
machineA:
  shard0
    photos_by_user
  shard1
    photos_by_user
  ~~shard2~~
    ~~photos_by_user~~
  ~~shard3~~
    ~~photos_by_user~~
```

```
machineC:
  ~~shard0~~
    ~~photos_by_user~~
  ~~shard1~~
    ~~photos_by_user~~
  shard2
    photos_by_user
  shard3
    photos_by_user
```

can do this as long as you have more logical shards than physical ones lesson: take tech/tools you know and try first to adapt them into a simple solution

2 which tools where?

where to cache / otherwise denormalize data we <3 redis

what happens when a user posts a photo?

1 user uploads photo with (optional) caption and location

2 synchronous write to the media database for that user

3 queues!

3a if geotagged, async worker POSTs to Solr

3b follower delivery

can’t have every user
who loads her timeline
look up all their followers
and then their photos

instead, everyone gets their own list in Redis

media ID is pushed onto
a list for every person
who’s following this user

Redis is awesome for this; rapid insert, rapid subsets when time to render a feed, we take small # of IDs, go look up info in memcached

Redis is great for…

data structures that are relatively bounded

(don't tie yourself to a solution where your in-memory DB is your main data store) caching complex objects where you want to more than GET ex: counting, sub-ranges, testing membership

especially when Taylor Swift posts live from the CMAs

follow graph

v1 : simple DB table (source_id, target_id, status) who do I follow?

who follows me?

do I follow X?

does X follow me?

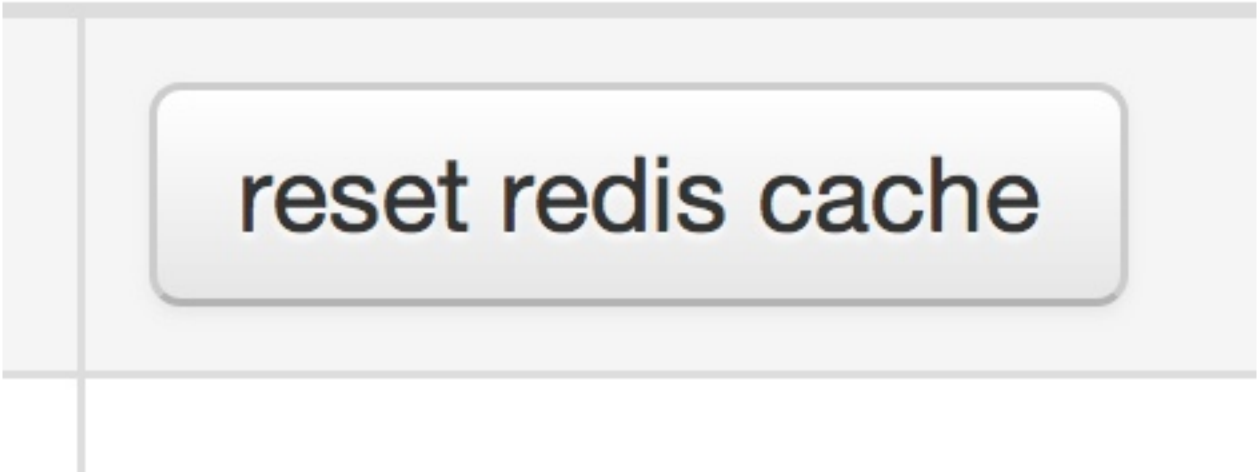
DB was busy, so we started storing parallel version in Redis follow_all(300 item list)

inconsistency

extra logic

so much extra logic

exposing your support team to the idea of cache invalidation


reset redis cache

redesign took a page
from twitter’s book

PG can handle tens of thousands of requests, very light memcached caching

two takeaways

1 have a versatile complement to your core data storage (like Redis)

2 try not to have two tools trying to do the same job

3 staying nimble

2010: 2 engineers

2011: 3 engineers

2012: 5 engineers

scarcity -> focus

engineer solutions that you’re not constantly returning to because they broke

1 extensive unit-tests and functional tests

2 keep it DRY

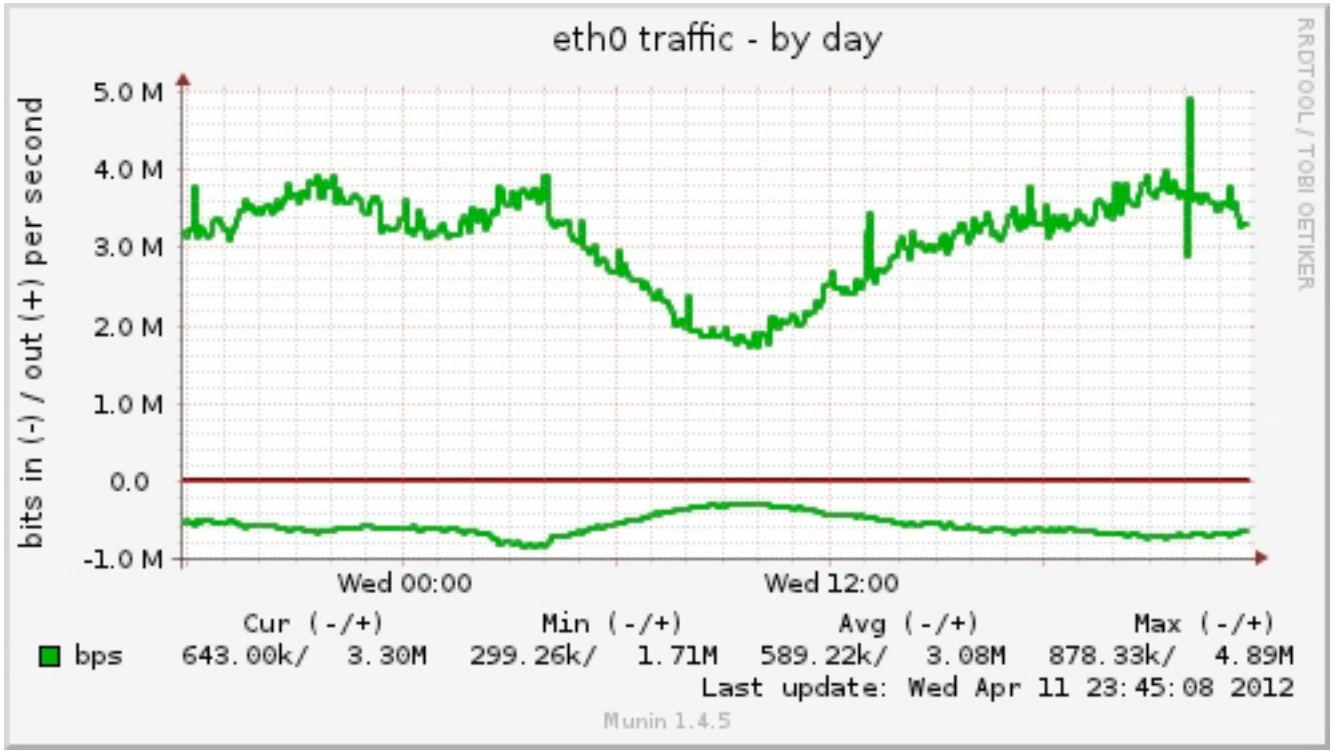
3 loose coupling using notifications / signals

4 do most of our work in Python, drop to C when necessary

5 frequent code reviews, pull requests to keep things in the ‘shared brain’

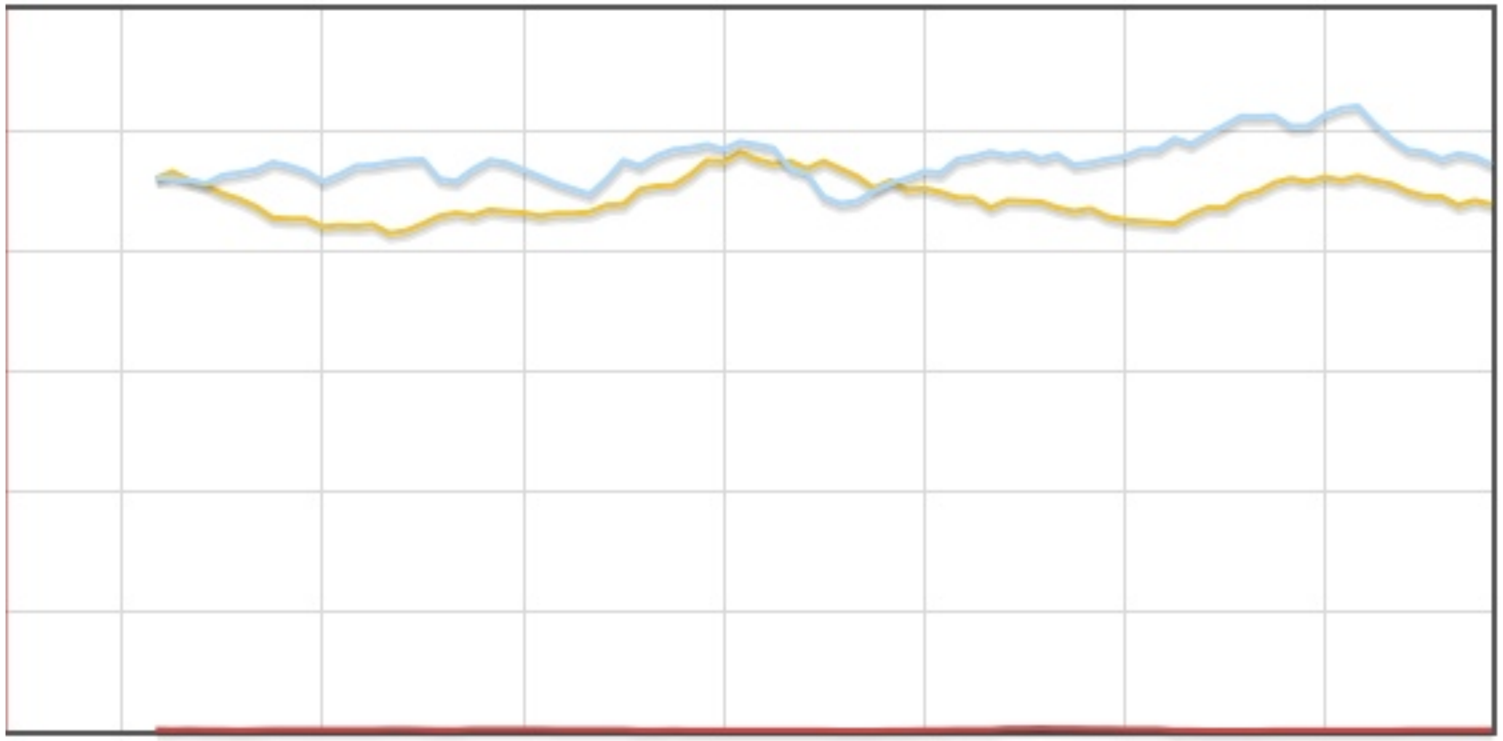
6 extensive monitoring

munin



```
          Cur (-/+)          Min (-/+)          Avg (-/+)          Max (-/+)
🟩 bps  643.00k/   3.30M  299.26k/   1.71M  589.22k/   3.08M  878.33k/   4.89M
                                      Last update:  Wed Apr 11 23:45:08 2012
```

statsd



PAGERDUTY

pingdom

“how is the system right now?”

“how does this compare
to historical trends?”

scaling for android

1 million new users in 12 hours

great tools that enable
easy read scalability

redis: slaveof <host> <port>

our Redis framework
assumes 0+ readslaves

tight iteration loops

statsd & pgfouine

know where you can
shed load if needed

(e.g. shorter feeds)

if you’re tempted to
reinvent the wheel…

don’t.

“our app servers sometimes kernel panic under load”

“what if we write a monitoring daemon…”

wait! this is exactly what HAProxy is great at surround yourself with awesome advisors

culture of openness
around engineering

give back; e.g.
node2dm

focus on making what
you have **better**

“**fast**, beautiful photo sharing”

“can we make all of our requests 50% the time?”

staying nimble = remind yourself of what's important your users around the world don't care that you wrote your own DB

wrapping up

unprecedented times

2 backend engineers can scale a system to 30+ million users

key word = **simplicity**

cleanest solution with the fewest moving parts as possible don't over-optimize or expect to know ahead of time how site will scale don’t think “someone else will join & take care of this” will happen sooner than you think; surround yourself with great advisors when adding software to stack: only if you have to, optimizing for operational simplicity few, if any, unsolvable scaling challenges for a social startup have fun