YELLOWPAGES.COM: Behind the Curtain

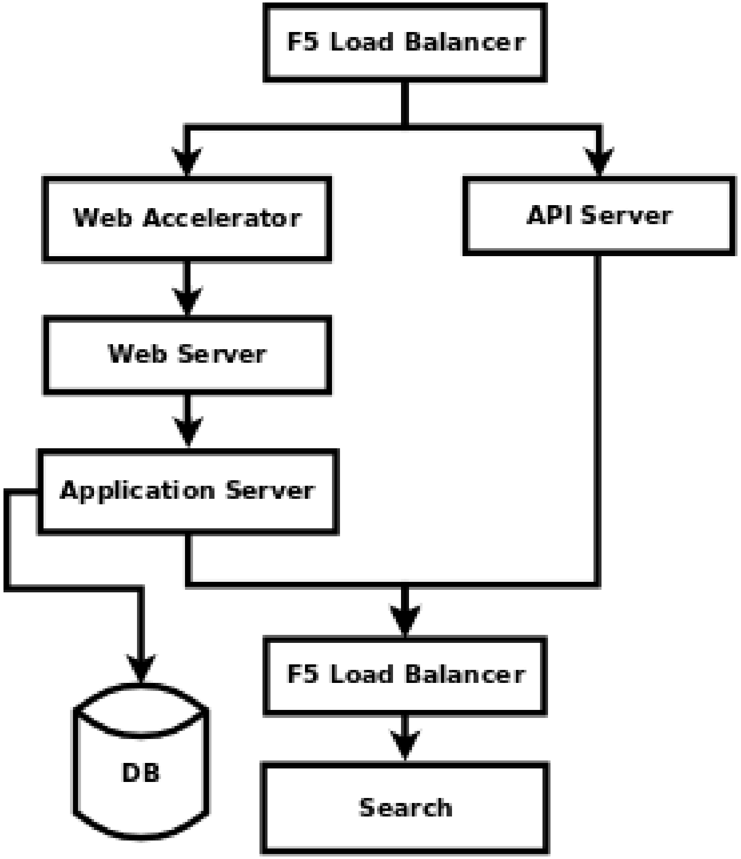
John Straw
AT&T Interactive

What is YELLOWPAGES.COM?

- Part of AT&T
- A local search website, serving
  - 23 million unique visitors / month
  - 2 million searches / day
  - More than 48 million requests / day
  - More than 1500 requests / sec
  - 30 Mbit/sec (200 Mbit/sec from Akamai)
- Entirely Ruby on Rails since June 2007

How we were

- Website and API applications written in Java
  - Website in application server
  - API in Tomcat
- Search code split between application and search layer



What was bad

- Problems with architecture
  - Separate search implementations in each application
  - Session-heavy website application design hard to scale horizontally
  - Pointless web accelerator layer
- Problems with application server platform
  - Technologically stagnant
  - No usable session migration features
  - Hard to do SEO

And also ...

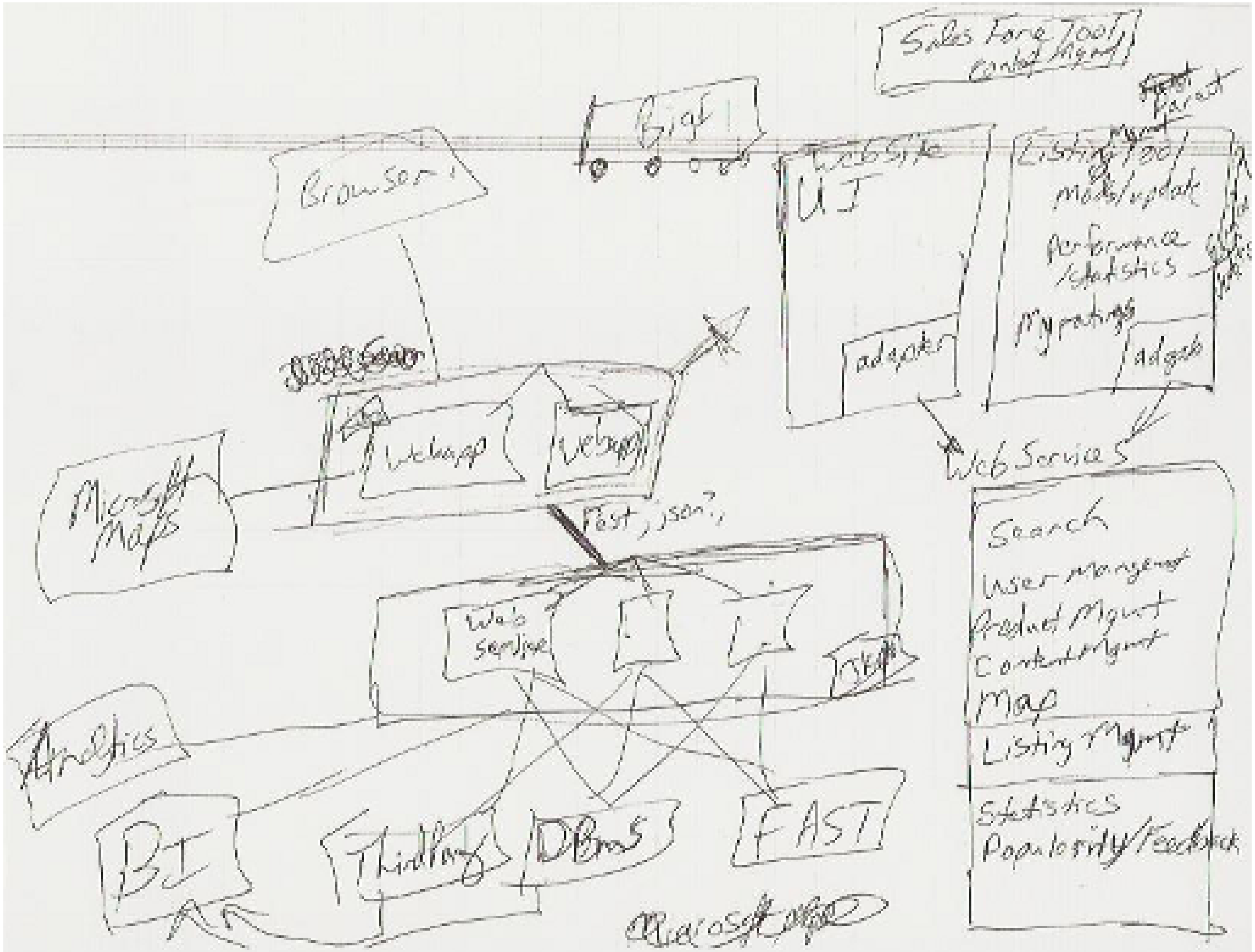
- Lots of code written by consultants 2004-2005
- Fundamental design problems
- Code extended largely by copy-and-modify since 11/2005 (to around 125K lines)
- No test code
- New features hard to implement

The Big Rewrite

- Several projects combined to become the big rewrite
  - Replacement of Java application server
  - Redesign of site look-and-feel
  - Many other wish-list projects, some of which were difficult to accomplish with existing application
- Project conception to completion: one year
- Development took about four months
- Project phases
  - **7/2006 - 12/2006:** Thinking, early preparation
  - **12/2006:** Rough architecture determination, kick-off
  - **1/2007 - 3/1/2007:** Technology research and prototypes, business rules exploration, UI design concepts
  - **3/1/2007 - 6/28/2007:** Site development and launch

Requirements for a new site architecture

1. Absolute control of urls
   - Maximize SEO crawl-ability
2. No sessions: HTTP is stateless
   - Anything other than cookies is just self-delusion
   - Staying stateless makes scaling easier
3. Be agile: write less code
   - Development must be faster
4. Develop easy-to-leverage core business services
   - Eliminate current duplicated code
   - Must be able to build other sites or applications with common search, personalization and business review logic
   - Service-oriented architecture, with web tier utilizing common service tier

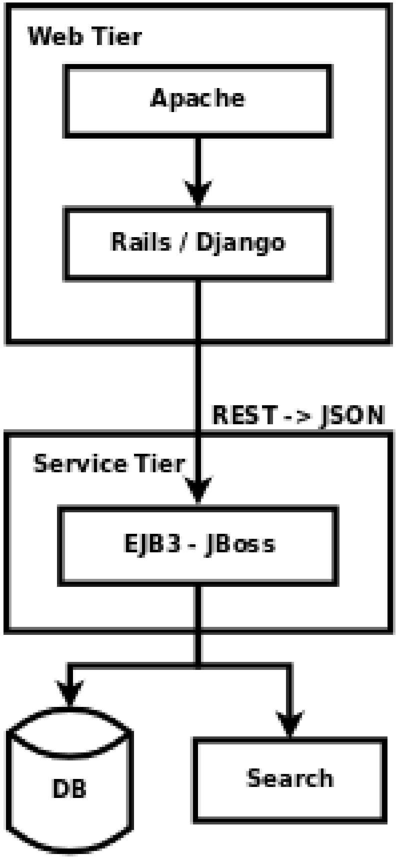

Sales Force Tool / Contract Mgmt
Bigf!
Target
Browser!
WebSite
UI
adapter
Listing Tool / mods/update
Performance /statistics
My ratings
adapter
Web Service
Search
User Mangement
Product Mgmt
Content Mgmt
Map
Listing Mgmt
Statistics
Popularity/Feedback
Webapp
Webapp
Fast, json?,
Microsoft Maps
Web service
Analytics
BI
Thirdparty
DBms
FAST

Rewrite team

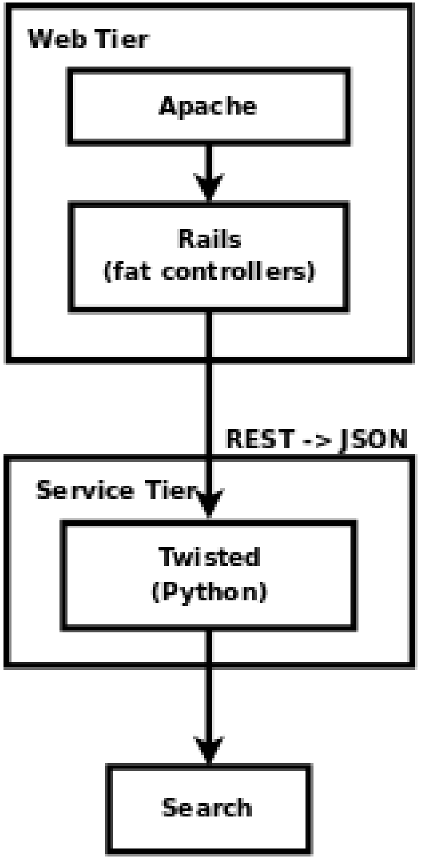
- Cross-functional team of about 20 people
  - Assemble stakeholders, not requirements
- Working closely together
  - Whole team sat together for entire project
  - Lunch provided several days per week
  - Team celebrations held for milestones
- Core development team deliberately small
  - Four skilled developers can accomplish a lot
  - Cost of communication low
  - Low management overhead

Picking the platform

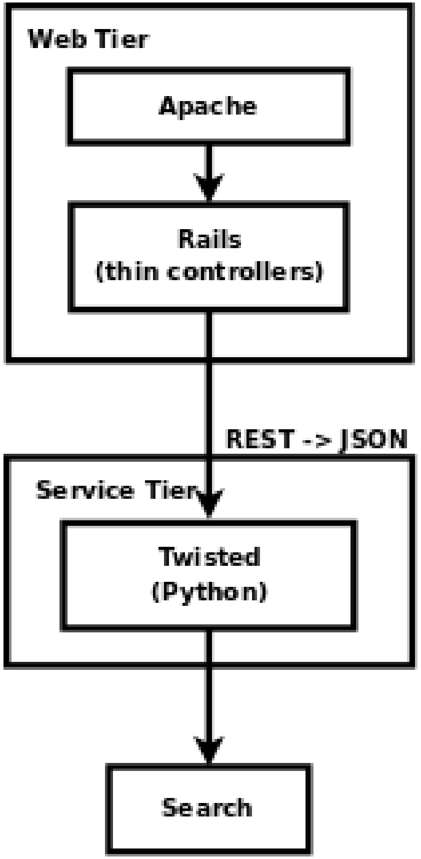
- Web tier:
  - Rails or Django
  - Utilizing common services for search, personalization, and ratings
- Service tier:
  - Java application
  - Probably EJB3 in JBoss
  - Exposing a REST API and returning JSON-serialized data
- Started writing prototypes ...



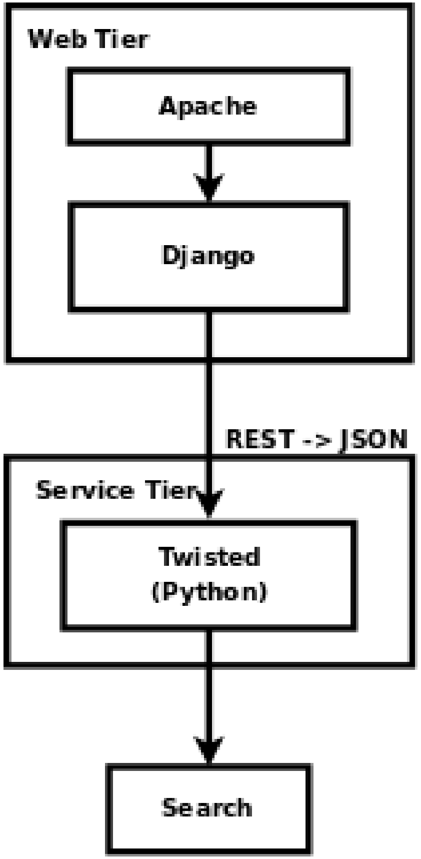
One


Web Tier
Apache
Rails
(fat controllers)
REST -> JSON
Service Tier
Twisted
(Python)
Search

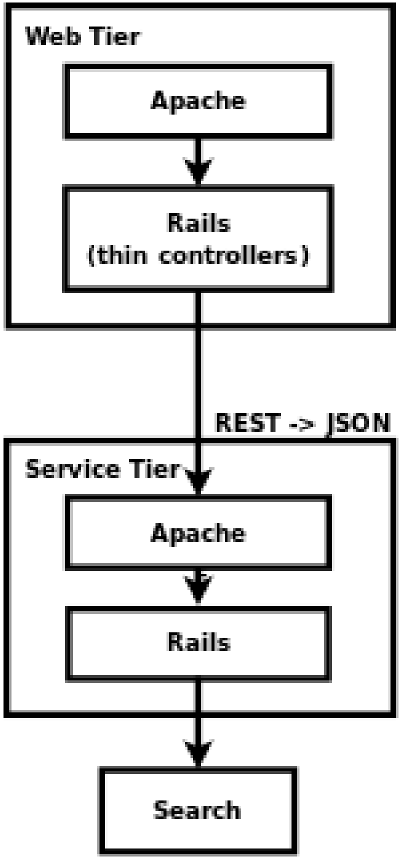
Two


Web Tier
Apache
Rails
(thin controllers)
REST -> JSON
Service Tier
Twisted
(Python)
Search

Three


Web Tier
Apache
Django
REST -> JSON
Service Tier
Twisted (Python)
Search

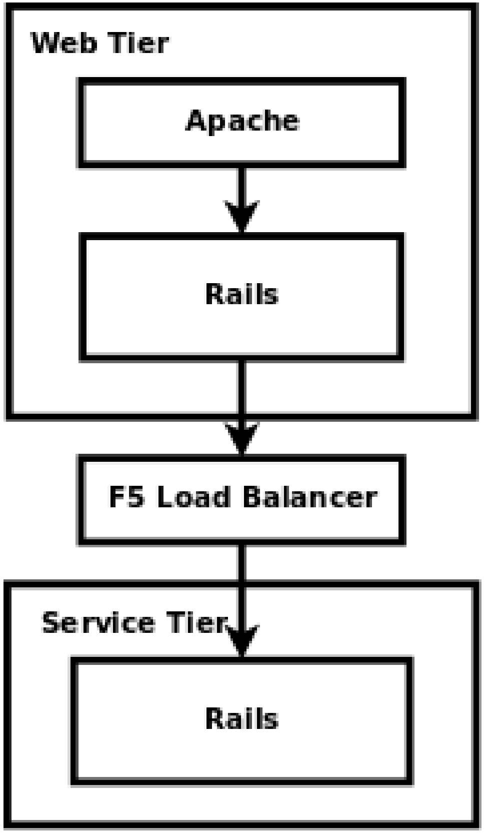
And finally ...

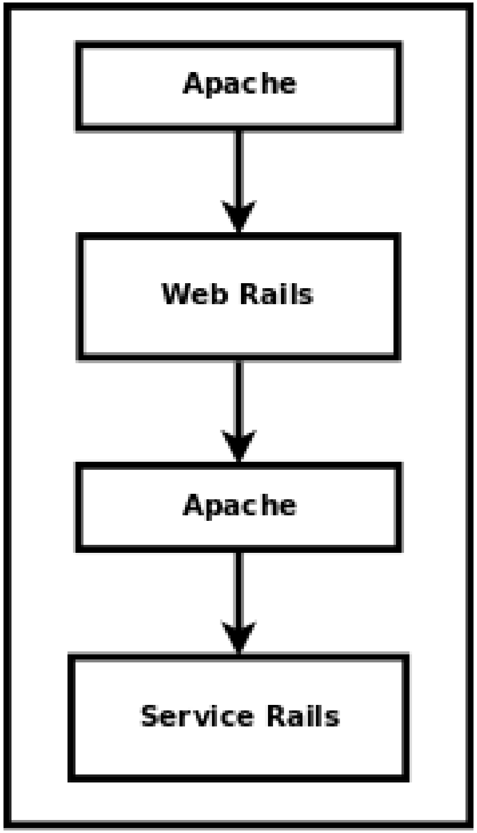

Web Tier
Apache
Rails
(thin controllers)
REST -> JSON
Service Tier
Apache
Rails
Search

Why Rails?

- Considered Java frameworks didn't provide enough control of url structure
- Web tier choice became Rails vs. Django
- Rails best web tier choice due to
  - Better automated testing integration
  - More platform maturity
  - Clearer path (for us!) to C if necessary for performance
  - Developer comfort and experience
- Team decided to go with Rails top-to-bottom
  - Evaluation of EJB3 didn't show real advantages over Ruby or Python for our application
  - Reasons for choosing Rails for web tier applied equally to service tier
  - Advantage of having uniform implementation environment

Separate or combined?





Other considerations

- How many servers?
- How many mongrels per server?
- How much memory for memcached?

Production configuration

- Acquired 25 machines of identical configuration for each data center
- Performance testing to size out each tier, and determine how many mongrels
- 4 GB of memory on each service-tier machine set aside for memcached
- Used 2 machines in each data center for database servers

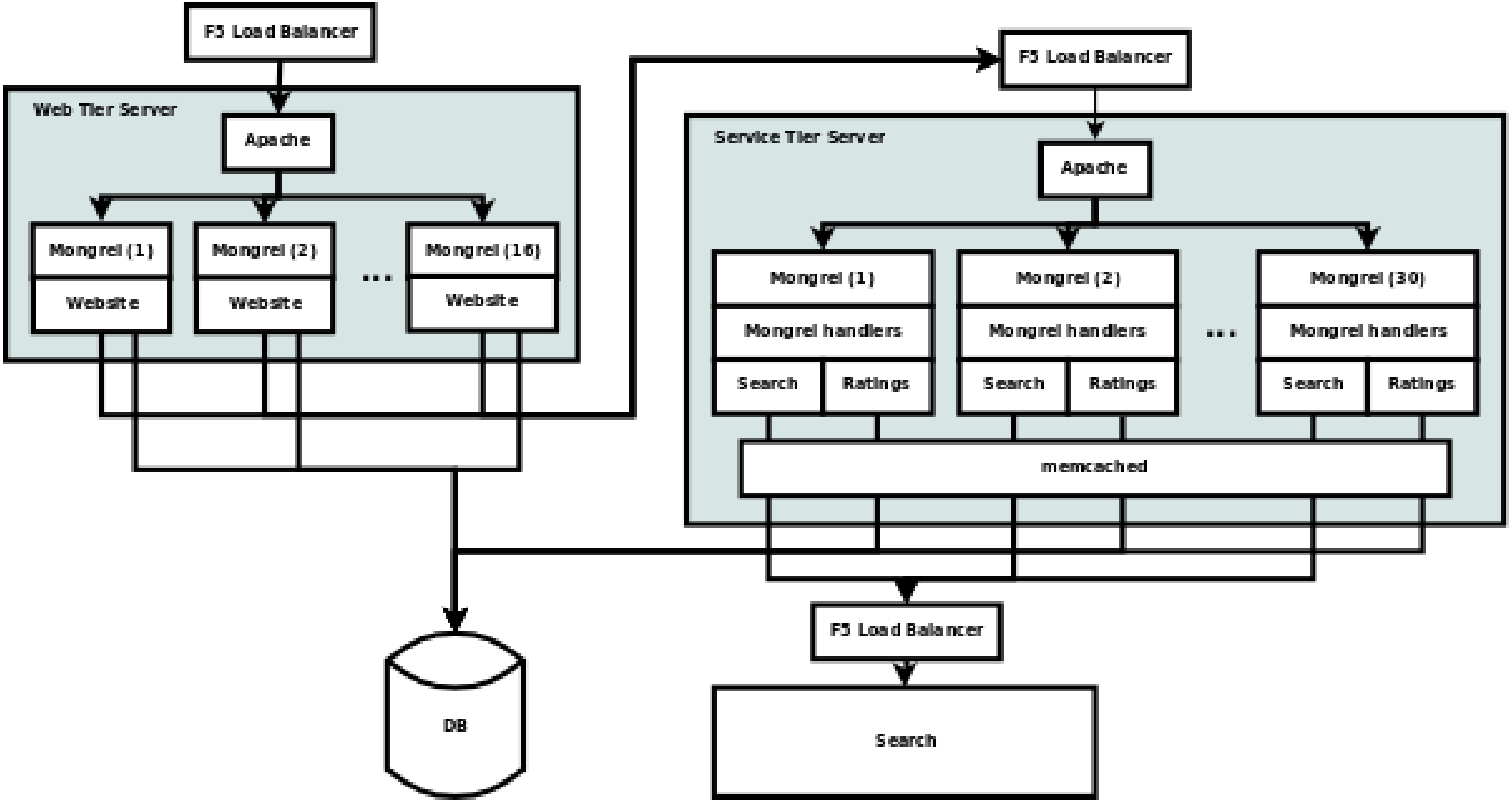
Performance goals

- Sub-second average home page load time
- Sub 4-second average search results page load time
- Handle traffic without dying

Performance optimizations

- mongrel_handlers in service tier application
- C library for parsing search cluster responses
- Erubis rather than erb in web tier

Site at launch



at&t Interactive

Database performance issues

- Machines inadequate to handle search load for ratings look-up
  - Additional caching added
- Oracle doesn't like lots of connections
- Use of named connections made this problem even worse
  - Additional memory required for database servers
  - All database look-up code moved to service tier
  - Changed to a single database connection

Page performance issues

- Slow page performance caused more by asset download times than speed of web framework
- Worked through the Yahoo! performance guidelines
- Minified and combined CSS and Javascript with asset_packager
- Moved image serving to Akamai edge cache
- Apache slow serving analytics tags -- moved to nginx for web tier
- Started using some fragment caching

Slow requests, etc.

- Slow requests in the web tier caused mongrel queueing
  - Developed qrp (http://qrp.rubyforge.org/)
  - Allows you to establish a backup pool where requests get parked until a mongrel is available
- Experimented with different malloc implementations
- Started using a custom MRI build -- ypc_ruby
- Started using a slightly-customized Mongrel

Overall performance

- Performance at launch was generally acceptable
- After web server & hosting changes performance better than previous site
- Extensive use of caching and elimination of obsolete queries lowered load on search cluster compared to previous site
- Over a year later, we need to do more profiling
  - Traffic has more than doubled since launch
  - Hardware evolution has invalidated original profiling
  - We now want sub 2-second search result pages

What else?

- New applications on Rails
  - Server-side component of native iPhone application
  - Working on moving current Java API application
  - Other internal applications
- Ruby but not Rails
  - Exploratory port of service tier to merb
  - Supporting development of Waves
  - Data-source ETL
  - Listing match-and-merge