ApacheCon

# Building a Vertical Search Site

(using lots of Apache software, of course)

Leading the Wave of Open Source

Hi, and welcome to my talk on "Building a vertical search site"

## Just the Facts, Ma'am

- Ken Krugler - CTO/co-founder of Krugle
- We use lots of Apache S/W at krugle.org
  - Httpd, Lucene, Nutch, Hadoop, Solr, Xerces, mod_perl, Commons, Jakarta, Maven, Perl, POI, and others I'm forgetting…sorry.
- I'll describe our architecture
- And the sometimes painful lessons learned

ApacheCon

Leading the Wave
of Open Source

Krugle is system for using search to help developers.

I'll talk about the details of what we do in the next slide, but the key points are that we built three different products in a short amount of time, and on a limited startup budget - and the way we did that was to use almost exclusively open source.

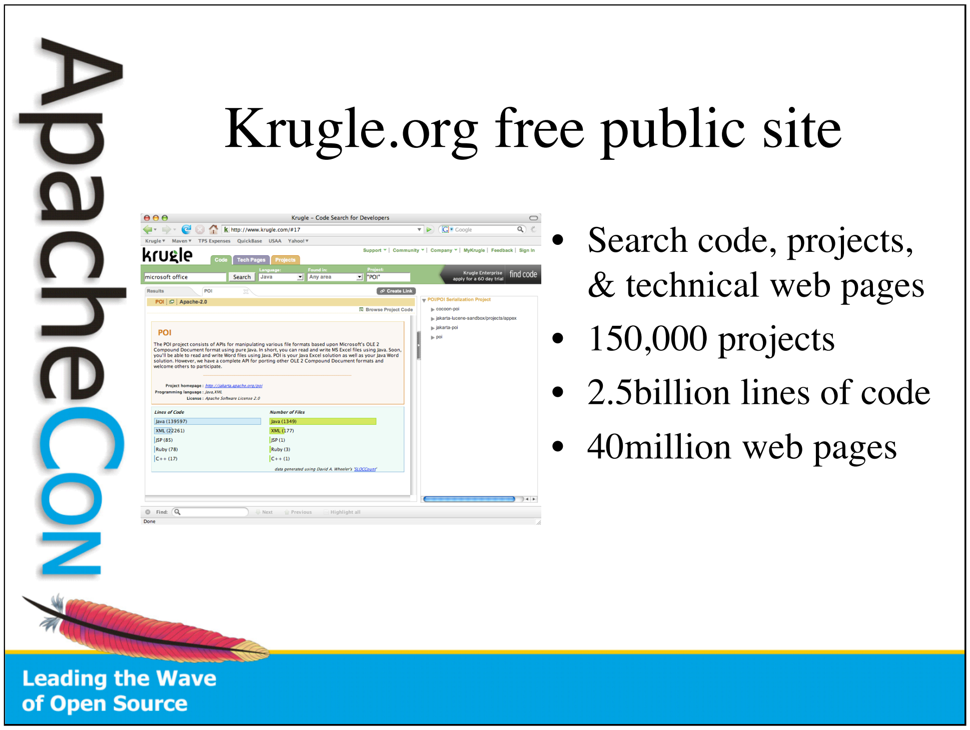
Along the way we learned some things about both architecture and open source, and that's what I'll be covering today. I'll focus mostly on the search aspects, specifically how we use Lucene.


I'd mentioned that we built three different products, or what we call the three faces of Krugle. It's not as schizophrenic as it sounds, since all three build on the same core technologies, and they provide the same key functionality for search-driven development.

What do I mean by "search driven development"? It's using search to help solve development problems, by making the right information available at the right time. The reason I bring this up is that it's important to understand what problems we're solving, and thus what functionality we need. And from that functionality flows the architectural requirements.

I think it's a too-common problem that we, meaning developers who are making technology decisions, focus on an architecture that we like. And that then in turn defines what's easy and what's hard, so the architectural decision winds up having a lot of say about functionality. And that then in turn defines what problems you can or can't solve.

We tried to flip that around, and make sure we focused on the problem space first, and used that to drive features and architecture. Now we did, early on, make the decision to use open source as much as possible. This wasn't an architectural decision so much as one based on cost and time-to-market, but it did wind up influencing many of the subsequent architectural choices.




So what's the problem we solve with the public site? There are three:

First, there's a lot of open source code out there, but it wasn't easily searchable.

Second, there's a lot of open source projects too, but it's hard to find what you're looking for.

Third, there's a lot of technical information, but it's sometimes hard to find the good stuff.

The common word in all three of the above is "lot". We knew we'd have to support fast search on many bytes of all three types of data.


When I do a search on "lucene" in code, I'm going to get a lot of hits.

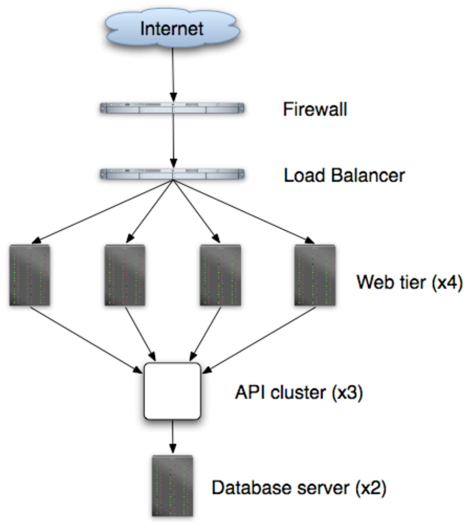
But I'm really looking for input on performance, so I can restrict to just comments that mention performance.

But I see a mix of languages, and I'm only interested in Java. And I don't want to get hits in Lucene itself, just code that uses Lucene.

Then I do a search on "lucene" in projects, where I also want it to be about databases.

So there's the requirement of fast, flexible search over lots of data. And you can see that we also have a requirement to provide a very rich, dynamic browser UI for exploratory search. So that also creates architectural requirements. For example, portlets are an interesting technology, but didn't seem right for this particular UI. We were going to need to write a lot of Javascript, where we were in control of the presentation layer. And we'd want to be able to access the services using HTTP requests.

# Krugle.org Architecture (web)




- Web tier runs Apache
- Also mod_perl
  - “glue” for Javascript to backend RESTful API
  - Partner APIs
- “Dirty” side of system

We've covered the problem space, and touched on functionality, so let's start talking about the actual nuts-and-bolts of the architecture.

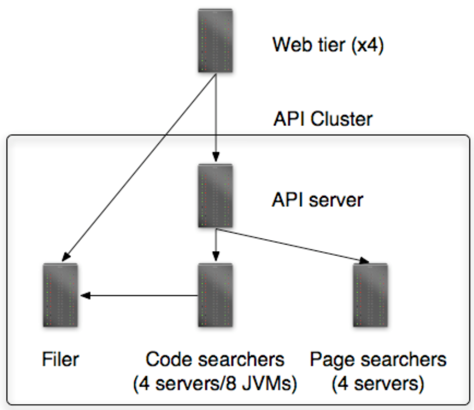
This is a pretty standard design, so I won't spend a lot of time on it.

There's the public-facing stuff, which we call the dirty side of the system. This consists of the top three pieces you see here - industry-strength firewalls from Juniper, load balancers from F5, and then four mid-range web servers.

The load balancers support sticky sessions using IP addresses, so a user keeps getting sent to the same web server once they've established a connection. This would cause problems if we had lots of AOL users coming from the same IP address, but fortunately that's not the case.

When we first went live, we only had two web servers, and we were exec-ing Perl code. This didn't work so well. Things backed up pretty quickly under load, mostly due to not using mod_perl. Though we made things worse by generating authentication tokens using "real" randomness, which means we quickly ran out of truly random bits. So then logins stacked up as the system waiting for cosmic rays to generate more randomness. Pretty quickly we realized that we didn't need that level of security.

# Krugle.org Architecture (API)




- API server uses Resin
- Webapps provide RESTful API services
- Filer is big disk array
  - LightTPD, NFS
- Searchers run Hadoop, Lucene

The API layer is more interesting, as here we get into things specific to the service that we provide.

As I mentioned earlier, requests to the API tier come from the web servers over HTTP. This "Krugle API" is reasonably RESTful. Requests to read or query "things" are received as HTTP GET requests, and the response is XML. A "thing" is a file, a project, a user-generated note, a codespace, and so on.

This REST approach has worked well for us, in general. We've benefitted from having these loosely coupled web apps, and it's been pretty easy to integration something like Solr into the mix.

There have been two significant downsides to this approach.

The first is that during development, you often wind up needing to hit one of these APIs to really test out your code. So we have a "development" API server that's available, but only if you're on VPN, and sometimes it's not available. People are changing code, so things can and do break. And when that happens, dependent services become harder to work on.

The other is that automated integration testing is also harder, for the

## Krugle API Server

- Webapps running inside of Resin
- Monitored using Big Brother
- NOT the performance bottleneck

ApacheCon

Leading the Wave
of Open Source

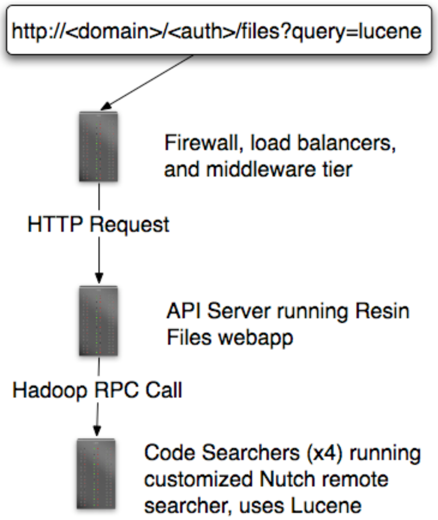
The API server a reasonably fast machine with some redundancy - things like dual power supplies, NICs, and spare drives. It turns out the API server isn't the bottleneck for performance, given how we use remote servers to do the searching. So we don't need a fire-breathing box here, just something that isn't likely to die.

On this API server, all of the services are running as Java webapps inside of Resin. Yes, we could have used Tomcat, but at the time when we were making this decision back in 2005, there was significant discussion on the mailing lists about Tomcat having stability problems. Mostly these seemed to be around it hanging at random times, under load. Resin got high marks here, so that's what we went with.

As you'll see later, we're using Jetty for part of the enterprise product, and we might wind up switching to it for everything. One of the advantages of Jetty is that it's better integrated with Maven and Eclipse, which we use internally.

But in general Resin has done well for us. Occasionally when we're having problems in the back end due to memory pressure, Resin can go into a mode where it returns bogus responses to requests - for example, we'll get an HTTP 200 status code, but no content. And that makes the middleware Perl code, the XSLT and Javascript all very unhappy.

# The Life of a Files Query



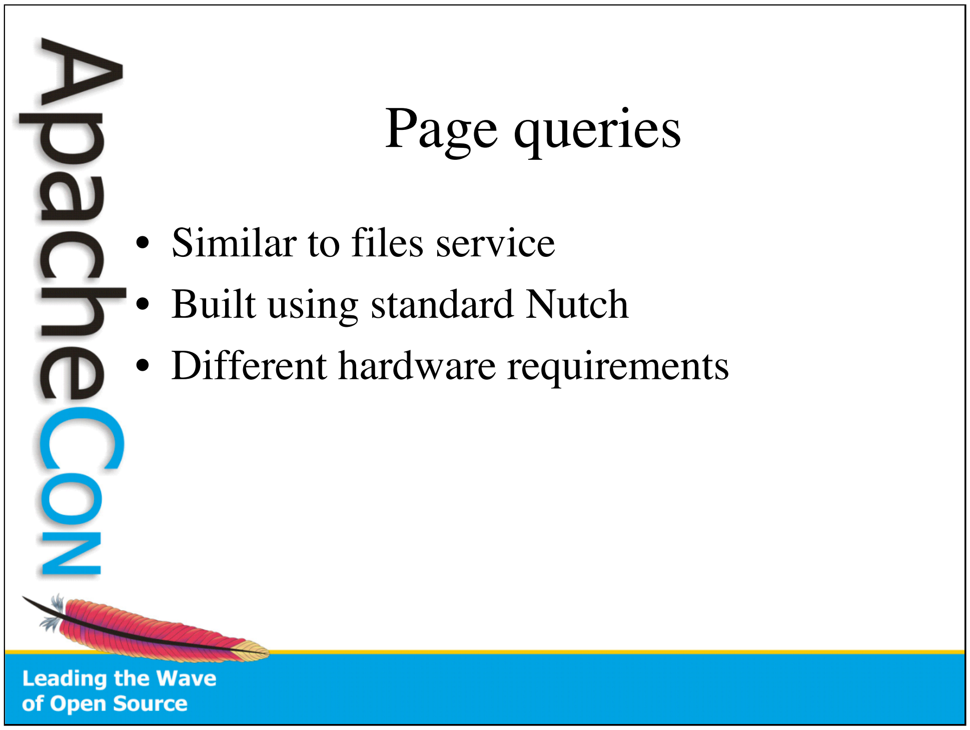

- HTTP to Resin
- Files webapp calls code searchers
- Results are combined
- XML response to caller

Going back to the code search we ran, the /files webapp service uses bits of Nutch to leverage the Nutch support for distributed searching. The query gets sent out, using Hadoop RPC, to four code searchers running a modified version of the Nutch remote searcher code. Each searcher is a 4GB server with two fast disks, so we run two 32-bit JVMs on each box, and two remote code searchers.

This lets us split our code index into 8 pieces, each with more than 5M documents. We randomly distribute the documents, to avoid skewing the inverse document frequencies. If, for example, we had one of these code searchers with nothing but Java source, and another searcher with only one Java file, then hits from the first server would get lower scores than what we want, and that one file from the second searcher would get a very high score.

Lucene's remote searcher implementation takes care of adjusting for this potential skew, but Nutch doesn't. If you take care when building the indexes, then this isn't a problem, and makes things faster. You can avoid another remote call required to gather the info you need to adjust for unbalanced document-level term frequencies.

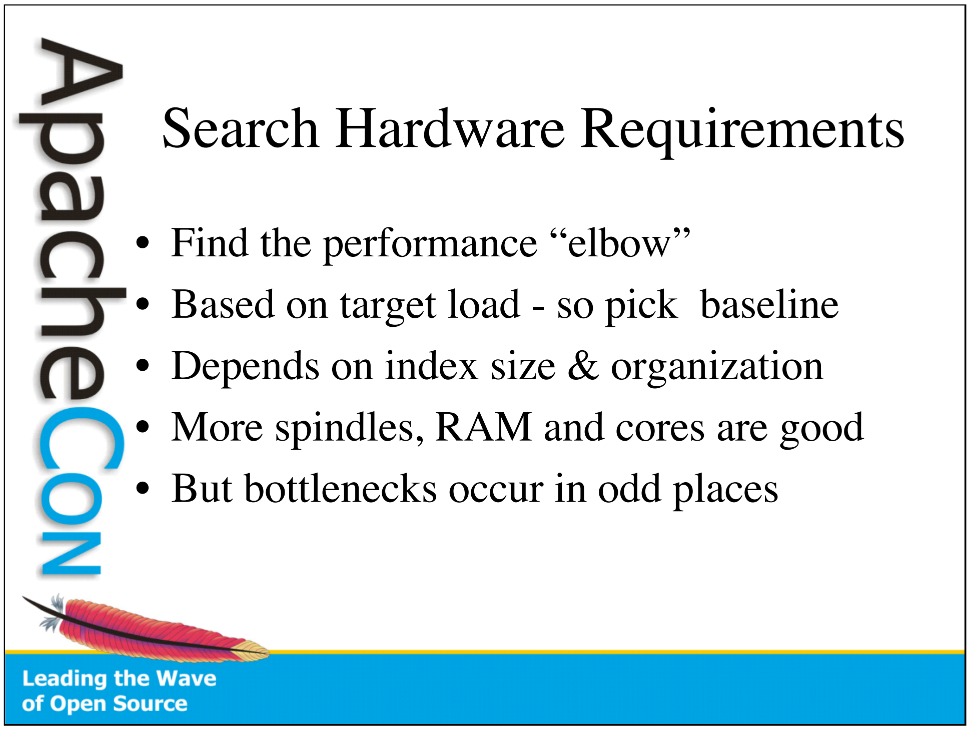
So getting back to that /files search request. It was sent out to each of the eight code searchers. They've returned their top N hits to the files webapp running on the API server. The files service picked out the top




For searching our 40M page tech page crawl, it winds up being very similar to files service. We using a pretty stock version of Nutch here, other than converting the Nutch results into our standard XML response format.

Page query results, like the files service, are cached on the API server using ehcache.

For the remote page searchers, we only run one JVM on each server, and we split up the index differently. Here we have one fast disk for the index, and a bigger, slower drive for the actual page data.



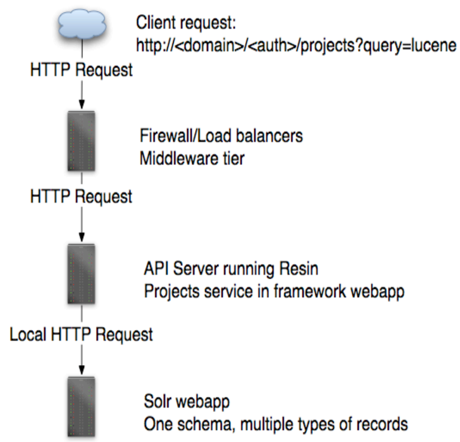

Now why did we go with 8 code searchers running on 4 servers? We set up a load test, and tried to figure out where the performance elbow existed. The question is when does adding servers stop improving performance significantly, for our target index and load? And for us, this was what we wound up with. The general rule of thumb seems to be that you want to have less than 10M documents per index, but that can vary widely. Why is that?

Note that I said "target index" previously. One of the changes we made, that let us get away with only 4 servers, was using the same technique on source code that Nutch uses on common words. If you leave in common words on a web page, your index size gets bigger and your search performance drops. But if you index combinations of common words, then you can avoid this problem. We did that type of thing for code like "i = 0", and that was a big win.

We'd get an even bigger win if we sorted our index by the static score we have for each file, and then do early search termination. That's a contrib that Doug Cutting made to Lucene a while back, but I haven't heard too much about people using it - or at least not using it successfully. Since we're currently fast enough, that's on the back burner.

We could also improve speed by combing more of our fields into combo

# The Life of a Projects Query




- HTTP to Resin
- Projects service in framework calls Solr
- Solr returns XML
- Framework remaps XML to standard format

Beyond code files and tech pages, there's a separate set of services that basically are front-ends to instances of Solr webapps that are also running inside of Resin.

When the projects service gets a request, it converts it into a standard Solr query, forwards it to the Solr webapp, then converts the response into our standard XML response and sends that back to the caller.

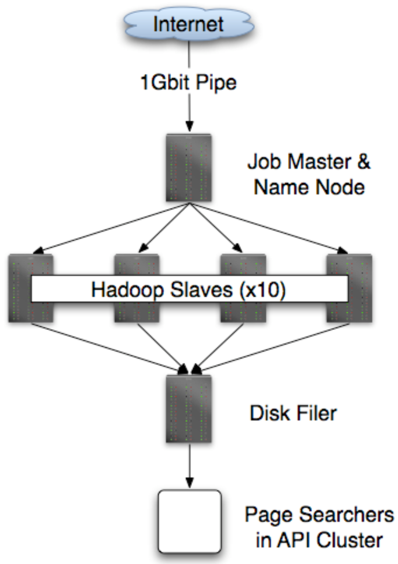
This seems inefficient, but given the query rate, the efficiency with which Resin handles local HTTP requests, and the performance of Solr/Lucene, this hasn't been an issue for us. We're using an index with about 150K entries, but I've heard stories on the list of much bigger indexes running without performance issues under high load.

Solr has been good to us in several different ways. It's easy to set up and get running, especially with the admin UI that gives you a view onto the index and lets you easily run test queries.

There are lots of useful analyzers that you can easily configure, using Solr's support for a Lucene index "schema".

It's never crashed.

# Krugle.org Architecture (CPI)




- Code crawl uses bits of Nutch, SCM adapters
- Project data pushed to Solr from MySQL
- Page crawl uses Nutch, custom page analyzers

A key point to remember here is that we've got three main sources of data that we use for our public site.

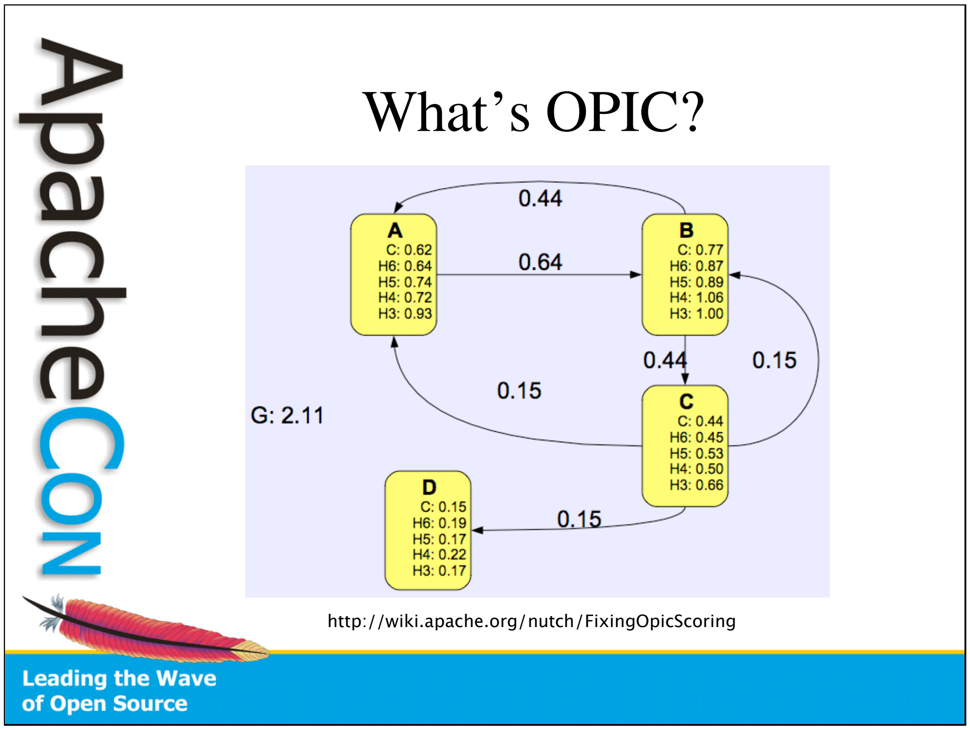
There's code, projects, and tech pages, and we handle each in a different way.

The diagram on the screen is for the page crawl, since the other two pieces, while interesting to me, don't really demonstrate exciting uses of Apache software.

For the page crawl, we use an 11 machine cluster running an older version of Nutch. It's version 0.8.2 with some customizations, and Hadoop 0.9.2. Why are we on an older version? Well, more than once we got bit by really bad bugs when updating to a newer version that "only had bug fixes". So last January we made the decision to stick with what we had, because it was good enough.

There's a lot of info on Nutch and Hadoop at the Apache web site, so I'm going to focus on the customizations we made, and the things we learned while using Nutch for a medium size web crawl.

First, a quick overview. A "crawl" in Nutch consists of multiple loops. A loop is a sequence of using the Nutch crawl database, the crawldb, to




If you're going to use Nutch, you need to understand at least a little bit about OPIC.

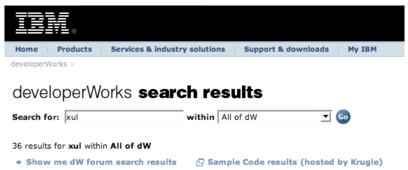
It's the "on-line page importance computation" that Nutch uses both during the crawl, and as the final static page score. In theory this is an incremental link analysis score that converges to something similar to a PageRank score if you recrawl enough. In practice, Nutch's implementation of this has some serious flaws.

For example, in a stable page crawl, page scores tend to "leak" out of any leaf pages. So the scores of pages continue to drop. Now since they're all dropping together, that's not so bad. But then when you inject new pages, these pages have scores that wind up being much higher than the older pages.

OPIC is also very sensitive to link farms, As you add new pages, the total energy of the web graph keeps going up. And so this winds up fighting with the leakage from leaf pages, which means that you wind up with spammy, highly linked pages having ridiculously high scores, and leaft pages have very low scores.

If you aren't recrawling, and you can stay away from link farms, then it's good enough to help guide the crawl. And the resulting scores are good enough for us, but we wind up having to essentially start each recrawl

# Krugle partner sites




- IBM developerWorks
- Sourceforge.net
- Amazon Web Services
- Yahoo! Dev Network
- Collabnet

Leading the Wave of Open Source

This is the second face of Krugle, where we provide code search for partners who have developer networks or communities and code that they want to make searchable.

These are hosted by us, with customized look and feel. In some cases we restrict searches to the subset of projects hosted by or associated with the partner.

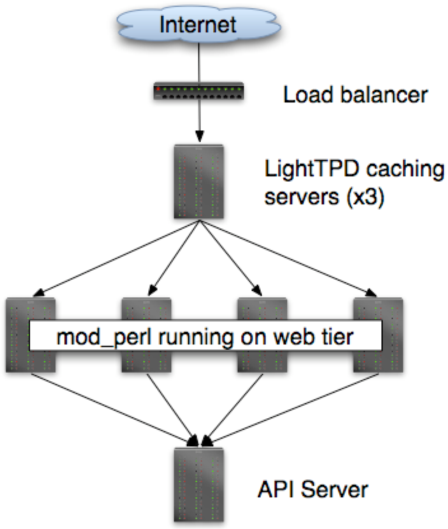
Each partner has their own set of challenges:

Sourceforge obviously has the most projects, and the biggest traffic.

Amazon has two types of projects, some of their own, and some hosted elsewhere that use Amazon web services. They want these searched as one set of "Amazon plus friendlies" projects.

IBM developerWorks has rapid update requirements. So they send us regular updates, via an HTTP feed, for new or modified projects. These need to be processed and pushed to the live site.

# Krugle Architecture (partners)

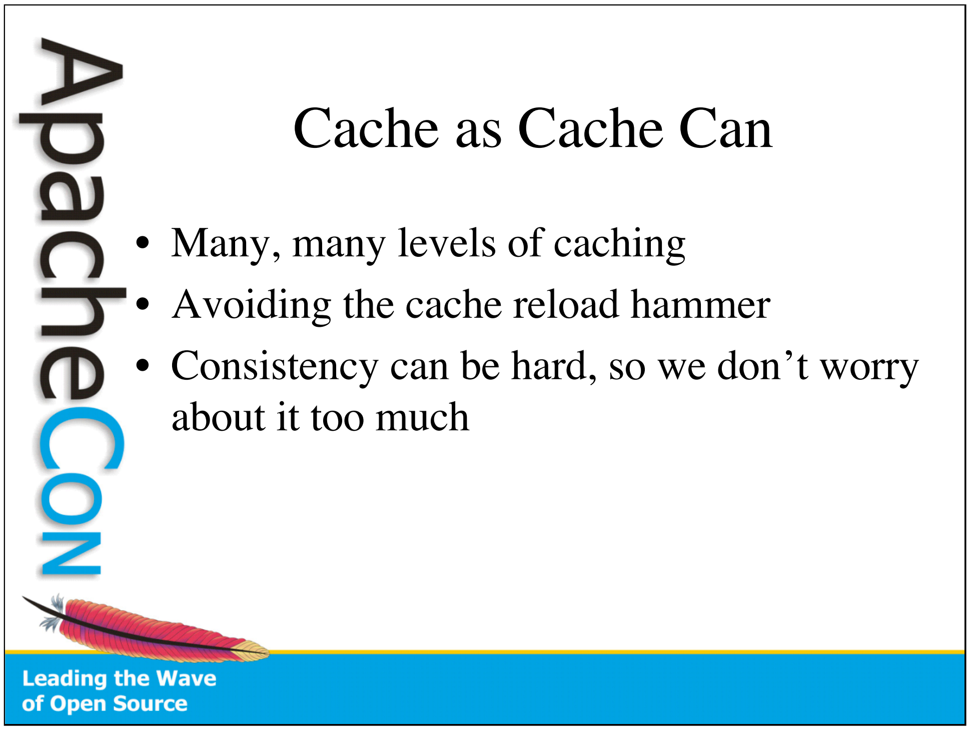



- Higher level API
- Wraps RESTful API
- Handled in web tier
- Big chunks of Perl
- LightTPD cache

Our partners make use of some specialized APIs provided by Perl code running in the middleware.

These requests first get round-robin dispatched to one of three LightTPD servers that use mod_cache to create a high performance cache. This significantly reduces the load on the API server, by up to 95%. For example, instead of 20 requests/second from a partner hitting the API layer, it's only 1 request/second.

Requests that are cache misses get sent on the web tier, where mod_perl executes Perl code that "wraps" our low-level APIs. Because of the standard API to services, it's easy in Perl to create customized functionality on top of these APIs.




Caching is clearly a big topic, so I won't go into this in depth. I'm just going to point out the many places where caching does occur throughout the system.

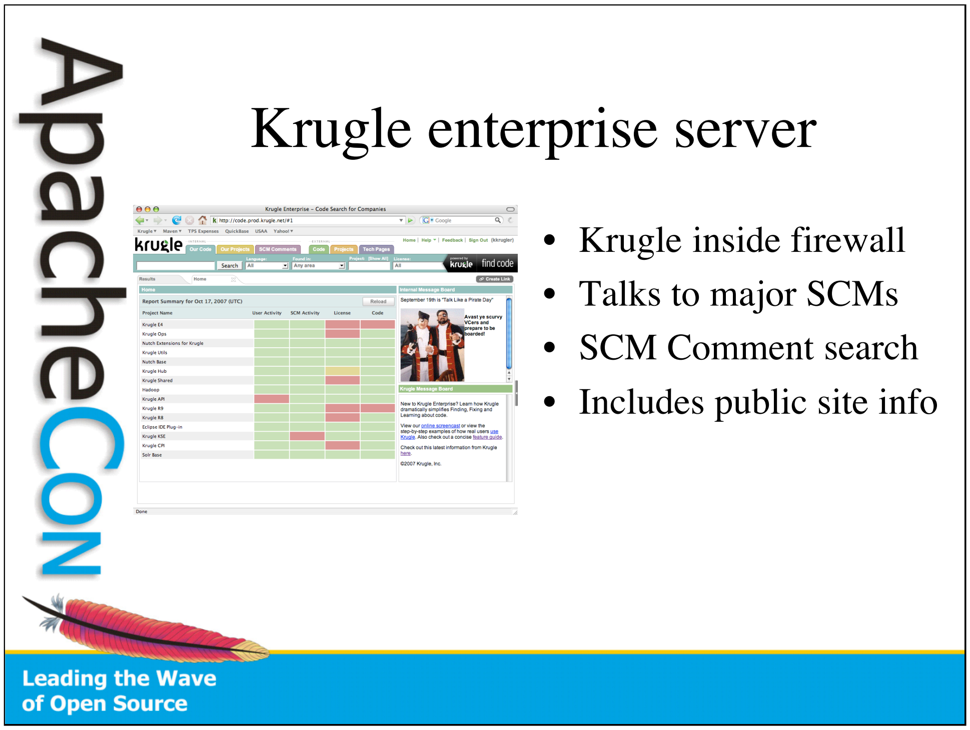
We've got the automatic file block caching that Linux does for us. As I mentioned earlier, this works surprisingly well for keeping key parts of a Lucene index in memory. In fact I know of one major company that warms up their Lucene-based searchers by cat'ing the index directory to /dev/null, as a way of forcing it all into the file system cache.

We cache search results at the service level in the API server using ehcache. And Solr has its own cache for queries.

There's also the LightTPD-based cache for partner APIs that I just mentioned.

And the web browser UI caches results during a user session, to avoid re-fetching content.

One problem that happens is when you start to depend on the cache to achieve target performance levels, and then the cache goes down or needs to be reloaded. Suddenly the back-end system gets hit with a huge load spike, and as we discussed when things start backing up,




Finally, we're at the third, and to us most important, face of Krugle. That's our enterprise product, which we've been working on for almost a year. It went into trials in May, and recently was released for general availability.

The enterprise product provides the same type of search-based functionality as the public site, but inside a company's firewall. So there are additional data sources - files and comments from internal SCMs, as well as project meta-data defined using what

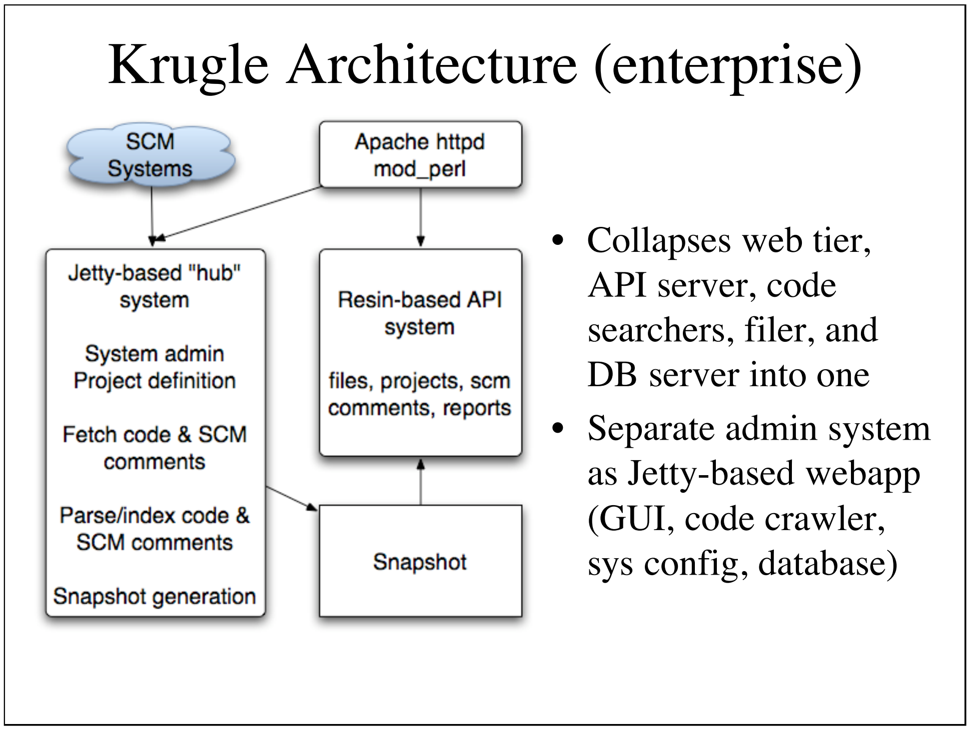
The servers we run this on have a fast 150GB Western Digital drive, 4GB RAM, and two dual-core CPUs.

ApacheCon

## Demo Krugle Enterprise

Leading the Wave
of Open Source

In some ways it looks very similar to the public site. But you can see there are three more search channels across the top here. The three on the right are the same as for the public project, then there are three more on the left, for internal code, internal projects, and SCM comments.




On the public site we've got probably 20 servers actively handling requests. But for our enterprise product, this all needs to fit into one box. So what all did we do to squeeze it down?

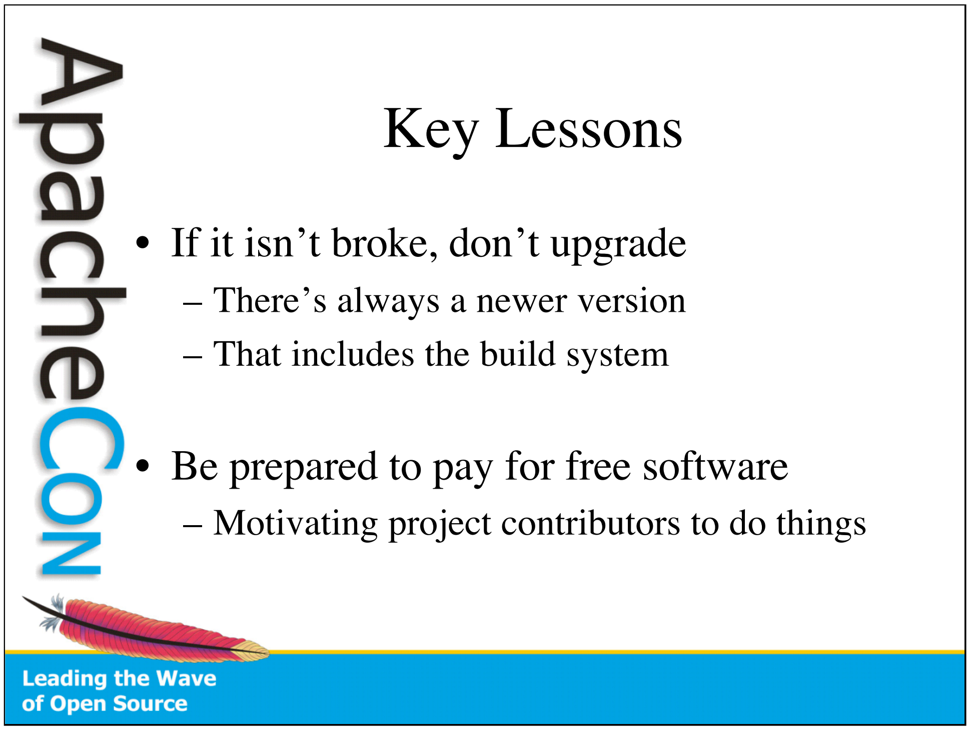
There's still an Apache httpd server, and mod_perl, but obviously only one of these instances running.

We still have Resin running webapps to implement the Krugle API, but there's no remote searchers for code. That all happens inside of the files webapp. And we don't support page crawling, so we can whack out a bunch of the Nutch crawl infrastructure.

But we need to add a nice UI for people who administer the system. And we also need to automate the code and SCM comment processing. So there's a Jetty-based webapp that implements the "hub" GUI I showed you previously, as well as this crawl process.

The configuration, including project definitions, is saved in a MySQL database that we interact with via Hibernate.

The result of a new or updated crawl is something we call a snapshot, which is a self-contained set of data stored in a directory. This includes the Solr index, the code index, the code files, reports generated on the




This is my final summary slide. If I had to do it all over again, what would be the most important things I wished I'd know in advance?

The first item is avoiding the powerful urge to get the latest released version. Most of our developers use Macs, and I blame Mac OS X as the source of that urge. There's the continuous PR blitz to get critical security fixes. Most of us upgrade regularly, without any problems. But the same isn't always the case for open source. That 1.3.2 version might fix some bugs, but it can also add in a gnarly thread lock bug.

I had to include the build system here, because we use Maven. And like many people who use Maven, it's a love/hate relationship. Sometimes more hate than love, actually. For example, we'd added a maven-enforcer-plugin to our build. In July a new version was made available on the Maven public repo, something like 1.0alpha3. That version wound up automatically being used, and it had a nasty bug that prevented inter-module dependencies from resolving correctly. So we starting generating builds with out-of-date modules. It's the kind of thing that makes you age faster than you want. The solution there was to really, really lock down the versions of everything we use that gets pulled from the public Maven repo, and disable auto-downloading to be extra safe.

On the subject of what free means, nothing is every really free, and if