



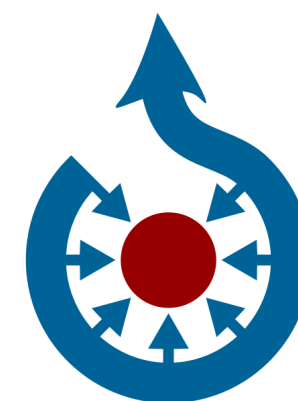
WIKIPEDIA  
The Free Encyclopedia



# Wikimedia architecture

Ryan Lane <[ryan@wikimedia.org](mailto:ryan@wikimedia.org)>

Wikimedia Foundation Inc.



# Topics

- Intro
- Our technical operations
- Global architecture
- Application servers
- Storage
- Caching
- Load balancing
- Content Delivery Network (CDN)
- The site architecture you can edit
- Community involvement

# Top five worldwide sites

Company	Users	Revenue	Employees	Server count
---------	-------	---------	-----------	--------------



920 million

\$23 billion

20,600

1,000,000+



740 million

\$58 billion

93,000

50,000+



600 million

\$6 billion

13,900

50,000+



500 million

\$300 million

1,200

30,000+



400 million

\$20 million

50

350

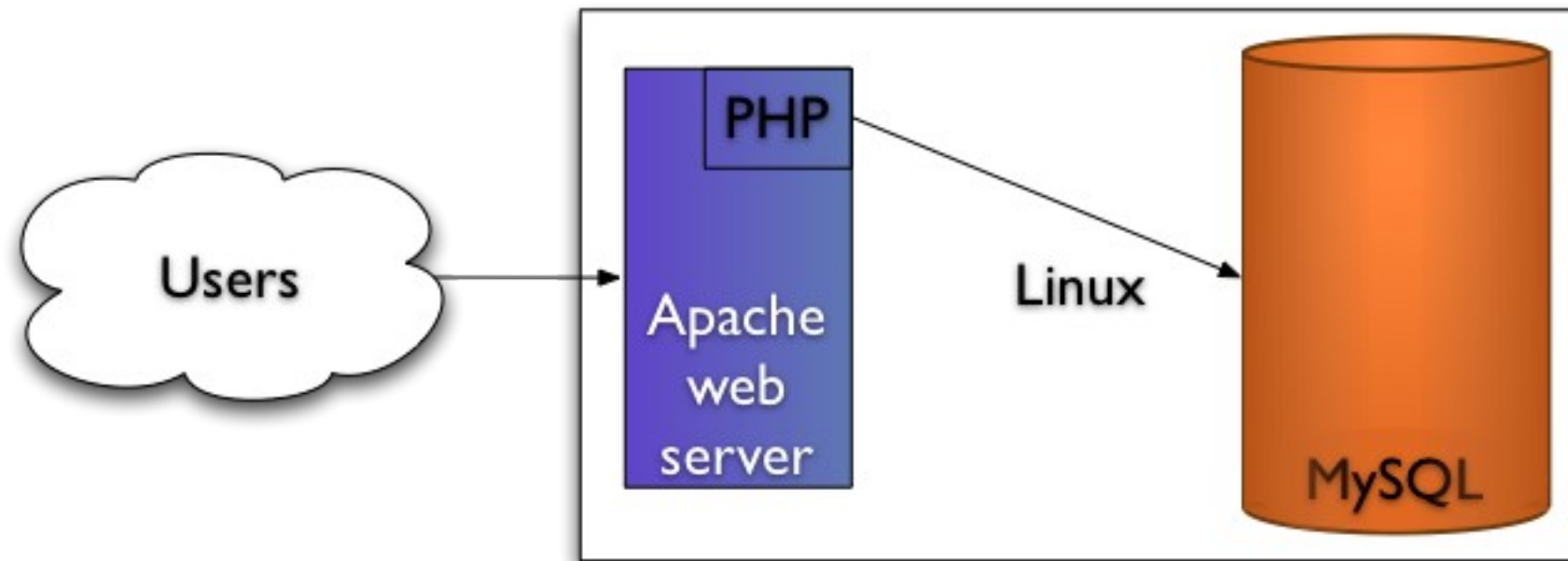
# Our operations

- Currently managed by ~6 ops engineers
- Historically ad-hoc, “fire fighting mode”
- Technical staff spread out globally
- Always someone awake...but no on-call
- Working to engage community operations contributors

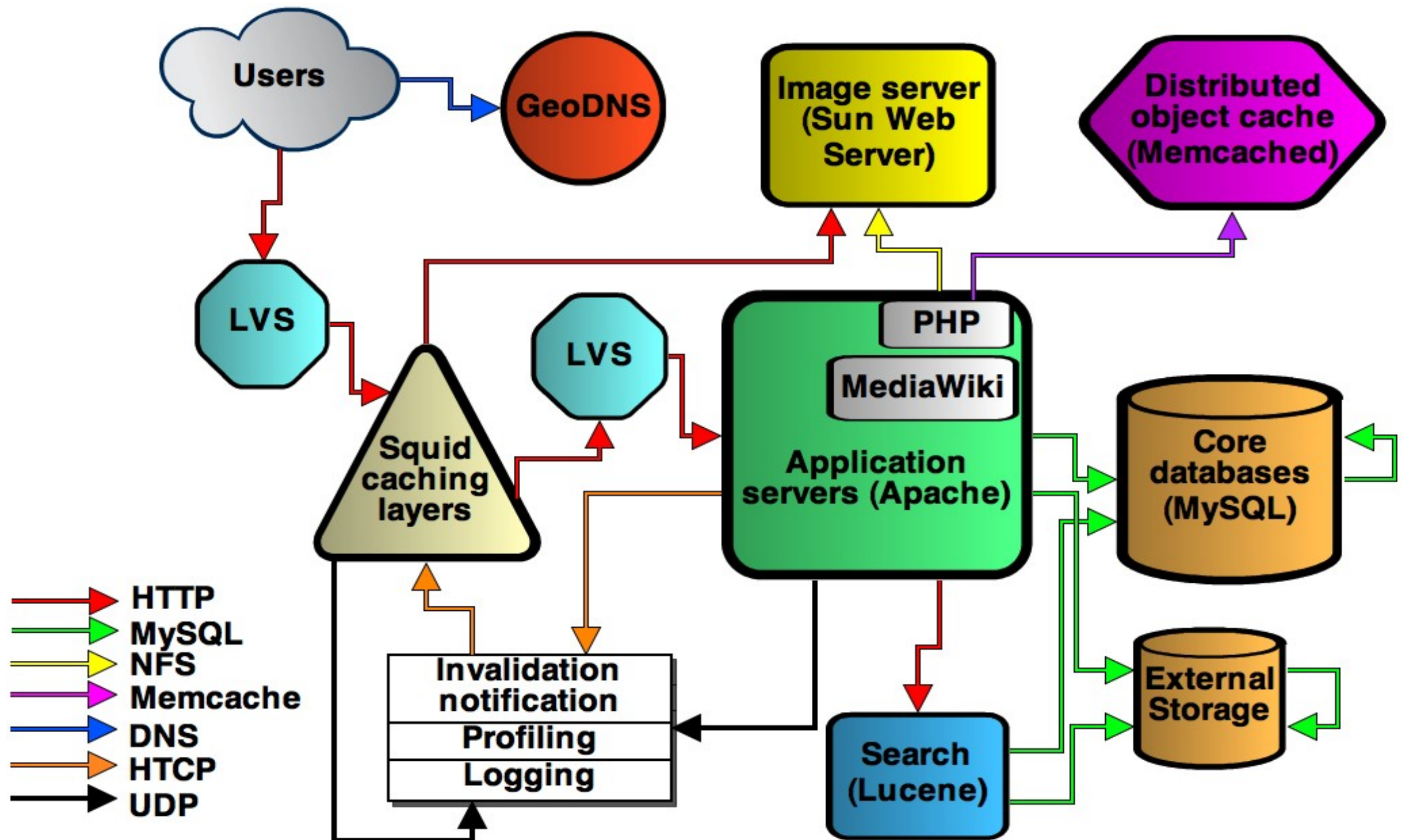
# Operations communication

- Most communication public on IRC
- Documentation in a wiki (<http://wikitech.wikimedia.org>)
- Sensitive communication via private lists and resource trackers

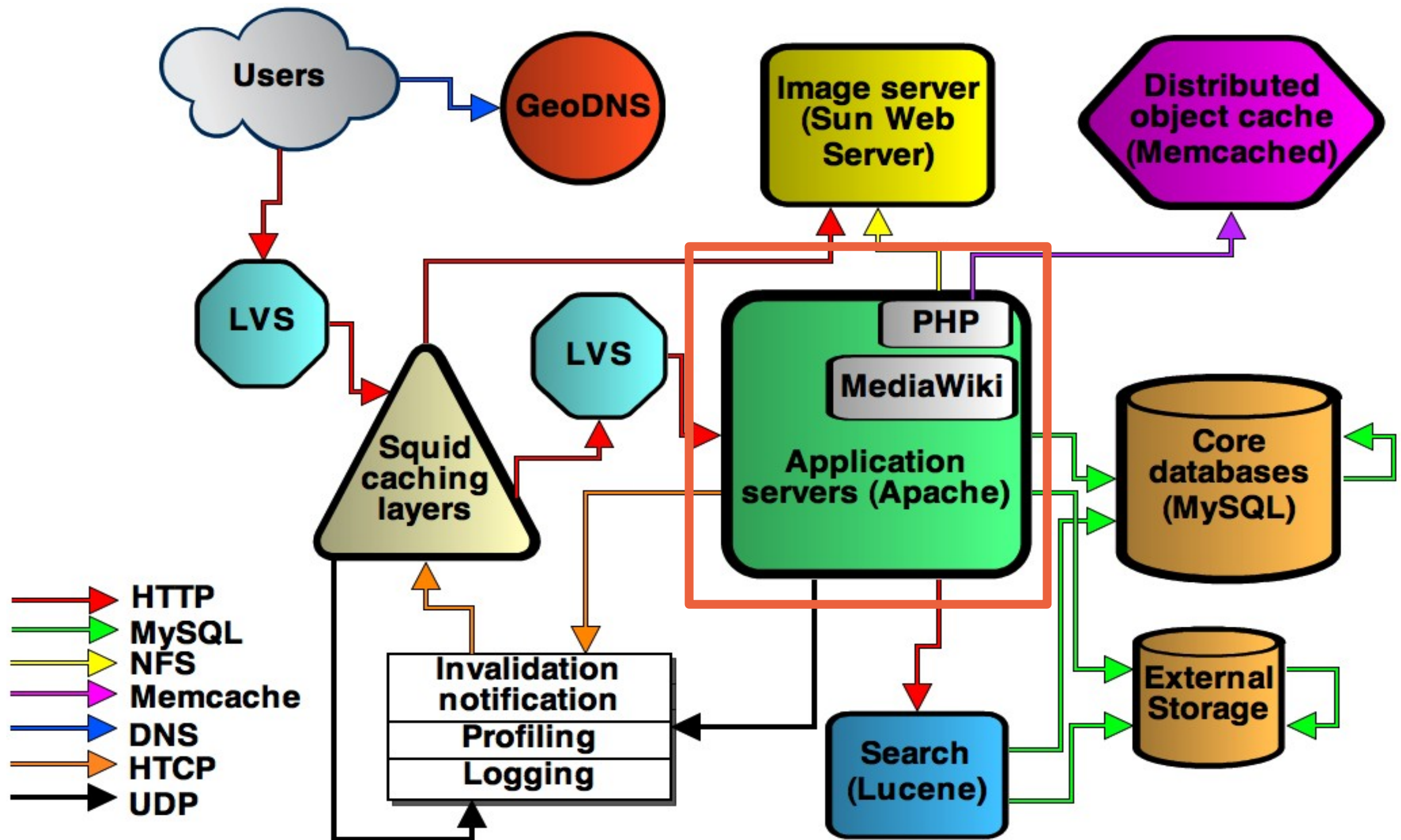
# Architecture: LAMP...



# ...on steroids.









# The Wiki software



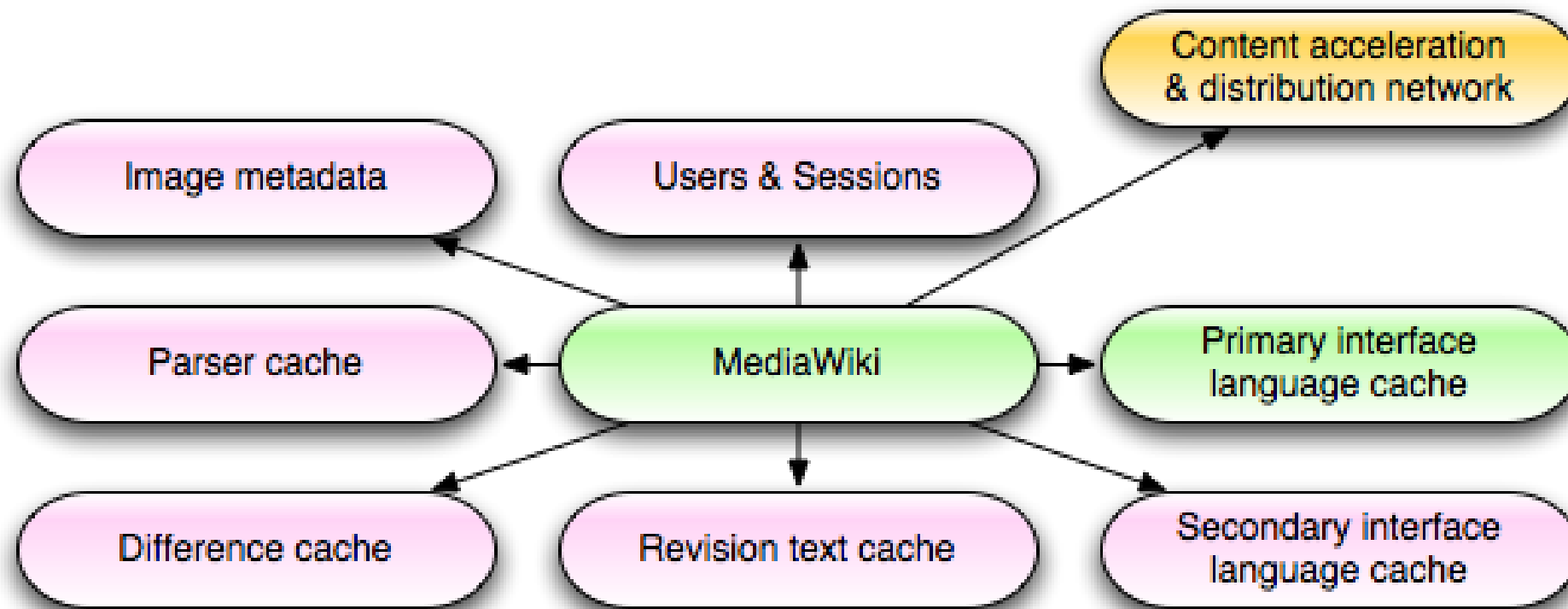
- All Wikimedia projects run on a MediaWiki platform
- Designed primarily for Wikimedia sites
- Open Source PHP software (GPL)
- Very scalable, very good localization

# MediaWiki optimization

- We optimize by...
  - caching expensive operations
  - focusing on the hot spots in the code (profiling!)
- If a MediaWiki feature is too expensive, it doesn't get enabled

# MediaWiki caching

- Caches everywhere
- Most of this data is cached in Memcached, a distributed object cache

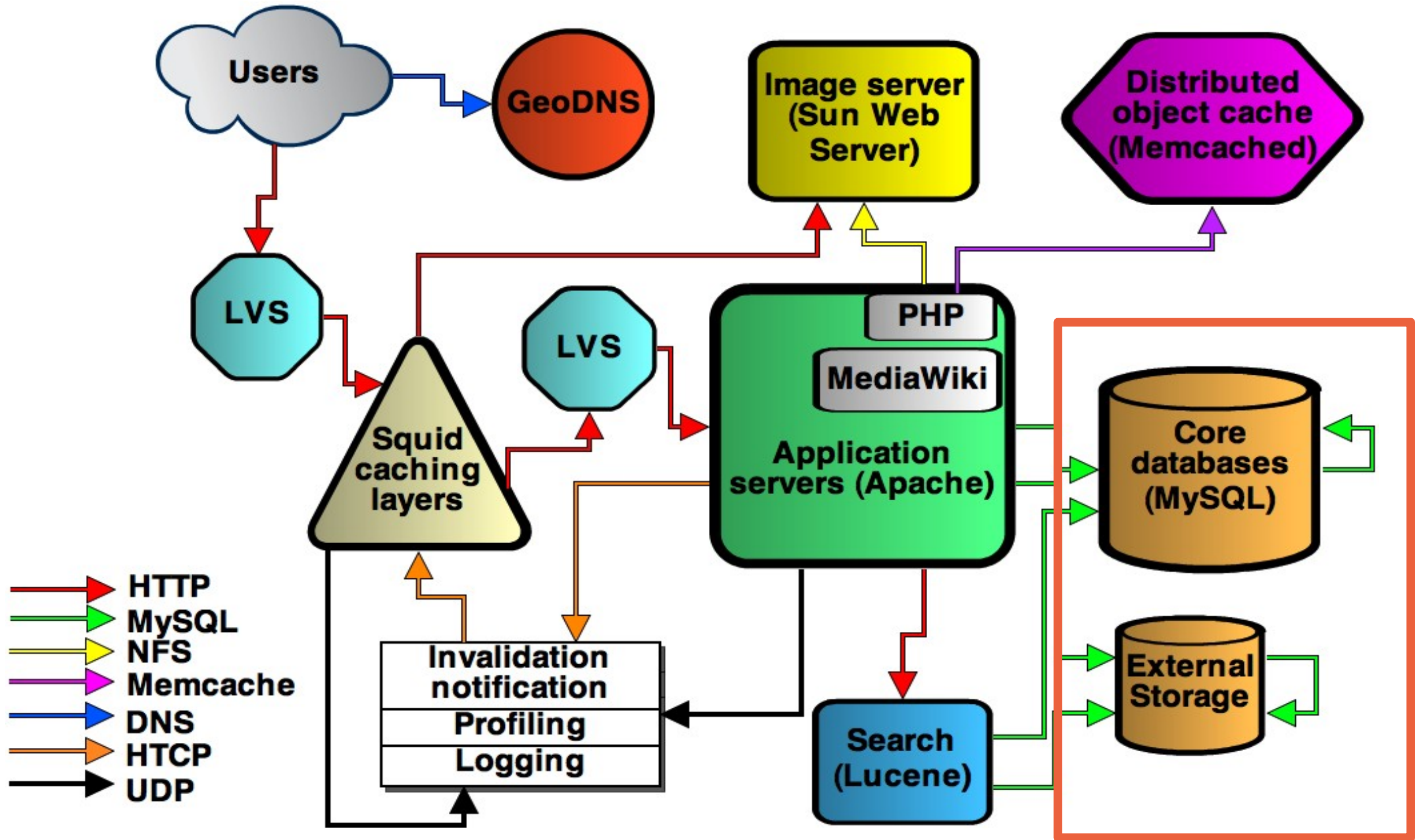


# MediaWiki profiling

<http://noc.wikimedia.org/cgi-bin/report.py>

[\[zhwiki\]](#) [\[thumb\]](#) [\[dewiki\]](#) [\[bigpage\]](#) [\[enwiki\]](#) [\[others\]](#) [\[flaggedrevs\]](#) [ showing 50 events, [show more](#) ]

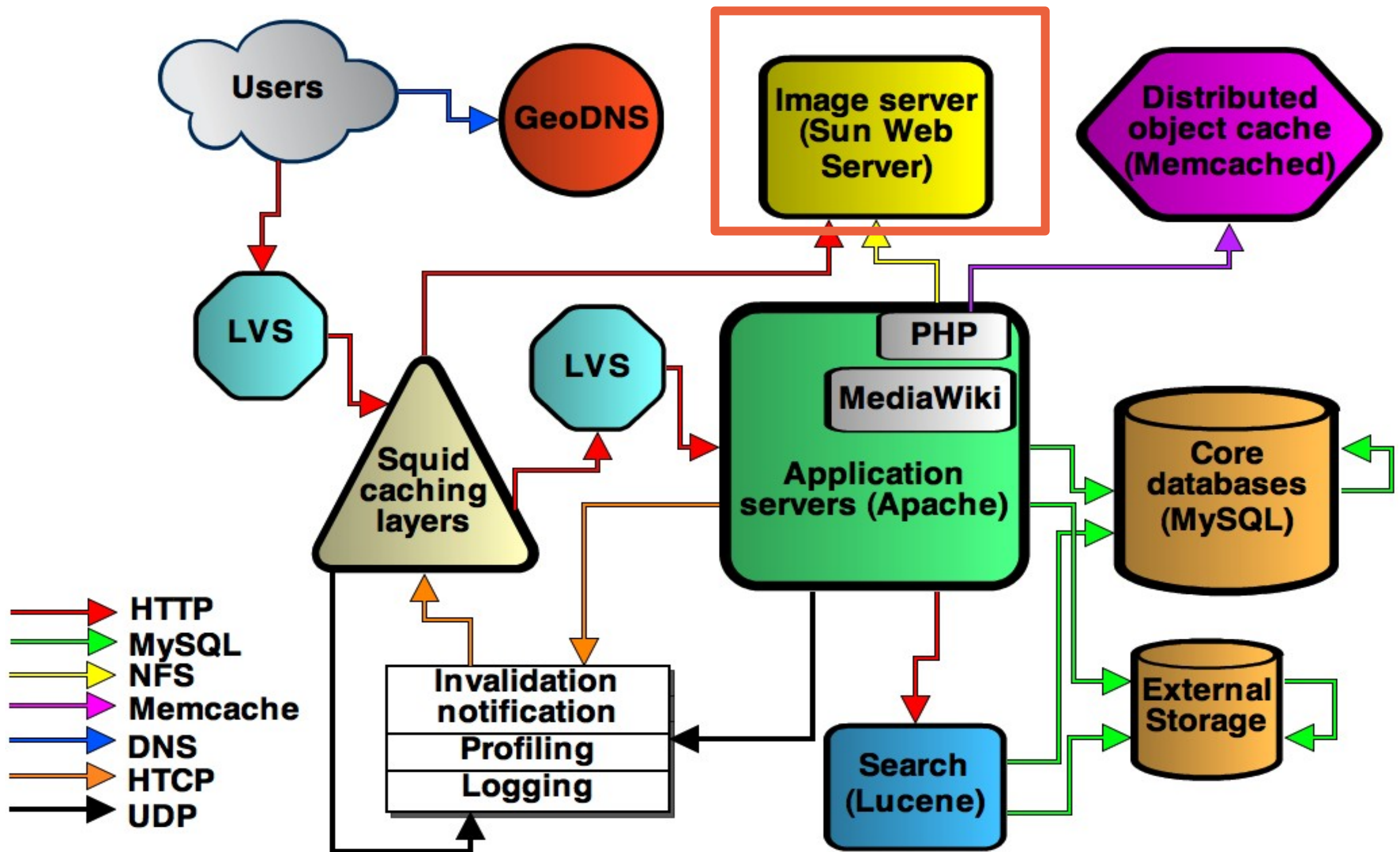
<a href="#">name</a>	<a href="#">count</a>	<a href="#">cpu%</a>	<a href="#">cpu/c</a>	<a href="#">real%</a>	<a href="#">real/c</a>
PPFrame_DOM::expand	2777300471	409	1.8	322	1.89
Parser::braceSubstitution	478045780	340	8.66	266	9.07
-total	7216450	100	169	100	226
Parser::braceSubstitution-pfunc	453314242	97.8	2.63	76.5	2.75
MediaWiki::performRequestForTitle	3445759	74.4	263	71.7	339
Parser::internalParse	6879781	78.4	139	61.7	146
Parser::replaceVariables	76312501	71.4	11.4	56.4	12
MediaWiki::performAction	1329950	65.8	604	54.8	671
Parser::parse	3242613	65.5	246	52.4	263
Article::view	956646	57.6	735	46.8	797
Parser::braceSubstitution-setup	478043151	53.9	1.38	41.9	1.43
Parser::parse-Article::getOutputFromWikitext	304682	49.8	1.99e+03	39.6	2.11e+03
Parser::argSubstitution	690835928	46	0.813	36.4	0.858
api.php	3720263	13.1	42.8	17.9	78.4
API:main	3720255	12.8	41.8	17.6	77.2



# Core databases

- One master, many replicated slaves
- Load balanced reads to slaves, write operations to master
- Separate database per wiki
- Separate big, popular wikis from smaller wikis (sharding)



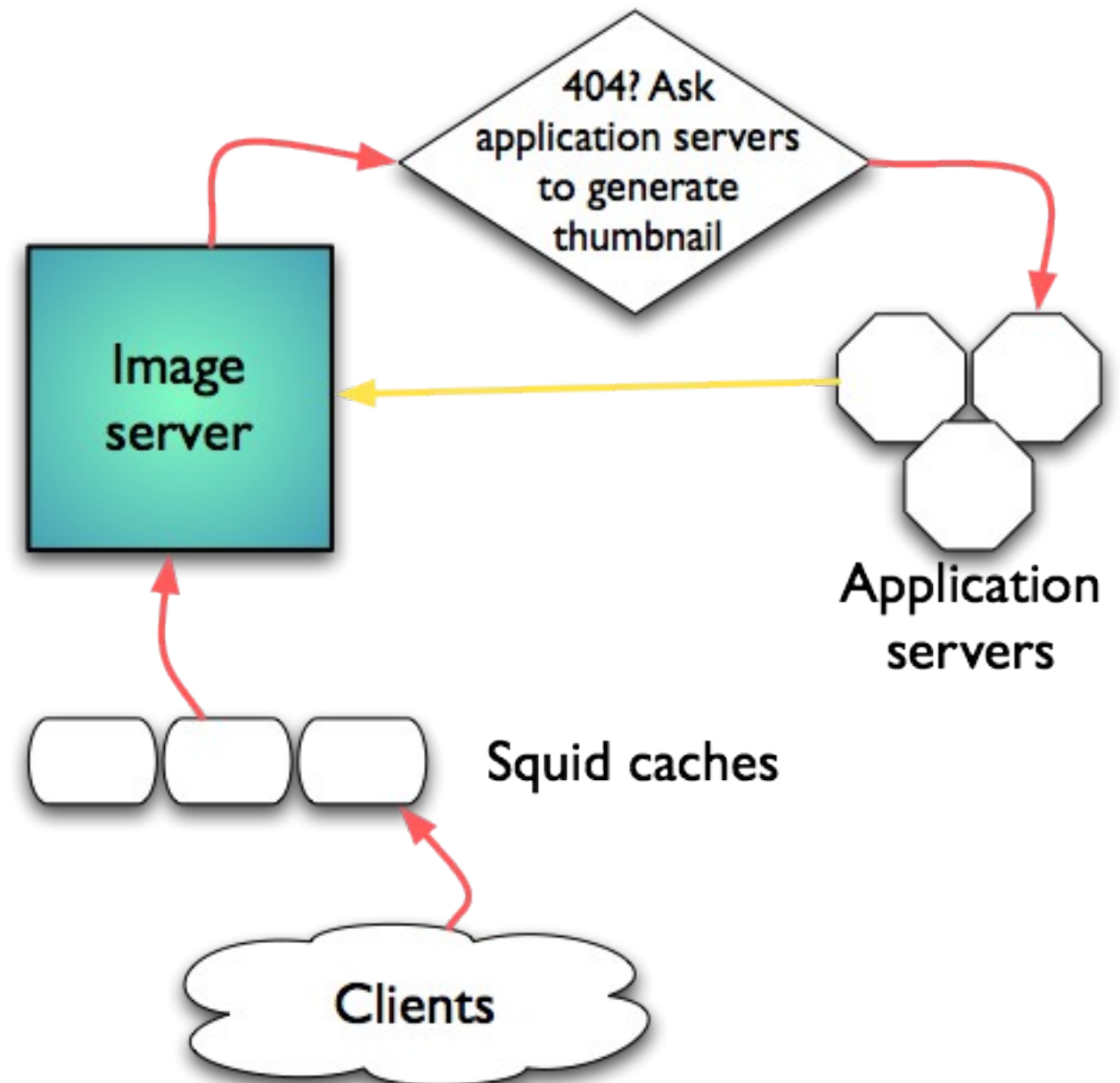


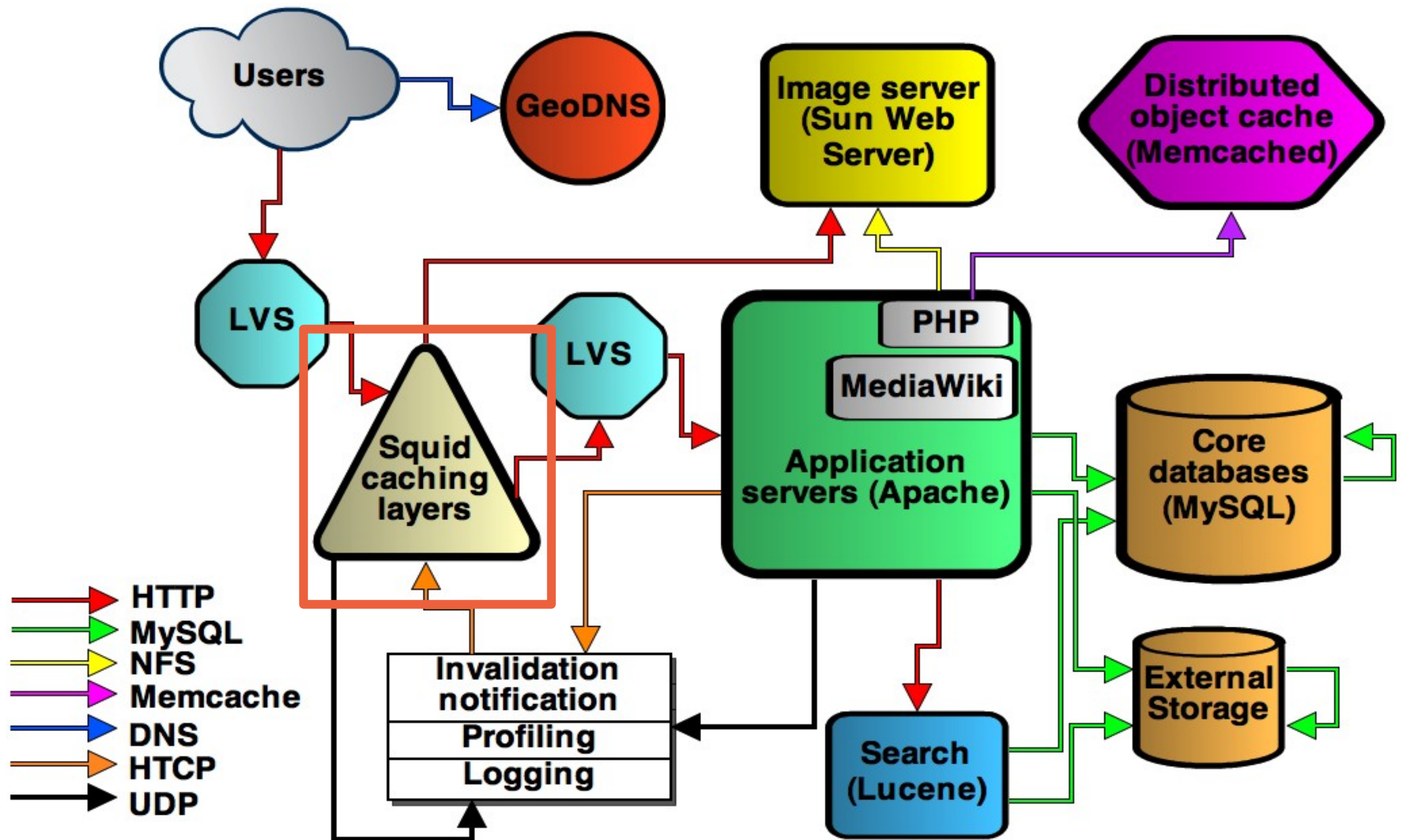
# Media storage

- Current solution is not scalable
- Replacing with an open source distributed file system
  - Likely OpenStack Swift

# Thumbnail generation

- `stat()` on each request is too expensive, so assume every file exists
- If a thumbnail doesn't exist, ask the application servers to render it



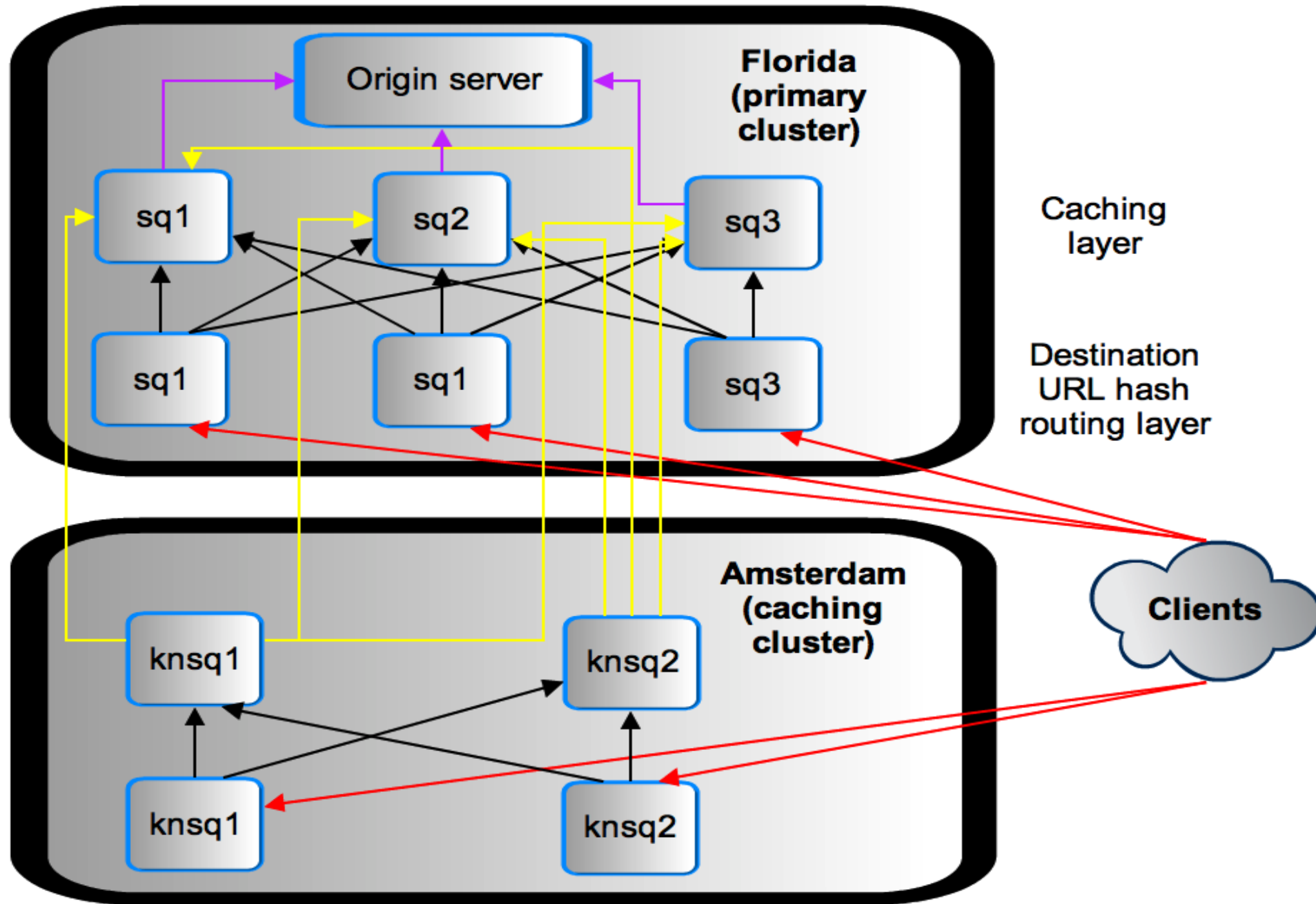


# Squid caching

- Caching reverse HTTP proxy
- Serves most of our traffic
- Split into two groups
- Hit rates: 95% for Text, 98% for Media, since the use of CARP



# CARP



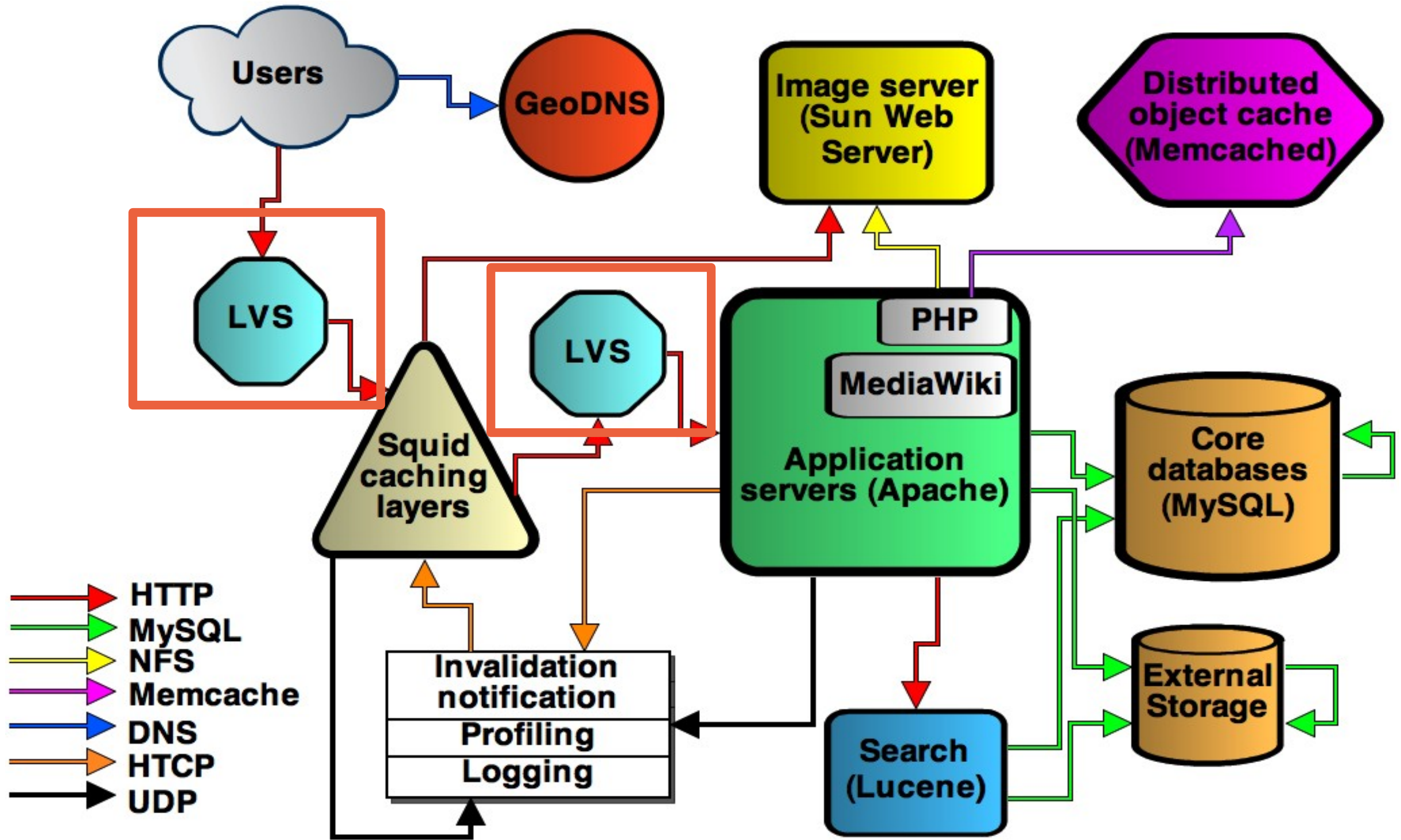


# Squid cache invalidation

- Wiki pages are edited at an unpredictable rate
- Users should always see current revision
- Invalidation through expiry times not acceptable
- Purge implemented using multicast UDP based HTCP protocol

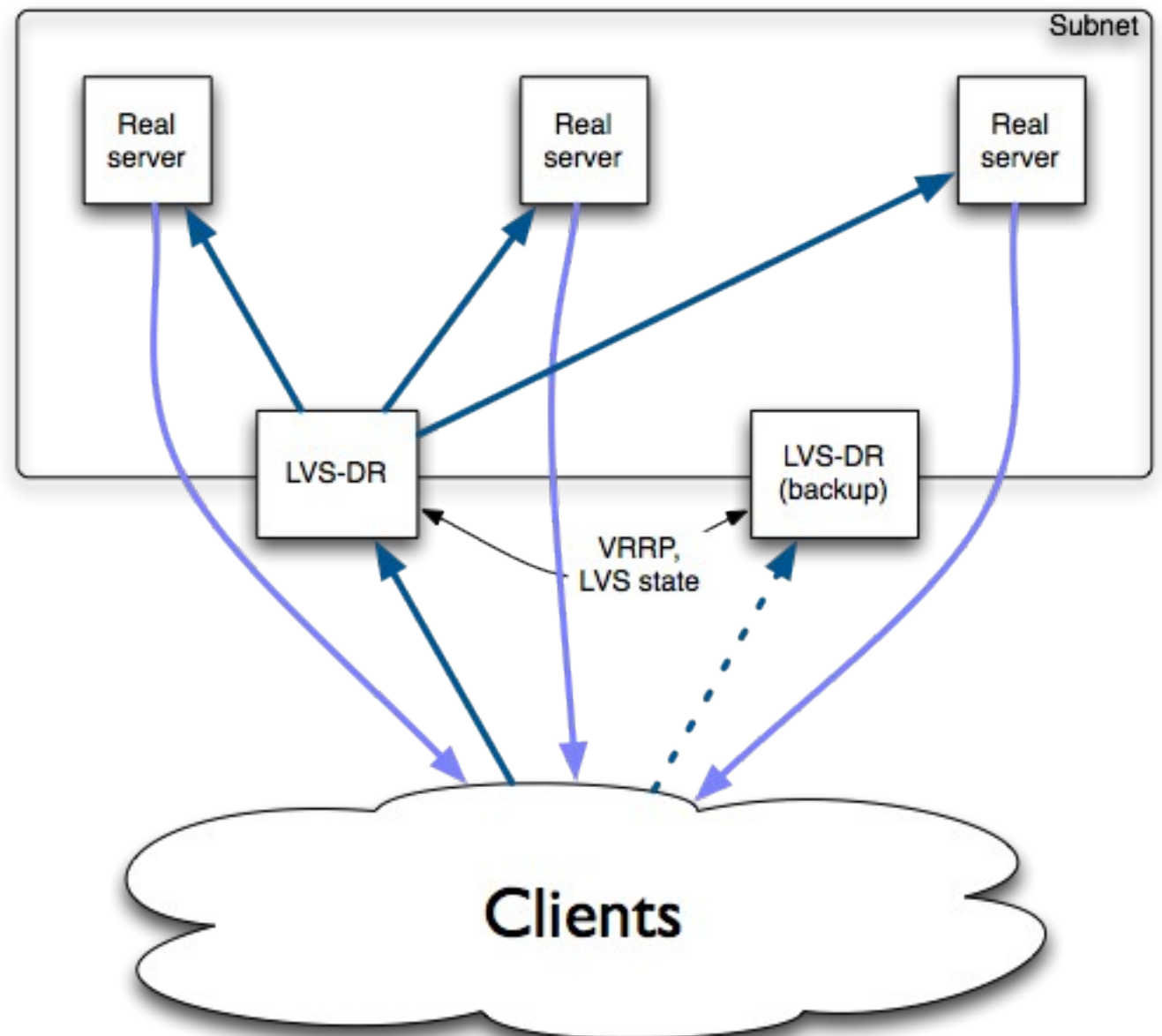
# Varnish Caching

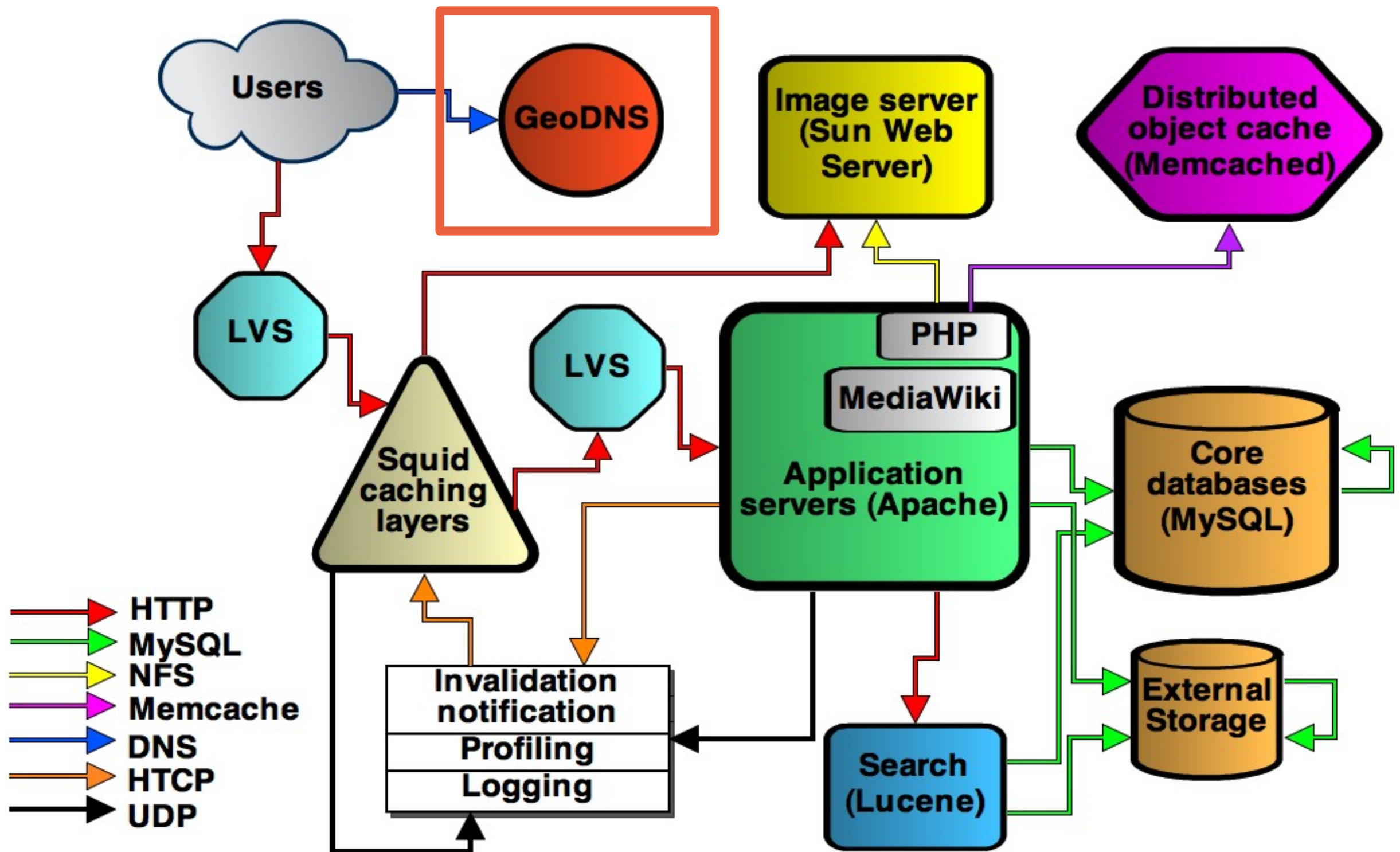
- 2-3 times more efficient than Squid
- Will eventually replace the Squid architecture
- Currently used for delivering static content such as javascript and css files (bits)



# Load Balancing: LVS-DR

- Linux Virtual Server
- Direct Routing mode
- All real servers share the same IP address
- The load balancer divides incoming traffic over the real servers
- Return traffic goes directly!

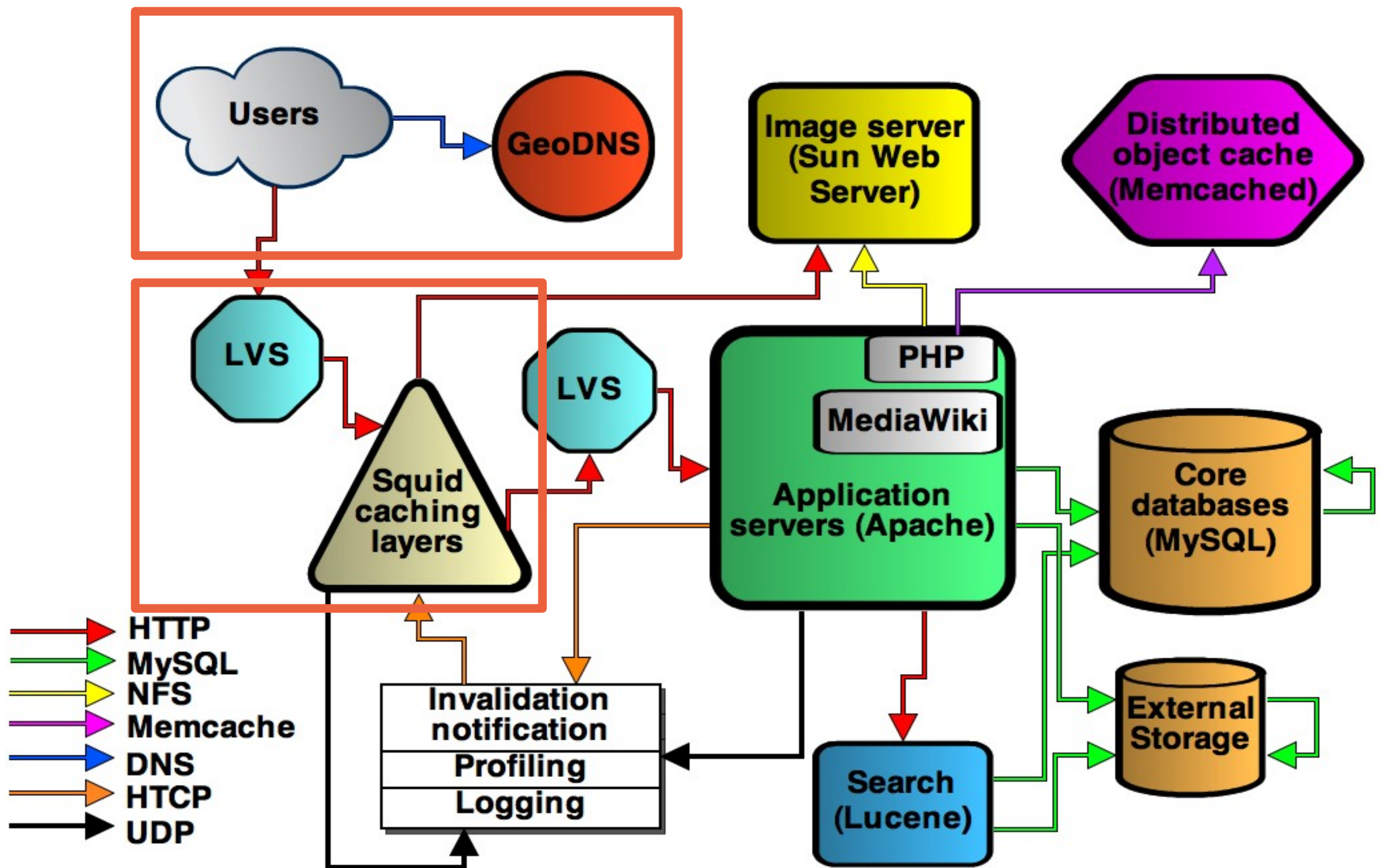




# Content Distribution Network (CDN)

- 2 clusters on 2 different continents:
  - Primary cluster in Tampa, Florida
  - Secondary caching-only cluster in Amsterdam
- Adding a new primary datacenter in Virginia

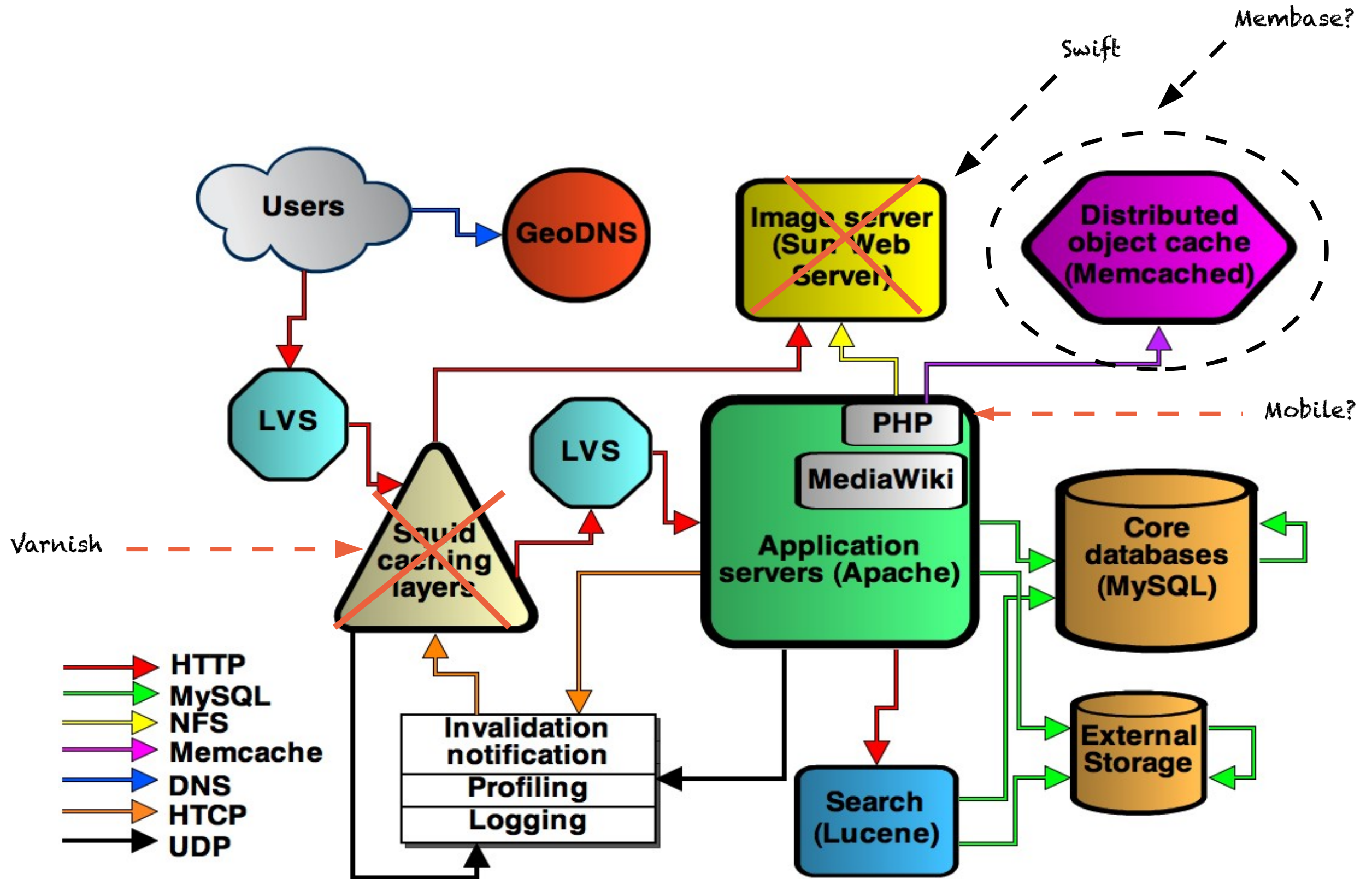




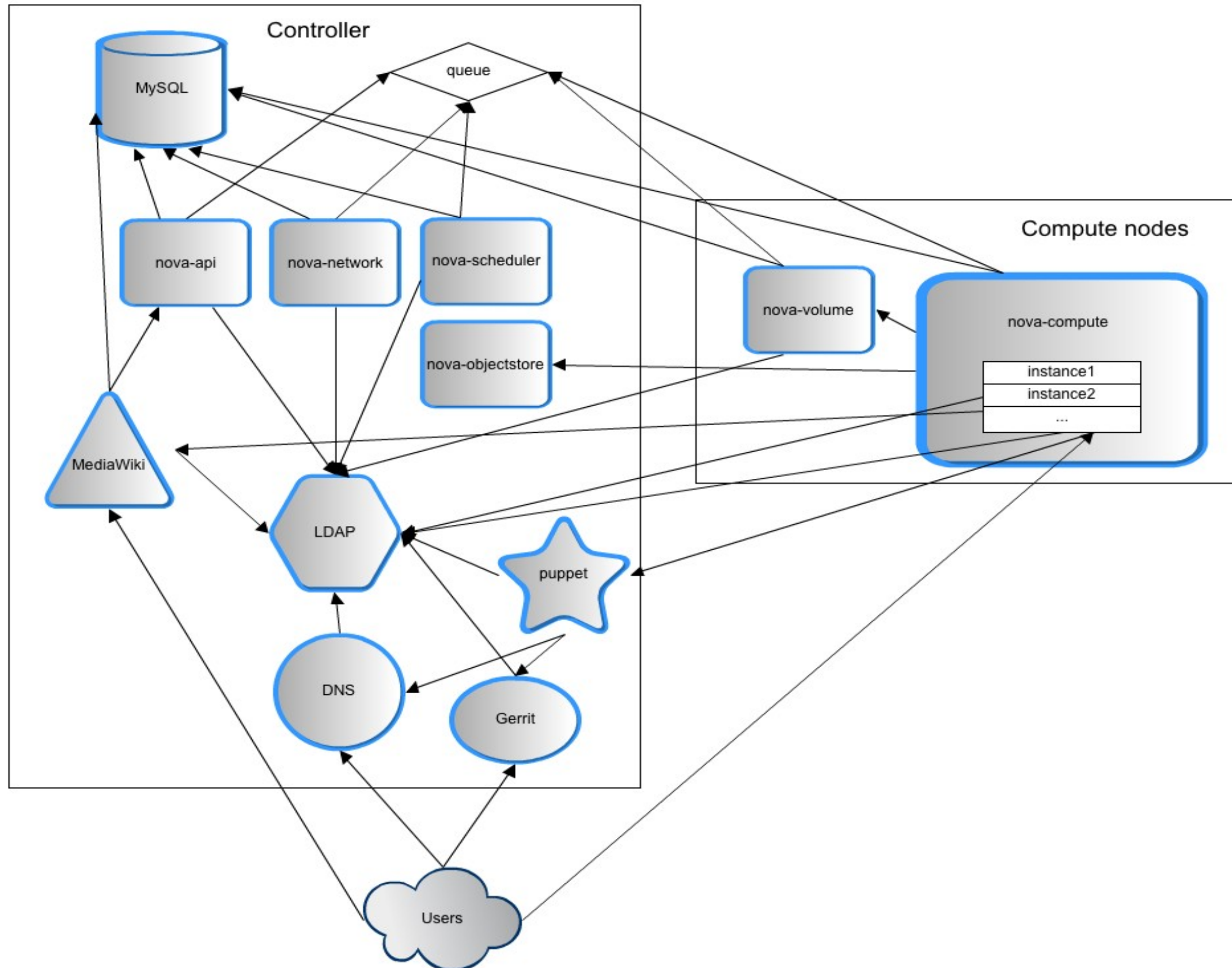
# Geographic Load Balancing

- Most users use DNS resolver close to them
- Map IP address of resolver to a country code
- Deliver CNAME of close datacenter entry based on country
- Using PowerDNS with a Geobackend

# The Site Architecture You Can Edit



# Test/Dev Architecture





# Basic use case

- Ops makes initial default project
  - Clone of production cluster
  - Used for most test/dev
- New projects mirror community or foundation initiatives
  - Devs build architecture in new project
  - Devs request merge for puppet changes via gerrit
  - Project instances moved to default project and tested
  - Project moved to production cluster

# How to engage the community

- Discuss
- Commit
- Participate



# How to engage the community

- Document
- Communicate changes

# Our philosophy

- Engage early
- Release early, release often
- Scratch your own itch

# Coding for WMF: Security

- Security is important. ***Really.***
- People rely on developers to write secure code, so:
  - An insecure extension in SVN...
  - An insecure extension on Wikipedia...

# Common vulnerabilities to avoid

- SQL injection
- Cross site scripting (XSS)
- Cross site request forgery (CSRF)
- Register Globals

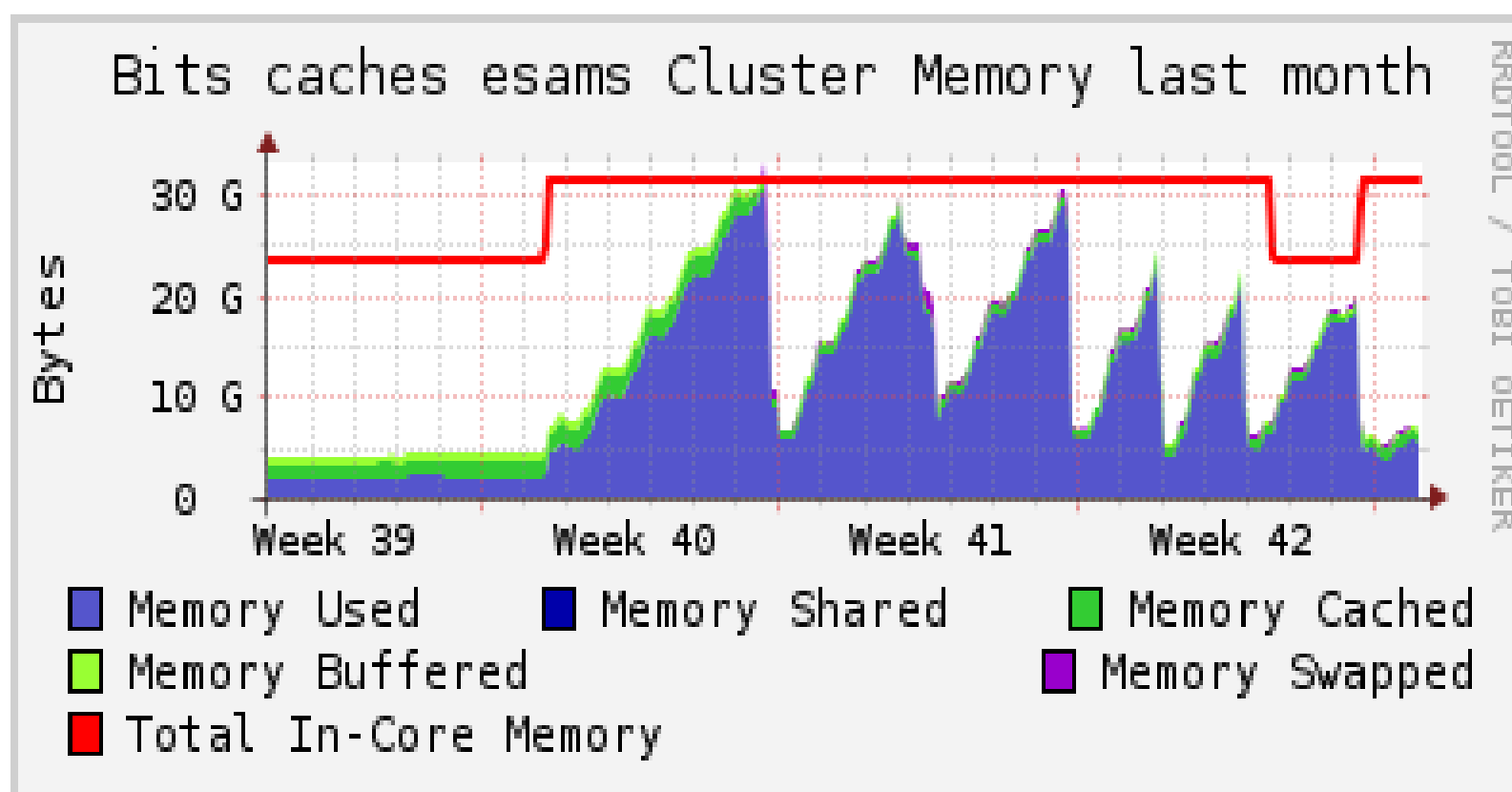
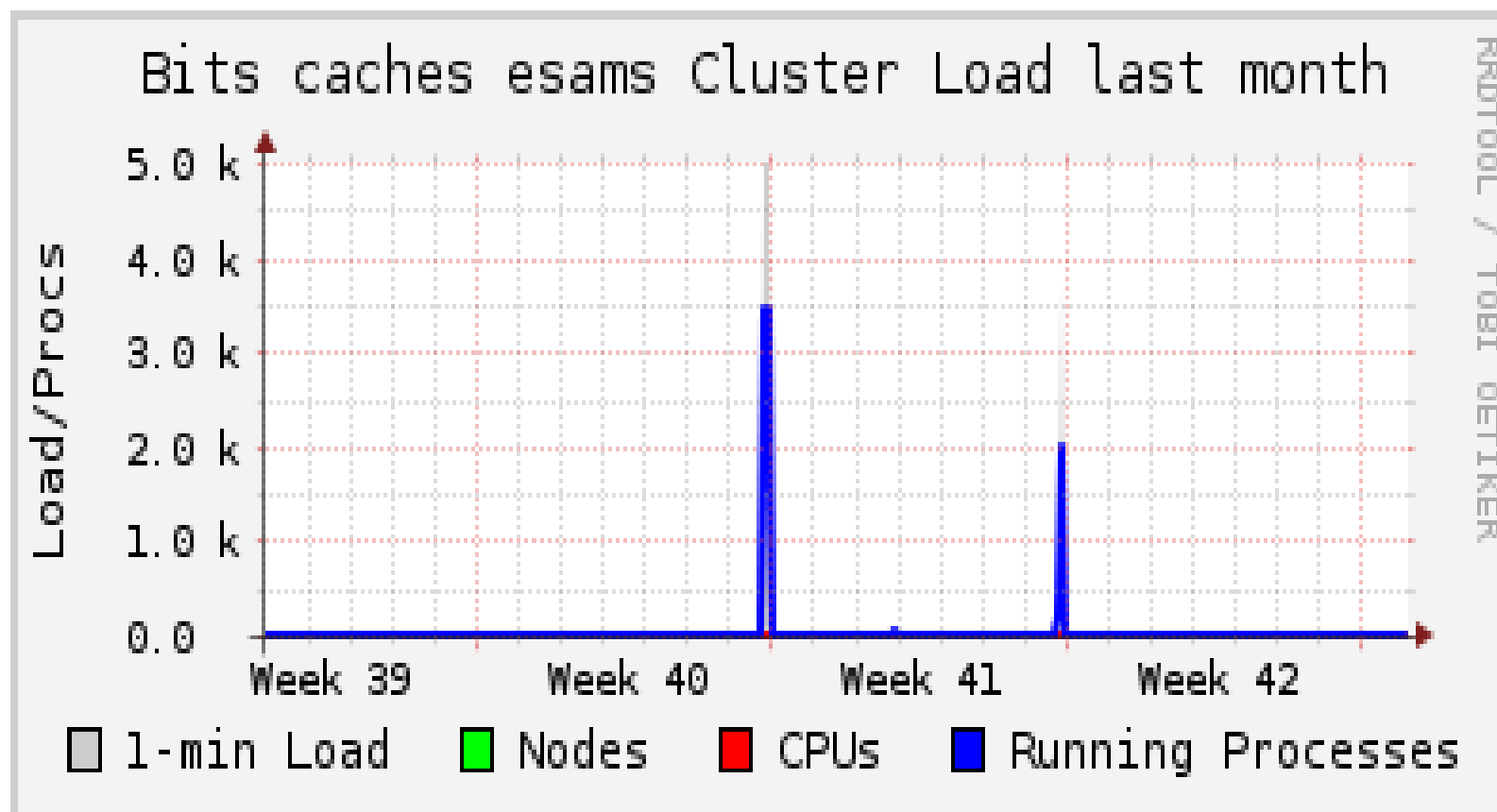
# General notes on security

- Don't trust *anyone*
- Sanitize all input
- Write code that is *demonstrably secure*
- Best of all: try to break and hack your own code

# Coding for WMF: Scalability and performance

- Wikimedia sites are huge
  - 5<sup>th</sup> most visited web presence
- Code must be:
  - Performant
  - Scalable





# Coding for WMF: Scalability and performance

- Cache
- Profile
- Optimize
- Ask for advice!

# Coding for WMF: Concurrency

- Assume a clustered architecture, *always*
- Your code will run concurrently
  - It can result in strange bugs

# Closing notes

- We rely heavily on open source
- Always looking for efficiencies
- Looking for more efficient management tools
- Looking for more contributors

# Questions, comments?

- E-mail: Ryan Lane <[ryan@wikimedia.org](mailto:ryan@wikimedia.org)>
- IRC: Freenode, Nick: Ryan\_Lane, Channels: #wikimedia-tech, #wikimedia-operations, #mediawiki, #openstack

# Communication resources

- Mailing lists
  - [http://www.mediawiki.org/wiki/Mailing\\_lists](http://www.mediawiki.org/wiki/Mailing_lists)
  - Important lists:
    - mediawiki-l: A MediaWiki support list
    - wikitech-l: A MediaWiki developer's list
    - mediawiki-api: A MediaWiki developer's list for the API
- IRC channels (on freenode)
  - #mediawiki: A MediaWiki support channel



# Developer resources

- [http://www.mediawiki.org/wiki/Developer\\_hub/ja](http://www.mediawiki.org/wiki/Developer_hub/ja) - developer hub
- Developer hub: lists resources, guidelines, and code documentation
- [http://www.mediawiki.org/wiki/How\\_to\\_become\\_a\\_MediaWiki\\_hacker/ja](http://www.mediawiki.org/wiki/How_to_become_a_MediaWiki_hacker/ja)
- How to become a MediaWiki hacker: introduction into how to do MediaWiki development
- [http://www.mediawiki.org/wiki/Security\\_for\\_developers](http://www.mediawiki.org/wiki/Security_for_developers)
- Security for developers: essential security documentation

# Developer resources

- [http://www.mediawiki.org/wiki/Manual:Coding\\_conventions/ja](http://www.mediawiki.org/wiki/Manual:Coding_conventions/ja)
- Coding conventions: conventions required for all Wikimedia run software
- <http://www.mediawiki.org/wiki/Localisation/ja>
- Localisation: resources to write code that can be easily localised
- [http://www.mediawiki.org/wiki/Code\\_review\\_guide](http://www.mediawiki.org/wiki/Code_review_guide)
- Code review guide: how your code will be reviewed before inclusion

When I was at the job I had previous to the WMF, I used the presentations and documentation given by WMF to scale the web architecture there. The information was great, and it was one of the most open architectures I've ever gotten to read about. It made things really easy to follow, especially since a lot of the configuration files were shared. I'm glad to be able to now share this information with others as well.

## Topics

- Intro
- Our technical operations
- Global architecture
- Application servers
- Storage
- Caching
- Load balancing
- Content Delivery Network (CDN)
- The site architecture you can edit
- Community involvement

This presentation is very time limited, so I can't fit in all the topics I'd like to discuss, and won't be able to go into as much detail as I'd like, Please talk to me afterwards, or discuss this with me via email or IRC.

Note from this chart that WMF is the 5<sup>th</sup> largest web presence in the world. We are close to the others in terms of users and traffic, but have way less resources and employees, and have nowhere near the number of servers.

The numbers shown for employees is for **all** employees, not ops staff. We have far fewer operations engineers.

## Our operations

- Currently managed by ~6 ops engineers
- Historically ad-hoc, "fire fighting mode"
- Technical staff spread out globally
- Always someone awake...but no on-call
- Working to engage community operations contributors

In fact, we only have around 6 ops people, including volunteers.

Since we have so few ops people, we are constantly in fire fighting mode. This is less so now that our capacity is starting to catch up to the amount of traffic we get, thankfully.

We have ops people from all over the world:

Netherlands

Australia

England

United States

No one in Asia right now!!

We always have someone awake, but no one is on call, so outages are problematic. Even if someone is awake during an outage, that doesn't necessarily mean they are available to help. They could be on vacation, or very sick, or any other number of things. Usually this means we have to wake someone up when outages occur.

Waking someone up is fairly normal now anyway, since different people have different domain specific knowledge. More employees and contributors should help with this.

We can use help! If you are interested in helping, please let us know. We have a lot of interesting work.



## Operations communication

- Most communication public on IRC
- Documentation in a wiki (<http://wikitech.wikimedia.org>)
- Sensitive communication via private lists and resource trackers

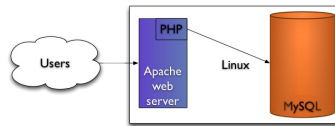
We're a small group that knows each other well. So it is fairly easily communicate. We make many decisions by consensus, but often decisions are made by whomever has the right domain specific knowledge.

We communicate primarily via IRC. This makes it easy to communicate across timezones and easy to see information that might have been missed while asleep or away, by reading backscrolls.

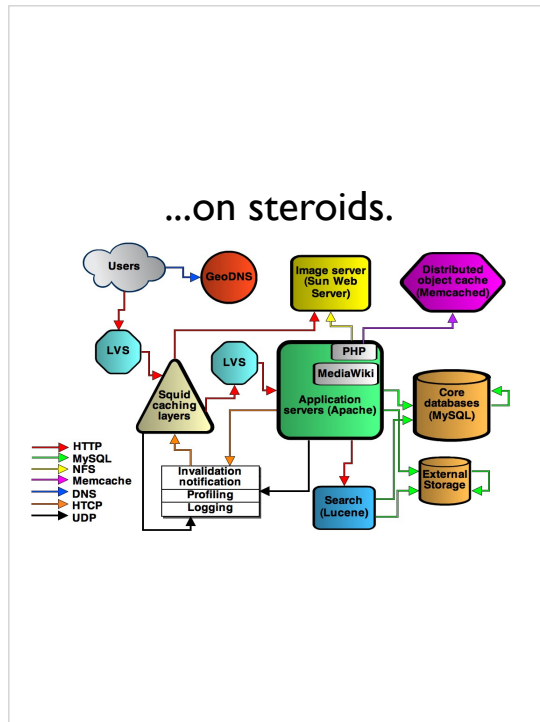
We keep our documentation in a public wiki (<http://wikitech.wikimedia.org>). Though our documentation isn't great, and not everything is documented, this gives a good overview of our architecture, and also goes fairly in depth on certain topics. This is a good site to study our architecture.

Though we'd like to discuss everything publicly, certain things like security vulnerabilities and vendor quotes can't be discussed publicly. We do sensitive communications via email lists and our closed resource tracker.

## Architecture: LAMP...



Our basic architecture is a LAMP stack. We've added a number of clever hacks over time due to our site's historic exponential growth.



This is a mostly full diagram of our architecture. Some things, like mobile, pdf servers, video renderers, and such, aren't listed since I haven't had time to update it lately though.

This diagram is complicated; however, I will break it down and explain each part separately

Our application servers are the core of our architecture. Every other thing in our architecture is built around serving this.

## The Wiki software

- All Wikimedia projects run on a MediaWiki platform
- Designed primarily for Wikimedia sites
- Open Source PHP software (GPL)
- Very scalable, very good localization

All WMF sites run MediaWiki as their application

Sites operate differently because of the extensions installed. MediaWiki is very extendable! It is also very scalable and localizable.

Runs the 5<sup>th</sup> largest web site  
Over 300 languages supported

The MediaWiki software is open source, and maintained by us; we have roughly 300 contributors

## MediaWiki optimization

- We optimize by...
- caching expensive operations
- focusing on the hot spots in the code (profiling!)
- If a MediaWiki feature is too expensive, it doesn't get enabled

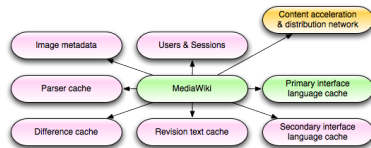
Caching and profiling are our main ways of optimization. Move of the architecture in this presentation is for caching. Most of the rest is optimizations made by noticing trends in profiling data.

If we determine that a feature is simply too expensive, even with caching and optimization, it simply doesn't get enabled



## MediaWiki caching

- Caches everywhere
- Most of this data is cached in Memcached, a distributed object cache



We have caches all over the codebase

User sessions  
Internationalized interface messages  
Parser objects  
Diff caches  
Metadata caches  
Content caches  
Image caches  
Static file caches  
etc...

We use APC for PHP opcode caching. PHP is an interpreted language, and every time a code block is run, PHP needs to interpret the code, and turn it into bytecode that can be run by the system. APC caches the bytecode and places it into shared memory so that the next time it is run, it runs the bytecode directly.

Memcache holds most of MediaWiki's caches. Memcache is an in-memory distributed object store with a simple interface

Get  
Set  
Modify

It is installed on a subset of our application servers

# MediaWiki profiling

<http://noc.wikimedia.org/cgi-bin/report.py>

[\[enwiki\]](#) [\[thumb\]](#) [\[dewiki\]](#) [\[bigpage\]](#) [\[enwiki\]](#) [\[others\]](#) [\[taggedrevs\]](#) [\[ showing 50 events, show more \]](#)

name	count	cpu%	cpu%	real%	real%
PPFrame_DOM:expand	2777300471	409	1.8	322	1.89
Parser::braceSubstitution	478045780	340	8.66	266	9.07
-total	7216450	100	169	100	226
Parser::braceSubstitution-ptime	453314242	97.8	2.63	76.5	2.75
MediaWiki::performRequestForTitle	3445759	74.4	263	71.7	339
Parser::internalParse	6879781	78.4	139	61.7	146
Parser::replaceVariables	76312501	71.4	11.4	56.4	12
MediaWiki::performAction	1329950	65.8	604	54.8	671
Parser::parse	3242613	65.5	246	52.4	263
Article::view	956646	57.6	735	46.8	797
Parser::braceSubstitution-setup	478043151	53.9	1.38	41.9	1.43
Parser::parse-Article::getOutputFromWikitext	304682	49.8	1.99e+03	39.6	2.11e+03
Parser::argSubstitution	690835928	46	0.813	36.4	0.858
api.php	3720263	13.1	42.8	17.9	78.4
API:main	3720255	12.8	41.8	17.6	77.2

MediaWiki's profile runs live at all times on WMF sites. This lets us see hot spots in our code under real conditions.

Our profiling data is sent via UDP to our UDP logging daemon. This is a core feature of MediaWiki, you can use this too! Note that you aren't required to use the UDP logging daemon. You can also write profiling data out to a file, or to a database.

Our relational databases are MySQL.

So far we are looking at a pretty traditional LAMP stack.

Like our application servers, we add a little complexity to make the database servers more scalable.

## Core databases

- One master, many replicated slaves
- Load balanced reads to slaves, write operations to master
- Separate database per wiki
- Separate big, popular wikis from smaller wikis (sharding)

Per database cluster we have one master and many replicated slaves

We do load balanced reads to slaves, and write operations to the master

- The master is used for some read operations in case the slaves are not yet up to date (*lagged*)

We have a separate database per wiki – note: not a separate server per wiki!

We scale by:

- Separating read and write operations (master/slave)
- Separating some expensive operations from cheap and more frequent operations (query groups)
- Separating big, popular wikis from smaller wikis (sharding)

This improves caching: temporal and spatial locality of reference and reduces the data set size per server

The image servers are currently the only non-open source solution in our infrastructure.

They hold all of our uploaded media, and are using the ZFS filesystem on Solaris.

## Media storage

- Current solution is not scalable
- Replacing with an open source distributed file system
- Likely OpenStack Swift

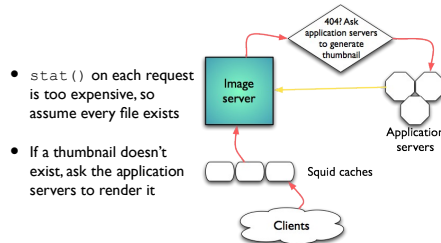
### Currently:

- 1 Image server containing ~8 TB of data
- Overloaded serving just 100 requests/s
- Media uploads and deletions over NFS, mounted on all application servers
- Not scalable
- Backups take ~ 2 weeks

### Future plans:

- A distributed file system accessed via a rest api
- Currently testing OpenStack Swift for this

## Thumbnail generation



Thumbnail generation is my favorite hack.

We noticed that `stat` operations were simply too expensive to handle, so we just assume every file exists.

We use a 404 handler in the web server to generate the thumbnail and return it. This eliminates the need for `stat`, and lets us deliver more requests per second.



Squid caching is the heart of our scalability. It's what lets us do things so cheaply, and with so few servers.

Our application's use case is what lets us get away with relying so heavily on caching. As mentioned earlier, we have 400 million users, and only about 200,000 of them are editors. This means most of our traffic is reads, which is easy to optimize.

## Squid caching

- Caching reverse HTTP proxy
- Serves most of our traffic
- Split into two groups
- Hit rates: 95% for Text, 98% for Media, since the use of CARP

HTTP reverse proxy caching implemented using Squid

Split into two groups with different characteristics

- 'Text' for wiki content (mostly compressed HTML pages)
- 'Media' for images and other forms of relatively large static files

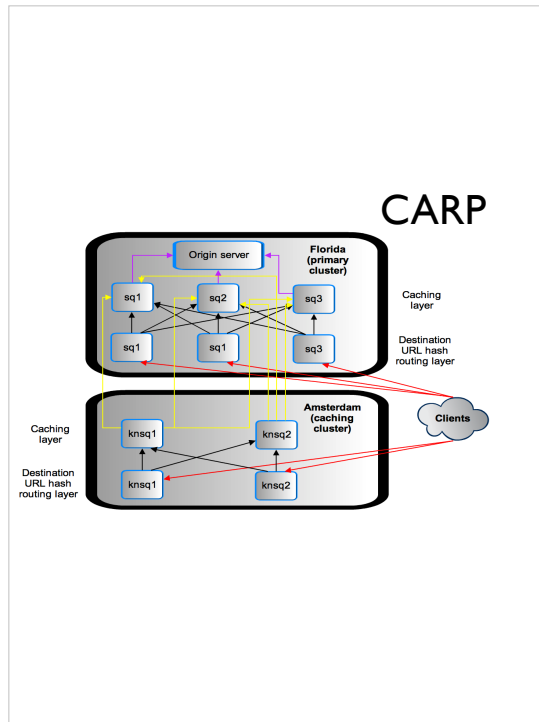
Up to 40 GB of disk caches per Squid server

Disk seek I/O limited

- The more disk spindles, the better!
- Up to 4 disks per server (1U rack servers)

8 GB of memory, half of that used by Squid

Hit rates: 95% for Text, 98% for Media, since the use of CARP algorithm



With CARP, when a user requests a resource, the CARP squids make a hash of the URL, and depending on the number of the hash, it'll send the request to a specific group of backend squid servers

The backend squid servers in the caching cluster also use CARP to directly access the backend servers of the primary datacenter if they do not have the resource available

If none of the squids have the resource available, the backend squids in the primary datacenter pull and cache the resource from the application servers, which dynamically create the resource.

## Squid cache invalidation

- Wiki pages are edited at an unpredictable rate
- Users should always see current revision
- Invalidation through expiry times not acceptable
- Purge implemented using multicast UDP based HTCP protocol

Wiki pages are edited at an unpredictable rate. Only the latest revision of a page should be served at all times in order to not hinder collaboration. Invalidation through expiry times are not acceptable, explicit cache purging needs to be done.

This is implemented using the UDP based HTCP protocol: on edit, application servers send out a single message containing the URL to be invalidated, which is delivered over multicast to all subscribed Squid caches

Squid was originally written to be a forward proxy, and has been beaten into submission over the years to be a reverse caching proxy.

Varnish is written specifically to be a reverse caching proxy, and as such, it is 3-4 times more efficient than squid. We are currently using it for our bits cache. Varnish will slowly replace squid over time in our architecture.

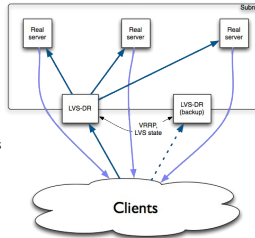
We are currently using varnish for a third caching group: 'Bits' for small static files like javascript, css, and interface images

Every service that we have is clustered. Some clustering, like MySQL and Memcached, is handled at the application level, but other clusters need to be handled at the network level.

To load balance our Apaches and Squids, we use LVS.

## Load Balancing: LVS-DR

- Linux Virtual Server
- Direct Routing mode
- All real servers share the same IP address
- The load balancer divides incoming traffic over the real servers
- Return traffic goes directly!



LVS is a standard feature of the Linux kernel

We are using LVS in direct routing mode. Incoming packets (such as a web request) are sent to the controller, who load balances the request to one of the real servers. The real servers talk back to the clients directly.

It is very efficient. A single server can easily handle over 100 kpps

- We use our oldest single CPU Pentium 4 servers, they've proven to be reliable and are not otherwise useful

Return traffic does *not* pass the load balancer, so 10 Gbit interfaces are not needed  
But, LVS works at layer 3-4 only, which means no data inspection, and as such, no high level load balancing based on things like requests, or user information.

So far we've talked about how things work in a single data center. I briefly mentioned a second datacenter in the squid portion of the talk.

We want our sites to be fast everywhere. Our primary datacenter is in the US though. This makes our sites fast in the US, but in other places they may not be fast even if everything is working perfectly in our primary datacenter because of latency or bandwidth issues between other countries and the US.

The solution is to place the data close to the user.



## Content Distribution Network (CDN)

- 2 clusters on 2 different continents:
  - Primary cluster in Tampa, Florida
  - Secondary caching-only cluster in Amsterdam
- Adding a new primary datacenter in Virginia

We currently have two datacenters: a primary datacenter in Tampa, Florida, and a caching-only datacenter in Amsterdam

We are in the process of opening a new datacenter in Virginia. When we get that datacenter up and operational, we will never again rely on a single primary data center.

The architecture in the caching-only cluster is a much simpler subset of the architecture in our primary datacenter.

In the caching-only cluster, we have DNS, LVS, Squid, and Varnish. We only have load balancing and caches.

But the question is: how do you send nearby traffic to the caching-only cluster?

## Geographic Load Balancing

- Most users use DNS resolver close to them
- Map IP address of resolver to a country code
- Deliver CNAME of close datacenter entry based on country
- Using PowerDNS with a Geobackend

### Observations:

- Most users use a DNS resolver relatively geographically close to them
- IP address to location mapping lists are available that are at least 90-95% accurate
- Geographically close often also means close *network topology* wise

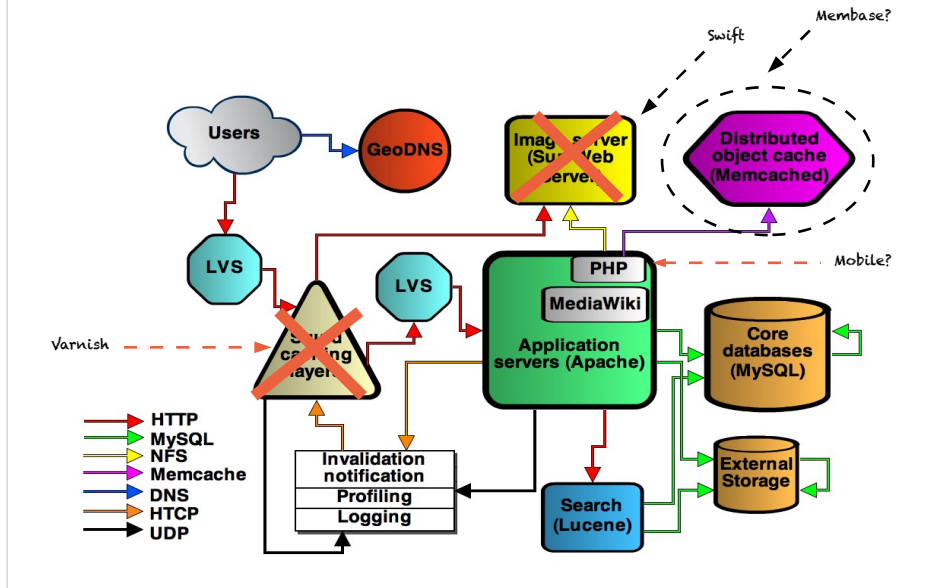
Solution: hand out DNS answers based on the estimated location of the querying DNS resolvers

Mark Bergsma wrote *Geobackend* for [PowerDNS](#) (for use by an IRC network)  
*Geobackend* reads a RBLDNS style zonefile into an efficient in-memory tree as IP map to numbers (countries)

Uses a flat file *director map* to map numbers (countries) to DNS CNAMEs. So if a user in London makes a request for en.wikipedia.org, they'll get the CNAME for the datacenter in Amsterdam, whereas if a user from New York makes a request for en.wikipedia.org, they'll get the CNAME for the datacenter in Florida.

This works pretty well for a coarsely distributed set of clusters. Since the geolookup is done by country, it wouldn't work as well for datacenters that are geographically close.

## The Site Architecture You Can [Edit](#)

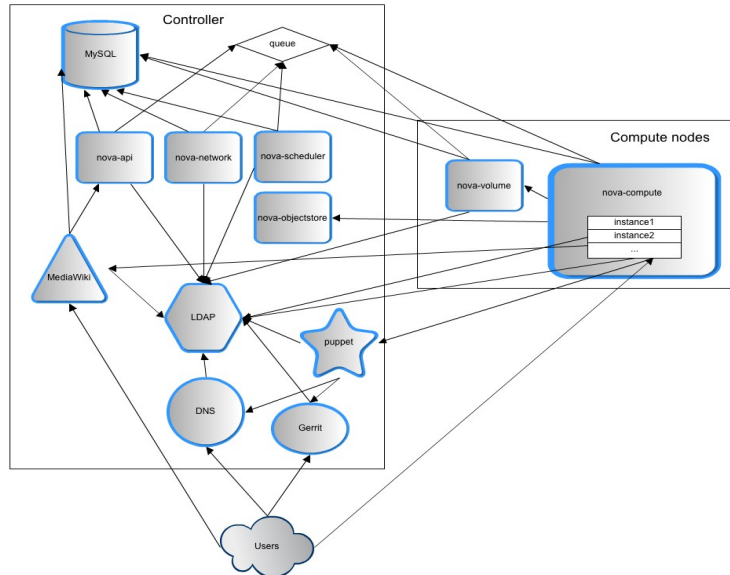


So far I've talked about what our current architecture looks like. The current architecture is the result of a number of clever hacks that have occurred on the live site. I'm building a test and development architecture so that we won't have to deploy most things directly on the live cluster.

This test/dev cluster will be virtualized, and will be designed so that it can be built by community developers, community operations engineers, staff developers and staff operations engineers.

The environment will be using OpenStack Nova, a virtualization technology similar to EC2 or RackspaceCloud.

# Test/Dev Architecture



Here's the current architecture I've built. It contains the following:

- MediaWiki, which is the user's interface for controlling most of the architecture
- LDAP, which is used for tight integration of all services, and instances
- DNS, which is controlled by MediaWiki
- Puppet, which from the instance POV is controlled by MediaWiki
- Gerrit, which is a Git interface for code review and will contain all puppet information
- Nova, which is used for managing infrastructure

## Basic use case

- Ops makes initial default project
  - Clone of production cluster
  - Used for most test/dev
- New projects mirror community or foundation initiatives
  - Devs build architecture in new project
  - Devs request merge for puppet changes via gerrit
  - Project instances moved to default project and tested
  - Project moved to production cluster

The basic use case is that the operations team will create an initial default project. This project will be a clone of our production cluster. Like our production cluster, direct root access will be limited here.

However, shell access will be given out fairly liberally. This environment will be used for most test and development. I hope for this to be used as a shared environment where staff and volunteers can collaboratively work together on projects.

New projects should mirror community or foundation initiatives. These will be used for new site architecture. For instance, we are implementing Open Web Analytics currently, and this required architecture that is separate from our normal architecture. In our production environment, it is difficult to give out root access, which made OWA

# How to engage the community

- Discuss
- Commit
- Participate

## How to engage the community

- If your software is for a Wikimedia site, begin by discussing your idea on mailing lists (ideally) or IRC
- Submit patches to bugzilla
- Get commit access
  - Submitting changes to our SVN server makes it easier for us to code review, and collaborate with you.
- Actively participate in code review
  - We have a code review tool where things are marked as “fixme” or “ok”, etc. The code review tool allows us to discuss each revision, and work towards solutions to problems with commits.
- Actively participate in mailing list threads about your software
  - Occasionally your software will be discussed on the lists. It is important to participate in these discussions. You can also make announcements to the lists about new releases of extensions, or upcoming changes to major pieces of code.

# How to engage the community

- Document
- Communicate changes

- If your software is an extension, document it on mediawiki.org on an extension page
- If you are making core code changes, update the appropriate documentation on mediawiki.org
- If you are adding hooks, add them to the hooks documentation
- Make sure you let people know what you are changing. Communication is key!
  - If you are working on a piece of core code, and are making major changes, it is very important to notify others, otherwise you may be breaking something they are working on, or may be changing something they are relying on.
- Don't be afraid of a language barrier. We have developers from all of the world, most of which english isn't their first language. We will try to always find a way to help. Recently we had a German developer in the #wikimedia channel asking for development help. I was trying my best to help, but was having problems. A german user also happened to be in the channel and helped translate for me. I was able to help the dev get through the problem they were having.
  - Being an ambassador is a great way to contribute, even if you can't help with technical changes. You can help others make these changes by helping us with the language barrier.



# Our philosophy

- Engage early
- Release early, release often
- Scratch your own itch

## •Engage early

- Let us know how you are interested in helping, and what you are working on. This lets us help you early in the process, and will make coding way easier.

## •Release early, release often

- Your code doesn't need to be totally finished to submit it. Give us your betas, your alphas, and your pre-alphas. Our appetite for code is insatiable. Check in code often, even if it isn't fully complete. This gives us the ability to review your code in increments over time, and give feedback on how it is architected.

## •Scratch your own itch

- You don't have to work on something we want. If you want to improve something, do it. This is how most of our developers operate. My itch is authentication. My first contribution was the LDAP plugin, because I wanted to use MediaWiki in my organization, but I didn't want to maintain another database of usernames and passwords. Over time I've incrementally added support for every major LDAP server on every major platform. I've written support for SSL client authentication, kerberos, Web SSO, etc. None of this is used on Wikimedia sites, but the code is used in thousands of third party wikis. Since these third parties can use the software, they can also participate in the community, and may contribute changes that help WMF and others that would not have otherwise been helped. Remember: your code can help the community even if you think it won't.

# Coding for WMF: Security

- Security is important. **Really.**
- People rely on developers to write secure code, so:
  - An insecure extension in SVN...
  - An insecure extension on Wikipedia...

•No, really, it seriously is. We are able to stay sane, even with a small team because our software is secure, and we don't have to constantly battle security problems.

•People rely on developers to write secure code, so:

- Insecure extension in SVN = security risk for unwitting third-party wiki admins and their users
- Insecure extension on Wikipedia = security risk for 300 million users
  - Also, third party sites are very likely to use extensions that are used on Wikimedia sites, as they are known to work well, be kept up to date, and are well known. This means you also cause even greater harm to third parties

# Common vulnerabilities to avoid

- SQL injection
- Cross site scripting (XSS)
- Cross site request forgery (CSRF)
- Register Globals

These are some of the most common vulnerabilities. We regularly find them in code review, and reject code submissions because of them. In general, they are fairly easy to avoid, if you follow good programming practices.

# General notes on security

- Don't trust *anyone*
- Sanitize all input
- Write code that is *demonstrably secure*
- Best of all: try to break and hack your own code

- Don't trust anyone, not even your users. *Especially* not your users.
- Sanitize all input, or ensure your output is rock solid, preferably using MediaWiki's `WebRequest` class, when possible. Unsanitized input is the cause of most vulnerabilities.
- Use tokens in forms, use the `HTMLForm` class, if possible as it does this for you to avoid CSRF
- Write code that is *demonstrably secure*. This keeps our code reviewers sane, makes your code more likely to pass review, and makes maintaining your code in a secure way much easier.
- Read *Security for developers* in the below resources slide.
- Best of all: try to break and hack your own code. Not only is this good practice, but it is also really fun! I do this daily as a developer and operations engineer and it keeps my job exciting. I like to do security reviews of other people's code for the same reason.

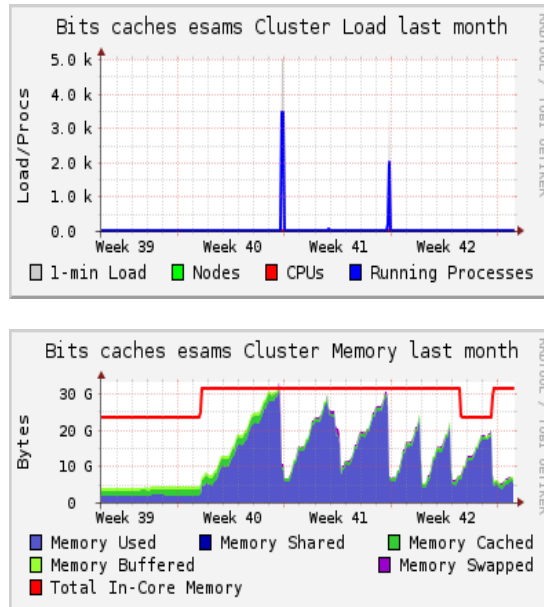
# Coding for WMF: Scalability and performance

- Wikimedia sites are huge
  - 5<sup>th</sup> most visited web presence
- Code must be:
  - Performant
  - Scalable

- Wikipedia is huge

- 50-100k requests per second
- 2-4k of those fire up MediaWiki
- enwiki alone has 20 million pages and almost 400 million revisions

- For your code to handle these conditions, you need to pay attention to scalability and performance



This is an example of the sites having problems cause of a memory leak in a patch to Varnish. At the beginning of week 40, we put in a patch to do geoip lookups. The patch had a few memory leaks. In weeks 40 and 41 we had site outages because of this because the Varnish servers ran out of memory, and went into swap. This caused a major spike in system load (over 3,000 system load).

The memory leaks themselves weren't really that large, but due to the huge number of requests occurring, the systems were leaking roughly 5GB of memory a day.

# Coding for WMF: Scalability and performance

- Cache
- Profile
- Optimize
- Ask for advice!

## • Things to keep in mind when implementing:


- Cache as much as you can
  - If it is expensive to generate, you should cache it
  - Remember though, that some things are more expensive to cache than to regenerate. Profiling will help here.
- Optimize database queries
  - Ask for advice! This is a complex subject. Even some of our best developers have problems with this occasionally. Discussing your database queries with some of our experts is a great way to learn, and will help ensure your code is performant
- Profile
  - MediaWiki has profiling functions built in that run live. Make use of them to find hot spots in your code.

# Coding for WMF: Concurrency

- Assume a clustered architecture, *always*
- Your code will run concurrently
  - It can result in strange bugs

- Always assume your code is running on a cluster. Always.
- WMF has ~180 Apache servers running multiple Apache processes
- Your code will likely run on many instances at the same time
- This can result in weird bugs you didn't/couldn't notice locally
- Remember that the database servers are clustered as well
  - Slaves generally have a lag time, so do reads from the master if the data *must* be up to date





We rely heavily on open source, and that also means we rely heavily on those open source communities. Every solution, in our infrastructure, except the image servers, is open source. When we run into a bug, either the community provides a fix, or we fix the problem, and provide the fix back to the community. This is something that isn't possible for us with proprietary solutions.

We are always looking for efficiencies in our environment. We are currently looking for better image storage, we are looking at a possible memcache replacement, we are looking at writing a kernel module for LVS to load balance LVS itself. We are also looking for better management tools so that even with our small team we'll be able to do things more efficiently and will hopefully have to deal with problems less often because of better management. If you have ideas on how we can operate more efficiently, let us know, or demo it for us. We'd be happy to have you help us.

We have historically been a team of volunteers and would love to get contributions from the community. We have a bunch of cool things to work on right now, and would love to get your help with it! If you have a solution, would like to help out, or even just have some ideas, let us know.

## Questions, comments?

- E-mail: Ryan Lane <[ryan@wikimedia.org](mailto:ryan@wikimedia.org)>
- IRC: Freenode, Nick: Ryan\_Lane, Channels: #wikimedia-tech, #wikimedia-operations, #mediawiki, #openstack

# Communication resources

- Mailing lists
  - [http://www.mediawiki.org/wiki/Mailing\\_lists](http://www.mediawiki.org/wiki/Mailing_lists)
  - Important lists:
    - mediawiki-l: A MediaWiki support list
    - wikitech-l: A MediaWiki developer's list
    - mediawiki-api: A MediaWiki developer's list for the API
- IRC channels (on freenode)
  - #mediawiki: A MediaWiki support channel

# Developer resources

- [http://www.mediawiki.org/wiki/Developer\\_hub/ja](http://www.mediawiki.org/wiki/Developer_hub/ja) - developer hub
  - Developer hub: lists resources, guidelines, and code documentation
- [http://www.mediawiki.org/wiki/How\\_to\\_become\\_a\\_MediaWiki\\_hacker/ja](http://www.mediawiki.org/wiki/How_to_become_a_MediaWiki_hacker/ja)
  - How to become a MediaWiki hacker: introduction into how to do MediaWiki development
- [http://www.mediawiki.org/wiki/Security\\_for\\_developers](http://www.mediawiki.org/wiki/Security_for_developers)
  - Security for developers: essential security documentation

# Developer resources

- [http://www.mediawiki.org/wiki/Manual:Coding\\_conventions/ja](http://www.mediawiki.org/wiki/Manual:Coding_conventions/ja)
- Coding conventions: conventions required for all Wikimedia run software
- <http://www.mediawiki.org/wiki/Localisation/ja>
- Localisation: resources to write code that can be easily localised
- [http://www.mediawiki.org/wiki/Code\\_review\\_guide](http://www.mediawiki.org/wiki/Code_review_guide)
- Code review guide: how your code will be reviewed before inclusion

MediaWiki is localized in over 300 languages.