Lecture 14:

Scaling a Web Site

Scale-out Parallelism, Elasticity, and Caching

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2014

Tunes

"Good Ol' Fashion Nightmare" (Matt & Kim)

Matt & Kim

"I think it's pretty clear what we were singing about."

- Matt & Kim

YES.
418 exams are long.

(we did warn you)

But…

In many situations.

THE QUESTIONS.

That were attempted.

had answers...
that seem to indicate...

that perhaps.

SOME

FUNDAMENTAL

CONCEPTS

may not be understood

may not be understood at the level

your instructor

might prefer.

AND SO

HERE

IS

THE DEAL.

The Exam 1 Deal

- No exam 1 solutions will be distributed at this time
- You have the opportunity to redo up to two questions (of your choosing) from the exam, on your own time.
  - You may discuss the problems with your classmates and the TAs.
  - You must write your solutions on your own.
  - You will get 50% credit for lost points on regraded questions.
  - This must be completed by April 11th

But... there's a catch.

Exam 1 Deal: The Catch

- You must hand in your solution to Kayvon at office hours

- And you are not allowed to hand in unless you are able to successfully answer a series of questions I ask you

- The questions will a subset of the seven questions on exam 1 (or simple follow up variants)

It's time to start thinking about projects

- **Timeline**
  - Project proposal due: April 4th
  - Project checkpoint: April 18th
  - Parallelism competition finals! (project presentations): May 9th
- **Ideas**
  - Pick an application, parallelize it, and analyze its performance
  - Modify a parallel library or compilation tool
  - Write a hardware simulator, play around with FPGAs, do real hardware design
  - Free to experiment with fun new parallel platforms: FPGAs, mobile devices, Tegra devkits, Raspberry Pis, etc.
- **We will be making a web page of ideas over spring break**
- **See examples from last year:**
  - http://15418.courses.cs.cmu.edu/spring2013/article/34

Today's focus: the basics of scaling a web site

- I'm going to focus on performance issues
  - Parallelism and locality

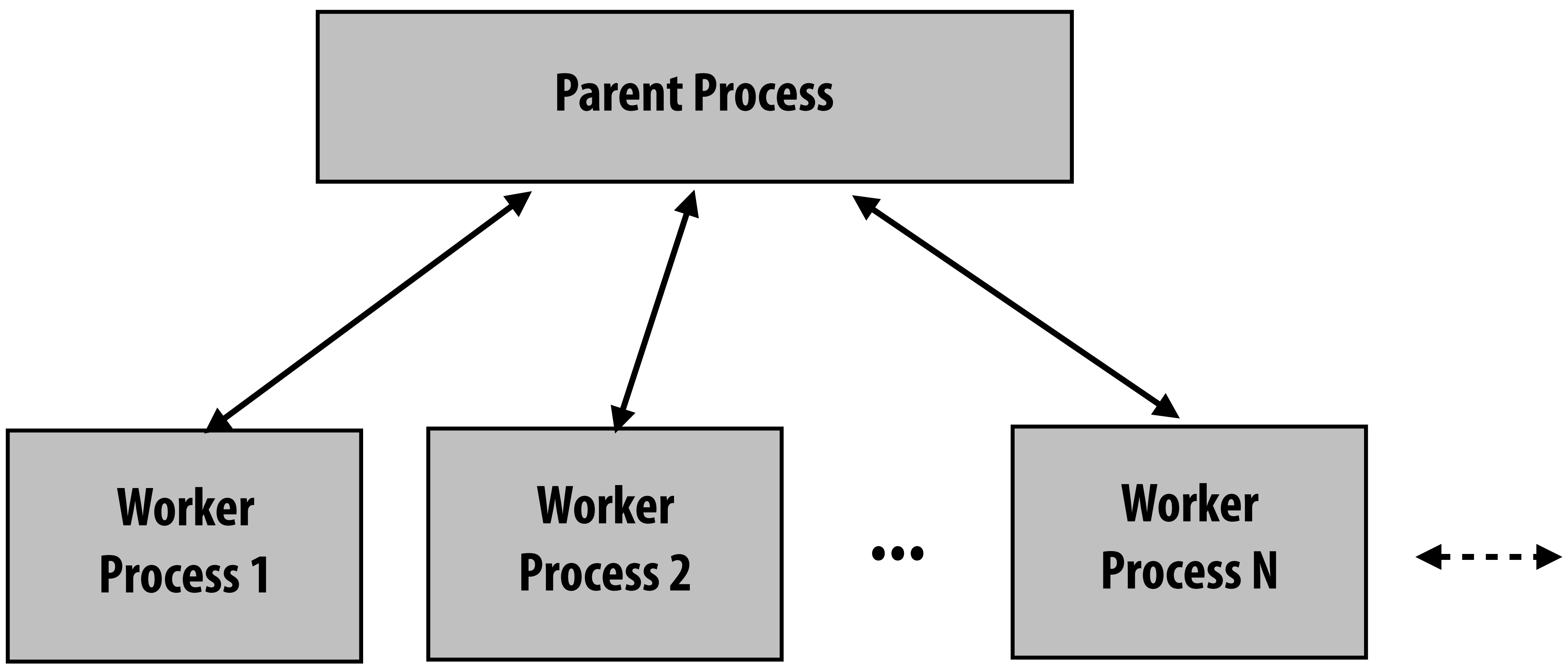
- Many other issues in developing a successful web platform
  - Reliability, security, privacy, etc.
  - There are other great courses at CMU for these topics

A simple web server for static content

```
while (1)
{

    request = wait_for_request();

    filename = parse_request(request);

    contents = read_file(filename);

    send contents as response

}
```

Question: is site performance a question of throughput or latency?
(we'll revisit this question later)

A simple parallel web server

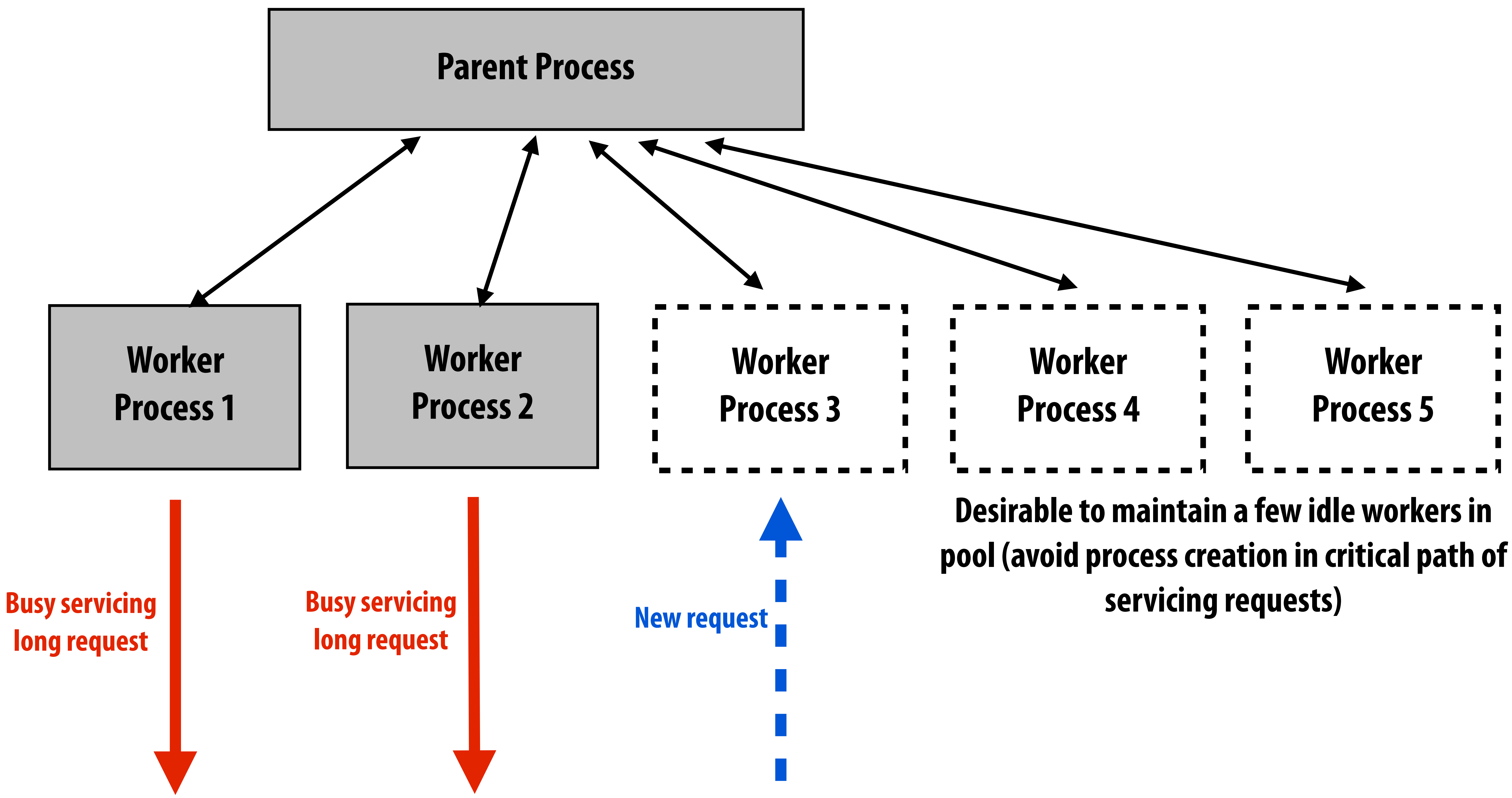



```
while (1)
{
    request = wait_for_request();

    filename = parse_request(request);

    contents = read_file(filename);

    send contents as response
}
```

What factors would you consider in setting the value of N for a multi-core web server?

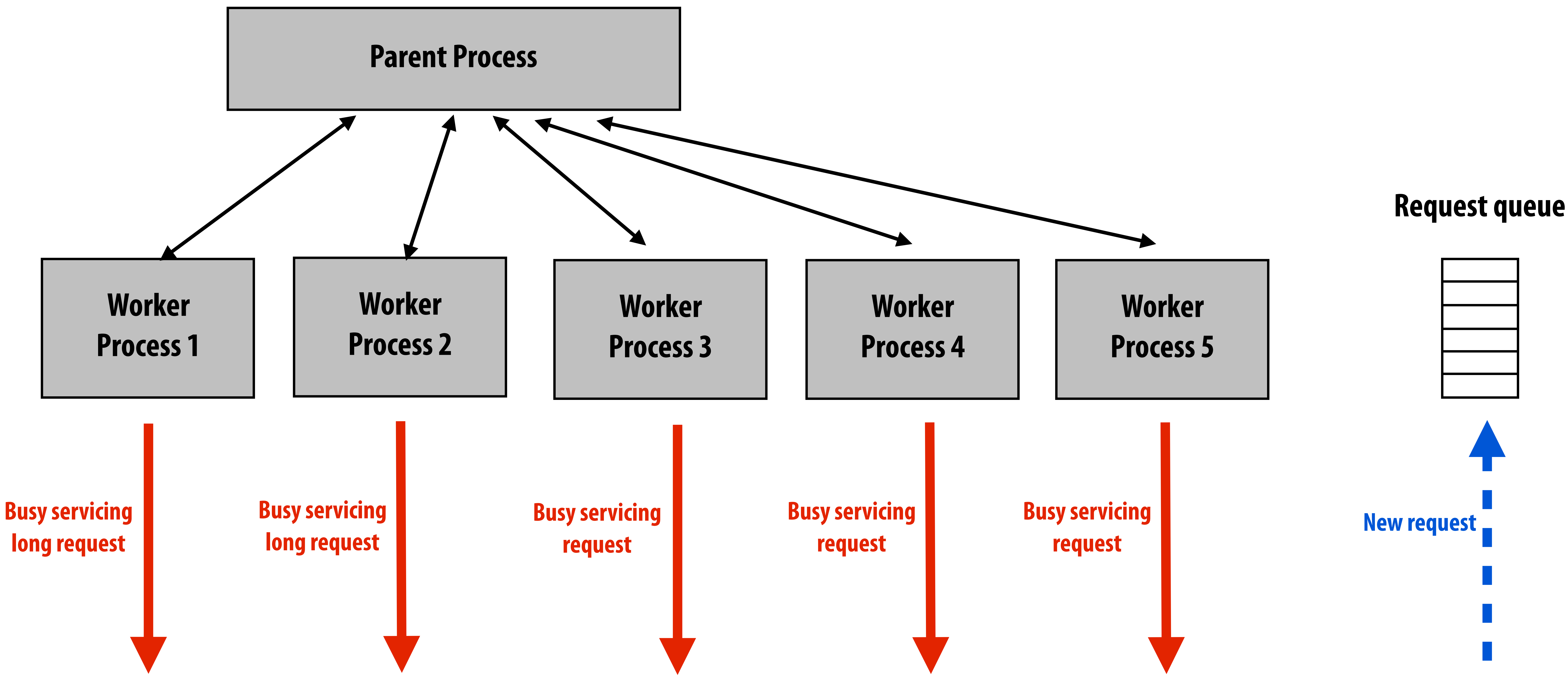
- Parallelism: use all the server's cores
- Latency hiding: hide long-latency disk read operations (by context switching between worker processes)
- Concurrency: many outstanding requests, want to service quick requests while long requests are in progress (e.g., large file transfer shouldn't block serving index.html)
- Footprint: don't want too many threads so that aggregate working set causes thrashing

Example: Apache's parent process dynamically manages size of worker pool



Desirable to maintain a few idle workers in pool (avoid process creation in critical path of servicing requests)

Limit maximum number of workers to avoid excessive memory footprint (thrashing)



Key parameter of Apache's "prefork" multi-processing module: `MaxRequestWorkers`

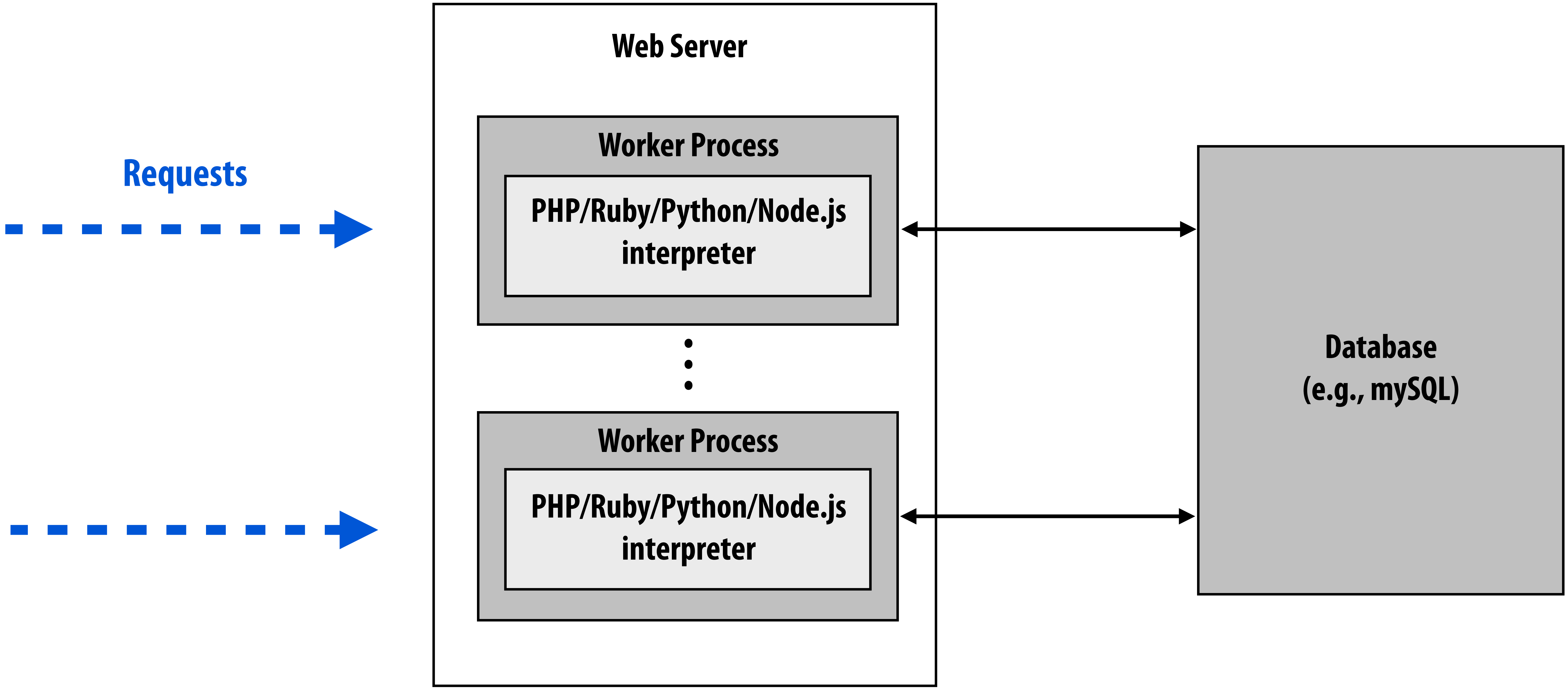
Aside: why partition server into processes, not threads?

- Protection
  - Don't want a crash in one worker to bring down the whole web server
  - Often want to use non-thread safe libraries (e.g., third-party libraries) in server operation

- Parent process can periodically recycle workers
  (robustness to memory leaks)

- Of course, multi-threaded web server solutions exist as well
  (e.g., Apache's "worker" module)

Dynamic web content



"Response" is not a static page on disk, but the result of application logic running in response to a request.

Consider the amount of logic and the number database queries required to generate your Facebook News Feed.

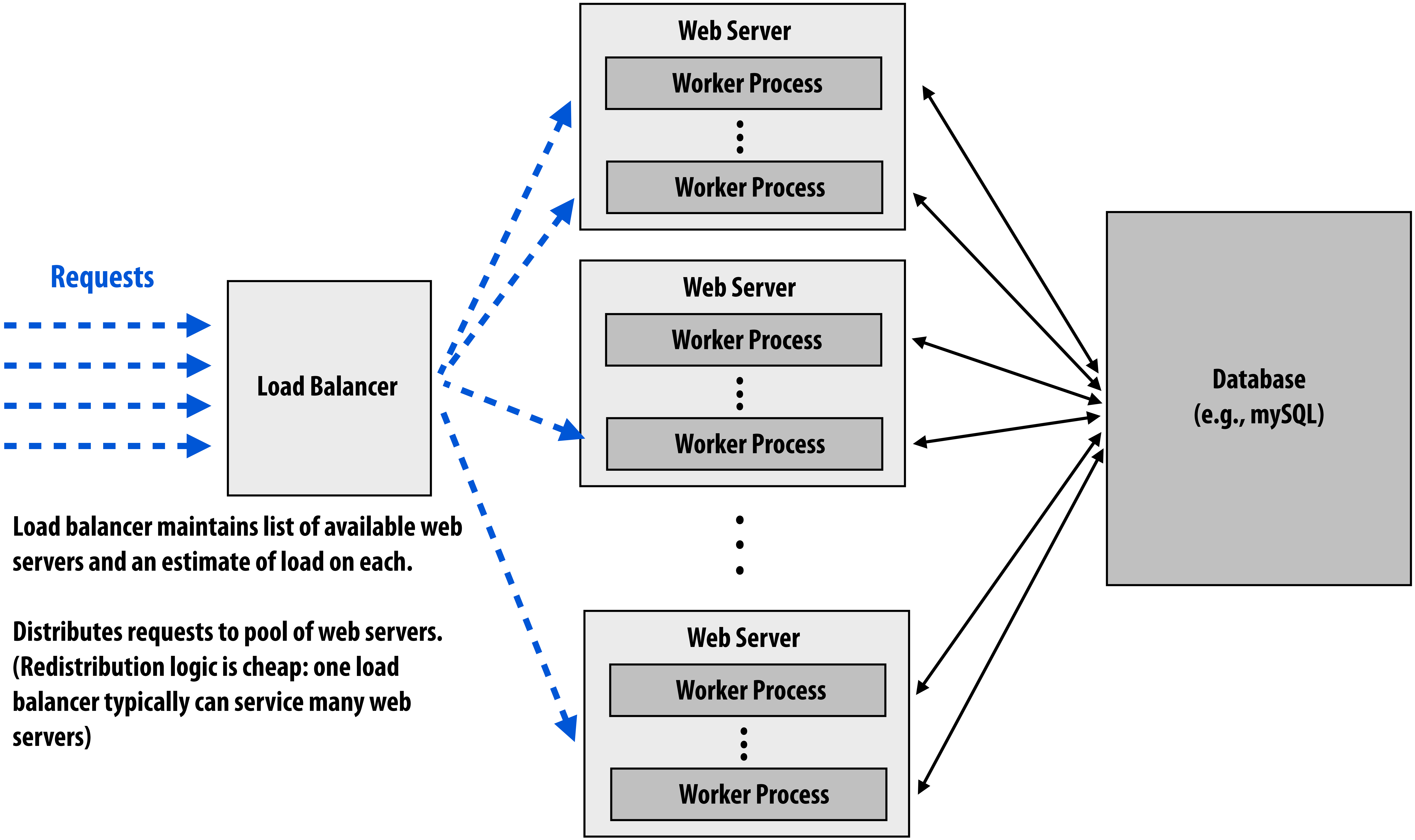
Scripting language performance (poor)

- Two popular content management systems (PHP)
  - Wordpress ~ 12 requests/sec/core (DB size = 1000 posts)
  - MediaWiki ~ 8 requests/sec/core
    [Source: Talaria Inc.]

- Recent interest in making making scripted code execute faster
  - Facebook's HipHop: PHP to C source-to-source converter
  - Google's V8 Javascript engine: JIT Javascript to machine code

"Scale out" to increase throughput

Use many web servers to meet site's throughput goals.

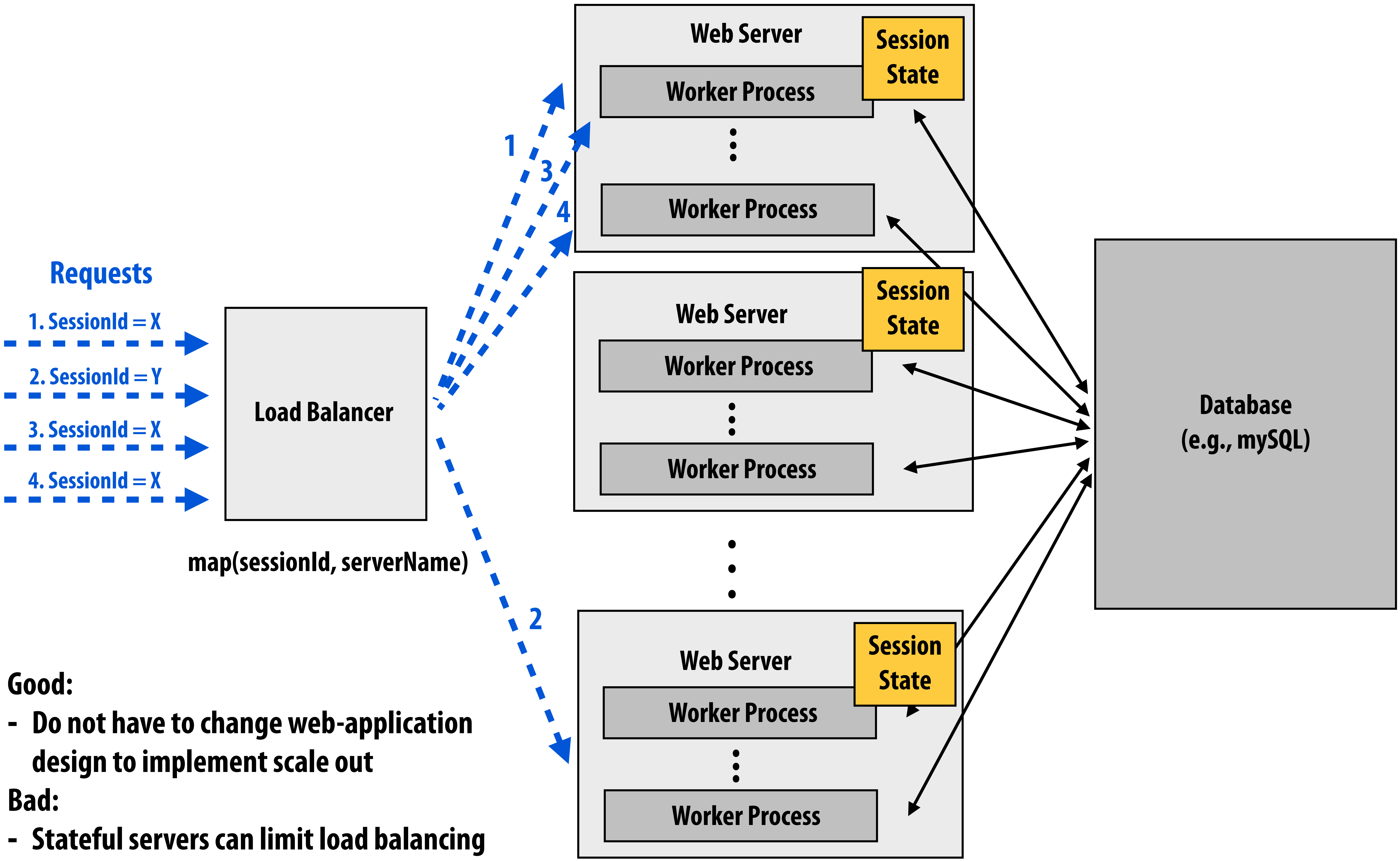


Load balancer maintains list of available web servers and an estimate of load on each.

Distributes requests to pool of web servers. (Redistribution logic is cheap: one load balancer typically can service many web servers)

Load balancing with persistence

All requests associated with a session are directed to the same server (aka. session affinity, "sticky sessions")

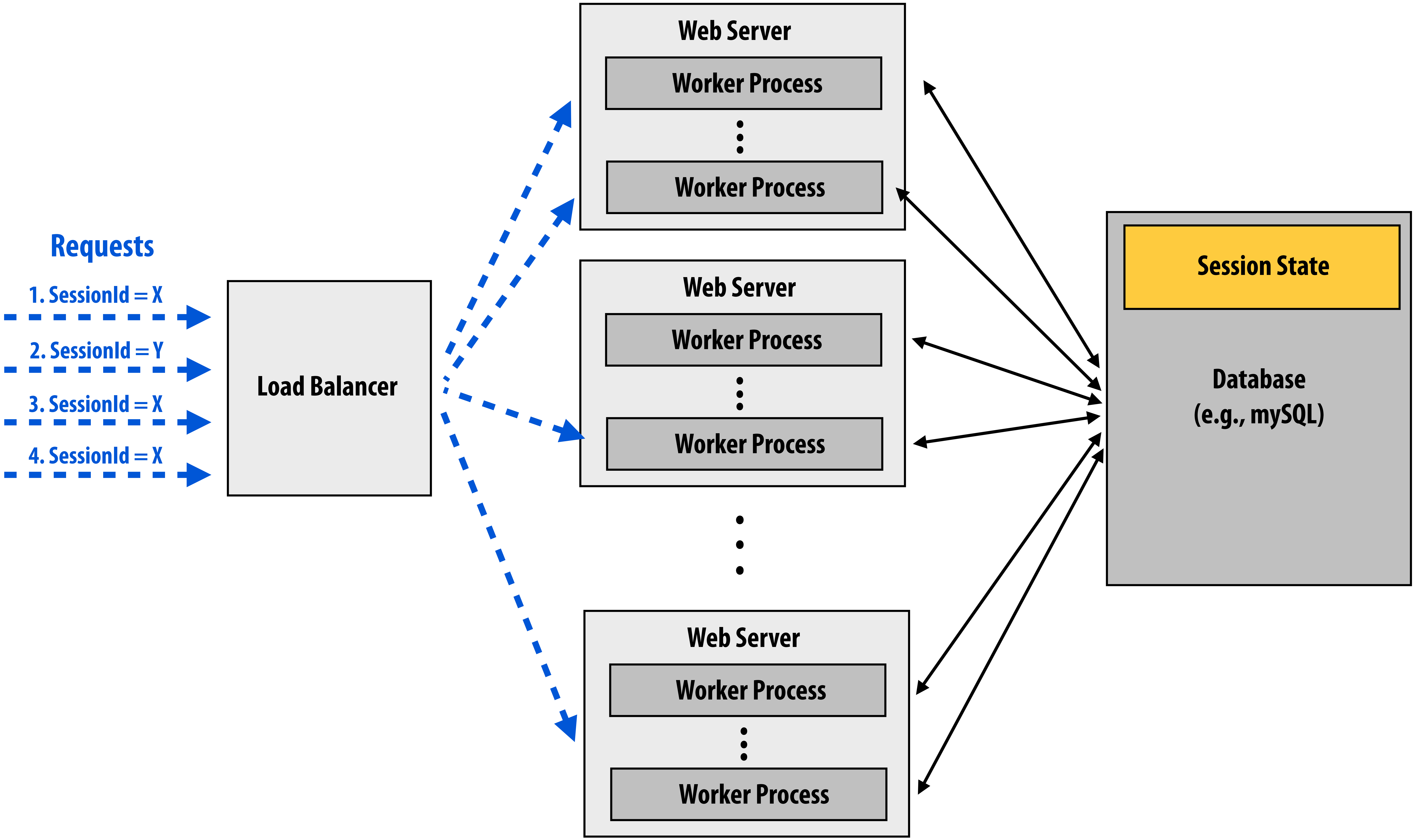


Good:
- Do not have to change web-application design to implement scale out

Bad:
- Stateful servers can limit load balancing options. Also, session is lost if server fails

Desirable: avoid persistent state in web server

Maintain stateless servers, treat sessions as persistent data to be stored in the DB.

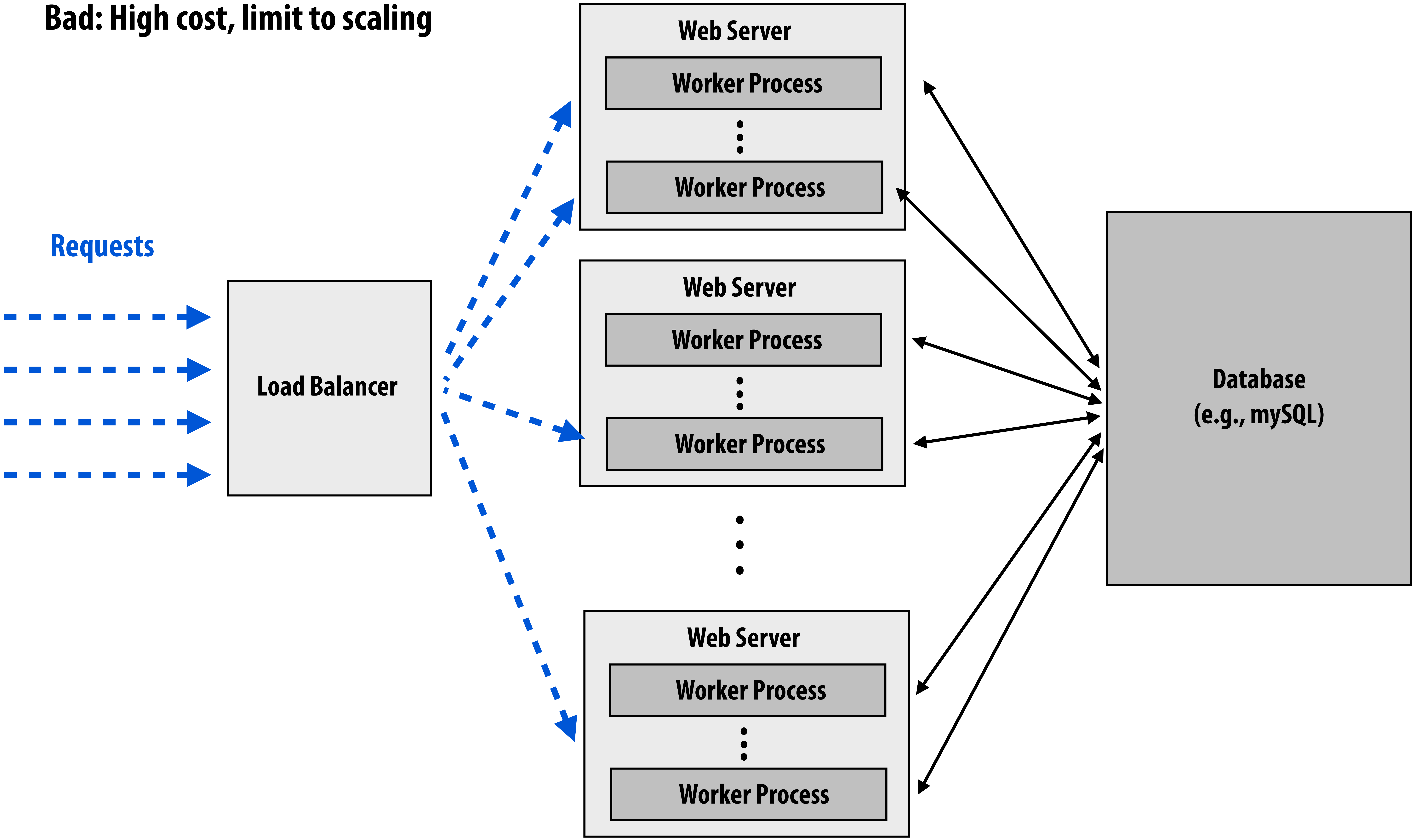


Dealing with database contention

Option 1: "scale up": buy better hardware for database server, buy professional-grade DB that scales

Good: no change to software

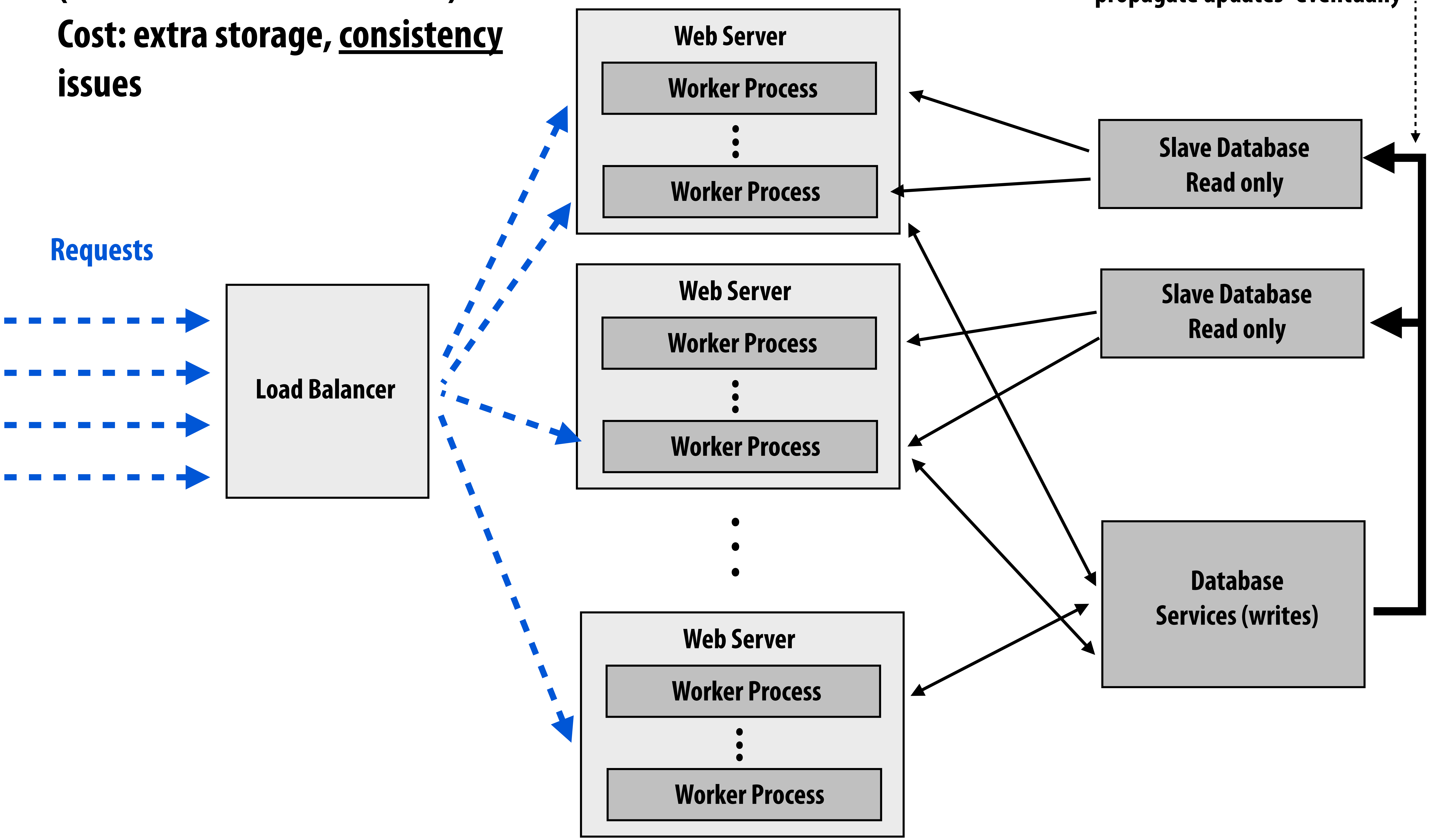
Bad: High cost, limit to scaling



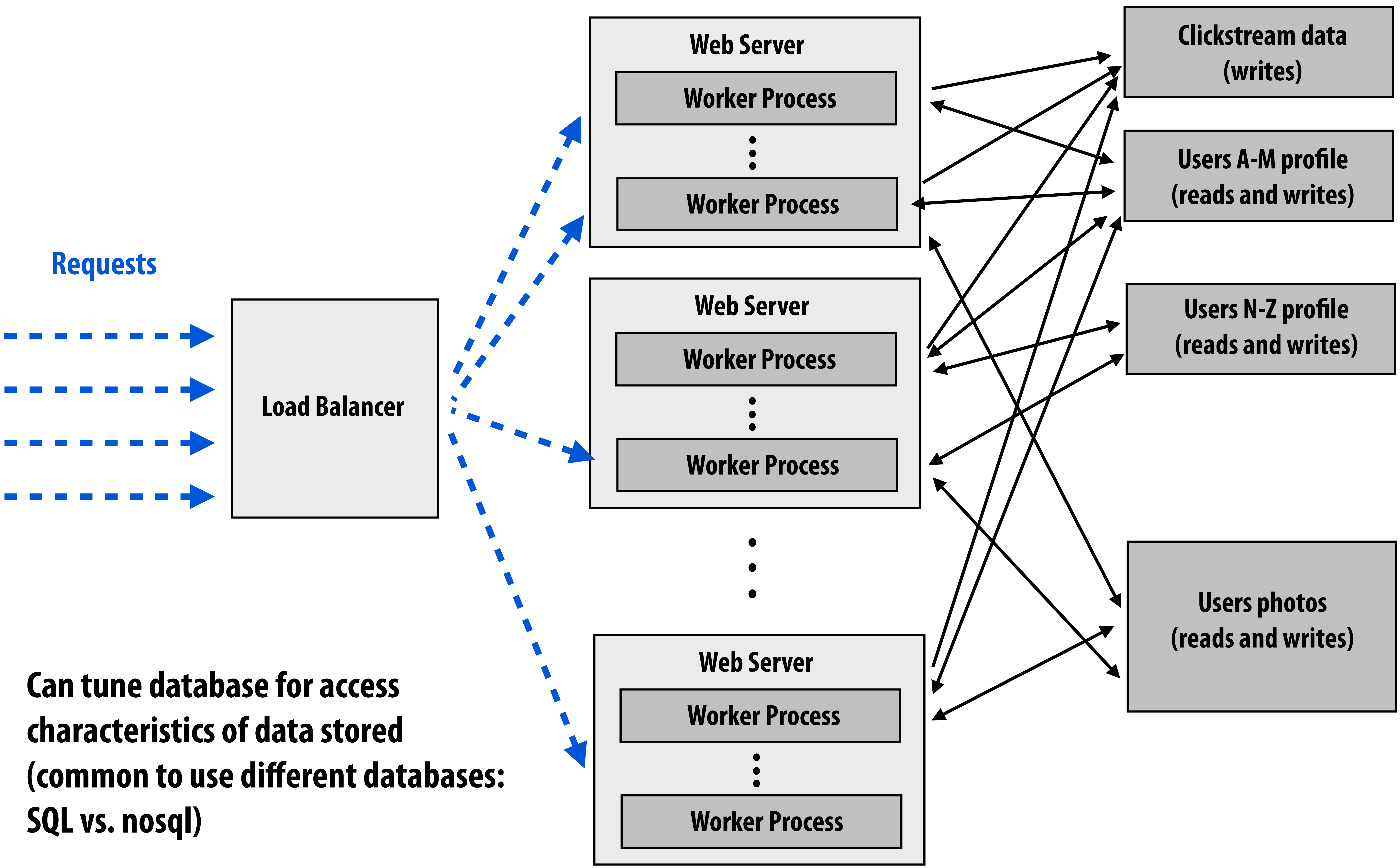
Scaling out a database: replicate

Replicate data and parallelize reads
(most DB accesses are reads)
Cost: extra storage, <u>consistency</u> issues

Adopt relaxed consistency models: propagate updates "eventually"



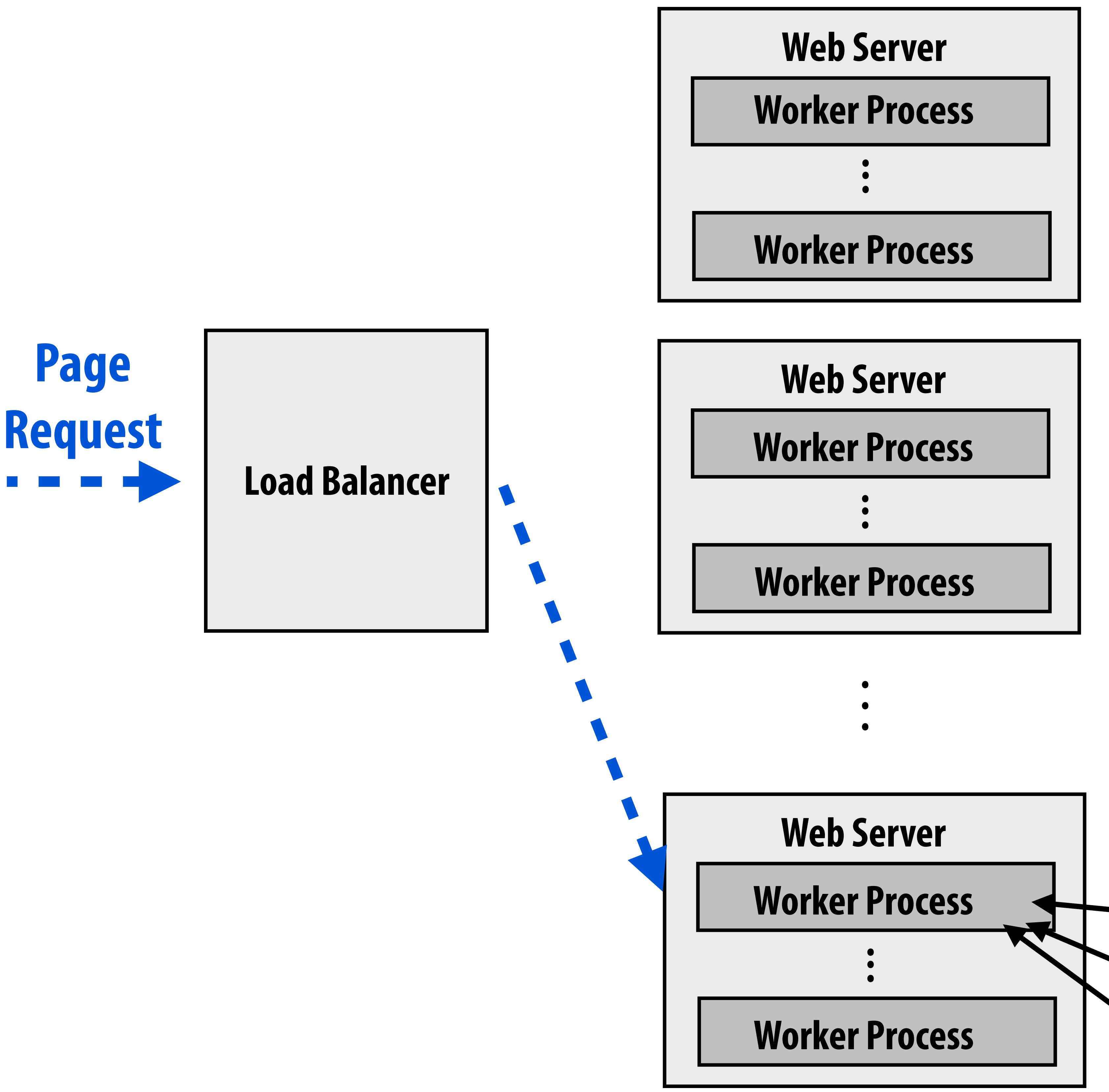
Scaling out a database: partition

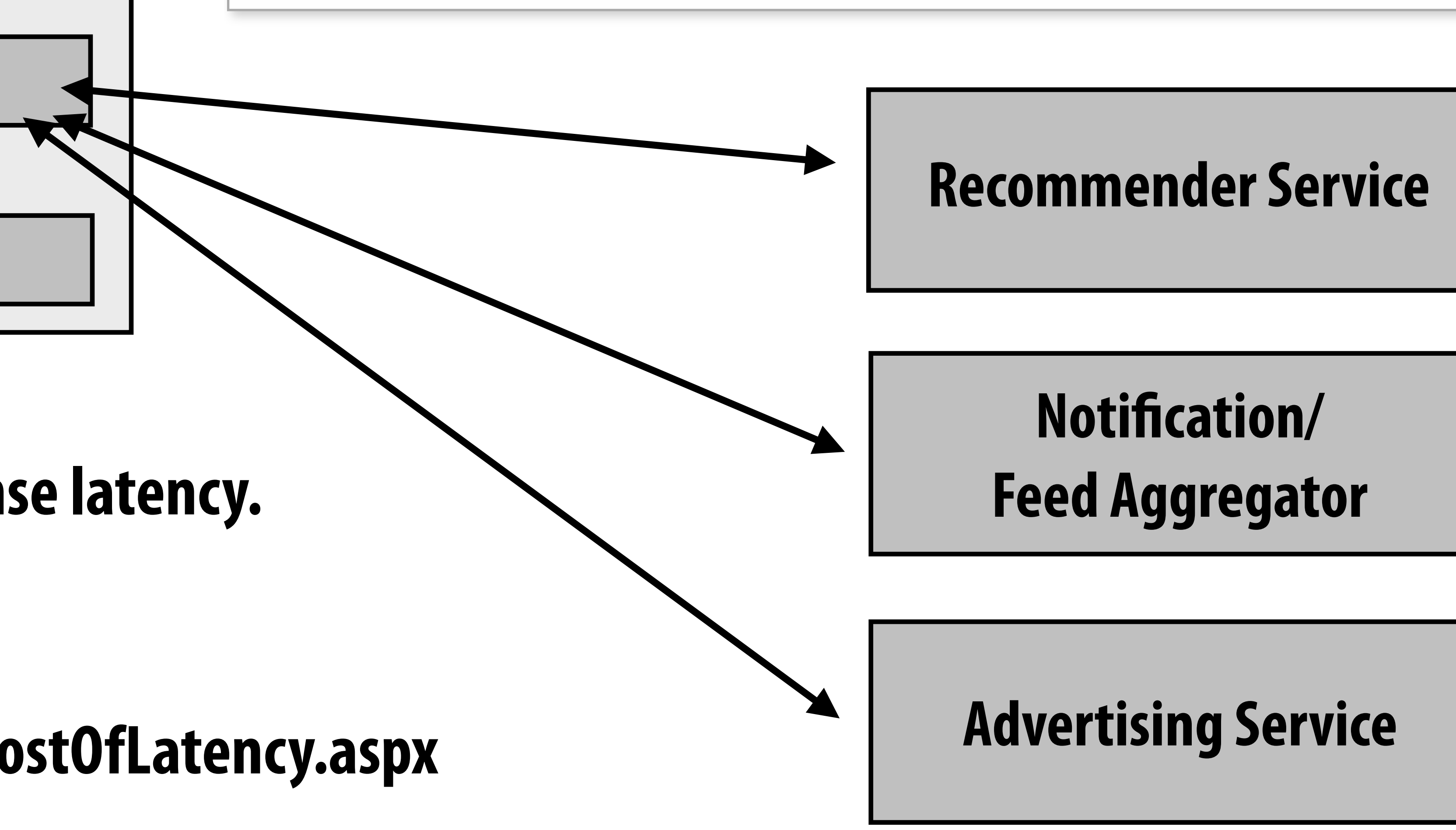


Can tune database for access characteristics of data stored (common to use different databases: SQL vs. nosql)

Inter-request parallelism

Parallelize generation of a single page





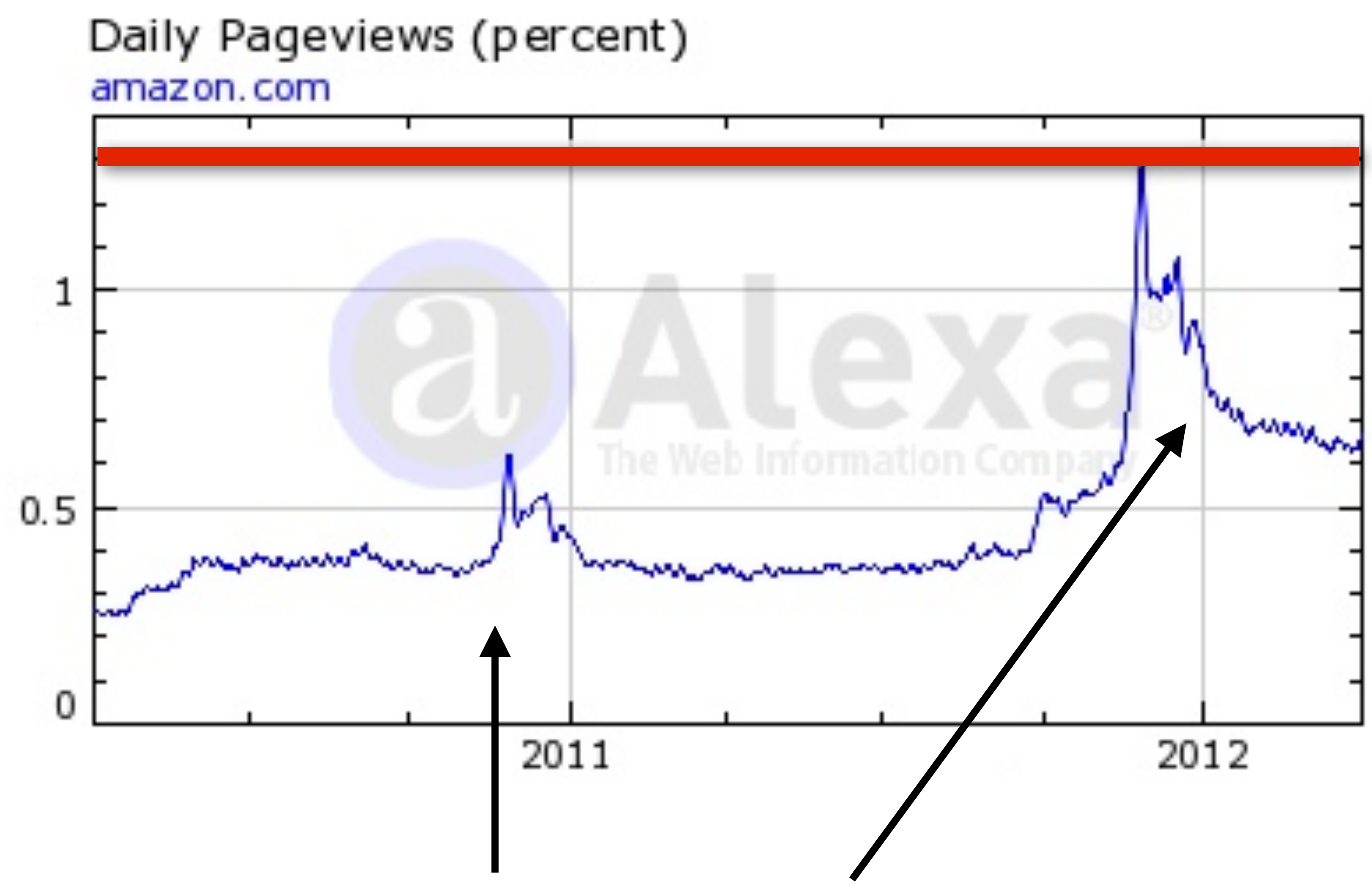
Amount of user traffic is directly correlated to response latency.

See great post:
http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx

How many web servers do you need?

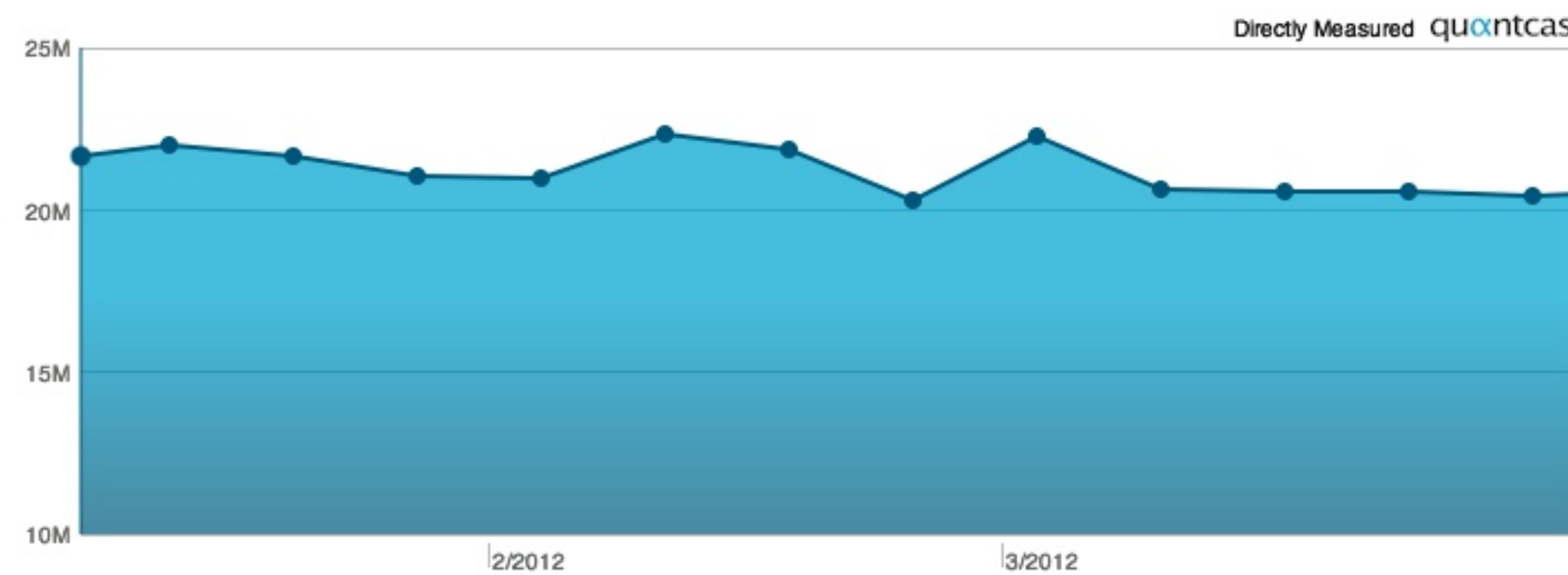
Web traffic is bursty

Amazon.com Page Views

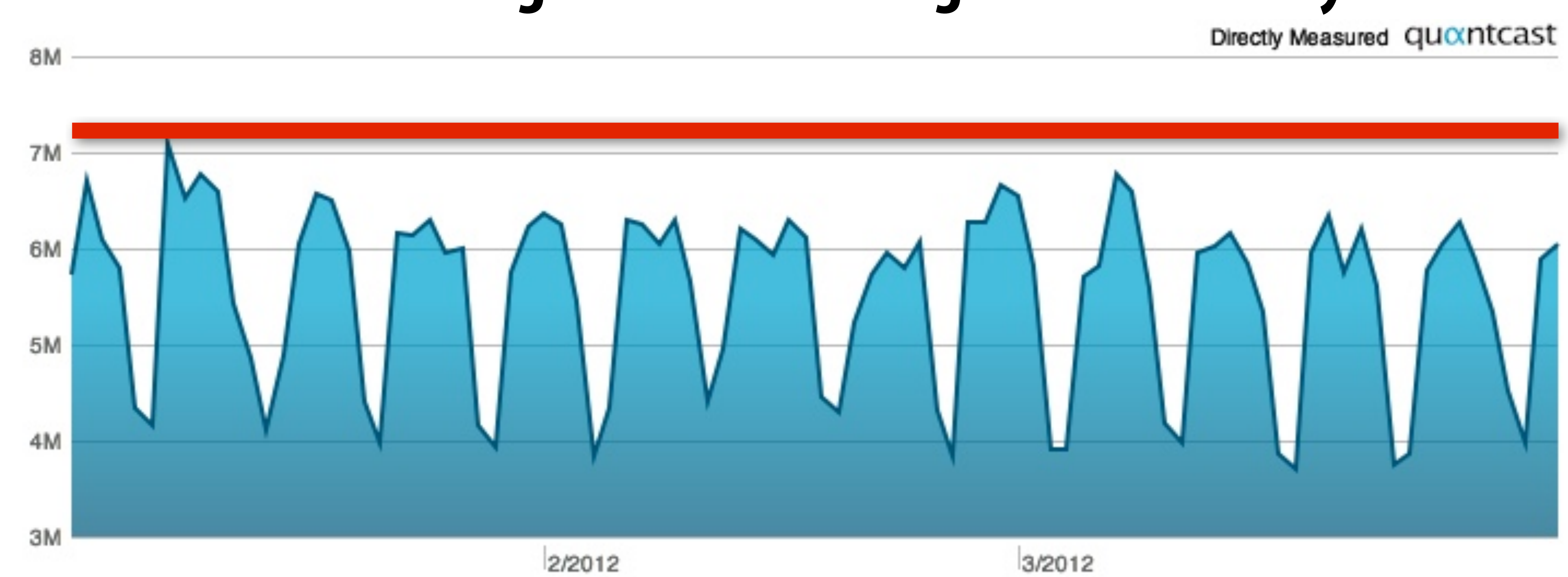


Holiday shopping season

HuffingtonPost.com Page Views Per Week



HuffingtonPost.com Page Views Per Day

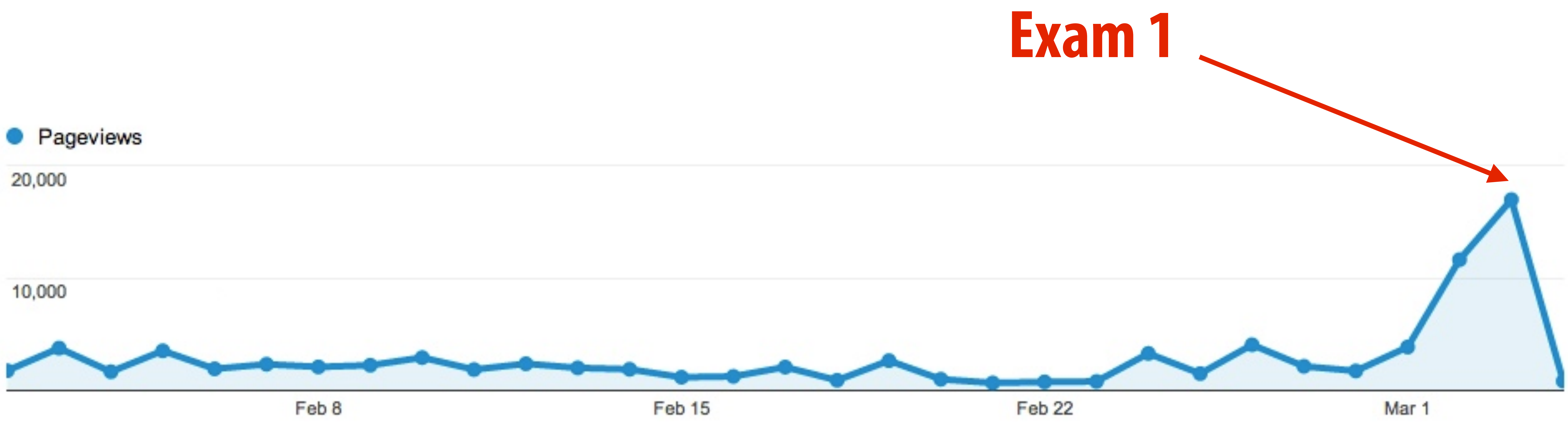


(fewer people read news on weekends)

More examples:

- **Facebook gears up for bursts of image uploads on Halloween and New Year's Eve.**
- **Twitter topics trend after world events**

15-418 Spring 2014 site traffic



Problem

- **Site load is bursty**

- **Provisioning site for the average case load will result in poor quality of service (or failures) during peak usage**
  - Peak usage tends to be when users care the most... since by the definition the site is important at these times

- **Provisioning site for the peak usage case will result in many idle servers most of the time**
  - Not cost efficient (must pay for many servers, power/cooling, datacenter space, etc.)

Elasticity!

- Main idea: site automatically adds or shuts down web servers based on measured load

- Need source of servers available on-demand
  - Example: Amazon.com EC2 instances

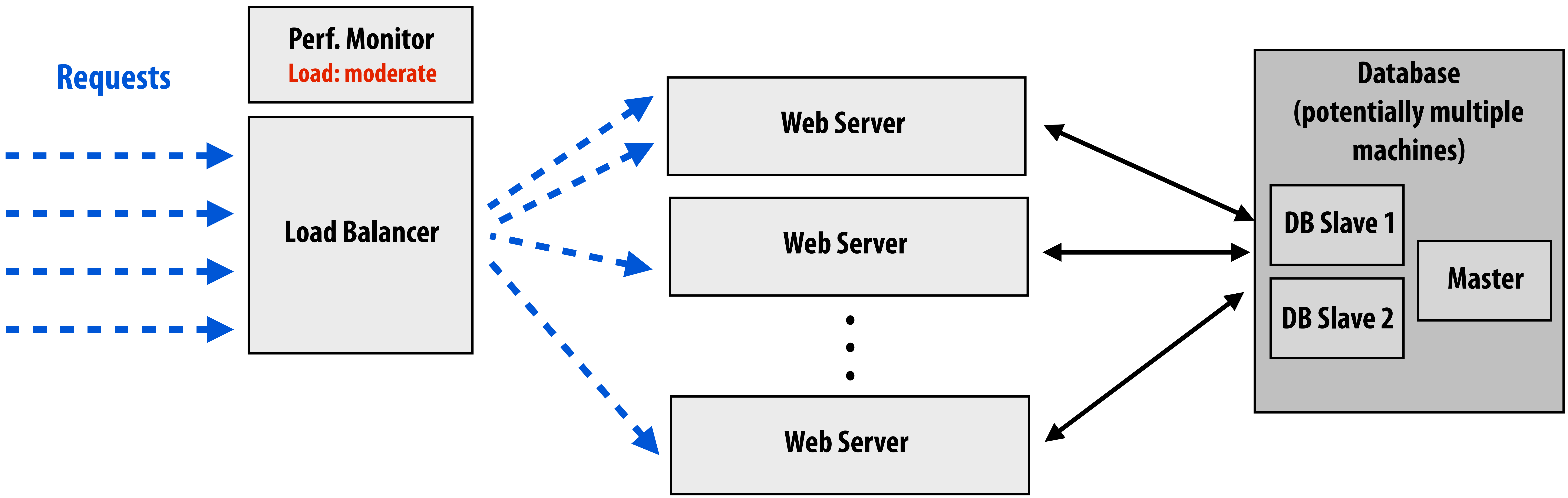
Example: Amazon's elastic compute cloud (EC2)

- Amazon had an over-provisioning problem
- Solution: make machines available for rent to others in need of compute
  - For those that don't want to incur cost of, or have expertise to, manage own machines at scale
  - For those that need elastic compute capability

Amazon.com Page Views

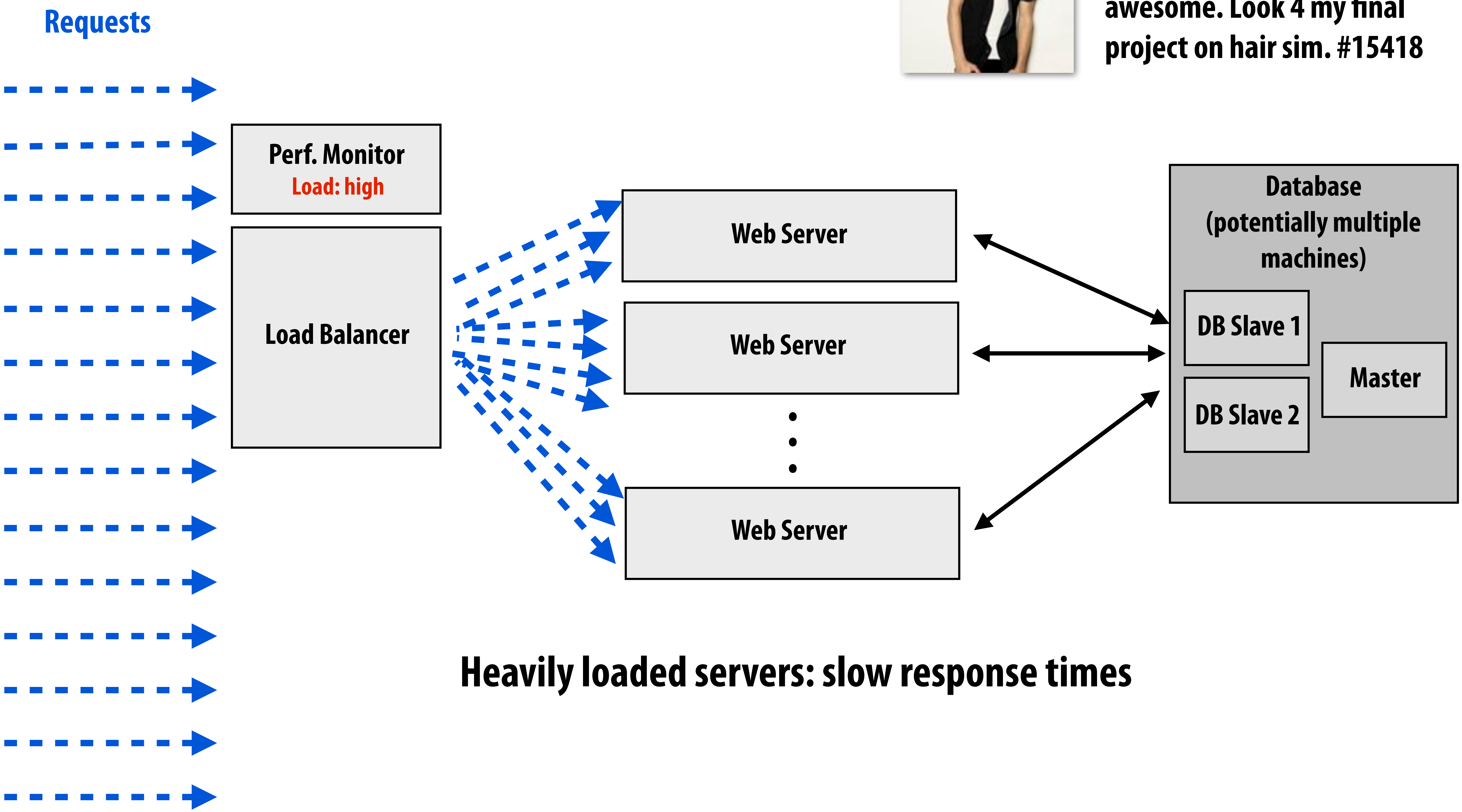
	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
General Purpose - Current Generation					
m3.medium	1	3	3.75	1 x 4 SSD	$0.113 per Hour
m3.large	2	6.5	7.5	1 x 32 SSD	$0.225 per Hour
m3.xlarge	4	13	15	2 x 40 SSD	$0.450 per Hour
m3.2xlarge	8	26	30	2 x 80 SSD	$0.900 per Hour
General Purpose - Previous Generation					
m1.small	1	1	1.7	1 x 160	$0.060 per Hour
m1.medium	1	2	3.75	1 x 410	$0.120 per Hour
m1.large	2	4	7.5	2 x 420	$0.240 per Hour
m1.xlarge	4	8	15	4 x 420	$0.480 per Hour

Site configuration: normal load



Event triggers spike in load

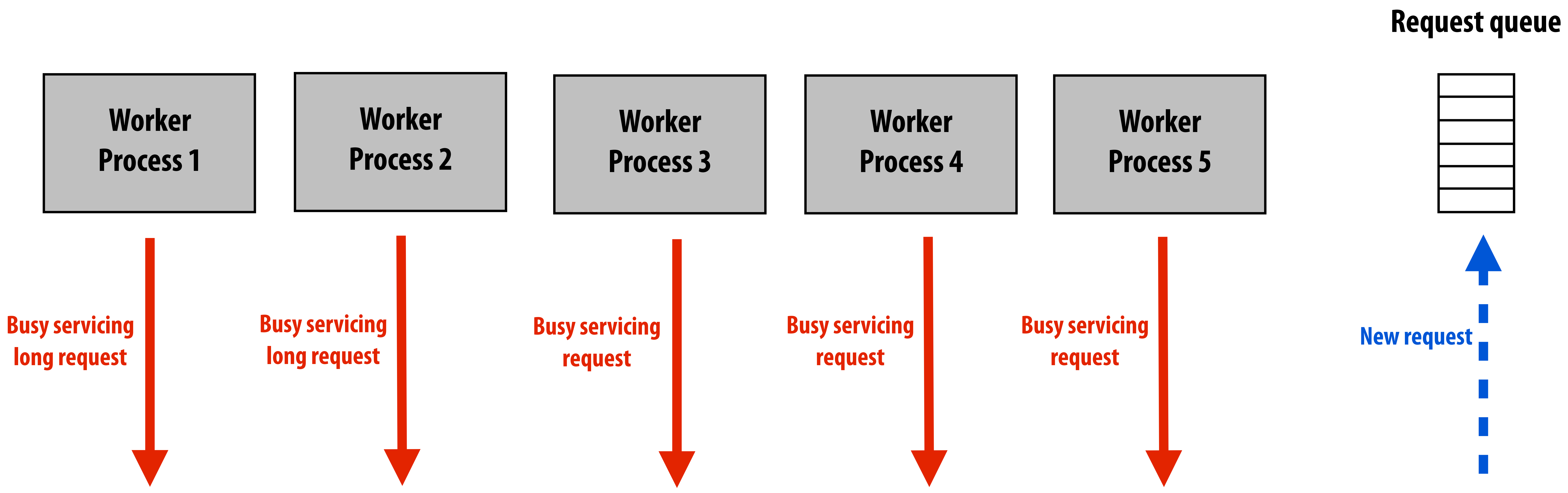
@justinbieber: OMG, parallel prog. class @ CMU is awesome. Look 4 my final project on hair sim. #15418



Heavily loaded servers: slow response times

Heavily loaded servers = slow response times

- If requests arrive faster than site can service them, queue lengths will grow
- <u>Latency</u> of servicing request is wait time in queue + time to actually process request
  - Assume site has capability to process R requests per second
  - Assume queue length is L
  - Time in queue = L/R
- How does site <u>throughput</u> change under heavy load?

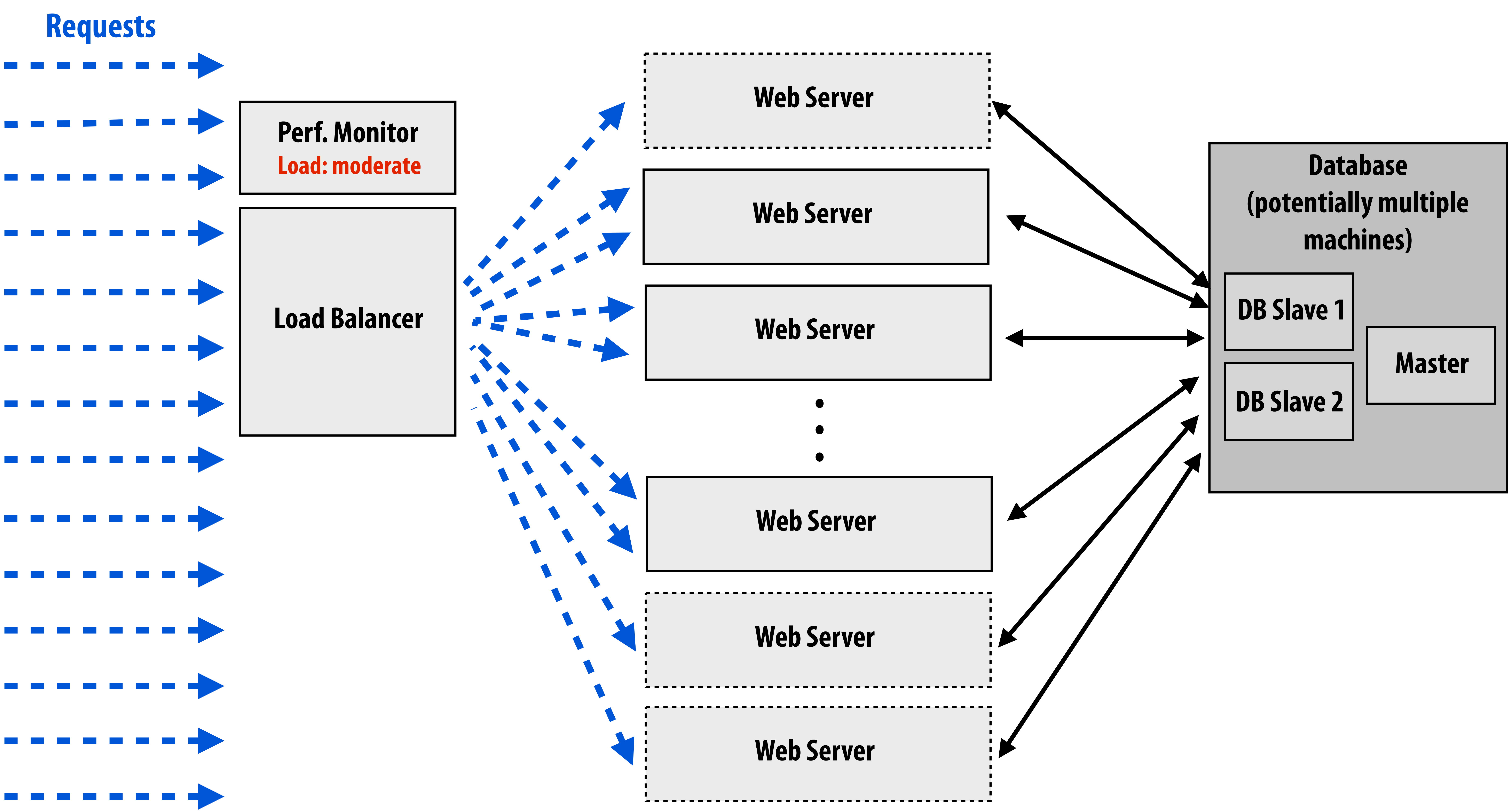


Site configuration: high load

Site performance monitor detects high load

Instantiates new web server instances

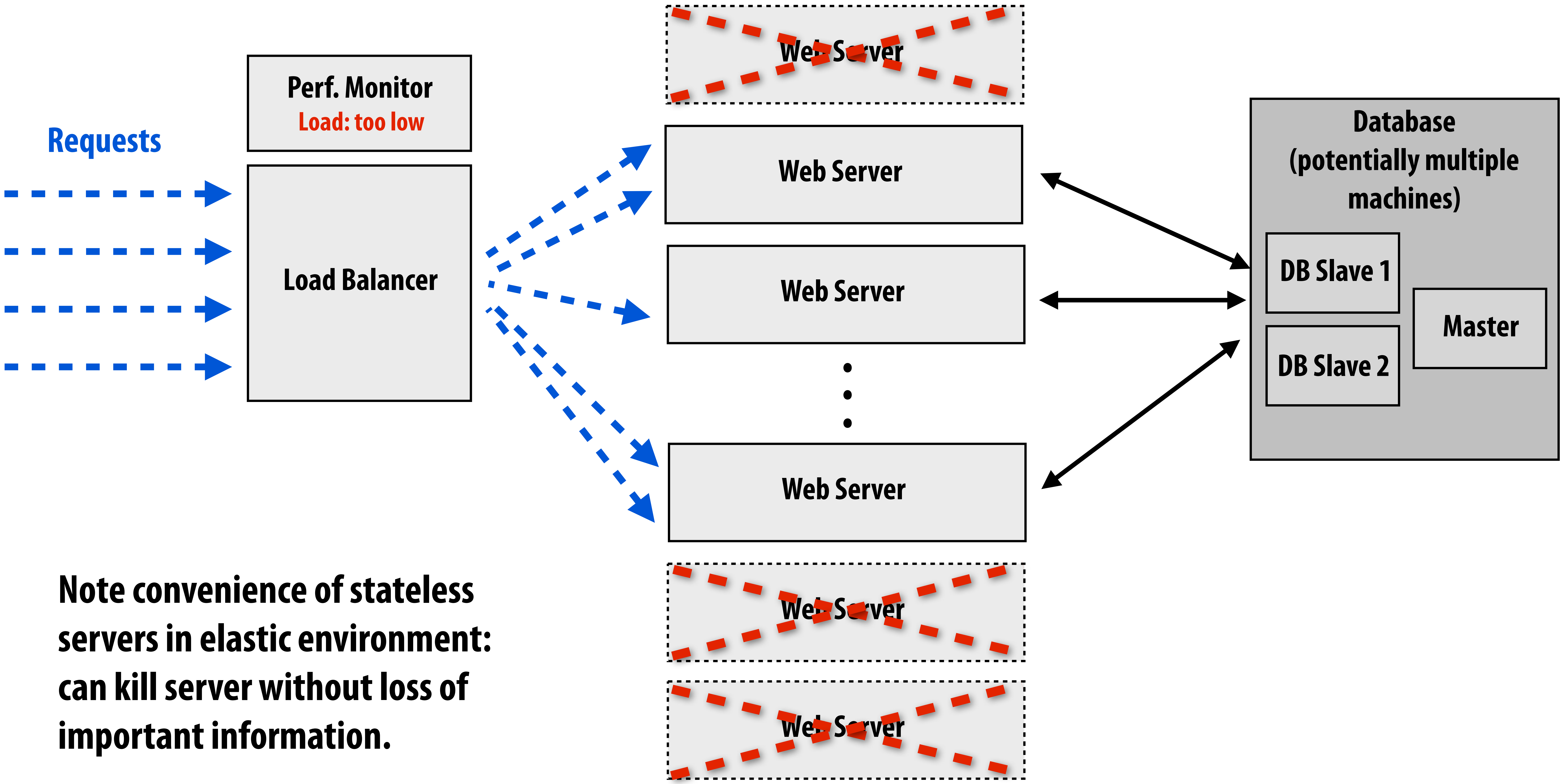
Informs load balancer about presence of new servers



Site configuration: return to normal load

Site performance monitor detects low load
Released extra server instances (to save operating cost)
Informs load balancer about loss of servers

@justinbieber: WTF, parallel programming is 2 hrd. Buy my new album.



Note convenience of stateless servers in elastic environment: can kill server without loss of important information.

Today: many "turn-key" environment-in-a-box services

Offer elastic computing environments for web applications

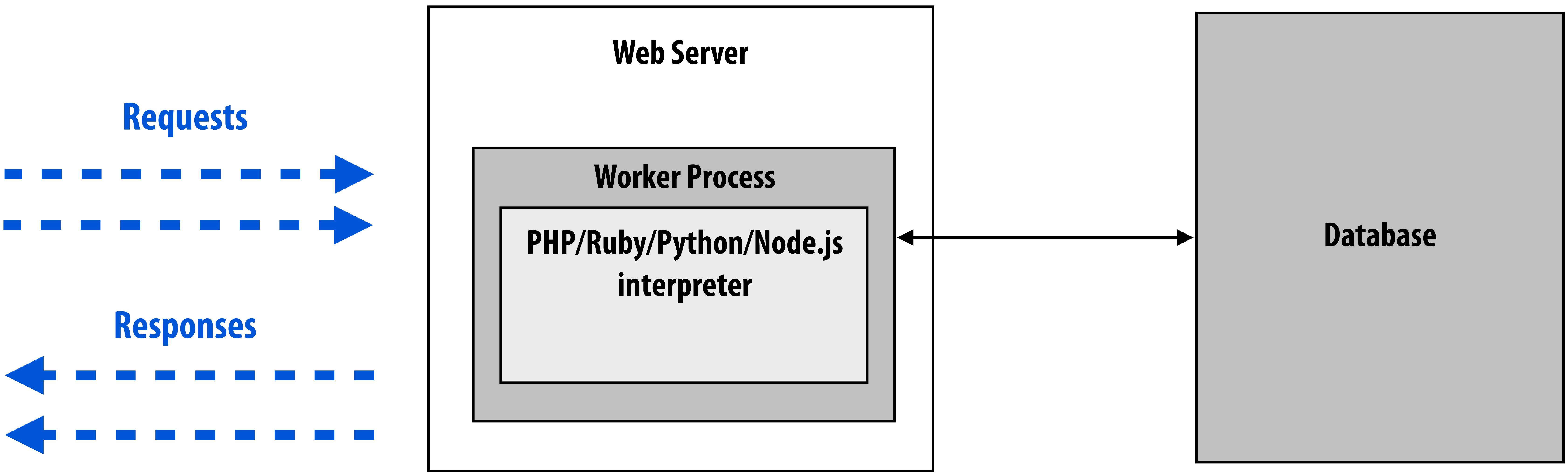
CloudWatch+Auto Scaling
Amazon Elastic Beanstalk

The story so far: parallelism
scale out, scale out, scale out

(+ elasticity to be able to scale out on demand)

Now: reuse and locality

Recall: basic site configuration



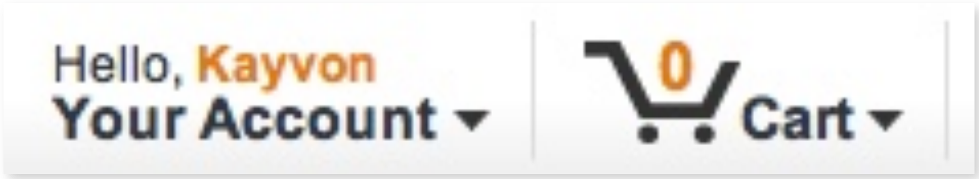
Example PHP Code

```
$query = "SELECT * FROM users WHERE username=‘kayvonf’;
$user = mysql_fetch_array(mysql_query($userquery));

echo “<div>” . $user[‘FirstName’] . “ “ . $user[‘LastName’] . “</div>”;
```

Response Information Flow

`<div>Kayvon Fatahalian</div>`

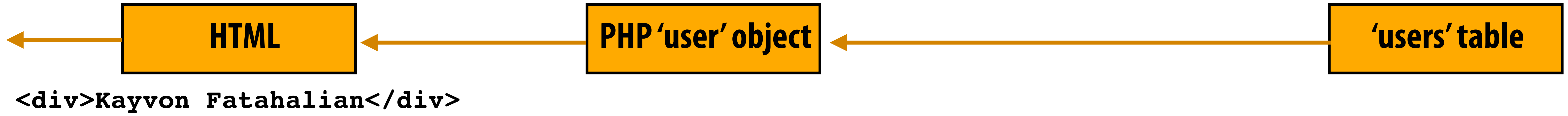
Work repeated every page



Example PHP Code

```
$query = "SELECT * FROM users WHERE username=‘kayvonf’;
$user = mysql_fetch_array(mysql_query($userquery));

echo “<div>” . $user[‘FirstName’] . “ “ . $user[‘LastName’] . “</div>”;
```

Response Information Flow

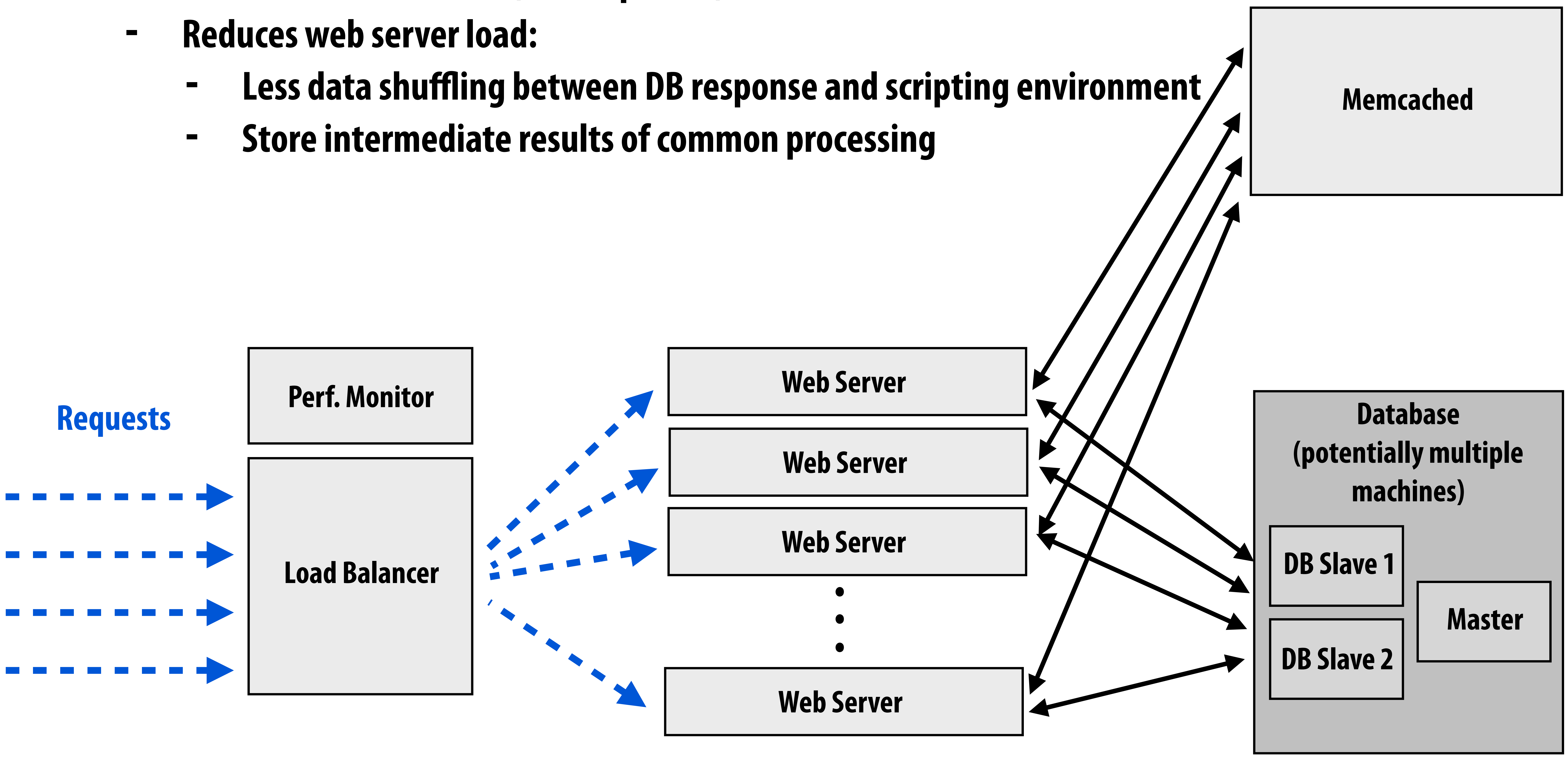


- Steps repeated to emit my name at the top of every page:
  - Communicate with DB
  - Perform query
  - Marshall results from database into object model of scripting language
  - Generate presentation
  - etc...

Remember, DB can be hard to scale!

Solution: cache!

- Cache commonly accessed objects
  - Example: `memcached`, in memory key-value store (e.g., a big hash table)
  - Reduces database load (fewer queries)
  - Reduces web server load:
    - Less data shuffling between DB response and scripting environment
    - Store intermediate results of common processing

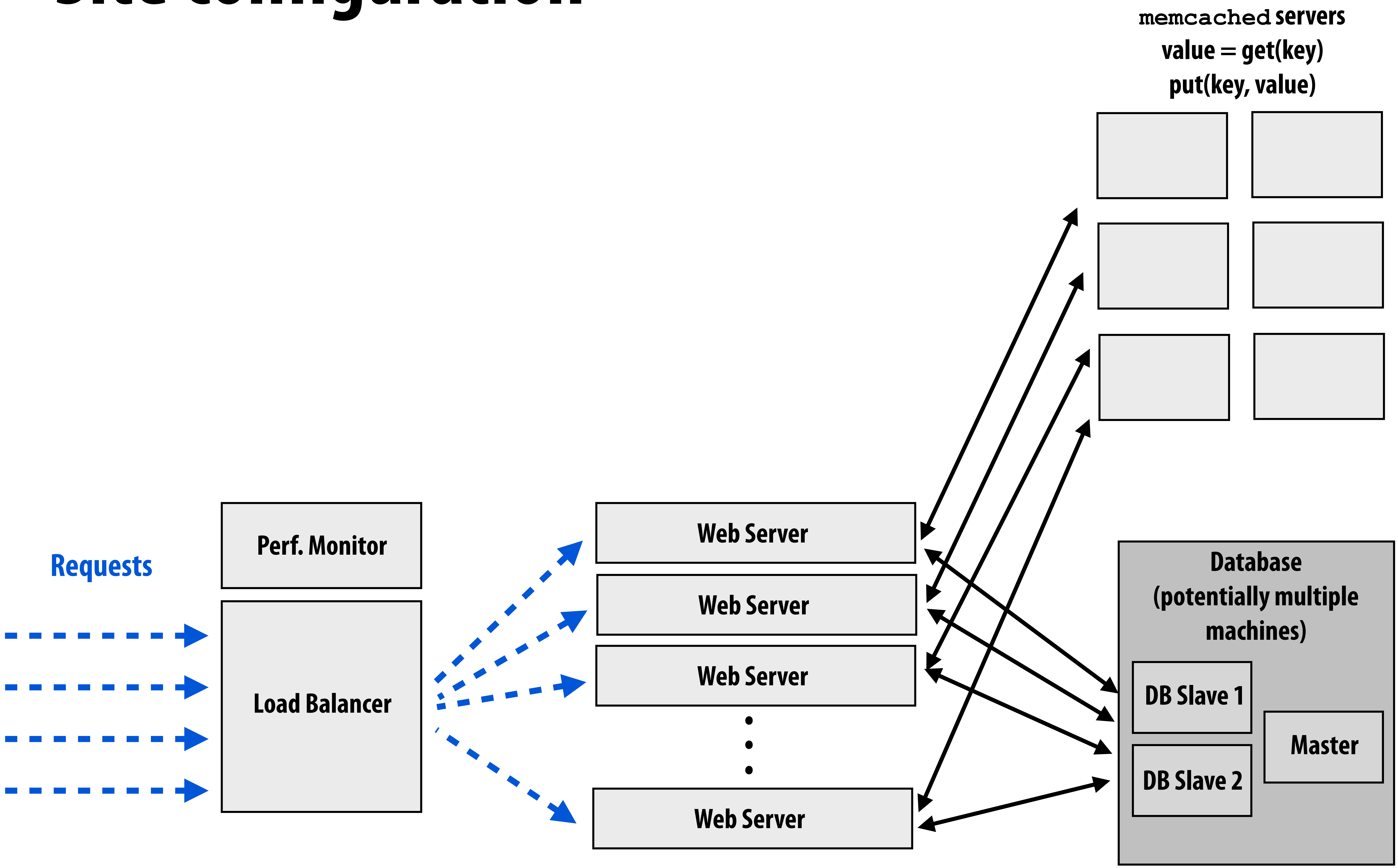


Caching example

```
userid = $_SESSION['userid'];

check if memcache->get(userid) retrieves a valid user object

if not:
   make expensive database query
   add resulting object into cache with memcache->put(userid)
   (so future requests involving this user can skip the query)

continue with request processing logic
```

- **Of course, there is complexity associated with keeping caches in sync with data in the DB in the presence of writes**
  - **Must invalidate cache**
  - **Very simple "first-step" solution: only cache read-only objects**
  - **More realistic solutions provide some measure of consistency**
    - **But we'll leave this to your distributed computing and database courses**

Site configuration



Example: Facebook memcached deployment

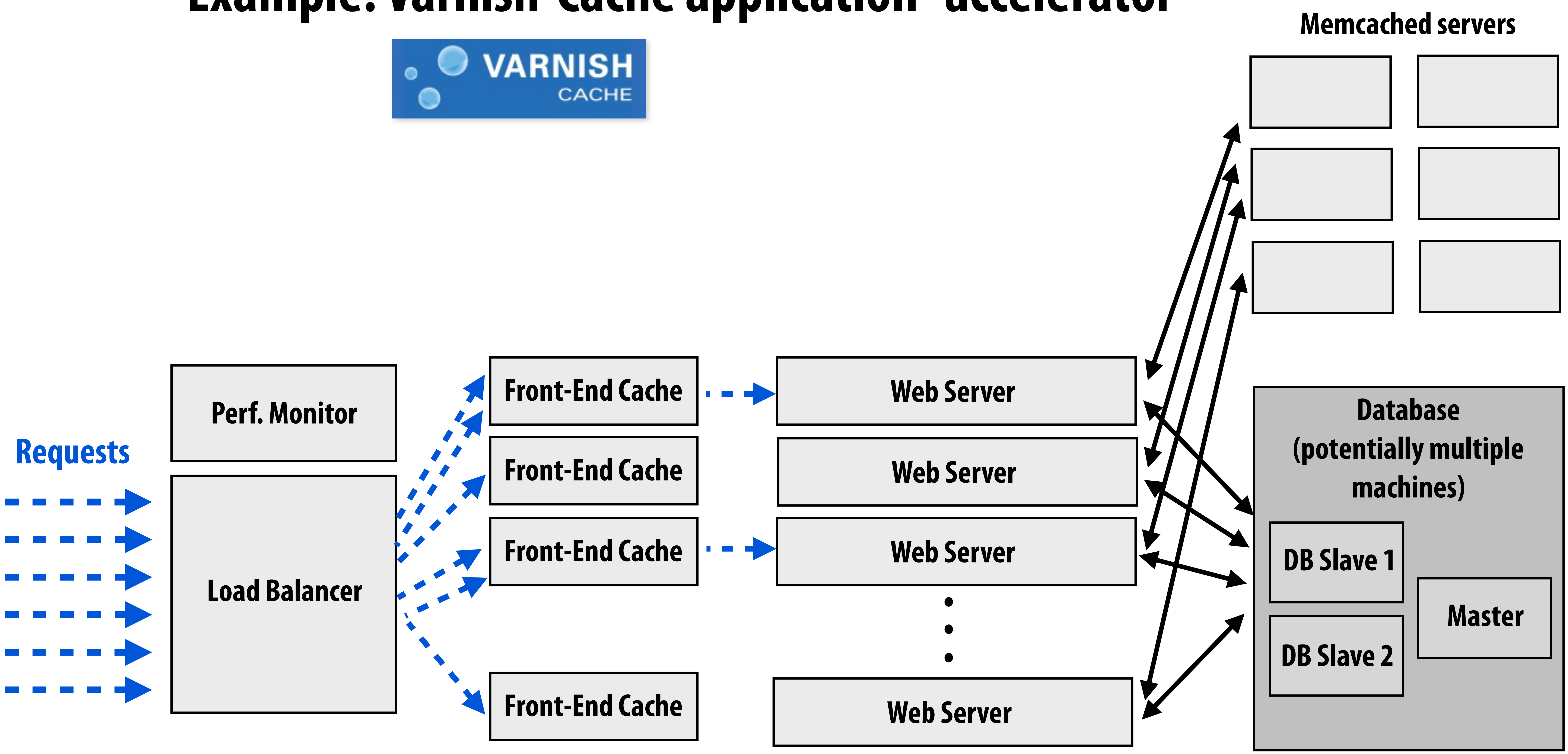
- Facebook, circa 2008
  - 800 memcached servers
  - 28 TB of cached data

- Performance
  - 200,000 UDP requests per second @ 173 msec latency
  - 300,000 UDP requests per second possible at "unacceptable" latency

Source: https://www.facebook.com/note.php?note_id=39391378919

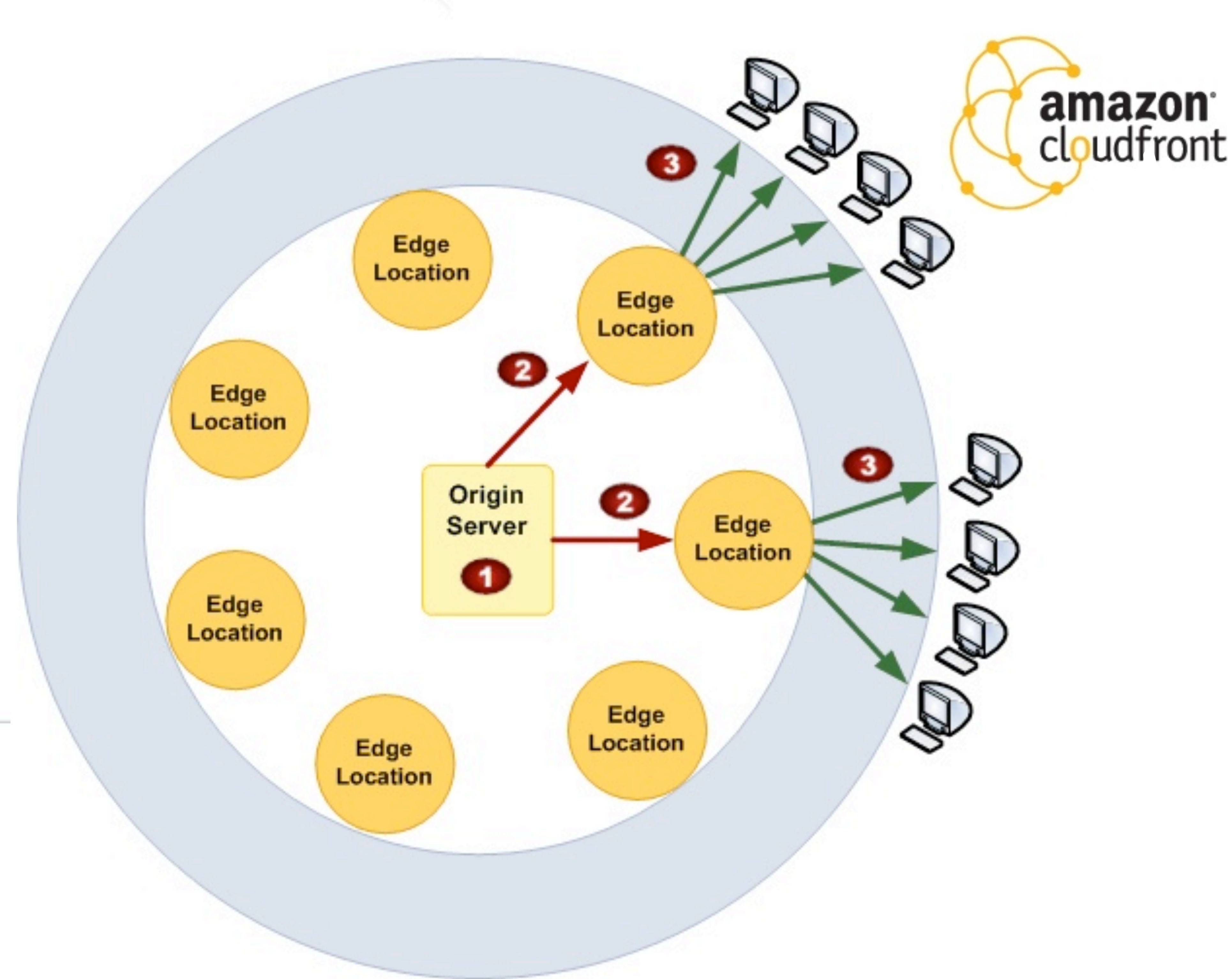
More caching

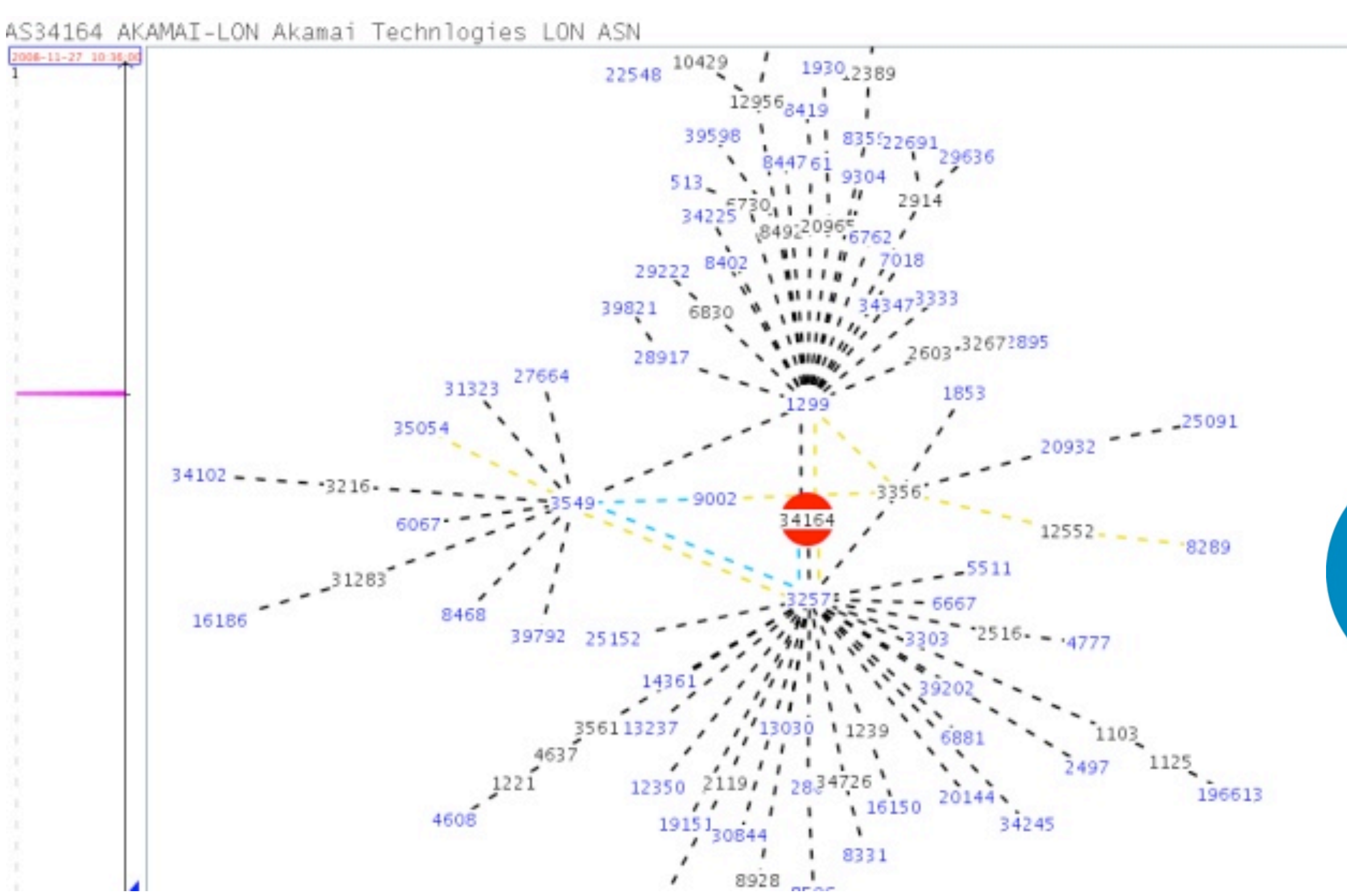
- Cache web server responses (e.g. entire pages, pieces of pages)
  - Reduce load on web servers
  - Example: Varnish-Cache application "accelerator"



Caching using content distribution networks (CDNs)

- Serving large media assets can be expensive to serve (high bandwidth costs, tie up web servers)
  - E.g., images, streaming video
- Physical locality is important
  - Higher bandwidth
  - Lower latency



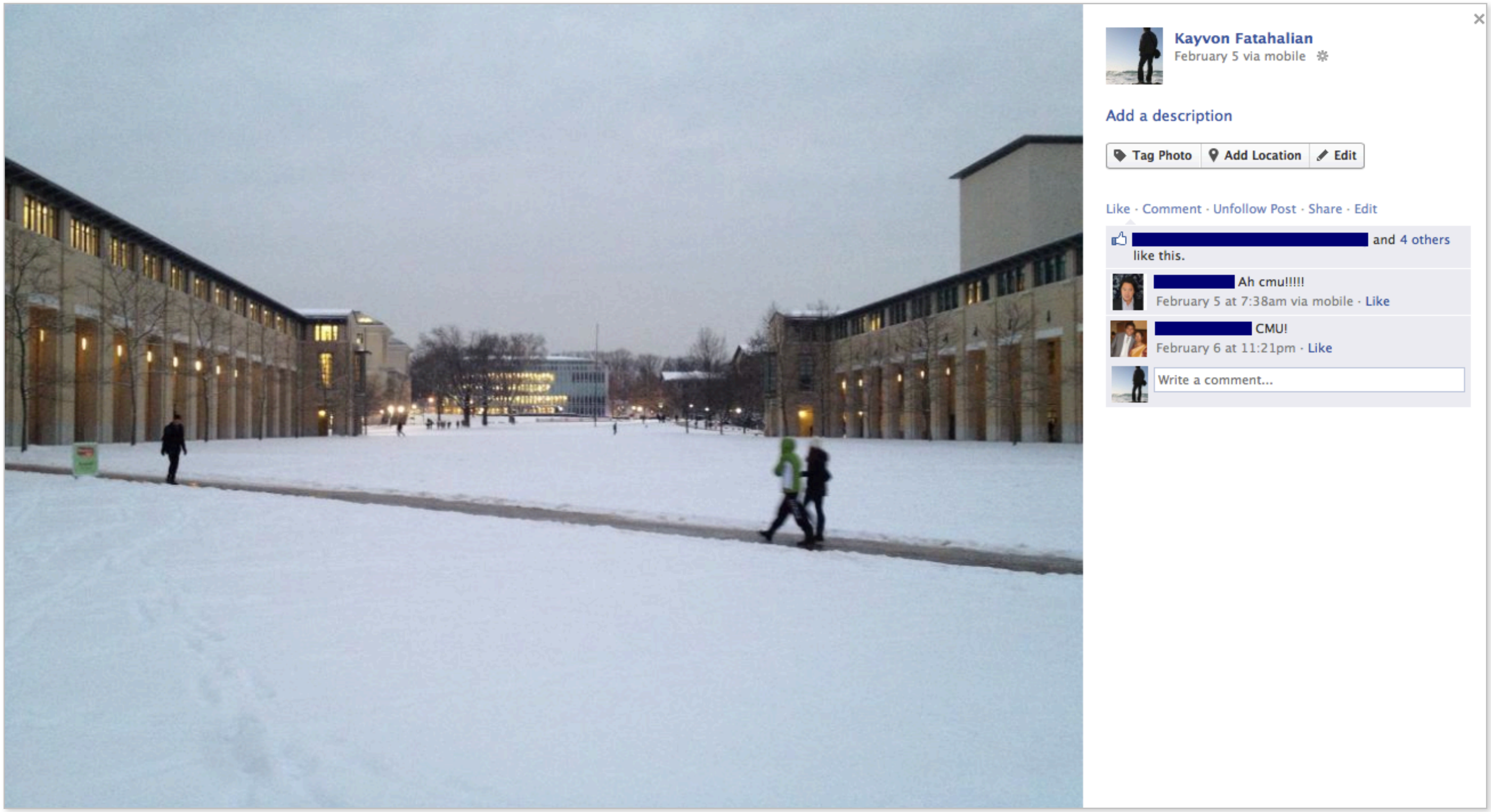


Akamai

London Content Distribution Network

Source: http://www.telco2.net/blog/2008/11/amazon_cloudfront_yet_more_tra.html

CDN usage example

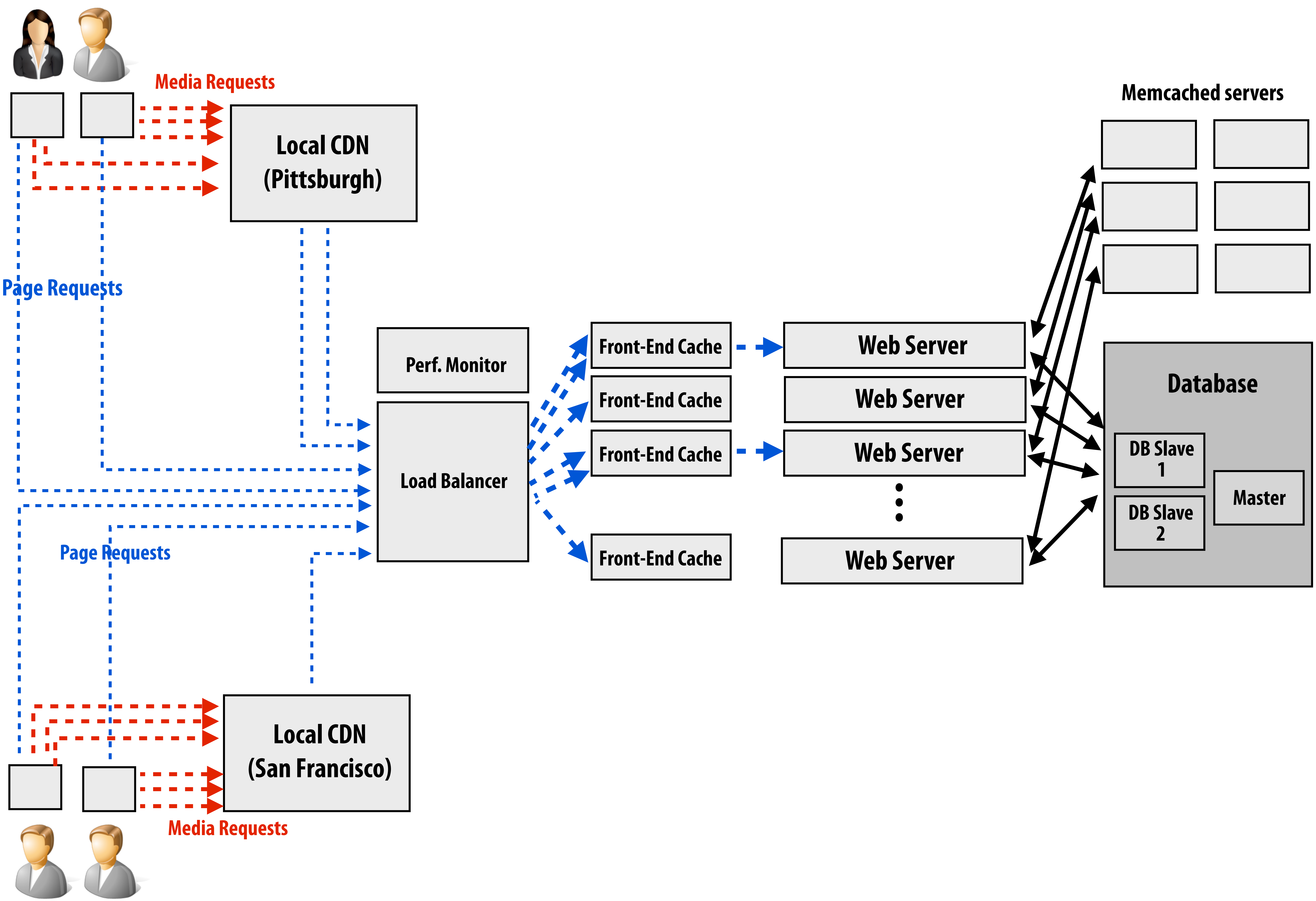


Facebook photo:

Page URL:
https://www.facebook.com/photo.php?fbid=10151325164543897&set=a.10150275074093897.338852.722973896&type=1&theater

Image source URL:
https://sphotos-a.xx.fbcdn.net/hphotos-prn1/522152_10151325164543897_1133820438_n.jpg

CDN integration



Summary: scaling modern web sites

- **Use parallelism**
  - Scale-out parallelism: leverage many web servers to meet throughput demand
  - Elastic scale-out: cost-effectively adapt to bursty load
  - Scaling databases can be tricky (replicate, shard, partition by access pattern)
    - Consistency issues on writes
- **Exploit locality and reuse**
  - Cache everything (key-value stores)
    - Cache the results of database access (reduce DB load)
    - Cache computation results (reduce web server load)
    - Cache the results of processing requests (reduce web server load)
  - Localize cached data near users, especially for large media content (CDNs)
- **Specialize implementations for performance**
  - Different forms of requests, different workload patterns

Final comments

- It is true that performance of straight-line application logic is often very poor in web-programming languages (orders of magnitude left on the table in Ruby and PHP).

- BUT... web development is not just trivial hacking in slow scripting languages. Scaling a web site is a very challenging parallel-systems problem that involves many of the optimization techniques and design choices studied in this class: just at different scales
  - Identifying parallelism and dependencies
  - Workload balancing: static vs. dynamic partitioning issues
  - Data duplication vs. contention
  - Throughput vs. latency trade-offs
  - Parallelism vs. footprint trade-offs
  - Identifying and exploiting reuse and locality

- Many great sites (and blogs) on the web to learn more:
  - www.highscalability.com has great case studies (see "All Time Favorites" section)
  - James Hamilton's blog: http://perspectives.mvdirona.com

Have a nice spring break!