Lecture 14:

Scaling a Web Site

Scale-out Parallelism, Elasticity, and Caching

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2017

Tunes

Taylor Swift

Shake it Off

(1989)

"Not happy with your Exam 1 grade? No worries. Plenty of chances to get better!"

- Taylor Swift

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 2 questions (of your choosing) from the exam, on your own time.
  - You may discuss the problems with your classmates, instructor, and TAs.
  - You must write your solutions on your own.
  - You will get 50% credit for lost points on regraded questions.
  - The revised solutions must be handed in by Friday, April 7th

But... there's a catch!

The Catch

- You must hand in your solution to the course staff at a designated office hours.
- And you are not allowed to hand in unless you are able to successfully answer a series of questions we ask you
- The questions will a subset of the questions on exam 1 (or simple follow up variants)
- The staff will post times to sign up for 6-minute time slots
  - But not until after Spring Break

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

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

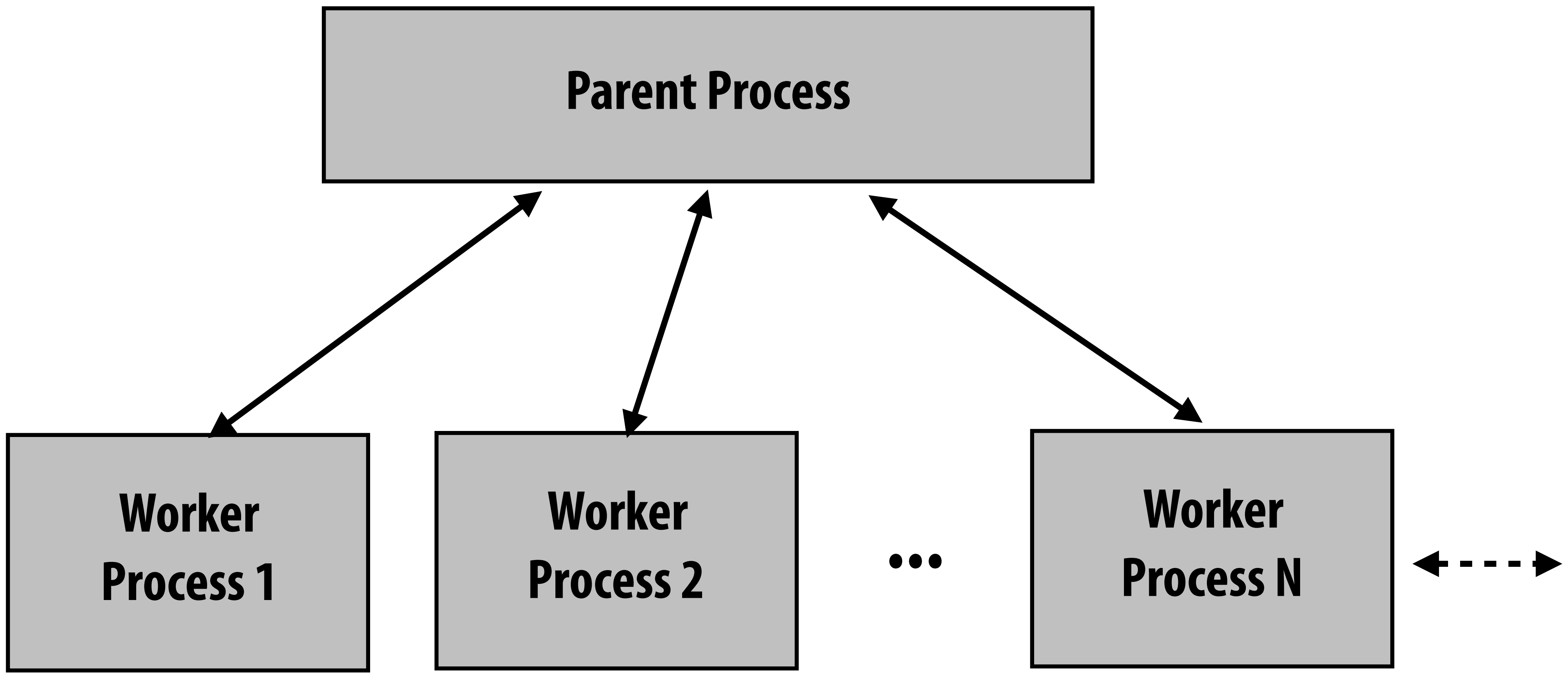
- Many other issues in developing a successful web platform
  - Reliability, security, privacy, etc.
  - There are other great courses at CMU for these topics (distributed systems, databases, cloud computing)

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

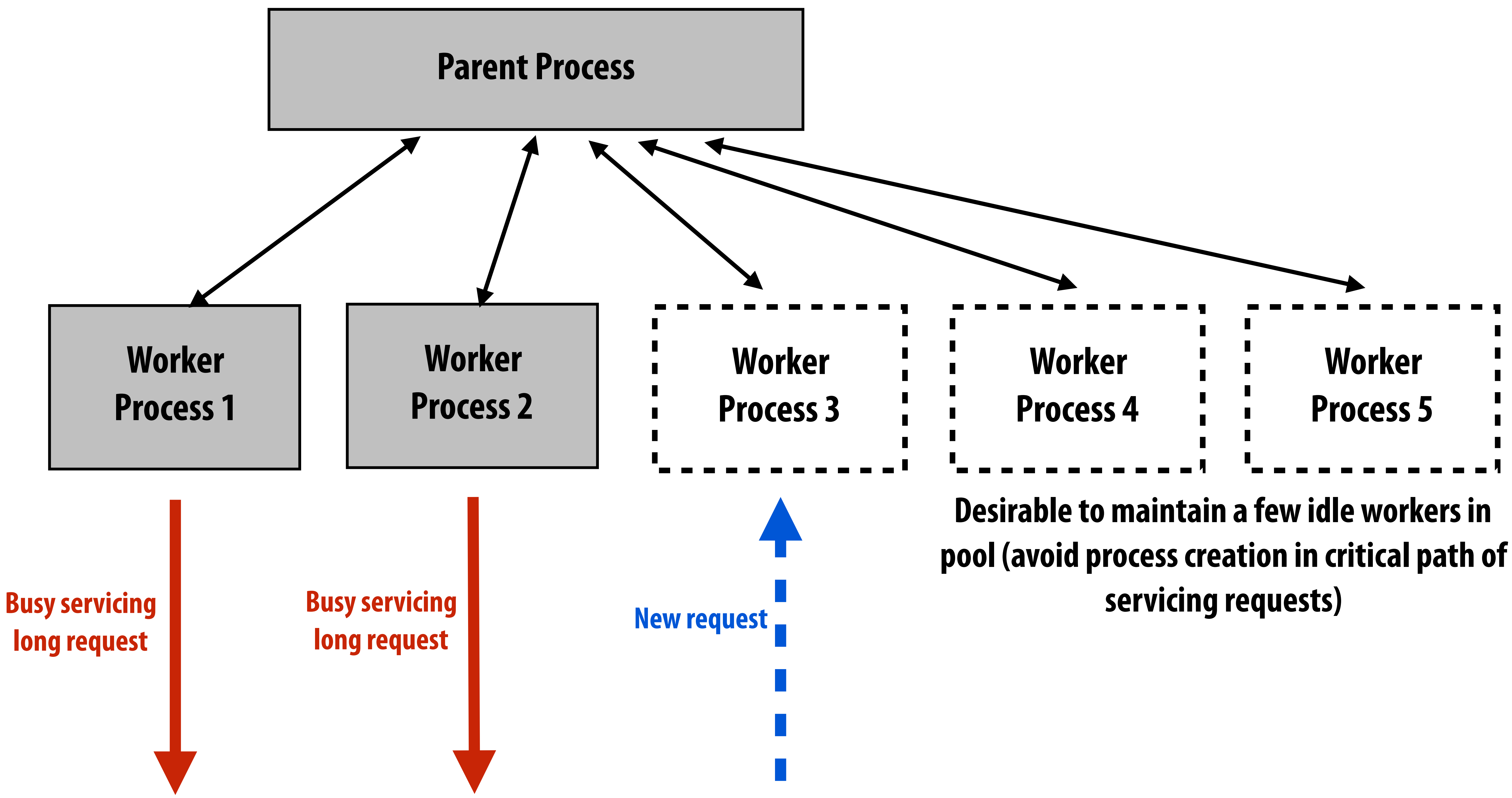



```
while (1)
{
    request = wait_for_request();

    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?

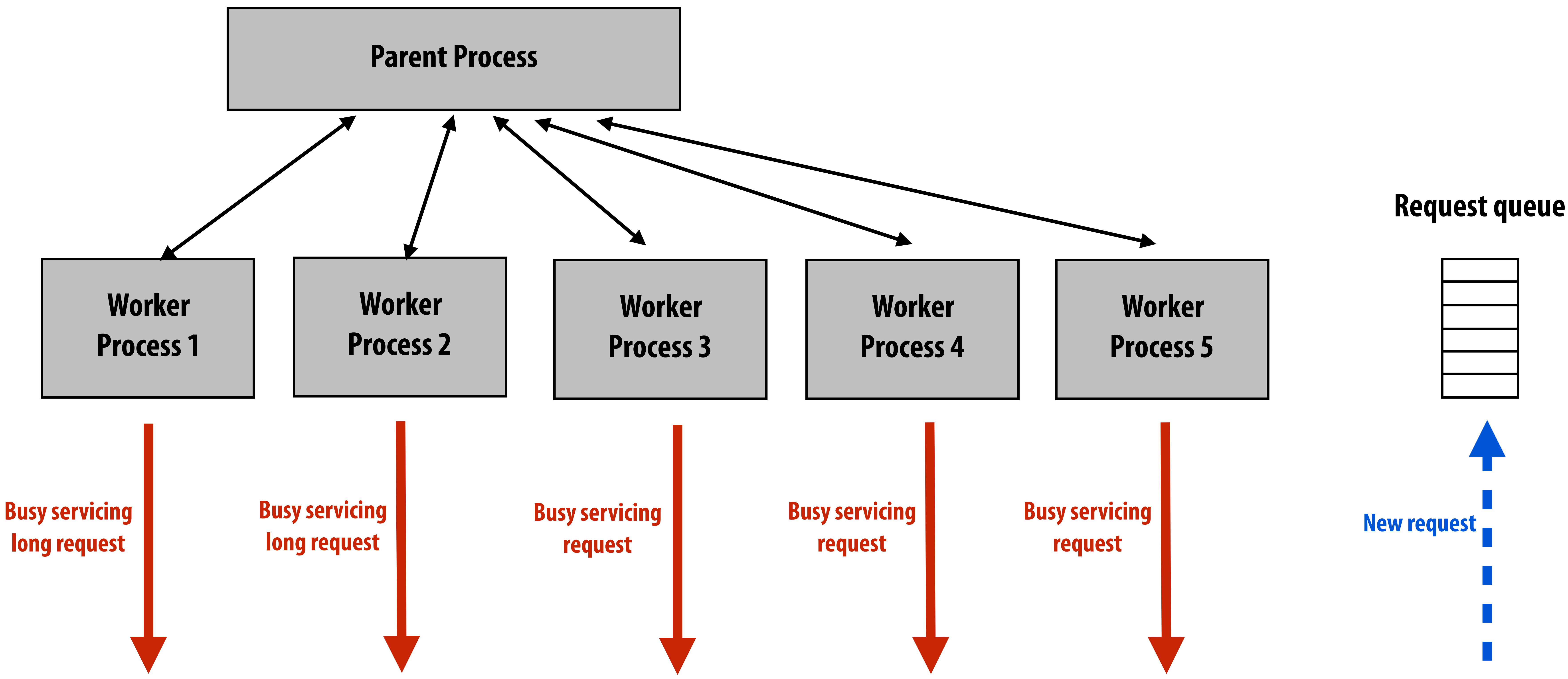
- Parallelism: use all the server's cores
- Latency hiding: hide long-latency disk read operations (by context switching between worker processes)
- Concurrency: many outstanding requests, want to service quick requests while long requests are in progress (e.g., large file transfer shouldn't block serving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

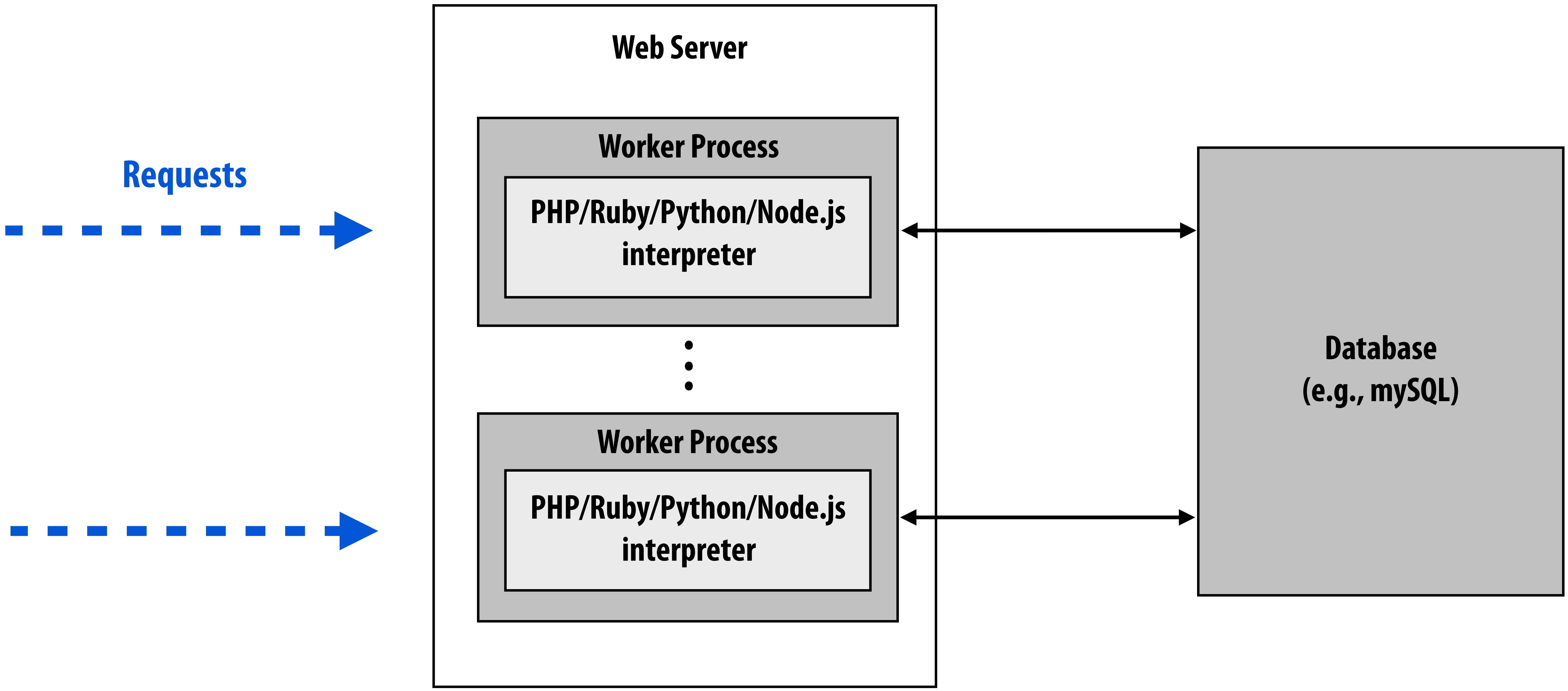
Aside: why partition server into processes, not threads?

- **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.

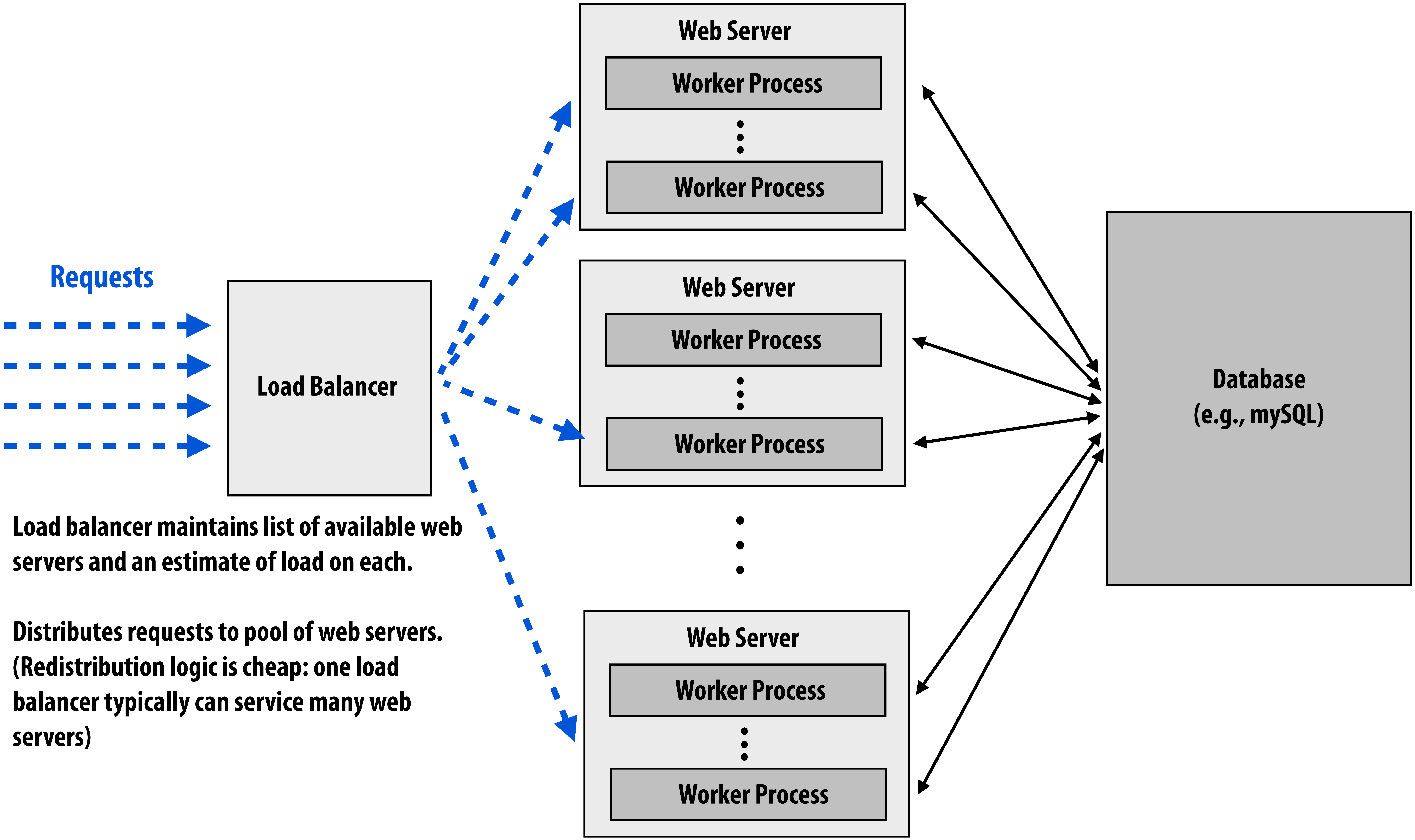
Scripting language performance (poor)

- Two popular content management systems (PHP)
  - Wordpress ~ 12 requests/sec/core (DB size = 1000 posts)
  - MediaWiki ~ 8 requests/sec/core
    [Source: Talaria Inc., 2012]

- 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.

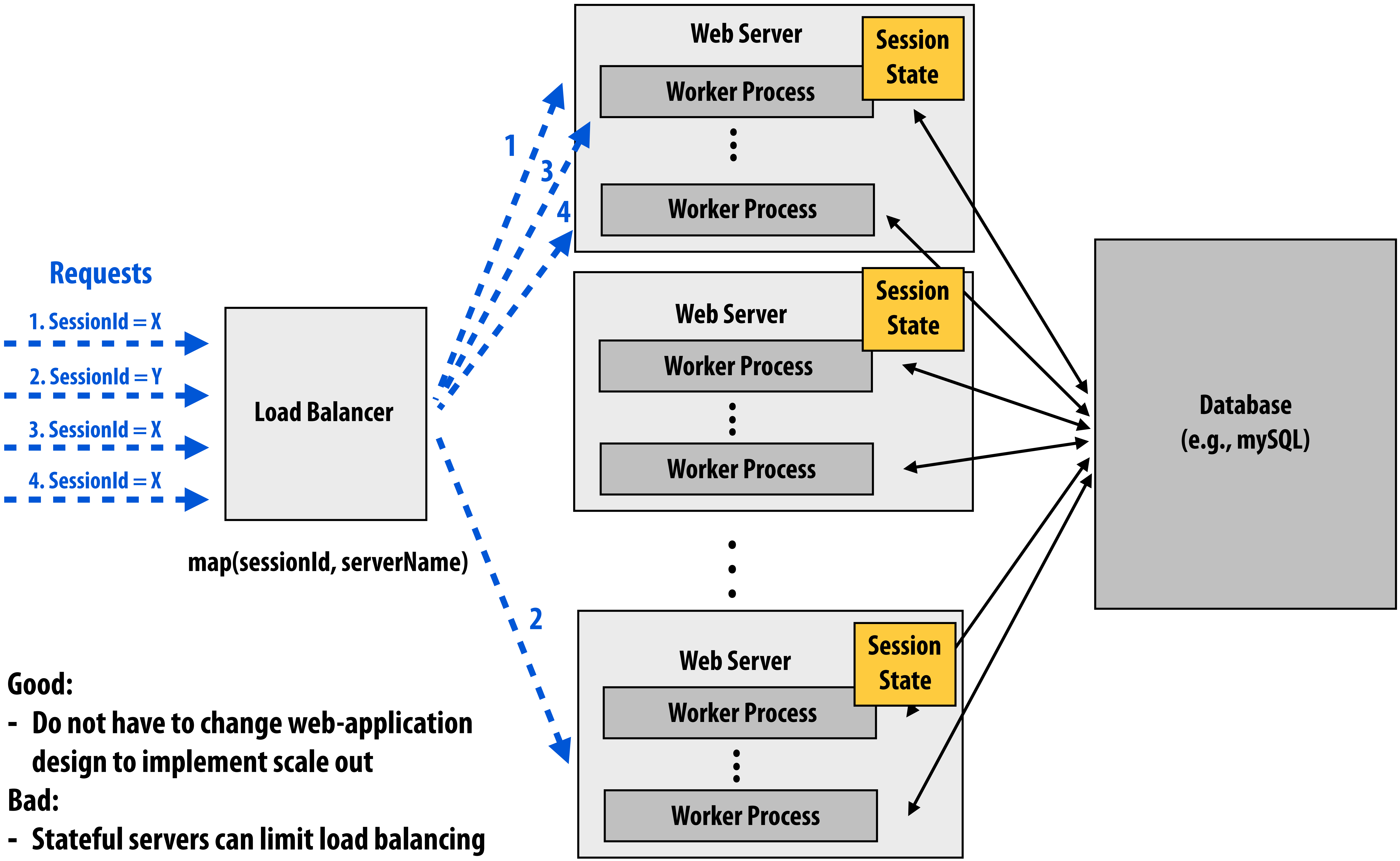


Load balancer maintains list of available web servers and an estimate of load on each.

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")

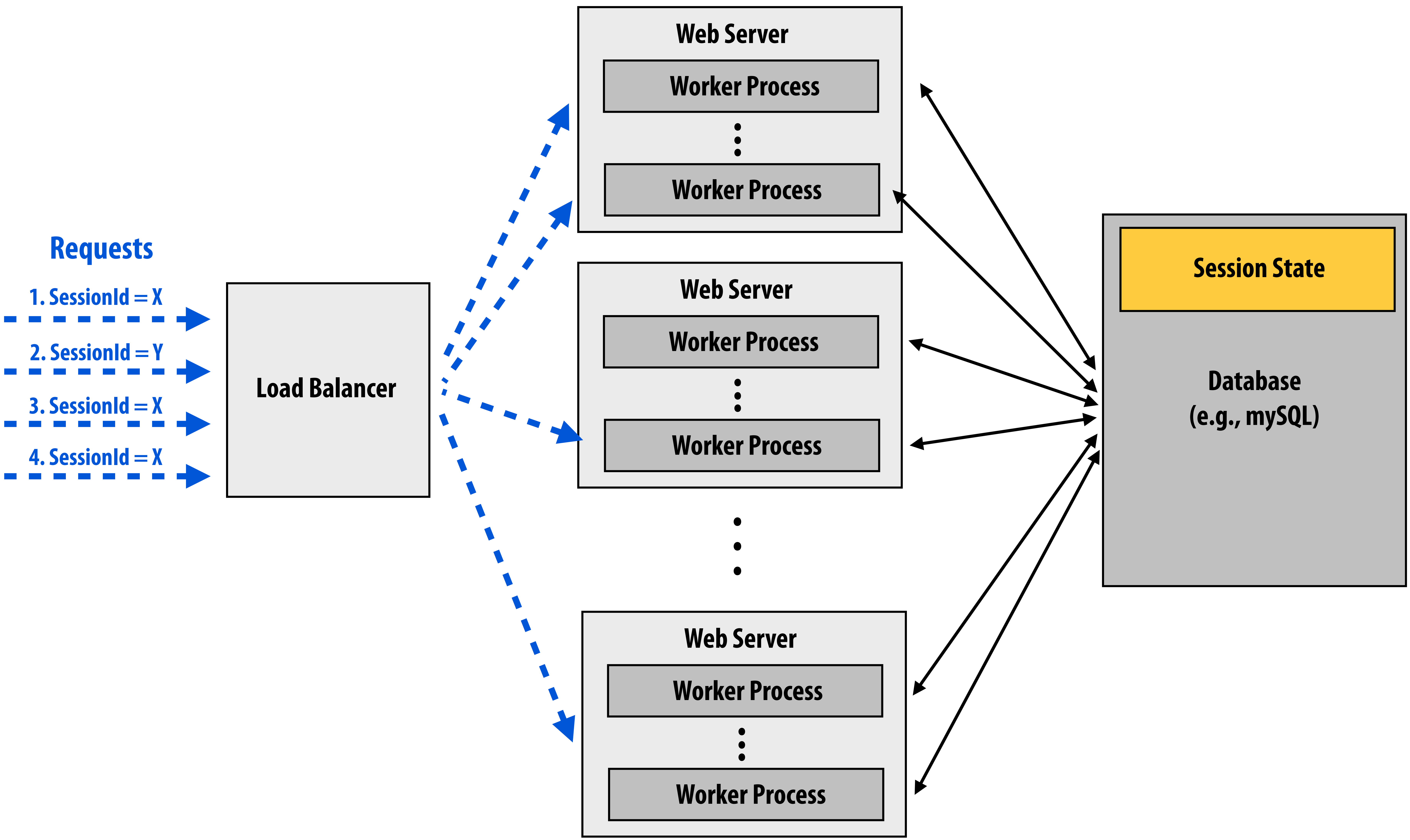


Good:
- Do not have to change web-application design to implement scale out

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.

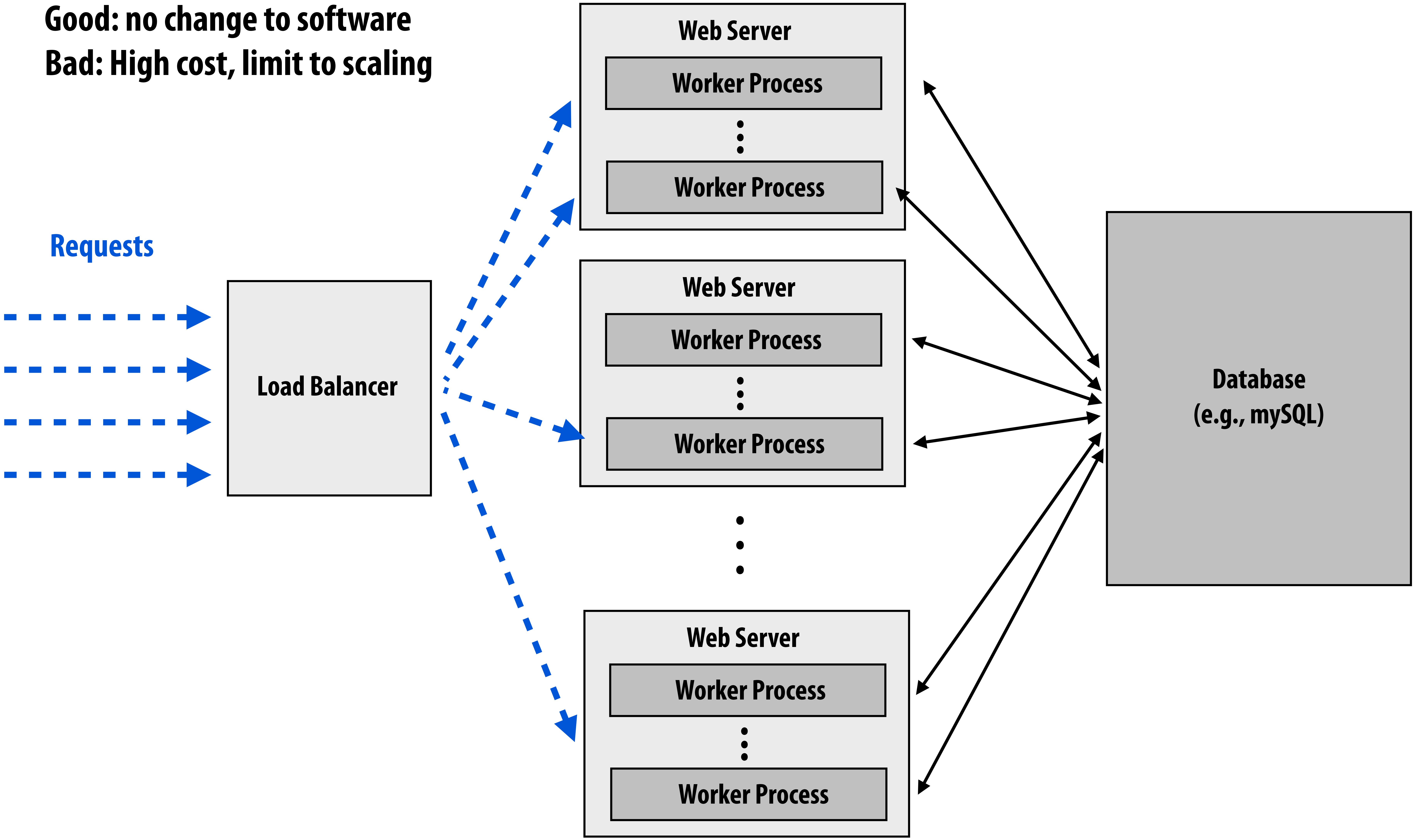


Dealing with database contention

Option 1: "scale up": buy better hardware for database server, buy professional-grade DB that scales (see database systems course by Prof. Pavlo)

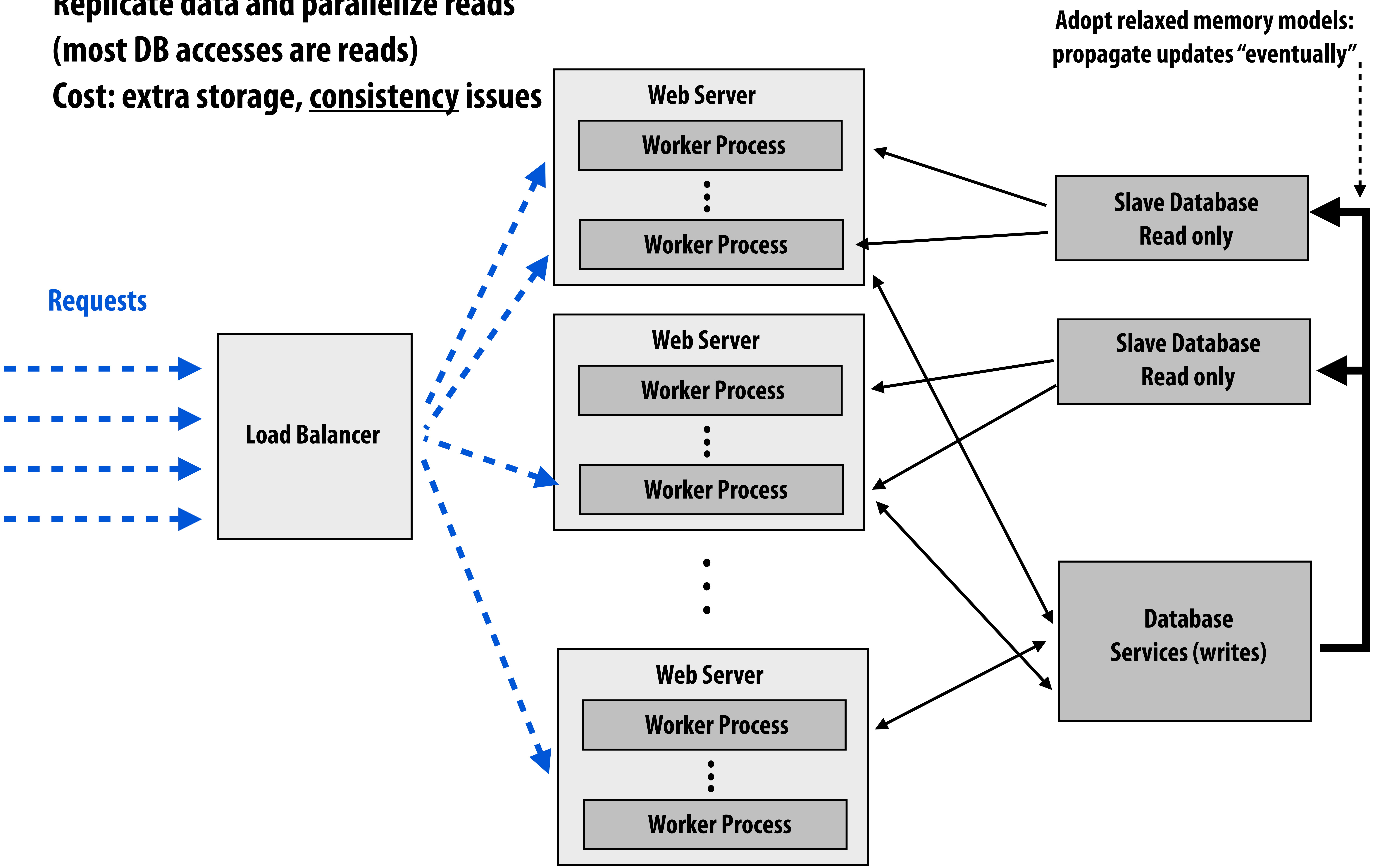
Good: no change to software

Bad: High cost, limit to scaling

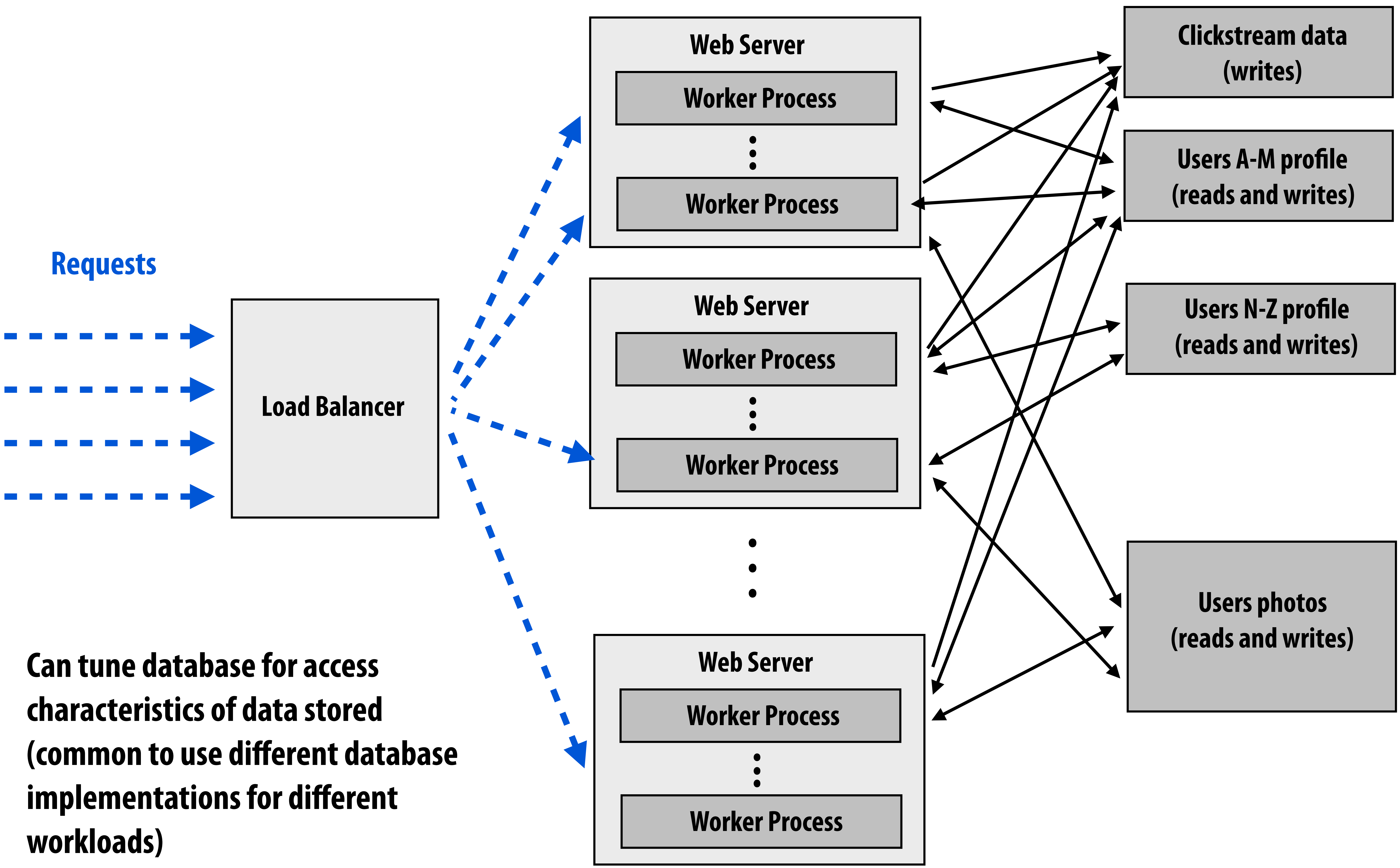


Scaling out a database: replicate

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



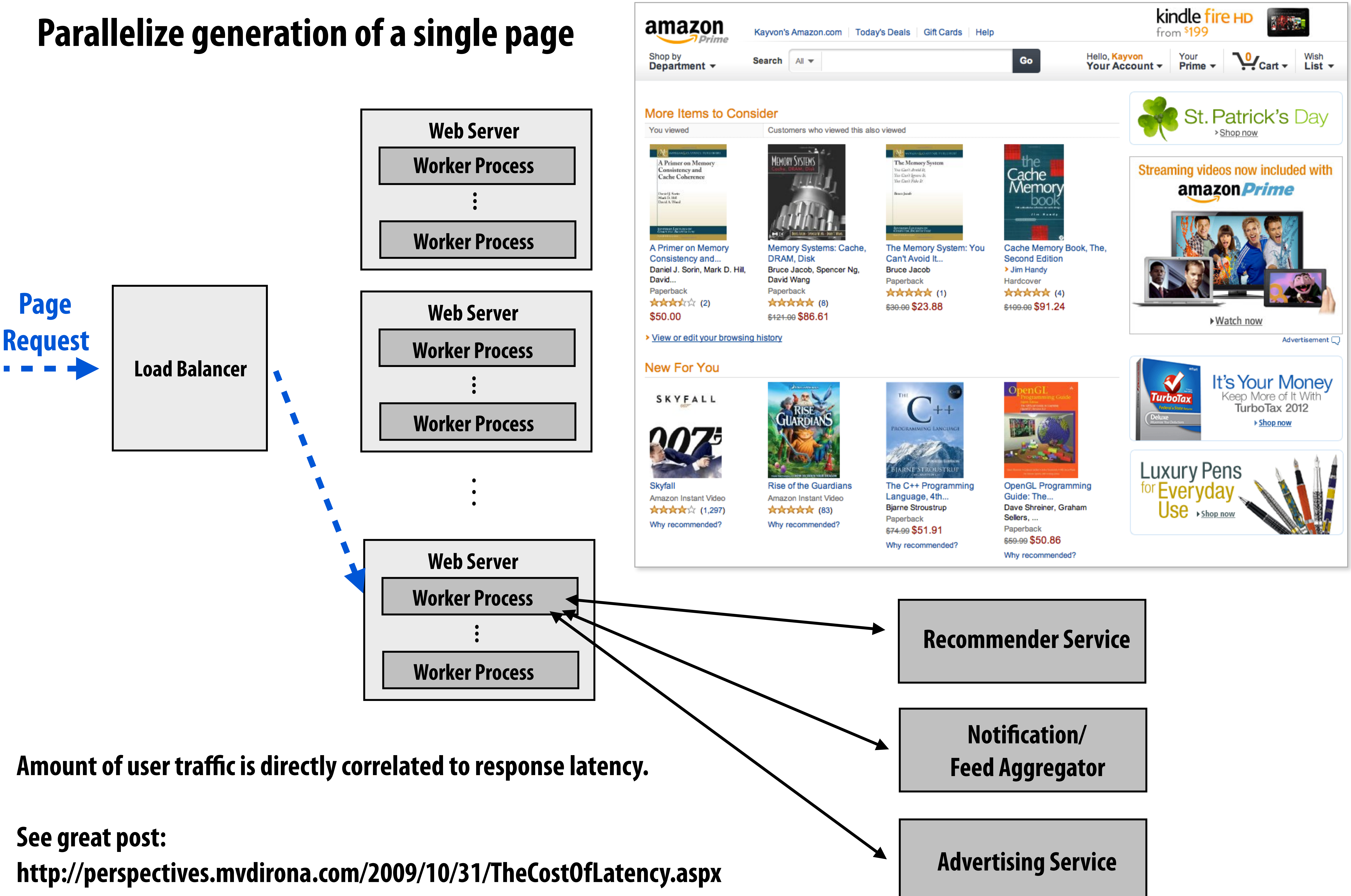
Scaling out a database: partition



Can tune database for access characteristics of data stored (common to use different database implementations for different workloads)

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

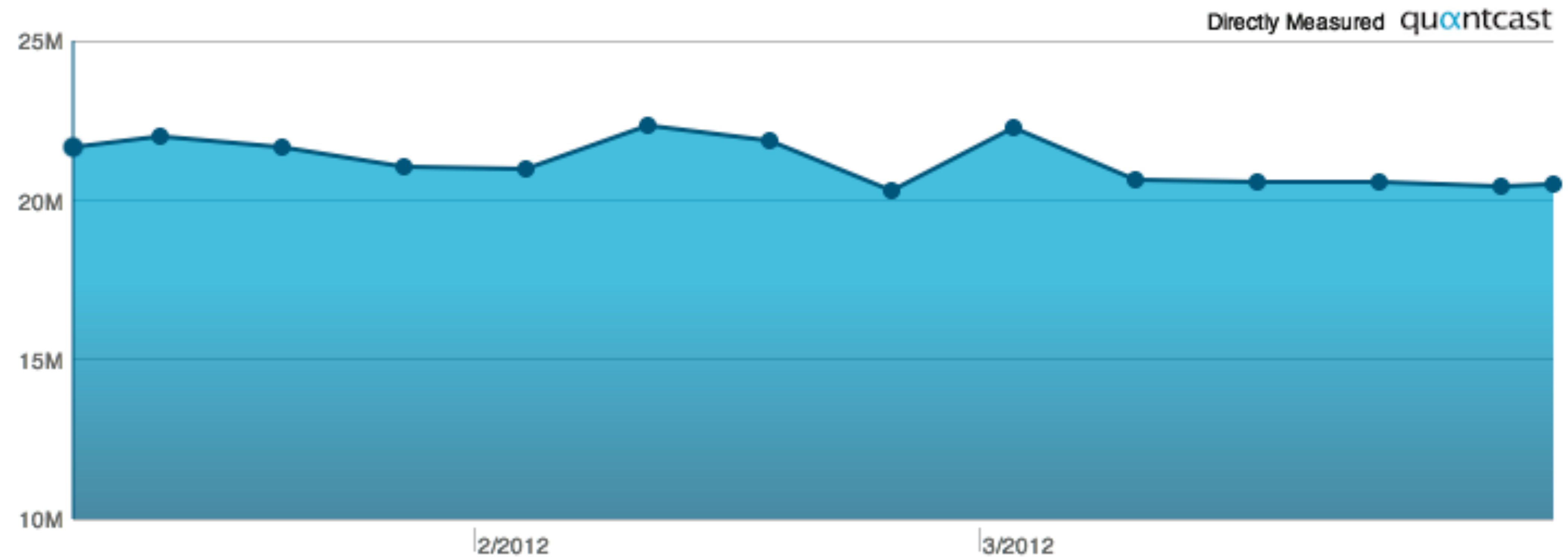
How many web servers do you need?

Web traffic is bursty

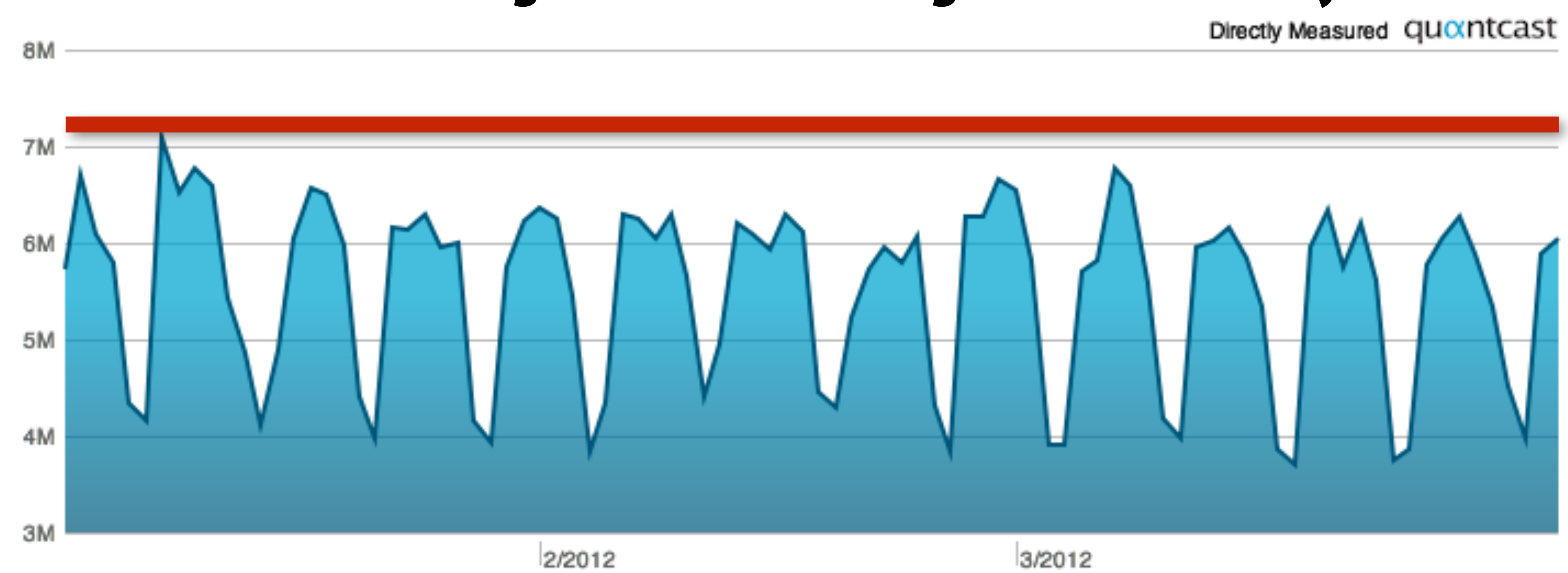
Amazon.com Page Views

Holiday shopping season

HuffingtonPost.com Page Views Per Week



HuffingtonPost.com Page Views Per Day

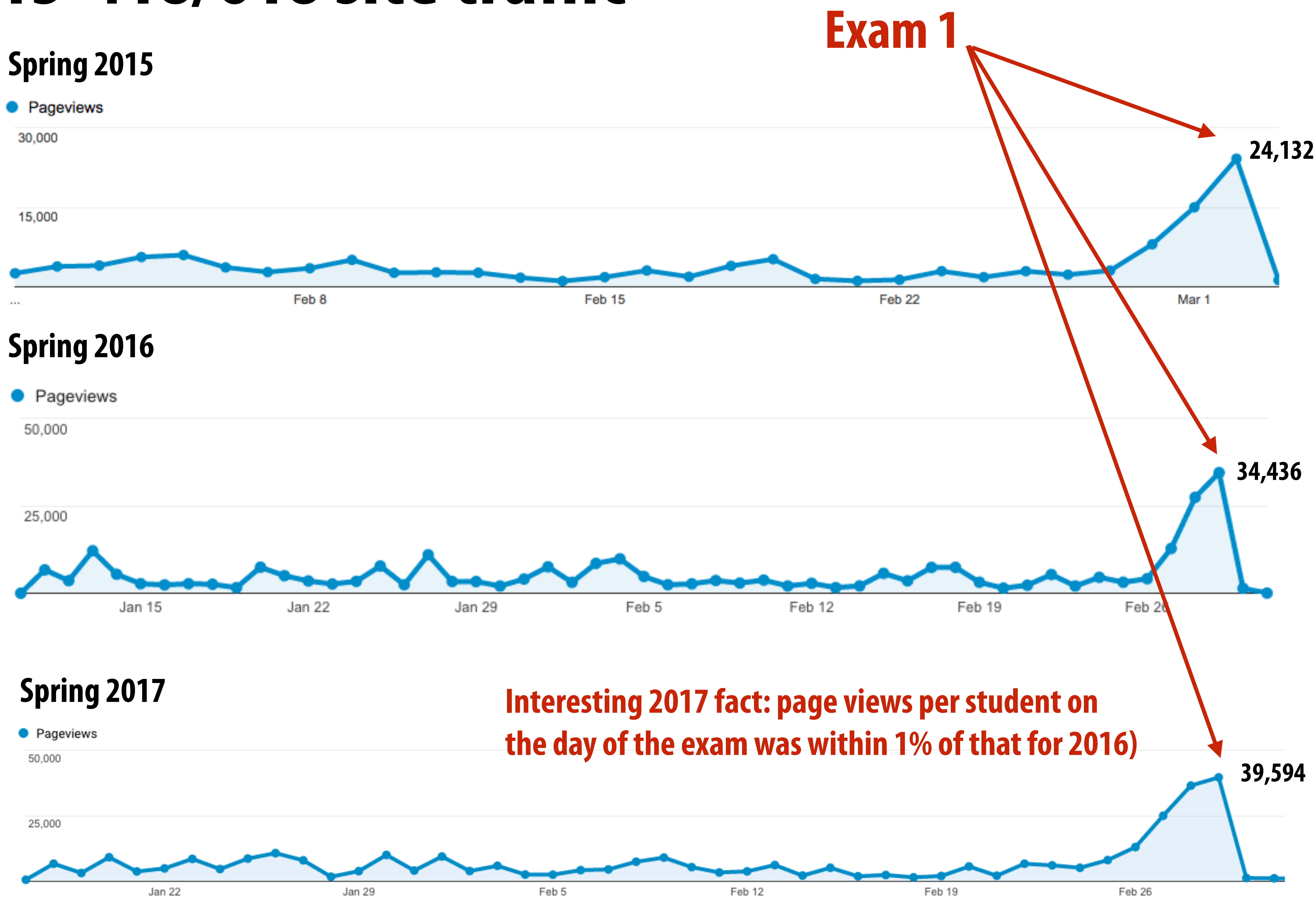


(fewer people read news on weekends)

More examples:

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

15-418/618 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 removes web servers from worker pool based on measured load

- Need source of servers available on-demand
  - Amazon.com EC2 instances
  - Google Cloud Platform
  - Microsoft Azure

Example: Amazon's elastic compute cloud (EC2)

Amazon.com Page Views

- Amazon had an over-provisioning problem
  - Need to provision for e-commerce bursts to avoid losing sales
  - Unused capacity during large parts of the year

- 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 EC2 US West (Oregon) on-demand pricing

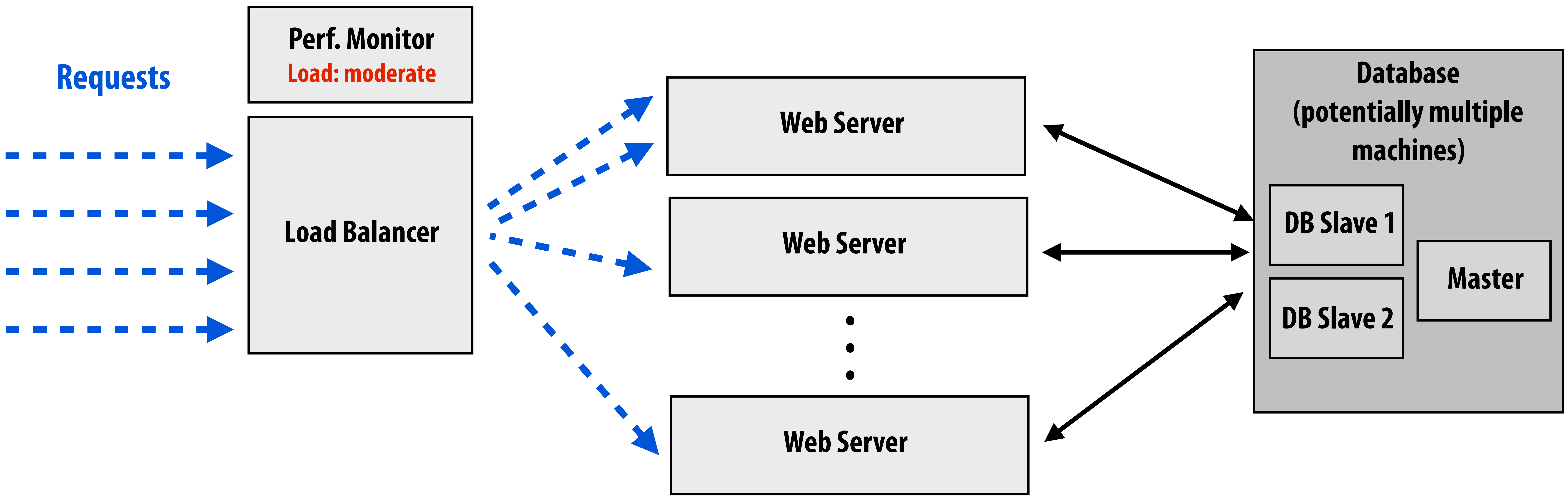
1 vCPU ~ 1 hyper thread on a "Haswell" E5-2666 v3 CPU

1 Tesla K80 → p2.xlarge

8 Tesla K80s → p2.8xlarge

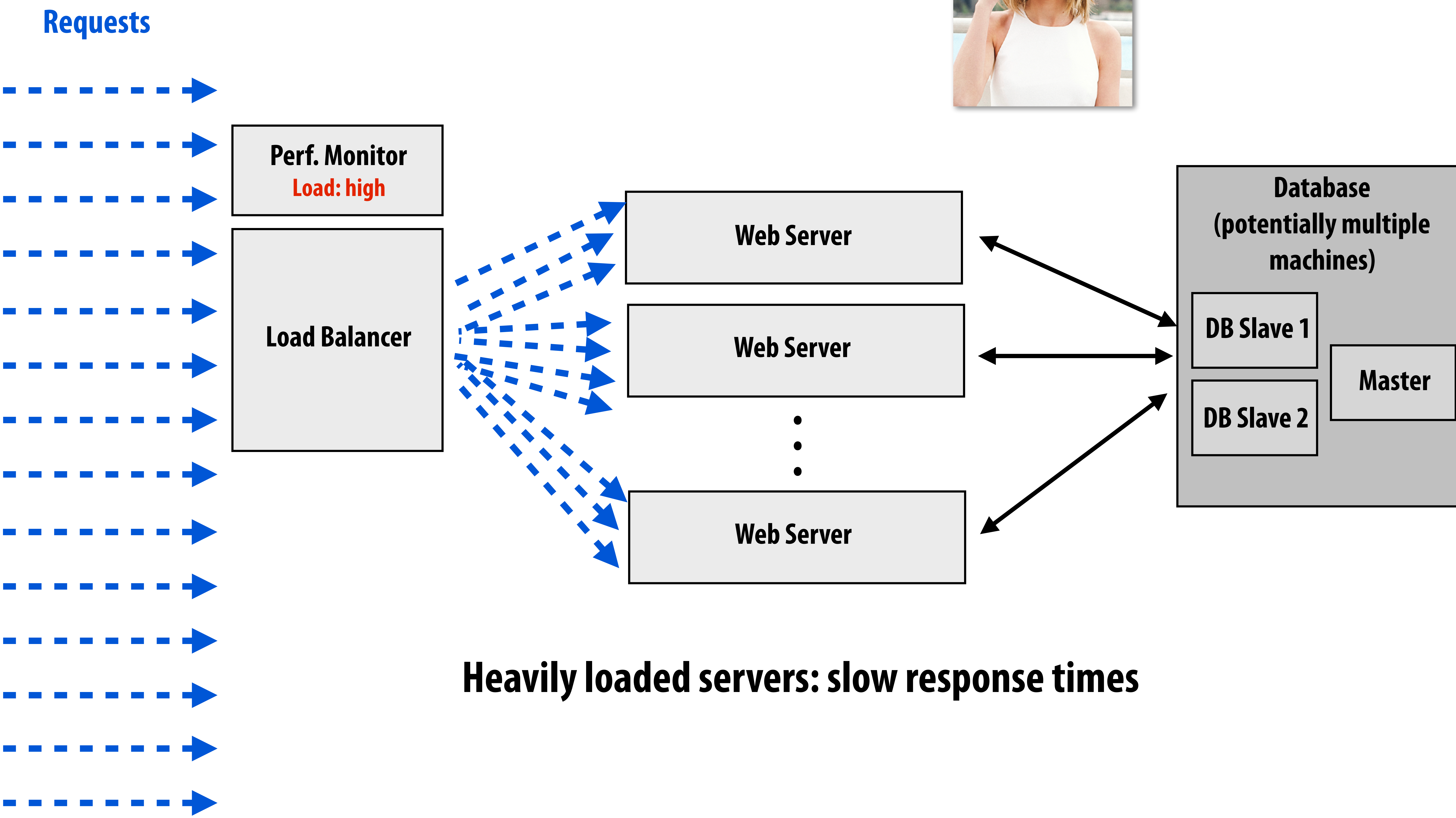
	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
Compute Optimized - Current Generation					
c4.large	2	8	3.75	EBS Only	$0.1 per Hour
c4.xlarge	4	16	7.5	EBS Only	$0.199 per Hour
c4.2xlarge	8	31	15	EBS Only	$0.398 per Hour
c4.4xlarge	16	62	30	EBS Only	$0.796 per Hour
c4.8xlarge	36	132	60	EBS Only	$1.591 per Hour
c3.large	2	7	3.75	2 x 16 SSD	$0.105 per Hour
c3.xlarge	4	14	7.5	2 x 40 SSD	$0.21 per Hour
c3.2xlarge	8	28	15	2 x 80 SSD	$0.42 per Hour
c3.4xlarge	16	55	30	2 x 160 SSD	$0.84 per Hour
c3.8xlarge	32	108	60	2 x 320 SSD	$1.68 per Hour
GPU Instances - Current Generation					
p2.xlarge	4	12	61	EBS Only	$0.9 per Hour
p2.8xlarge	32	94	488	EBS Only	$7.2 per Hour
p2.16xlarge	64	188	732	EBS Only	$14.4 per Hour
g2.2xlarge	8	26	15	60 SSD	$0.65 per Hour
g2.8xlarge	32	104	60	2 x 120 SSD	$2.6 per Hour
Memory Optimized - Current Generation					
x1.16xlarge	64	174.5	976	1 x 1920 SSD	$6.669 per Hour
x1.32xlarge	128	349	1952	2 x 1920 SSD	$13.338 per Hour
r3.large	2	6.5	15	1 x 32 SSD	$0.166 per Hour

Site configuration: normal load



Event triggers spike in load

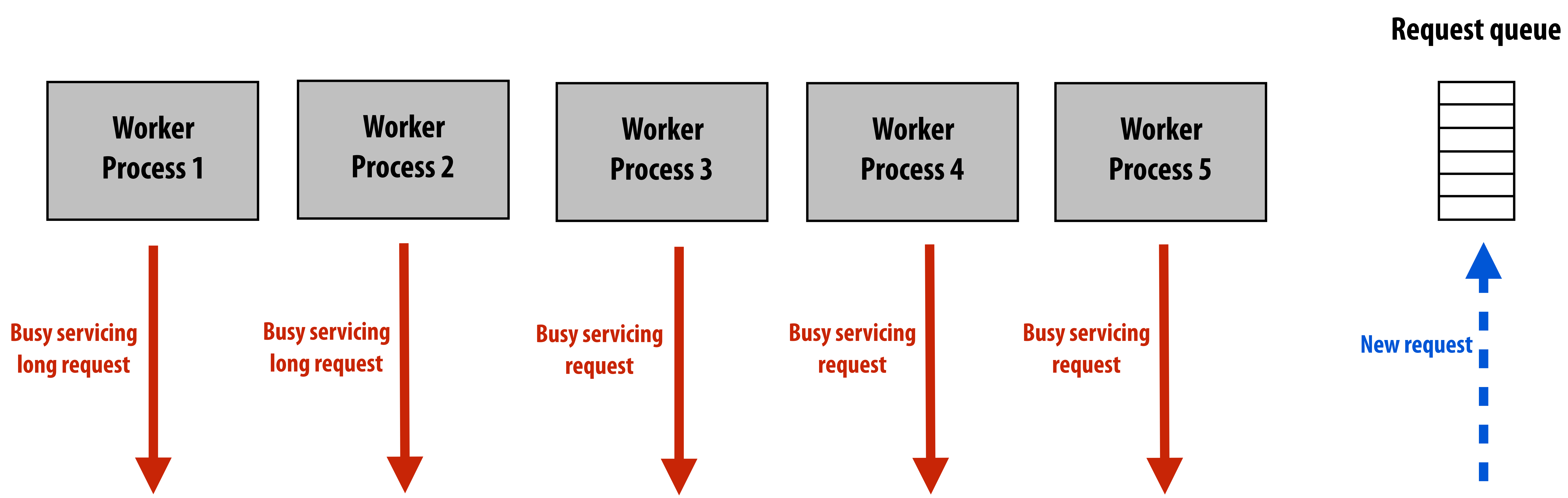
@taylorswift13: parallel class @ CMU is the bomb check it out! #15418.



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
- Latency 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 throughput change under heavy load?

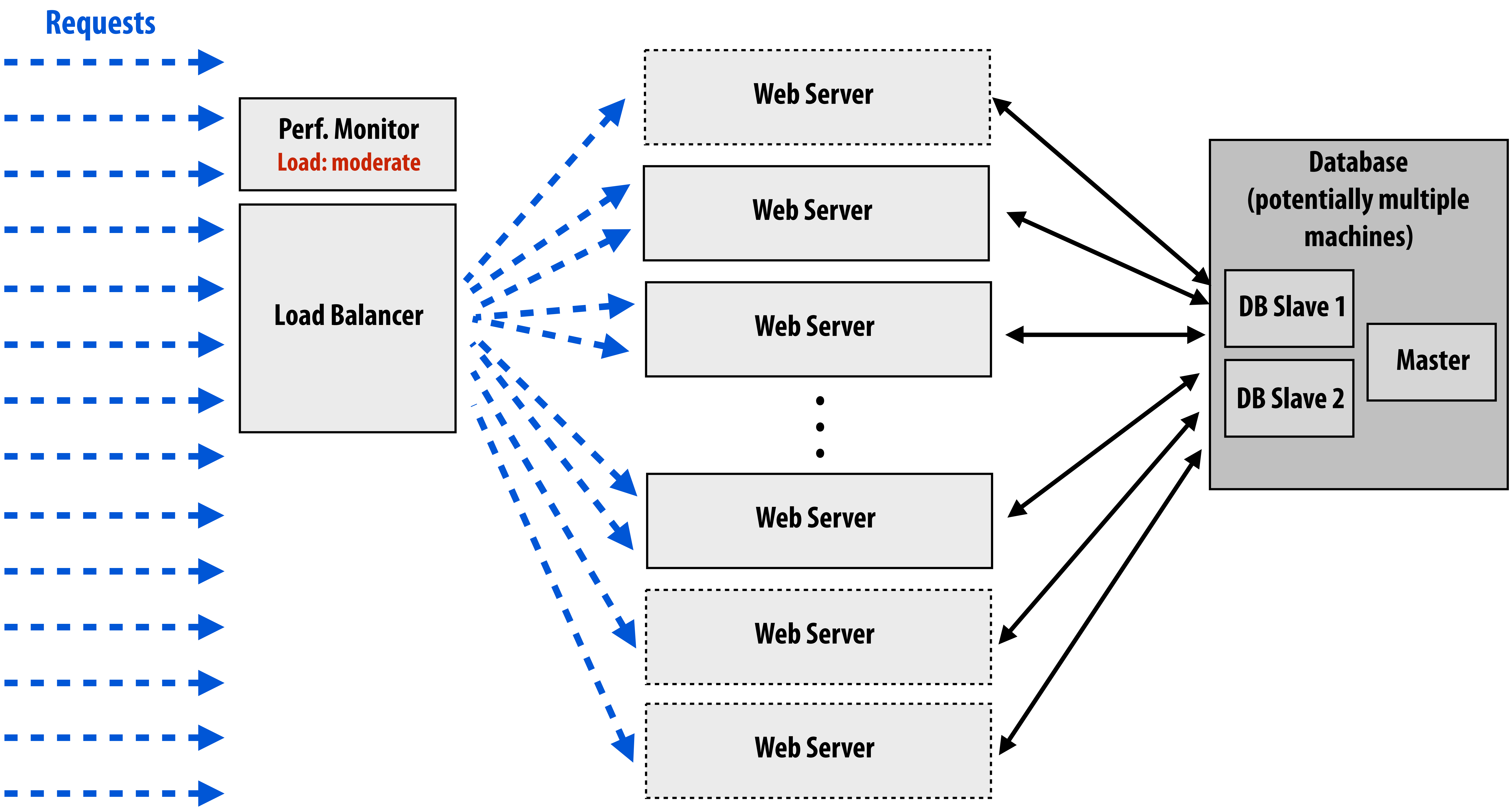


Site configuration: high load

Site performance monitor detects high load

Instantiates new web server instances

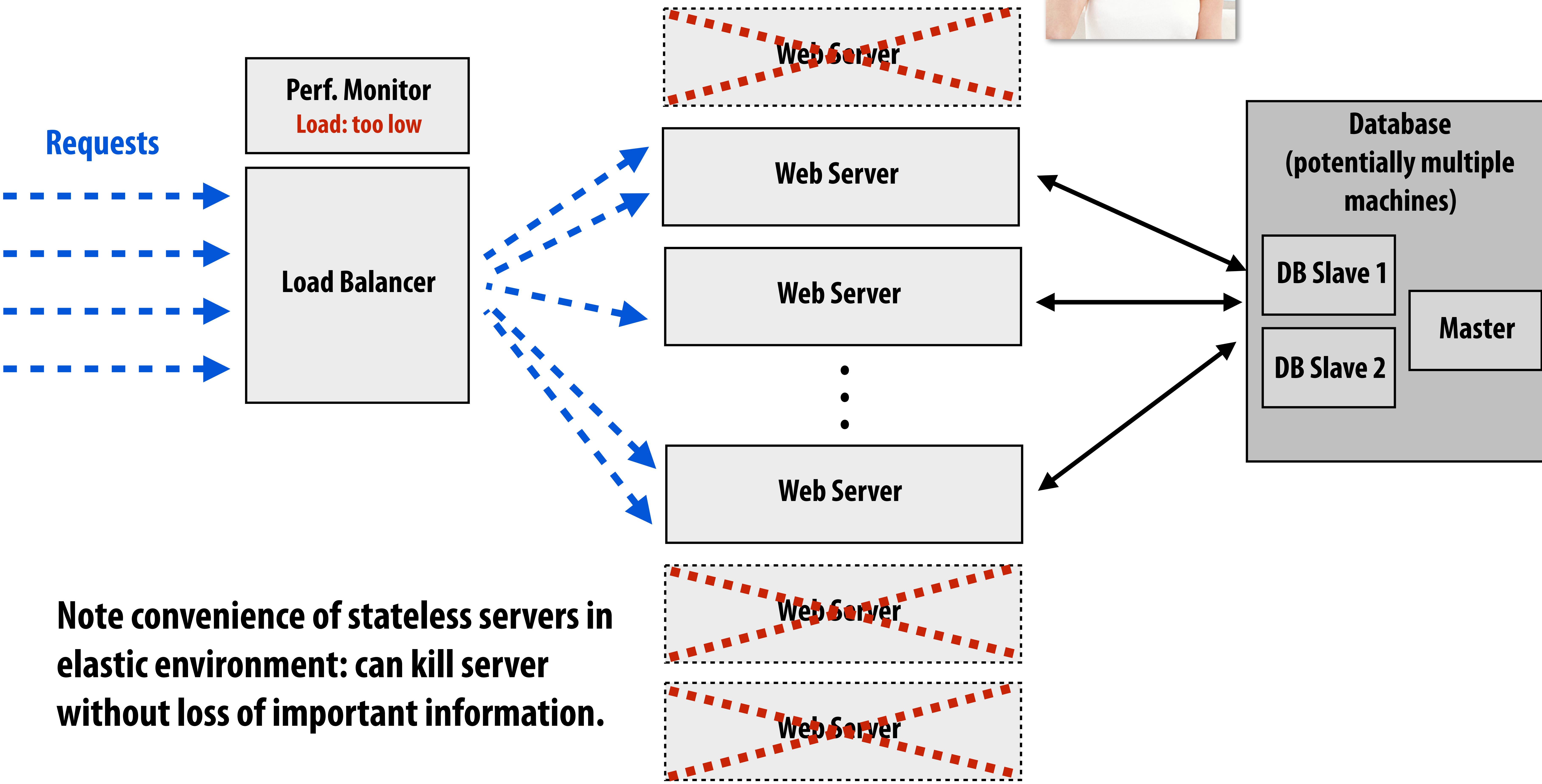
Informs load balancer about presence of new servers



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

@taylorswift13: hard midterm? Shake it off watchin' my new vids.



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

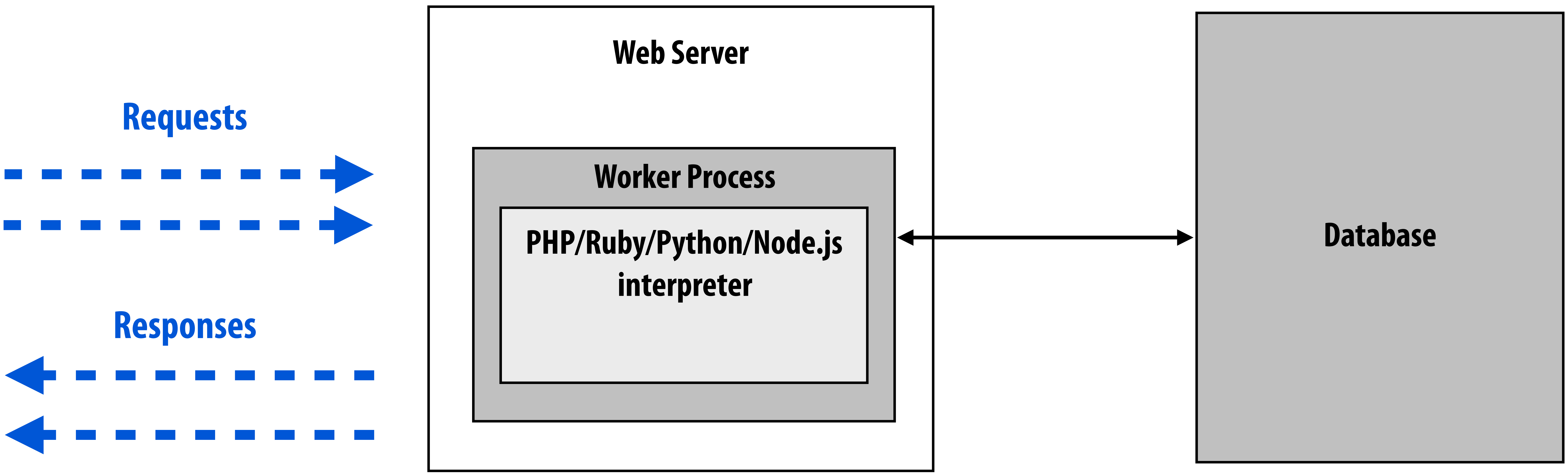
Amazon Elastic Beanstalk

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

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

Now: reuse and locality

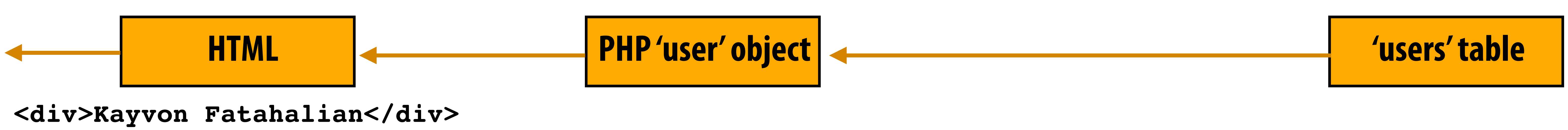
Recall: basic site configuration



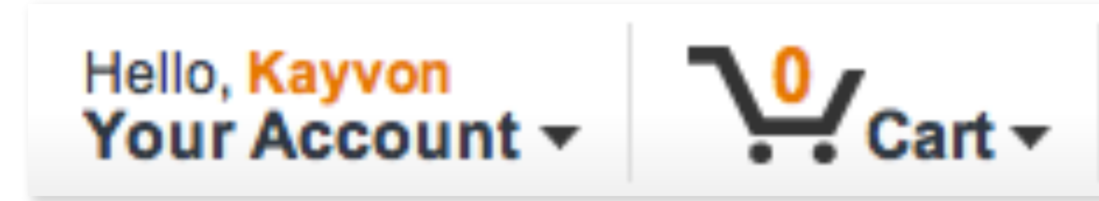
Example PHP Code

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

echo "<div>" . $user['FirstName'] . " " . $user['LastName'] . "</div>";
```

Response Information Flow



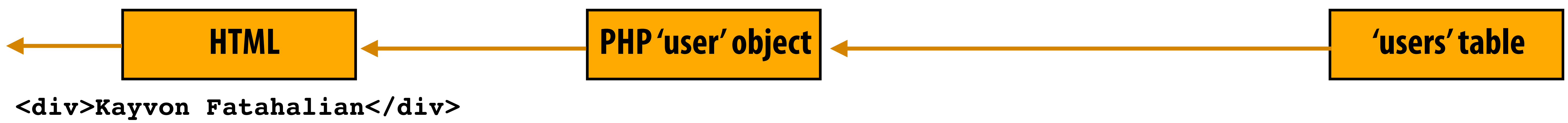
Work repeated every page



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

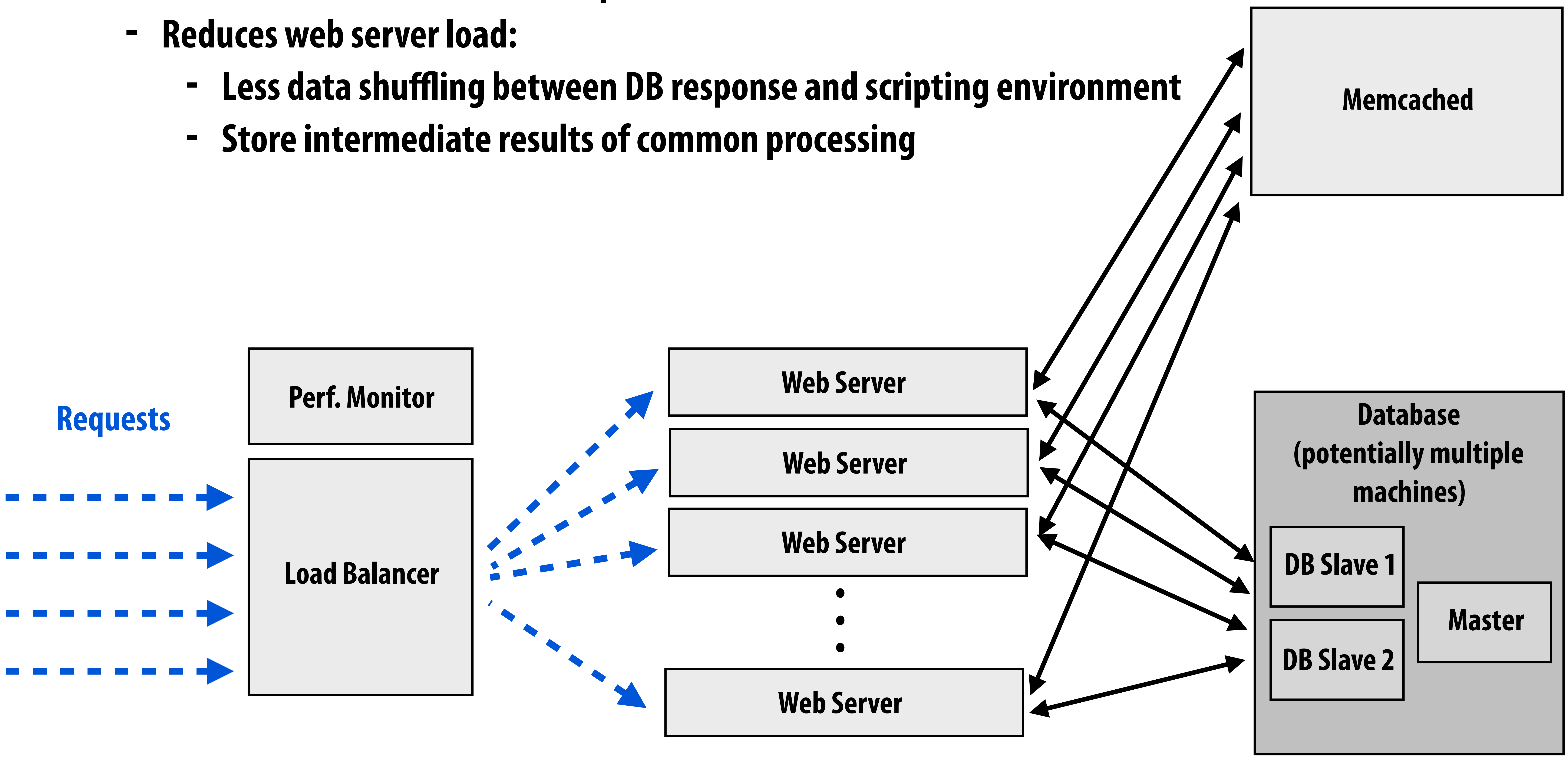


- Steps repeated to emit my name at the top of every page:
  - Communicate with DB
  - Perform query
  - Marshall results from database into object model of scripting language
  - Generate presentation
  - etc...

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

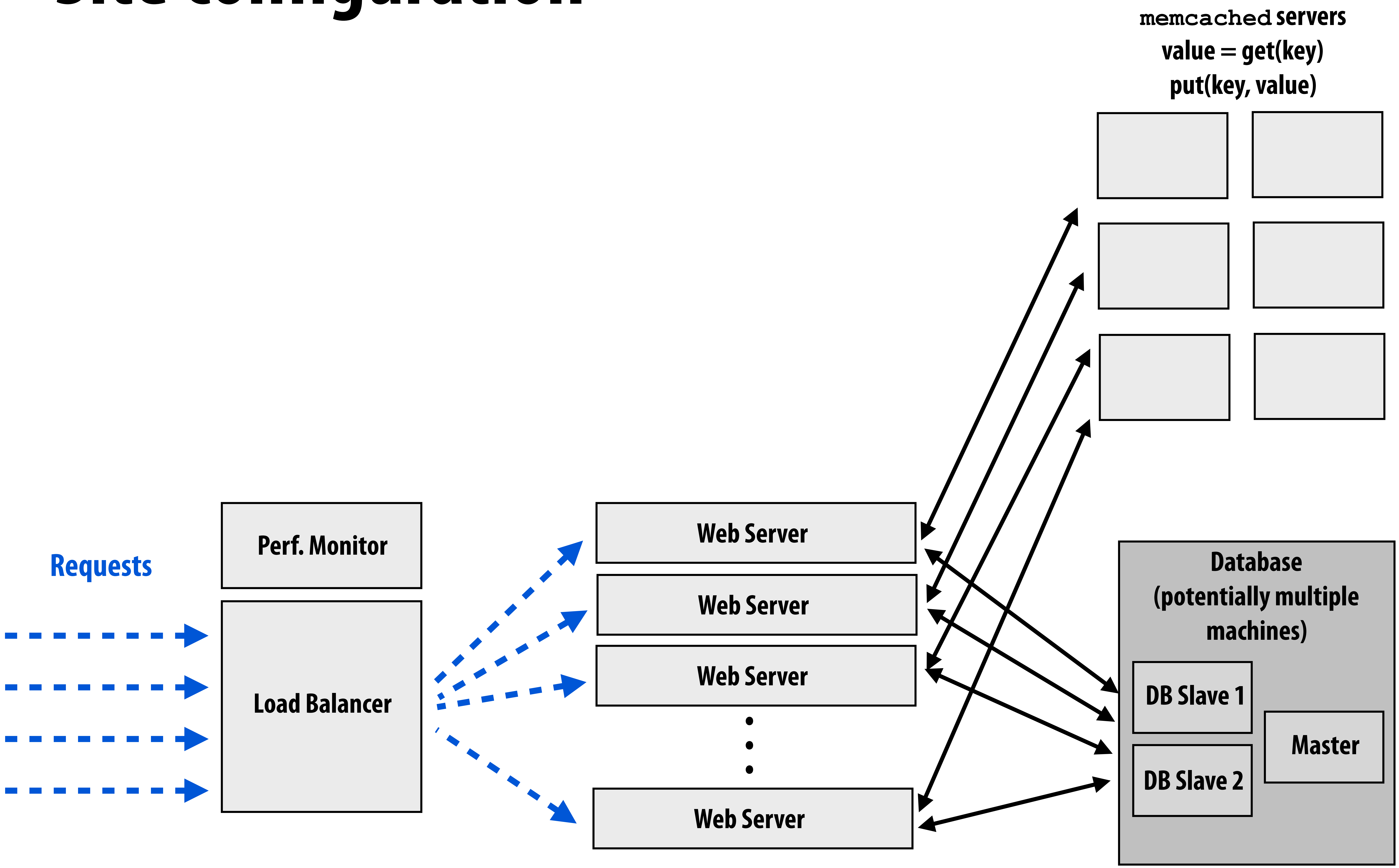


Caching example

```
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

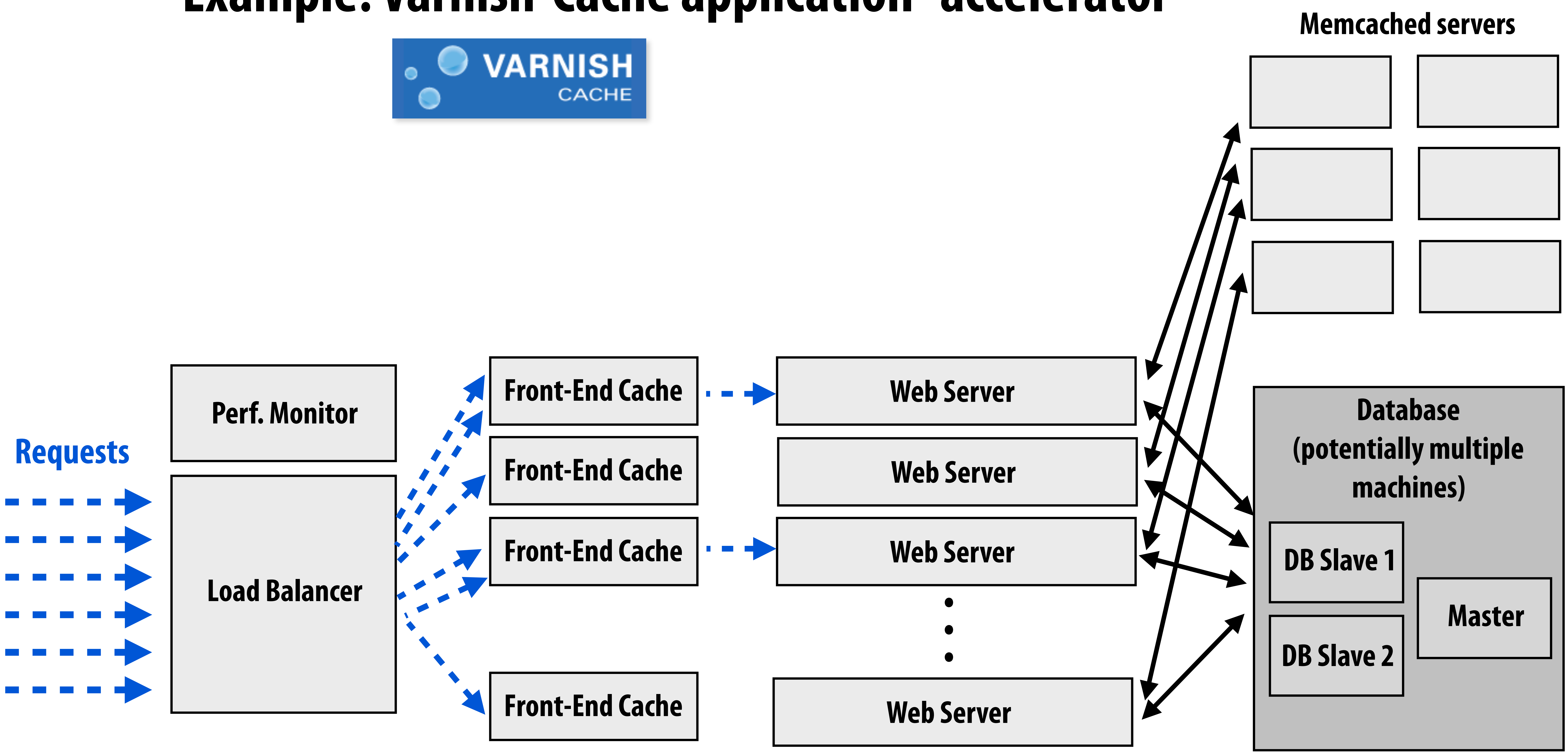
- Facebook, circa 2008
  - 800 memcached servers
  - 28 TB of cached data

- 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

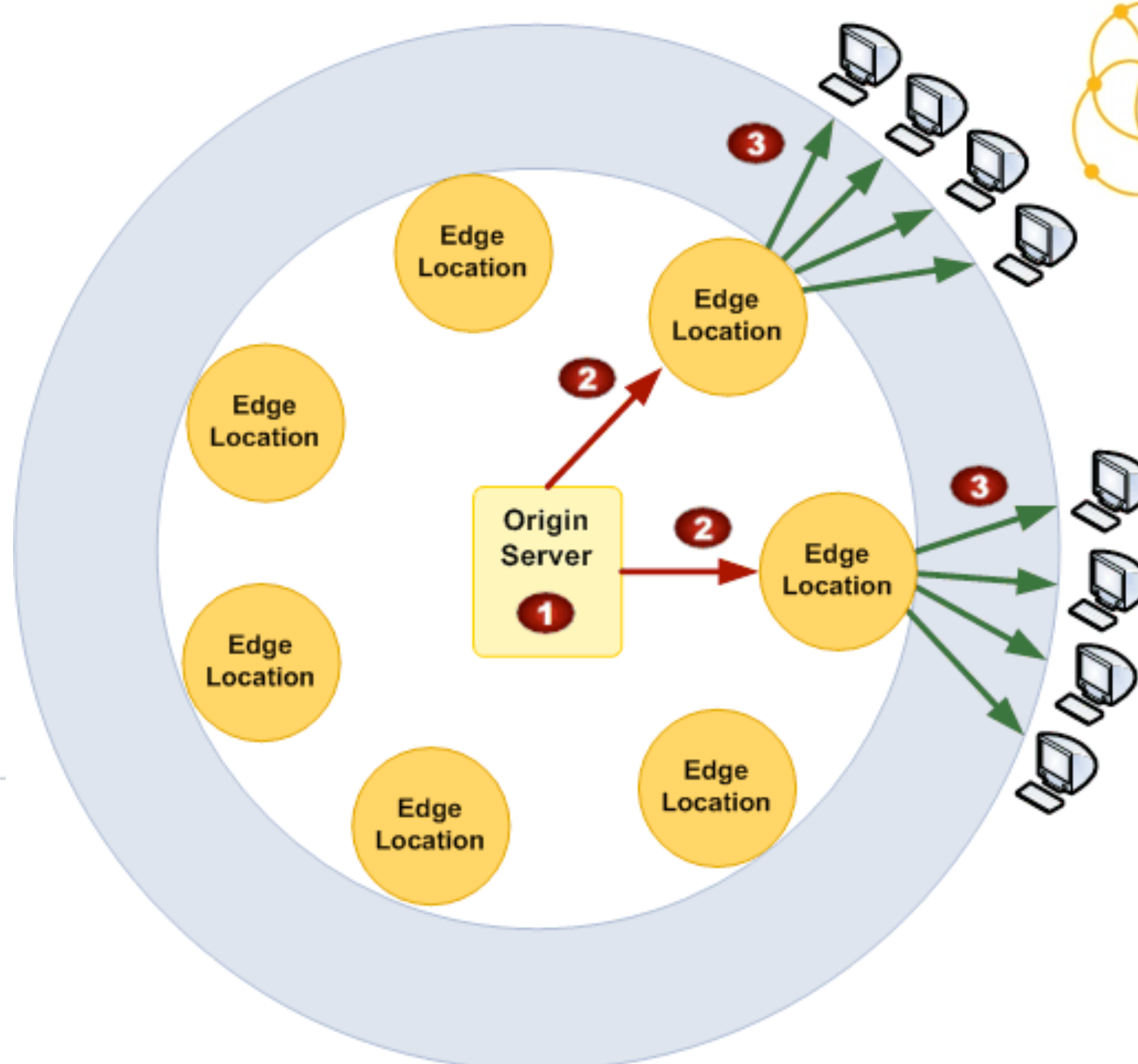
More caching

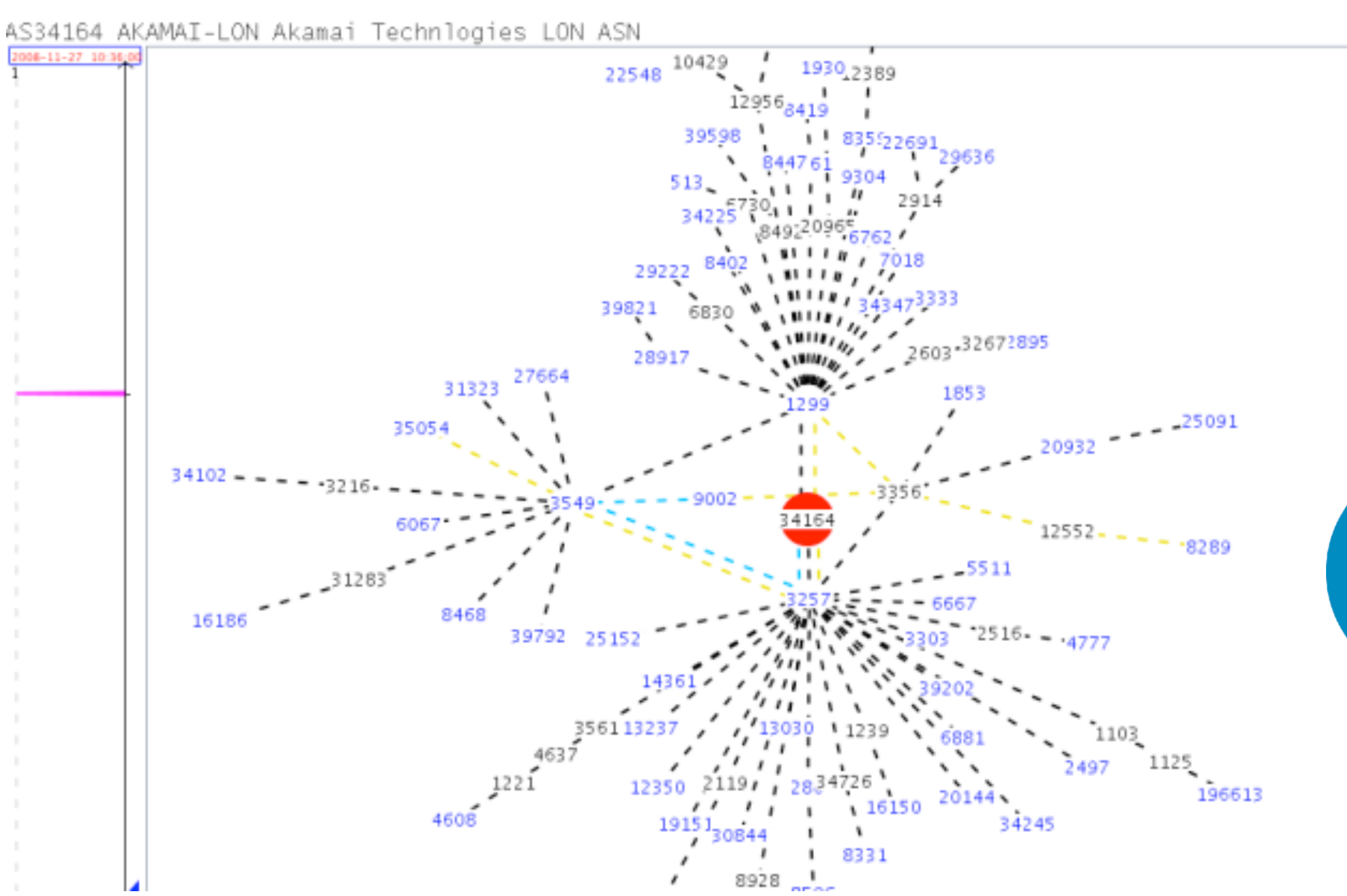
- Cache web server responses (e.g. entire pages, pieces of pages)
  - Reduce load on web servers
  - Example: Varnish-Cache application "accelerator"



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





London Content Distribution Network

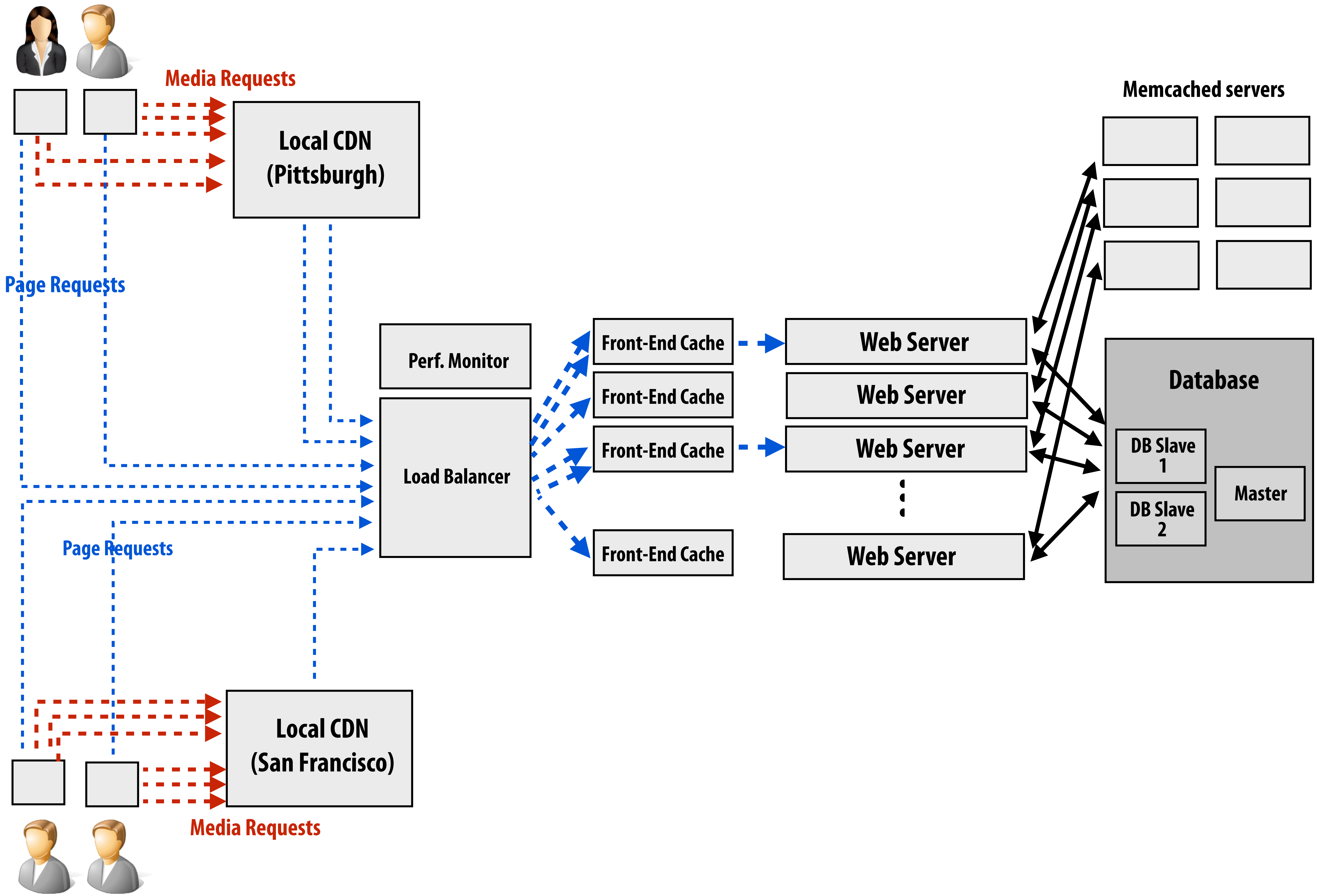
Source: http://www.telco2.net/blog/2008/11/amazon_cloudfront_yet_more_tra.html

CDN usage example (Facebook photos)

Facebook page URL: (you can't get here since you aren't a friend on my photos access list)
https://www.facebook.com/photo.php?fbid=10153875308143897&set=a.10150275074093897.338852.722973896&type=3&theater

Image source URL: (you can definitely see this photo… try it!)
https://scontent.fagc2-1.fna.fbcdn.net/v/t1.0-9/13466473_10153875308143897_4595852336757037043_n.jpg?oh=f5aac709574b85e58d14534a8770cecb&oe=5973BB23

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
  - Good example: different databases for different types of requests

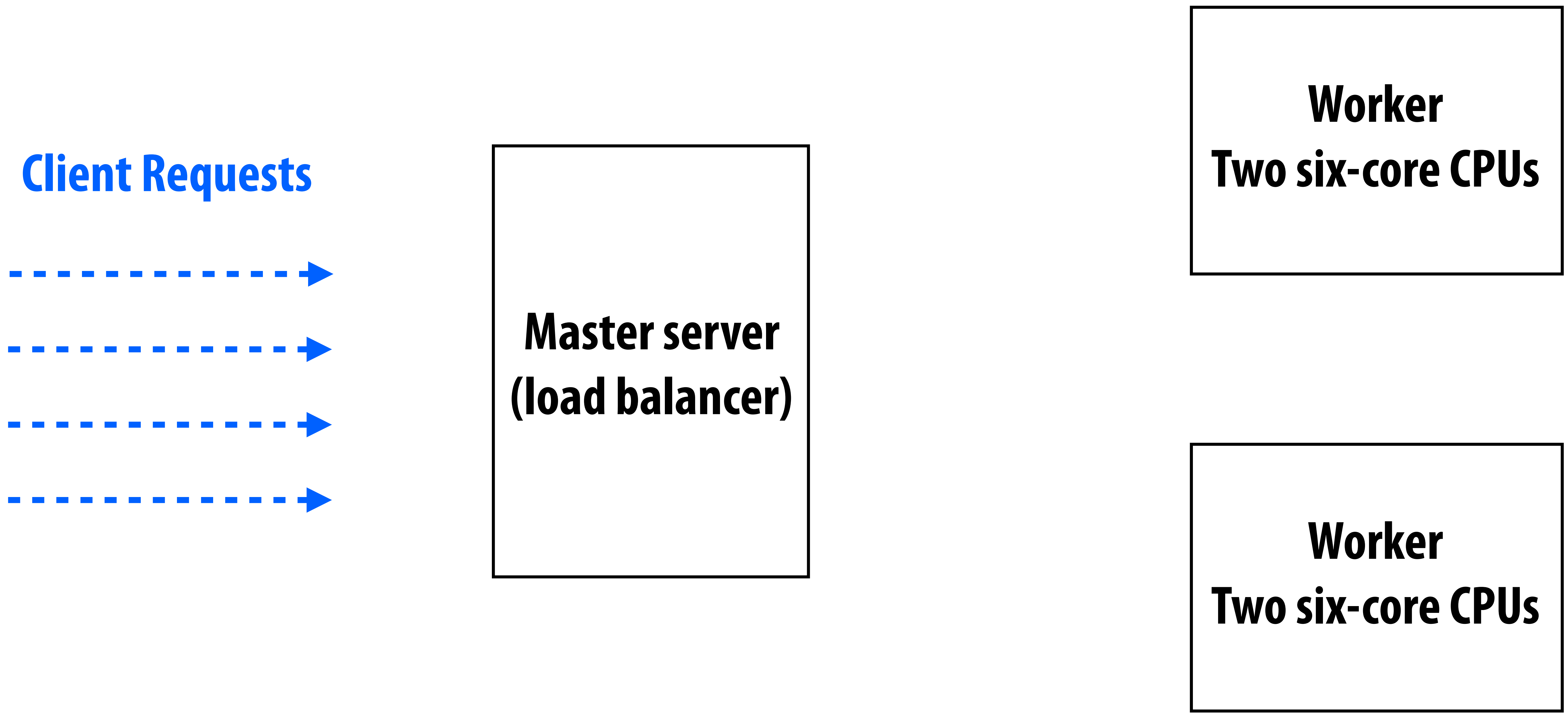
Final comments

- 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 quick 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

Assignment 4

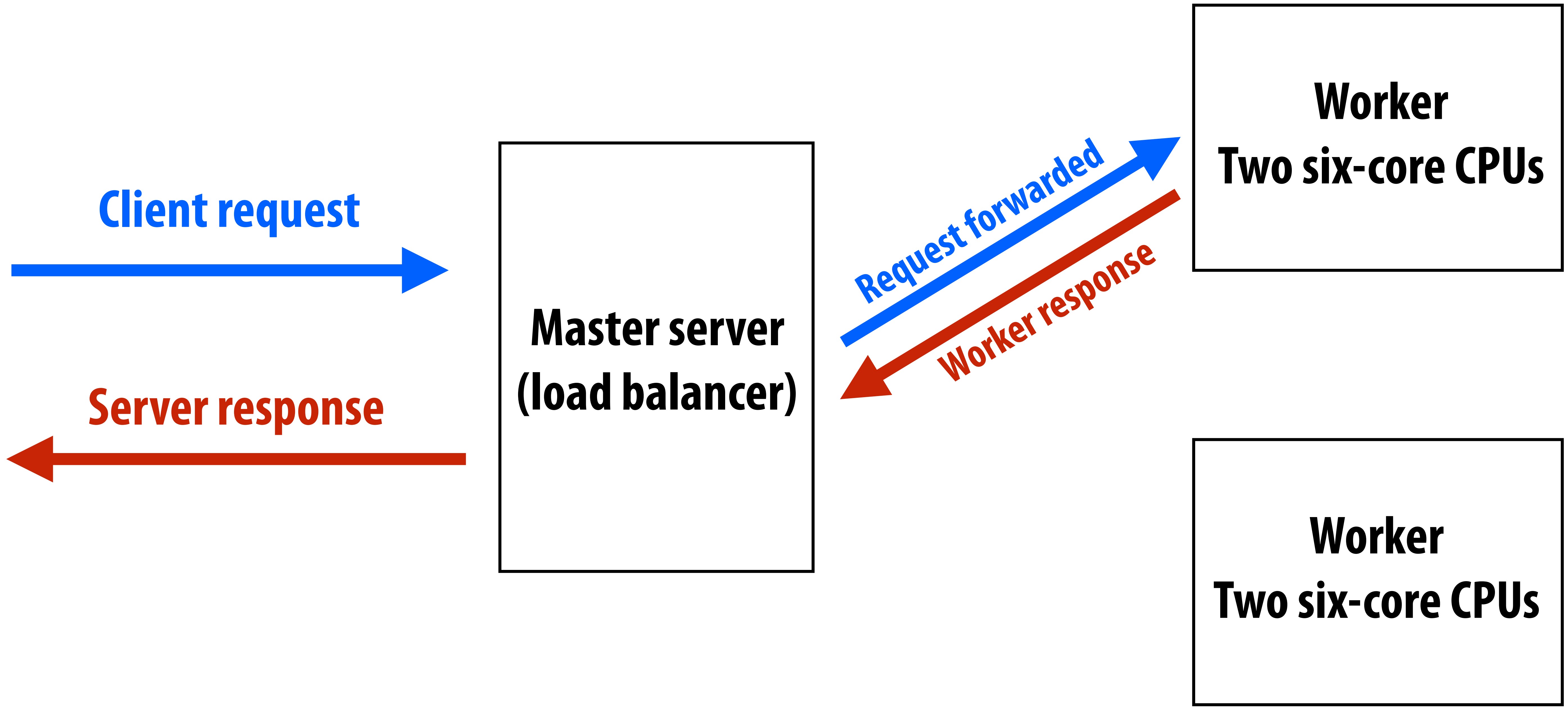
Assignment 4

- You will implement a simple web site that efficiently handles a request stream



Assignment 4

- You will implement a load balancer/scheduler to efficiently handle a request stream



Assignment 4: the master node

- The master is a load balancer
- The master is structured as an event-driven system
  - The master has only one thread of control, but the server as a whole processes client requests concurrently

Master server (load balancer)

You implement:

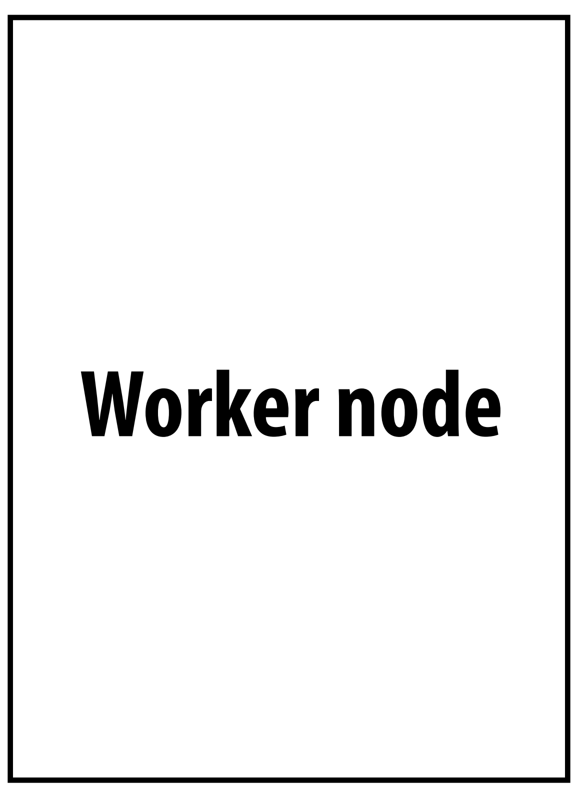
```
// take action when a request comes in
void handle_client_request(Client_handle client_handle, const RequestMsg& req);

// take action when a worker provides a response
void handle_worker_response(Worker_handle worker_handle, const ResponseMsg& resp);
```

We give you:

```
// sends a request to a worker
void send_job_to_worker(Worker_handle worker_handle, const RequestMsg& req);

// sends a response to the client
void send_client_response(Client_handle client_handle, const ResponseMsg& resp);
```

Assignment 4: the worker nodes

- The worker nodes are responsible for the "heavy lifting" (executing the specified requests)



You implement:

```
// take action when a request comes in
void worker_handle_request(const RequestMsg& req);
```

We give you:

```
// send a response back to the master
void worker_send_response(const ResponseMsg& resp);

// black-box logic to actually do the work (and populate a response)
void execute_work(const RequestMsg& req, ResponseMsg& resp);
```

Assignment 4: challenge 1

- There a number of different types of requests with different workload characteristics
  - Compute intensive requests (both long and short)
  - Memory intensive requests…


```
{"time": 0,  "work": "cmd=highcompute;x=5", "resp": "42"}
{"time": 10, "work": "cmd=highcompute;x=10", "resp": "59"}
{"time": 20, "work": "cmd=highcompute;x=15", "resp": "78"}
{"time": 21, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 22, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 23, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 24, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 30, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 40, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 50, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
```

Assignment 4: challenge 2

- **The load varies over time! Your server must be elastic!**


```
{"time": 0,  "work": "cmd=highcompute;x=5", "resp": "42"}
{"time": 10, "work": "cmd=highcompute;x=10", "resp": "59"}
{"time": 20, "work": "cmd=highcompute;x=15", "resp": "78"}
{"time": 21, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 22, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 23, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 24, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 30, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 40, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 50, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
```


We give you:

```
// ask for another worker node
void request_boot_worker(int tag);

// request a worker be shut down
void kill_worker(Worker_handle worker_handle);
```

You implement:

```
// notification that the worker is up and running
void handle_worker_boot(Worker_handle worker_handle, int tag);
```

Assignment 4

- **Goal: service the request stream as efficiently as possible (low latency response time) using as few workers as possible (low website operation cost)**

- **Ideas you might want to consider:**
  - **What is a smart assignment of jobs (work) to workers?**
  - **When to [request more/release idle] worker nodes?**
  - **Can overall costs be reduced by caching?**