Lecture 15:

Scaling a Web Site

Scale-out Parallelism, Elasticity, and Caching

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2020

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

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

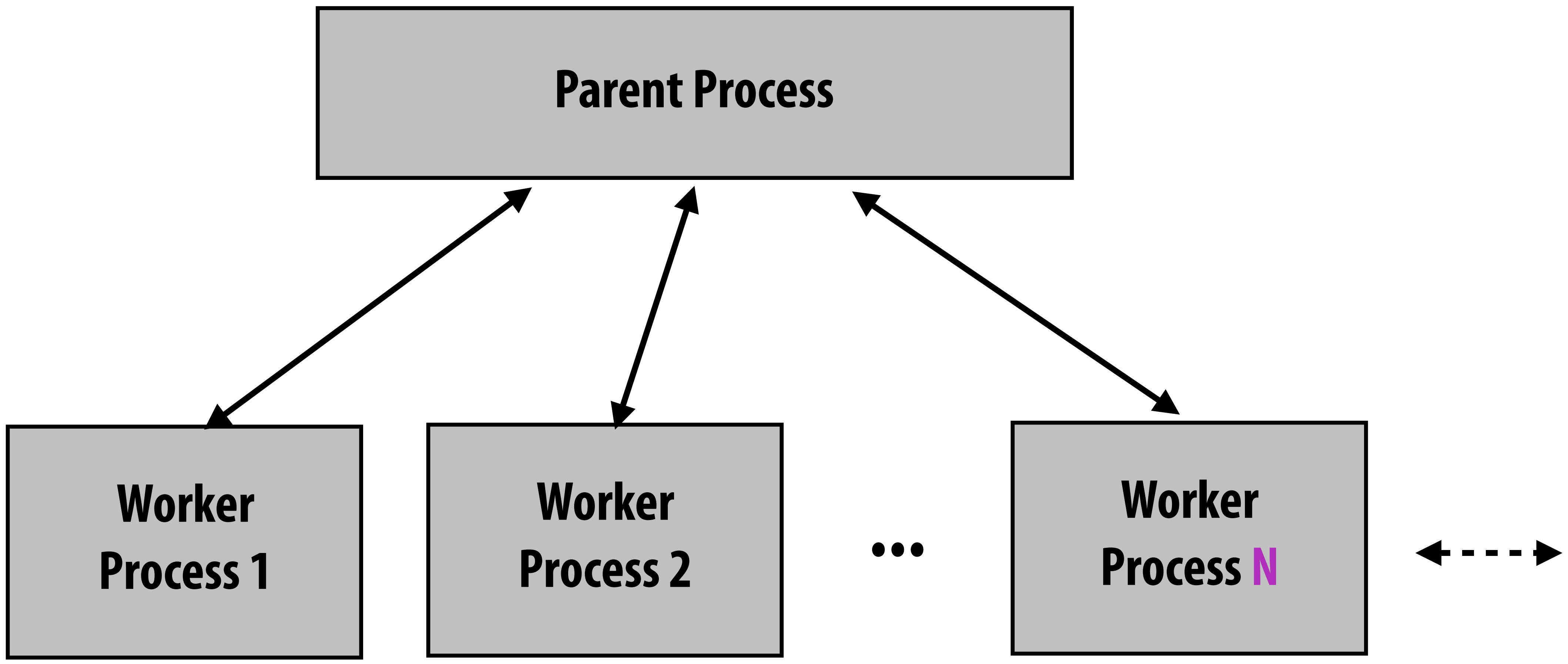
- Many other issues in developing a successful web platform
  - Reliability, security, privacy, etc.
  - There are other great courses at CMU for these topics (distributed systems, databases, cloud computing)

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

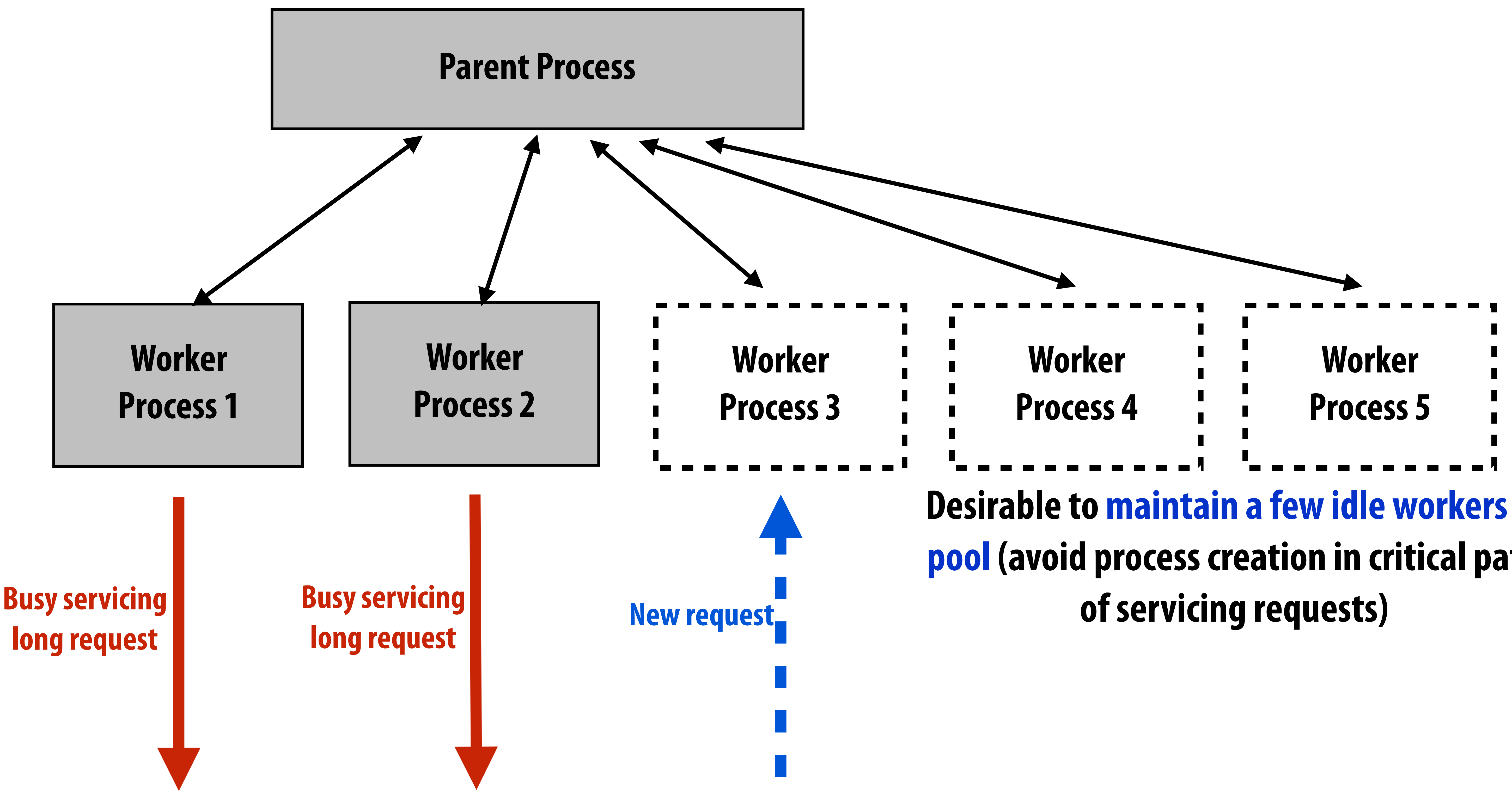



```
while (1)
{
    request = wait_for_request();

    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?

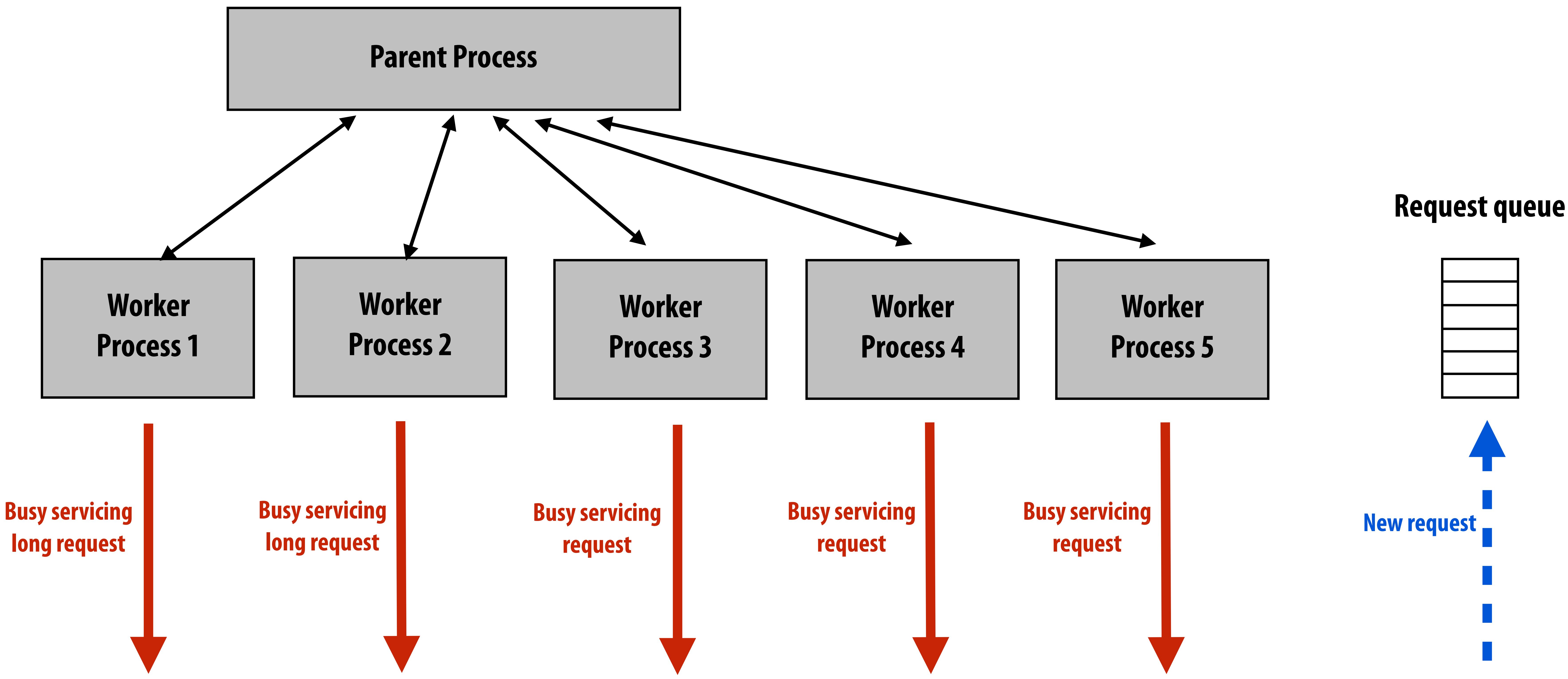
- Parallelism: use all the server's cores
- Latency hiding: hide long-latency disk read operations (by context switching between worker processes)
- Concurrency: many outstanding requests; service quick requests while long requests are in progress
  - (e.g., large file transfer shouldn't block serving 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`

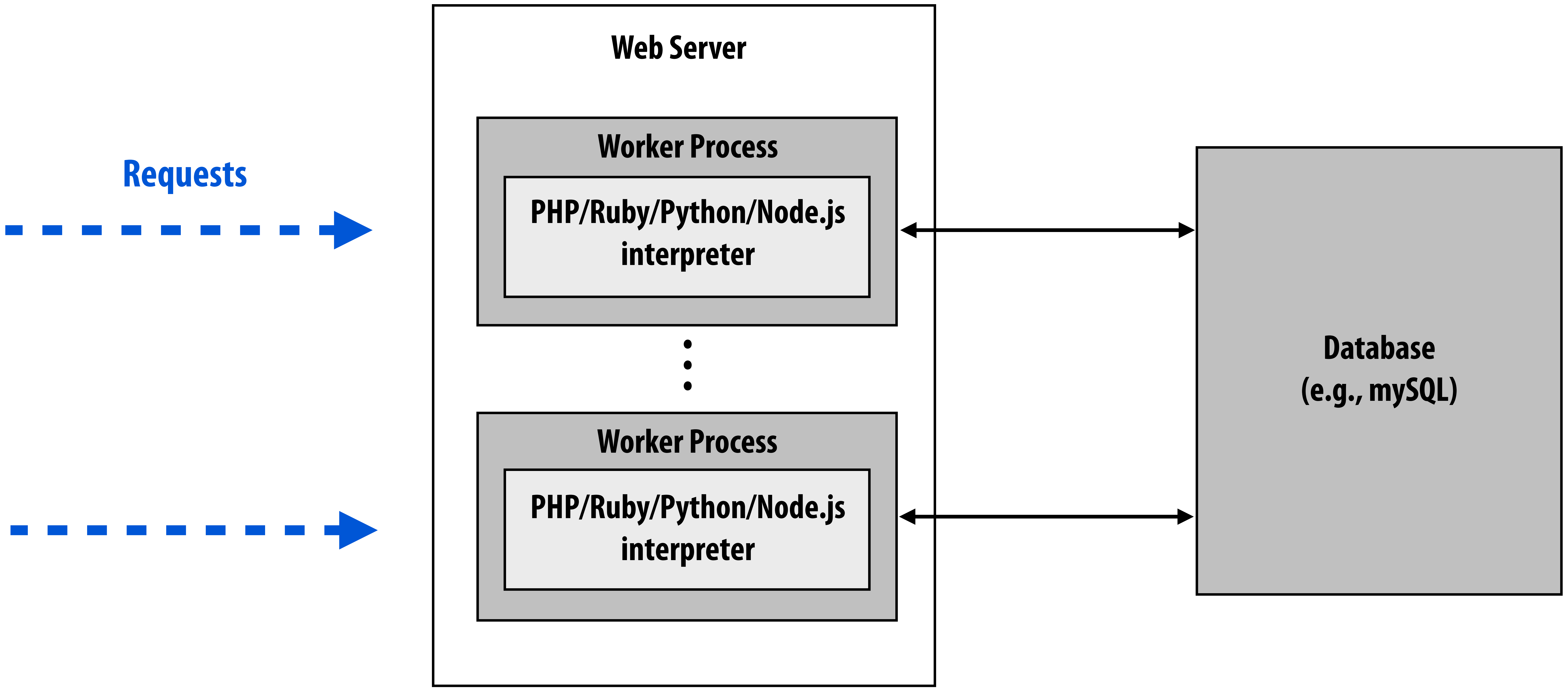
Aside: why partition server into processes, not threads?

- **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)

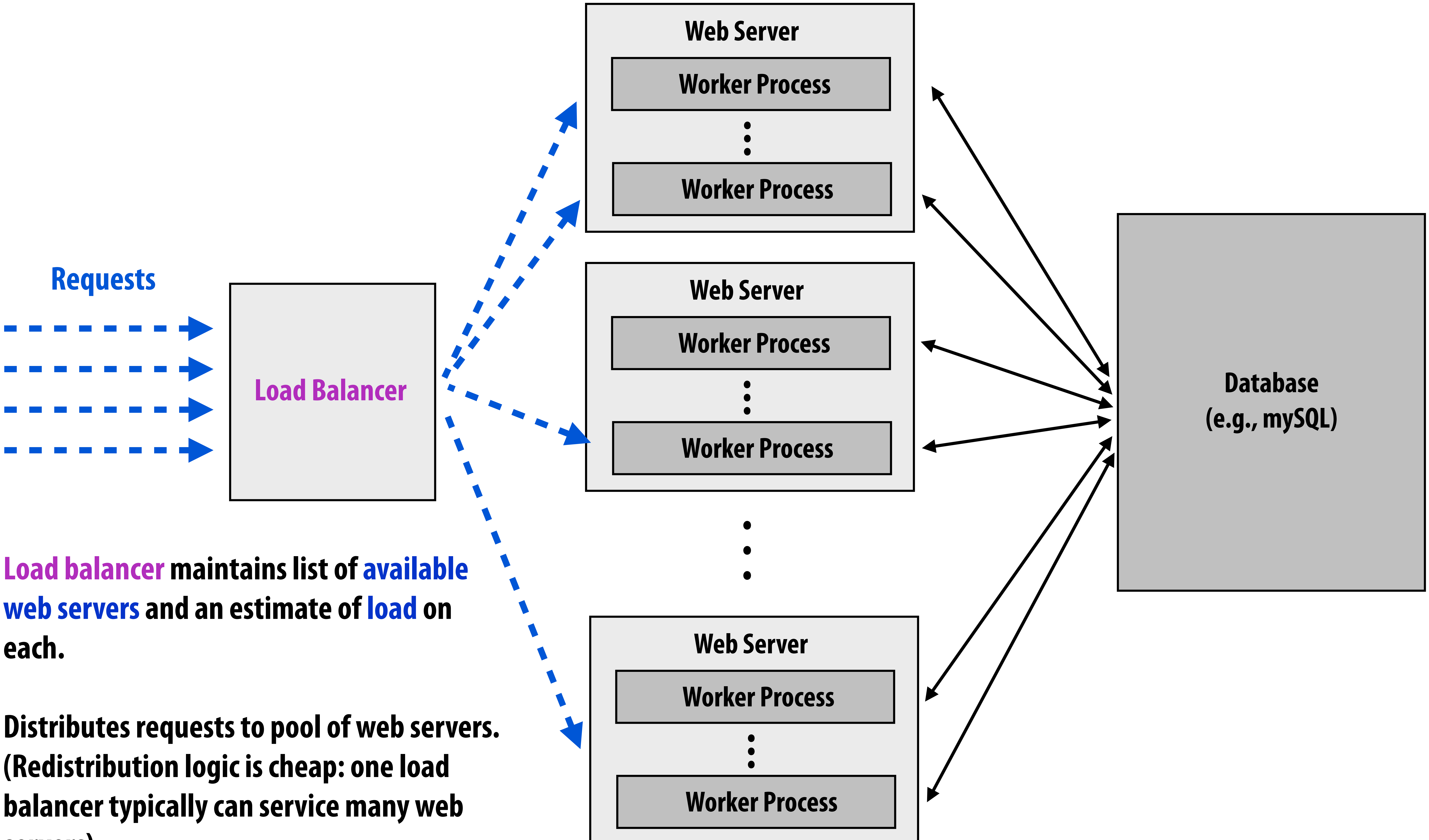
- Two popular content management systems (PHP)
  - Wordpress ~ 12 requests/sec/core (DB size = 1000 posts)
  - MediaWiki ~ 8 requests/sec/core

  [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.

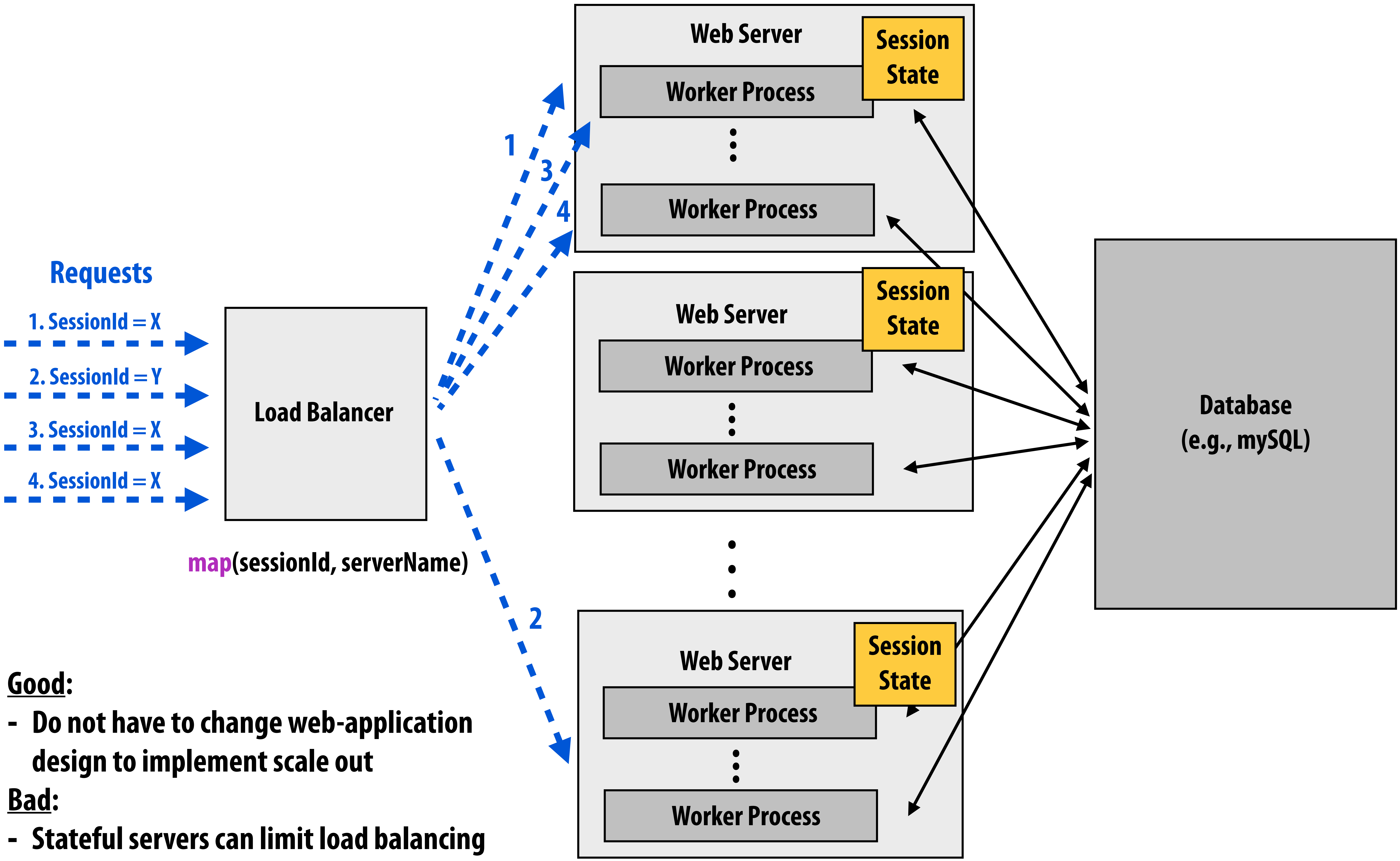


Load balancer maintains list of available web servers and an estimate of load on each.

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")

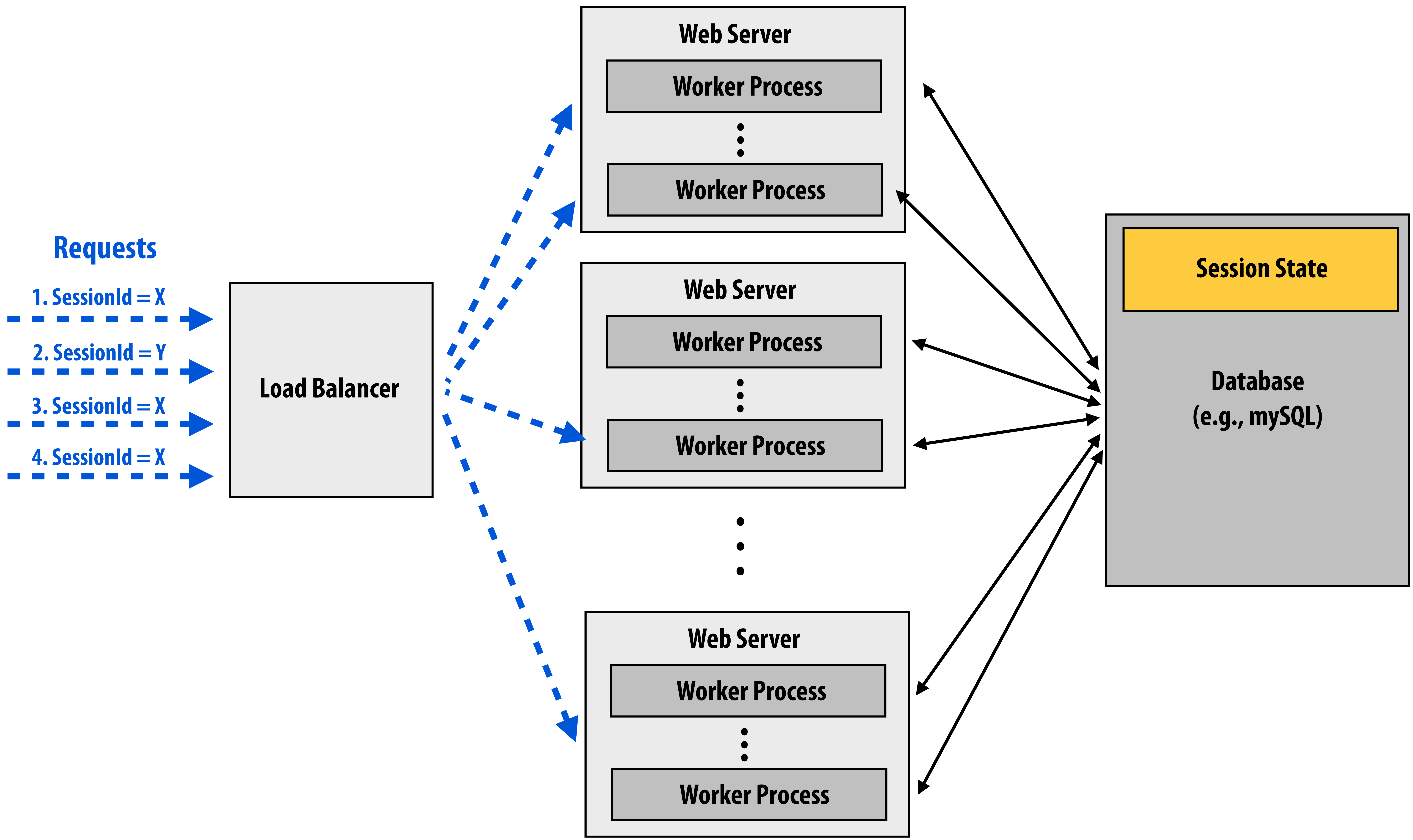


Good:
- Do not have to change web-application design to implement scale out

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.

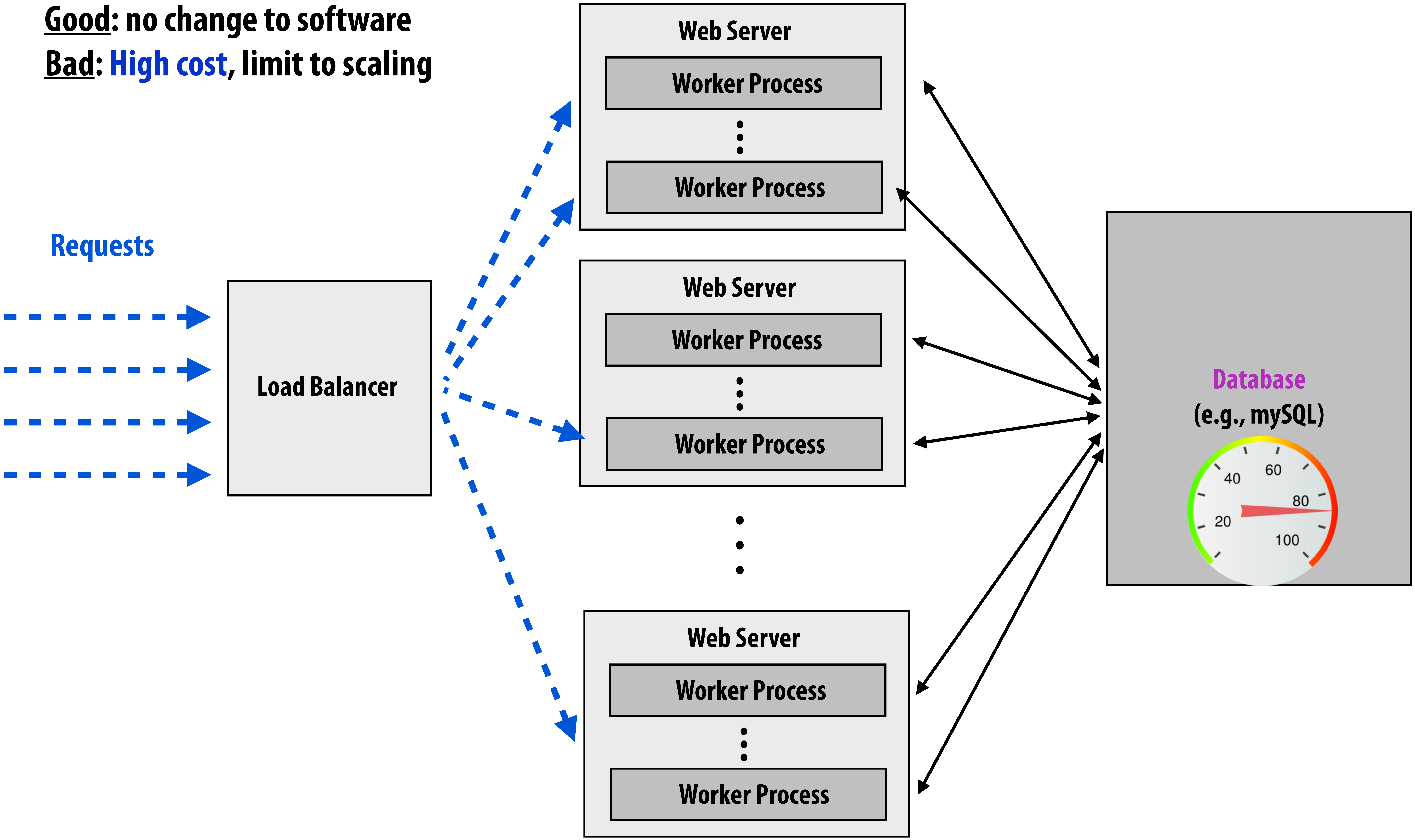


Dealing with database contention

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

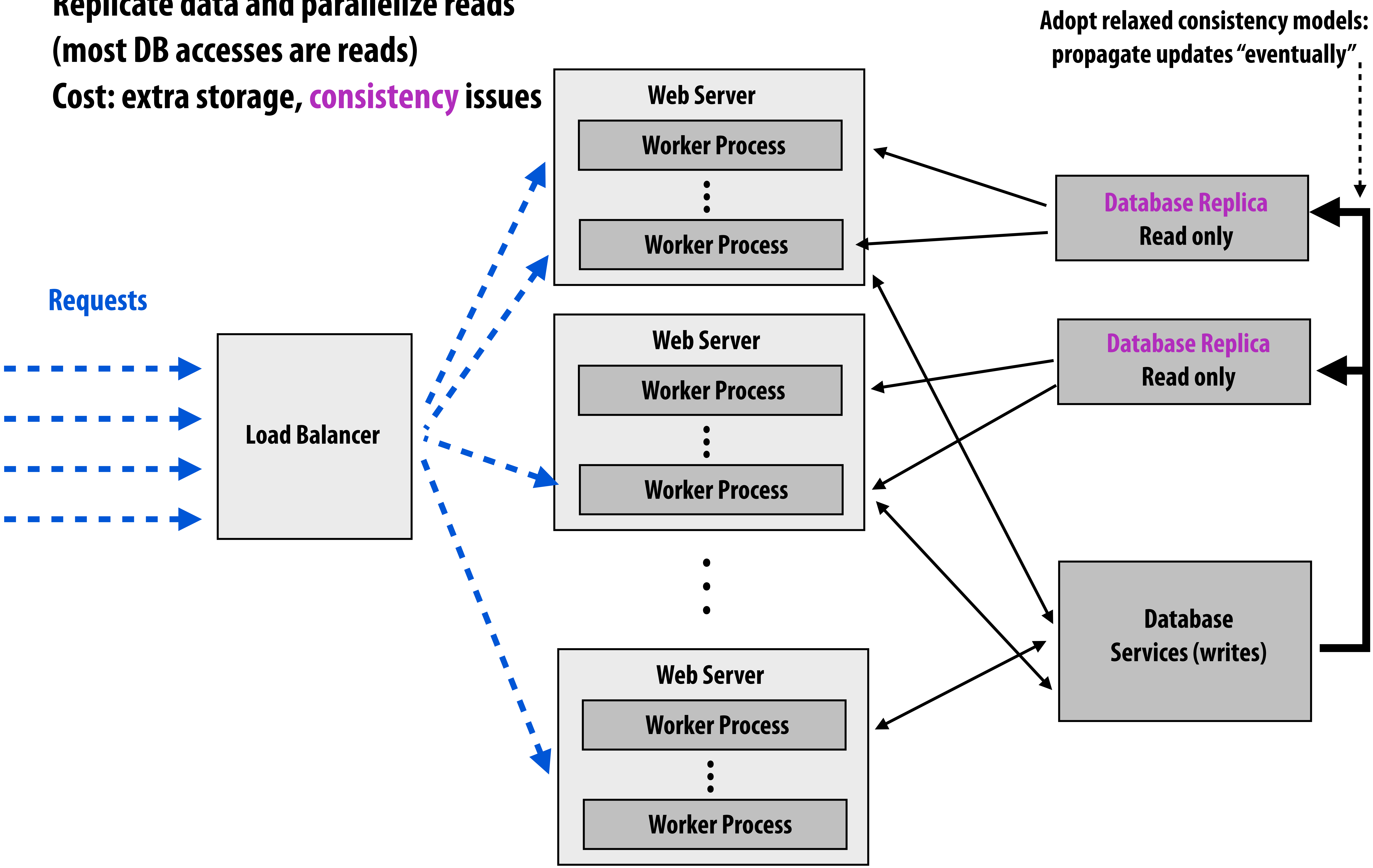
Good: no change to software

Bad: High cost, limit to scaling

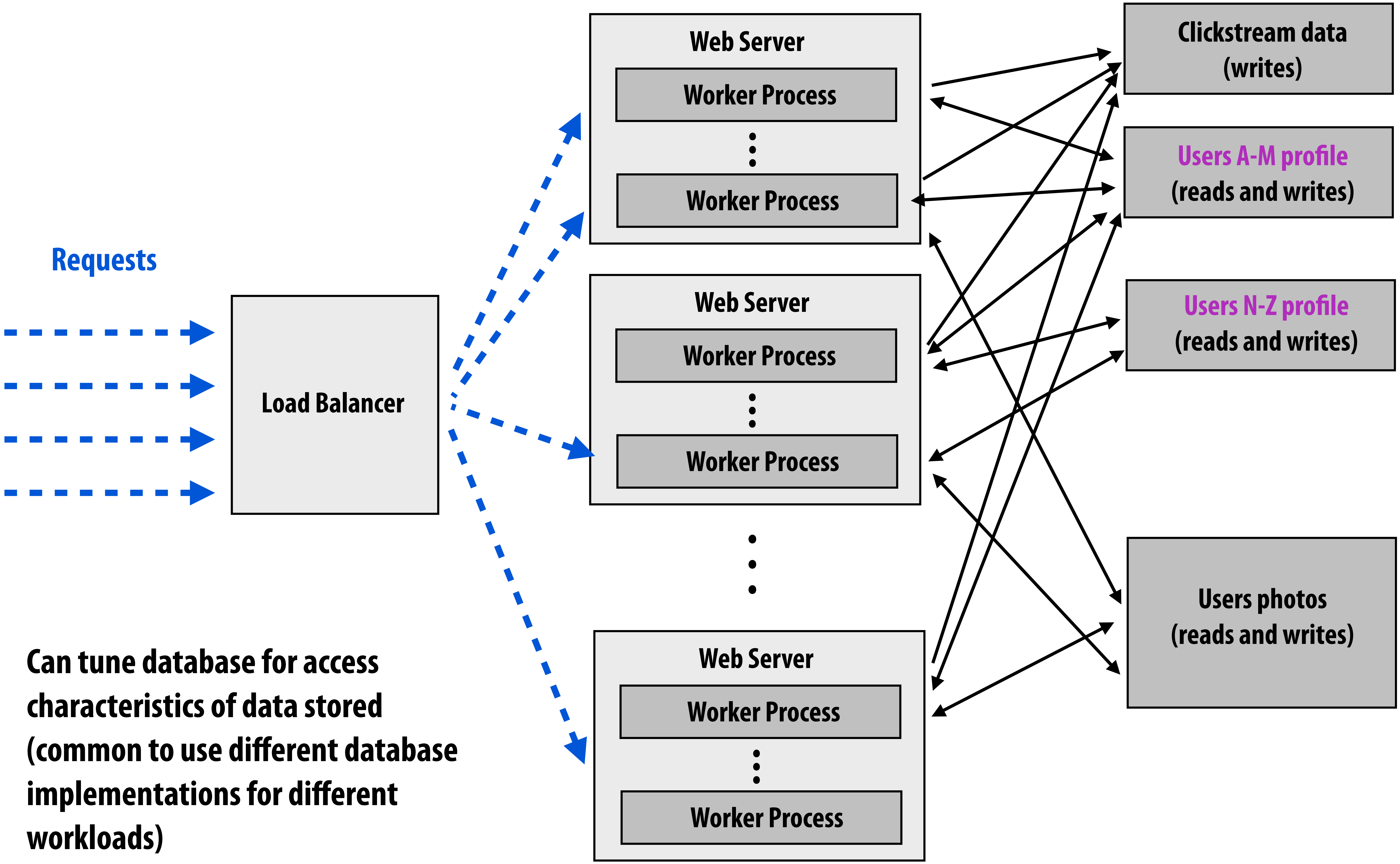


Scaling out a database: replicate

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



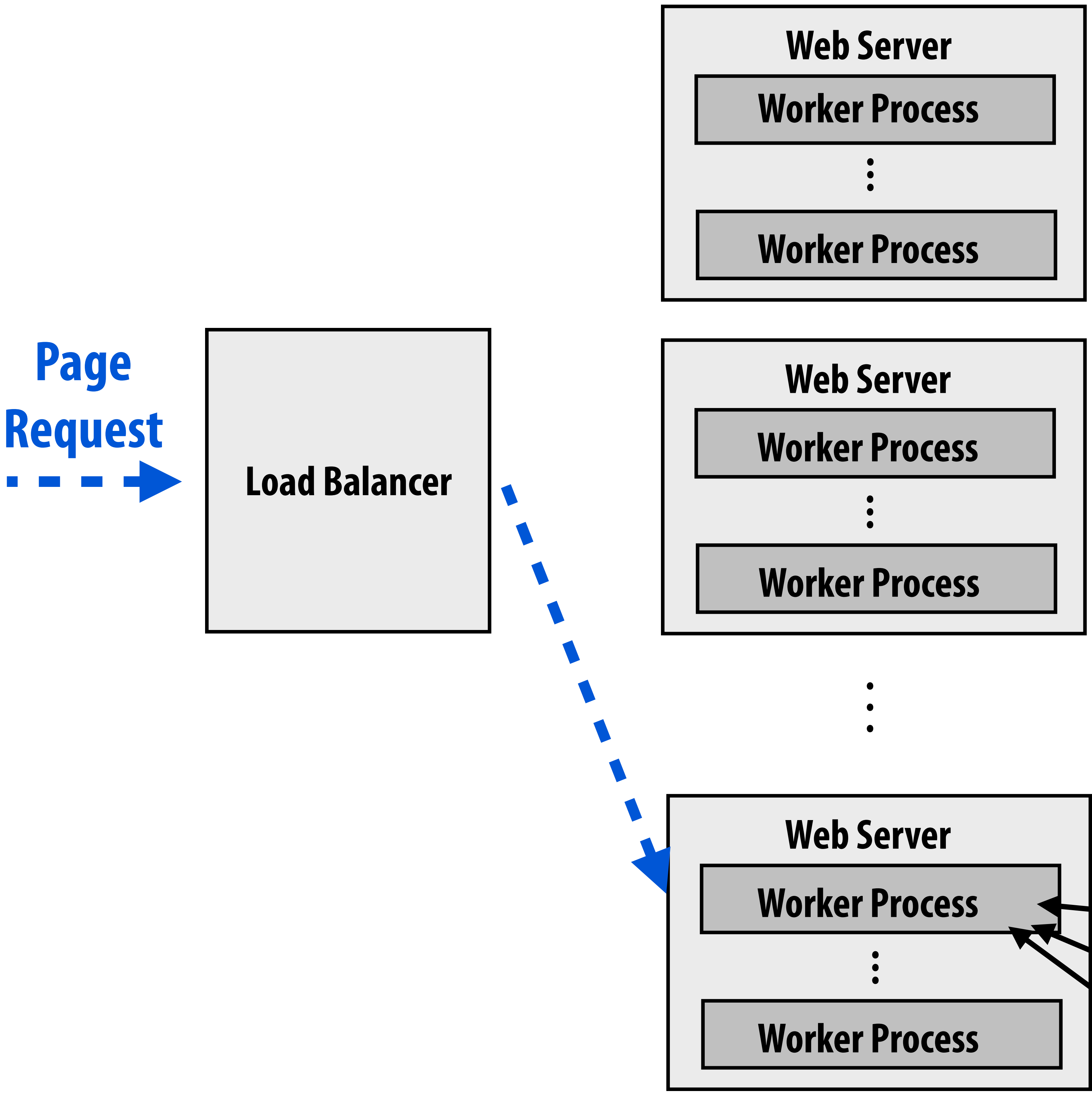
Scaling out a database: partition

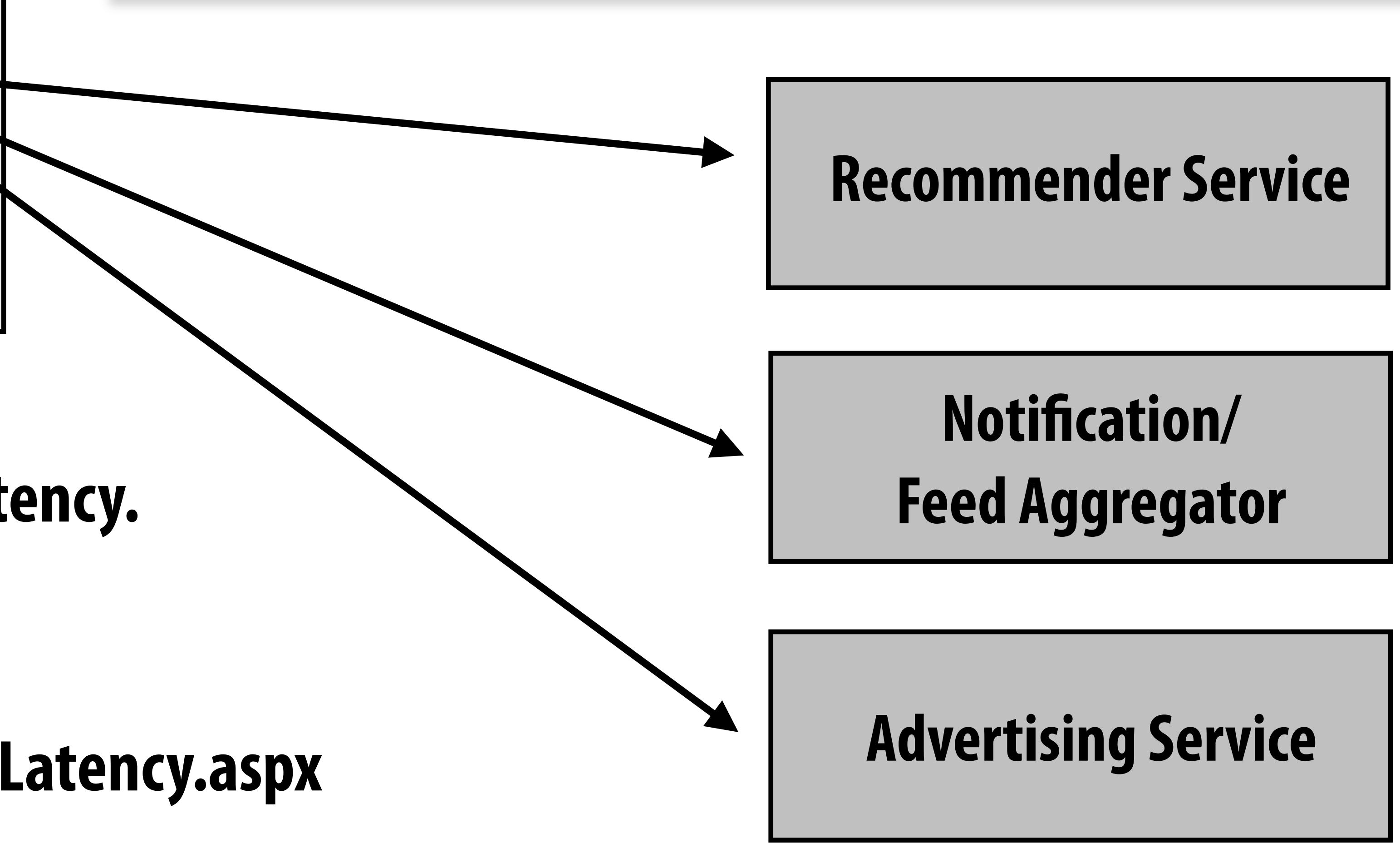


Can tune database for access characteristics of data stored (common to use different database implementations for different workloads)

Intra-request parallelism

Parallelize generation of a single page





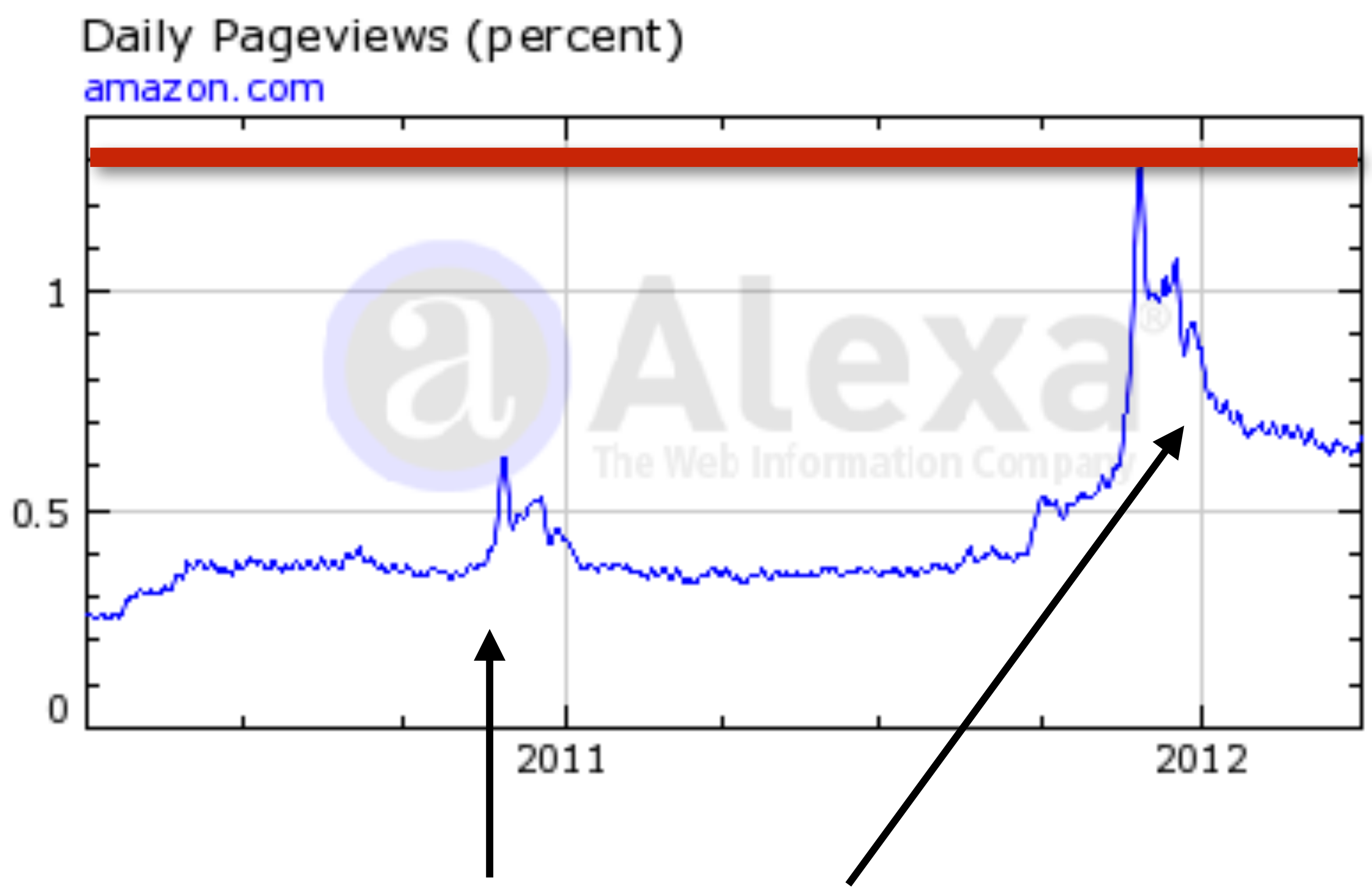
Amount of user traffic is directly correlated to response latency.

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

How many web servers do you need?

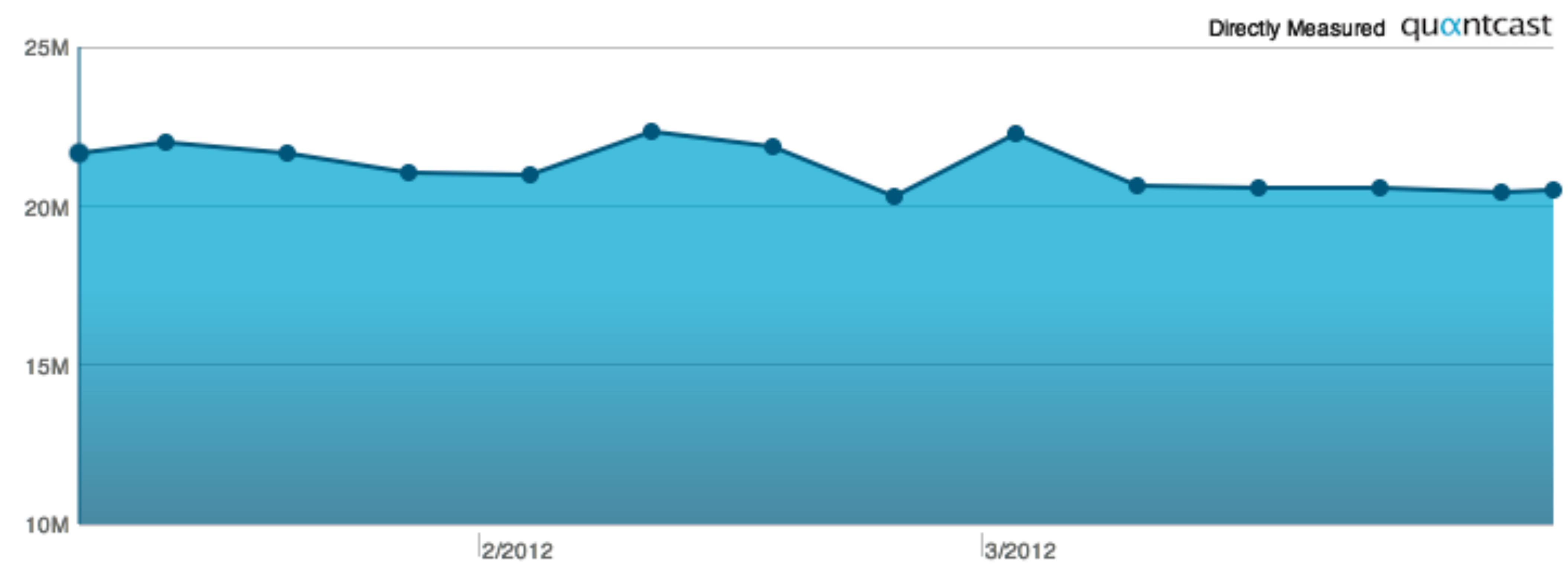
Web traffic is bursty

Amazon.com Page Views

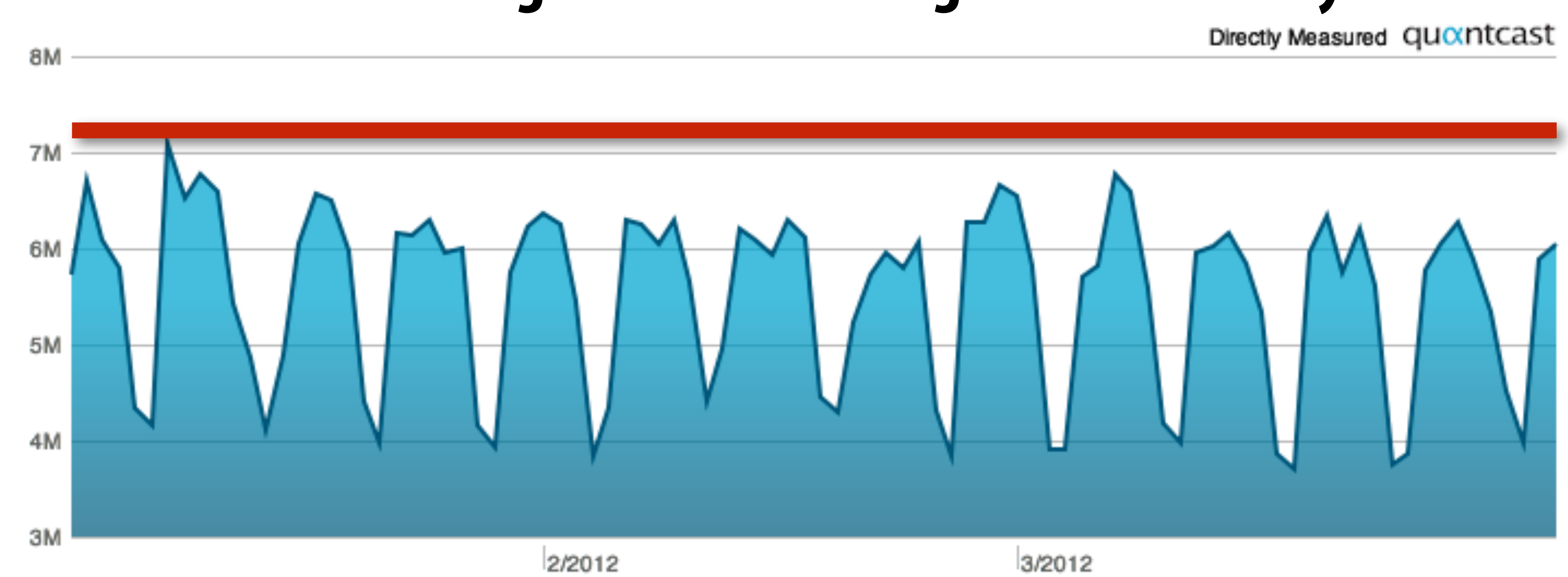


Holiday shopping season

HuffingtonPost.com Page Views Per Week



HuffingtonPost.com Page Views Per Day



(fewer people read news on weekends)

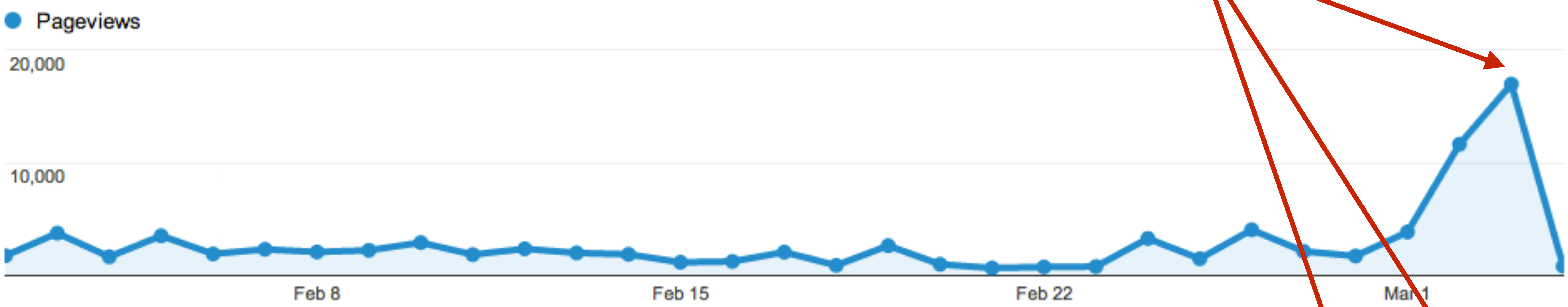
More examples:

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

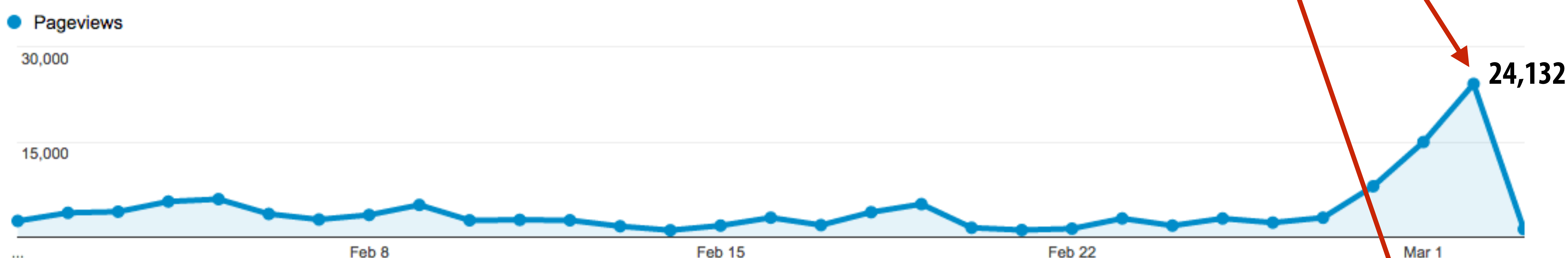
15-418/618 site traffic

Exam 1

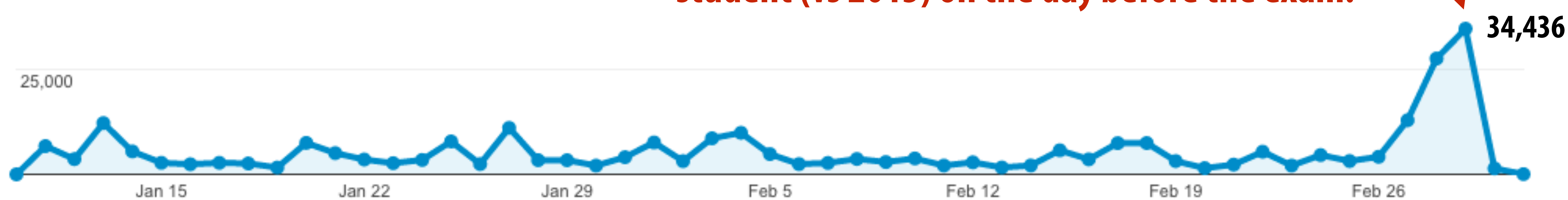
Spring 2014



Spring 2015



Spring 2016



Interesting 2016 fact: 10% fewer page views per student (vs 2015) on the day before the exam.

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

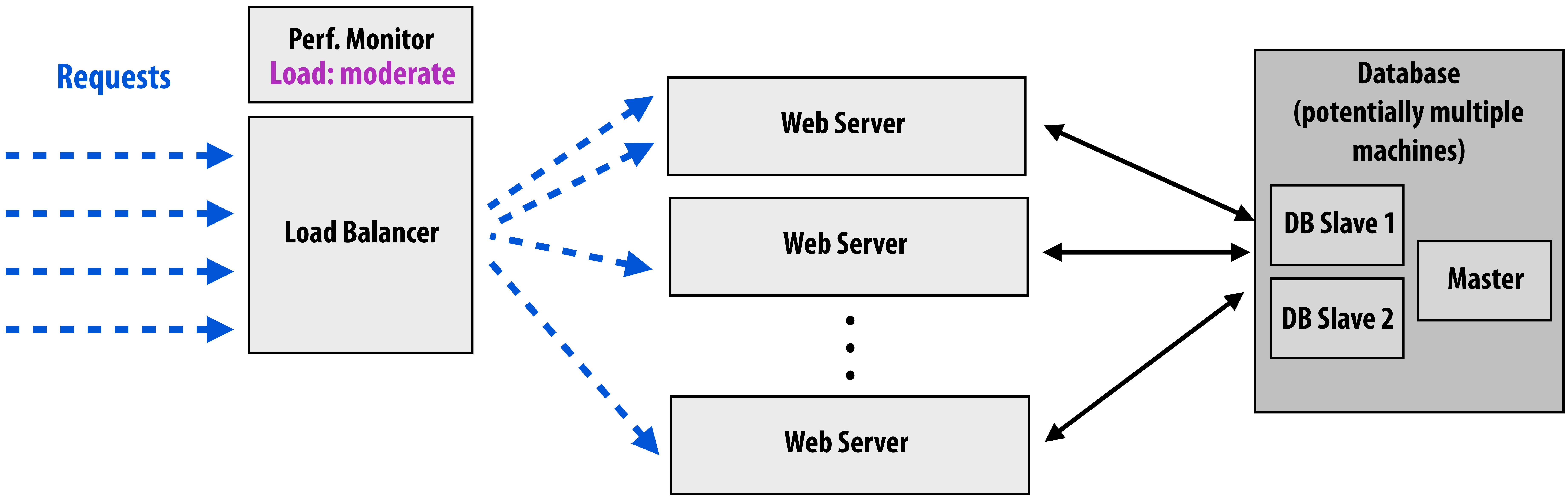
Example: Amazon's elastic compute cloud (EC2)

- Amazon 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

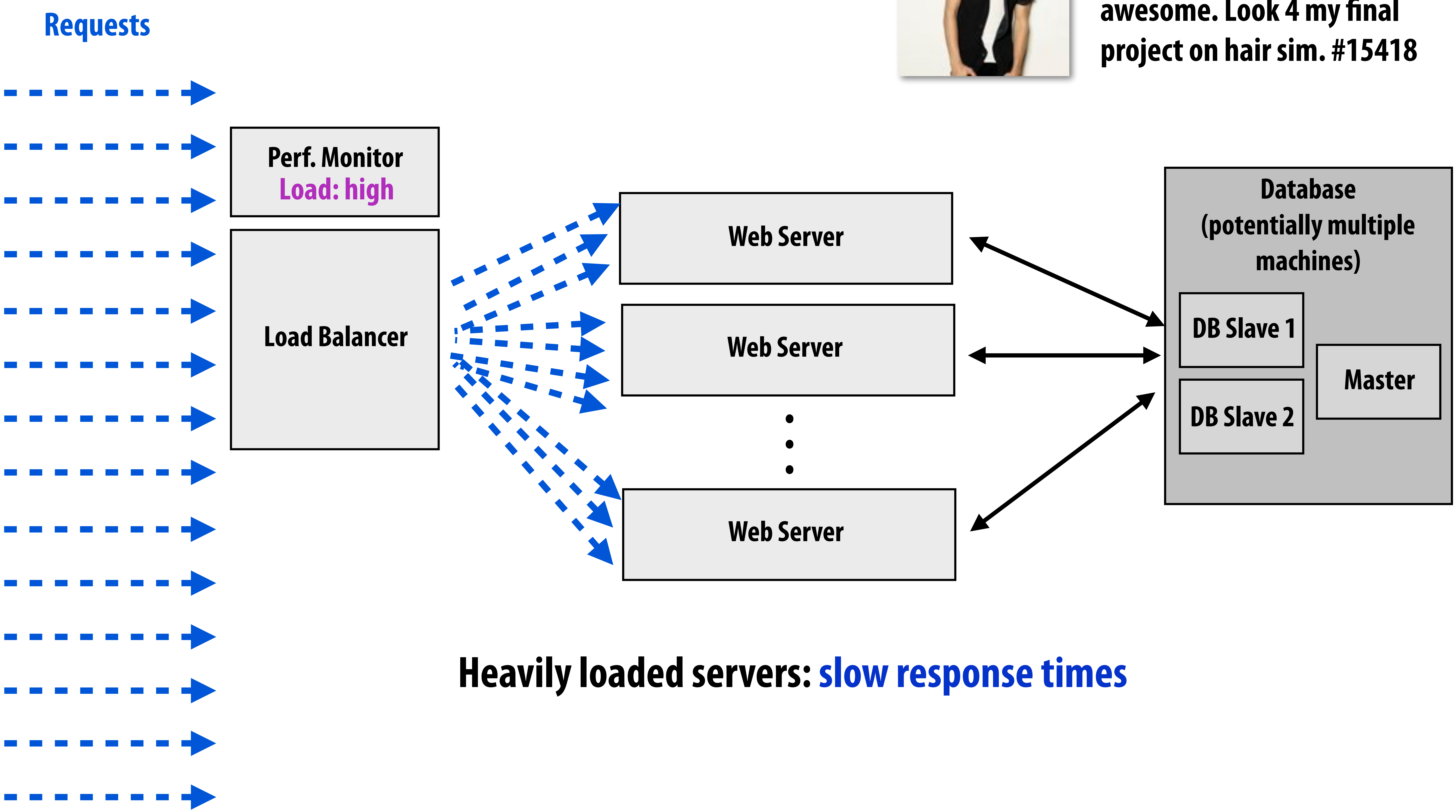
	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
Compute Optimized - Current Generation					
c4.large	2	8	3.75	EBS Only	$0.105 per Hour
c4.xlarge	4	16	7.5	EBS Only	$0.209 per Hour
c4.2xlarge	8	31	15	EBS Only	$0.419 per Hour
c4.4xlarge	16	62	30	EBS Only	$0.838 per Hour
c4.8xlarge	36	132	60	EBS Only	$1.675 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					
g2.2xlarge	8	26	15	60 SSD	$0.65 per Hour
g2.8xlarge	32	104	60	2 x 120 SSD	$2.6 per Hour

Site configuration: normal load



Event triggers spike in load

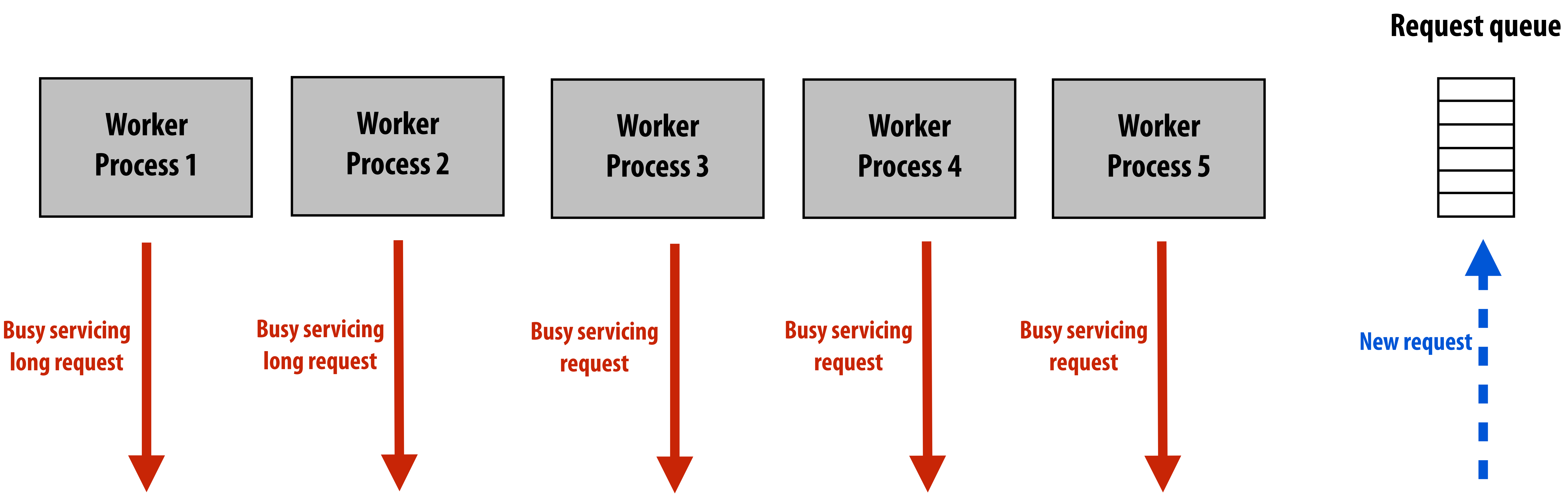
@justinbieber: OMG, parallel prog. class @ CMU is awesome. Look 4 my final project on hair sim. #15418



Heavily loaded servers: slow response times

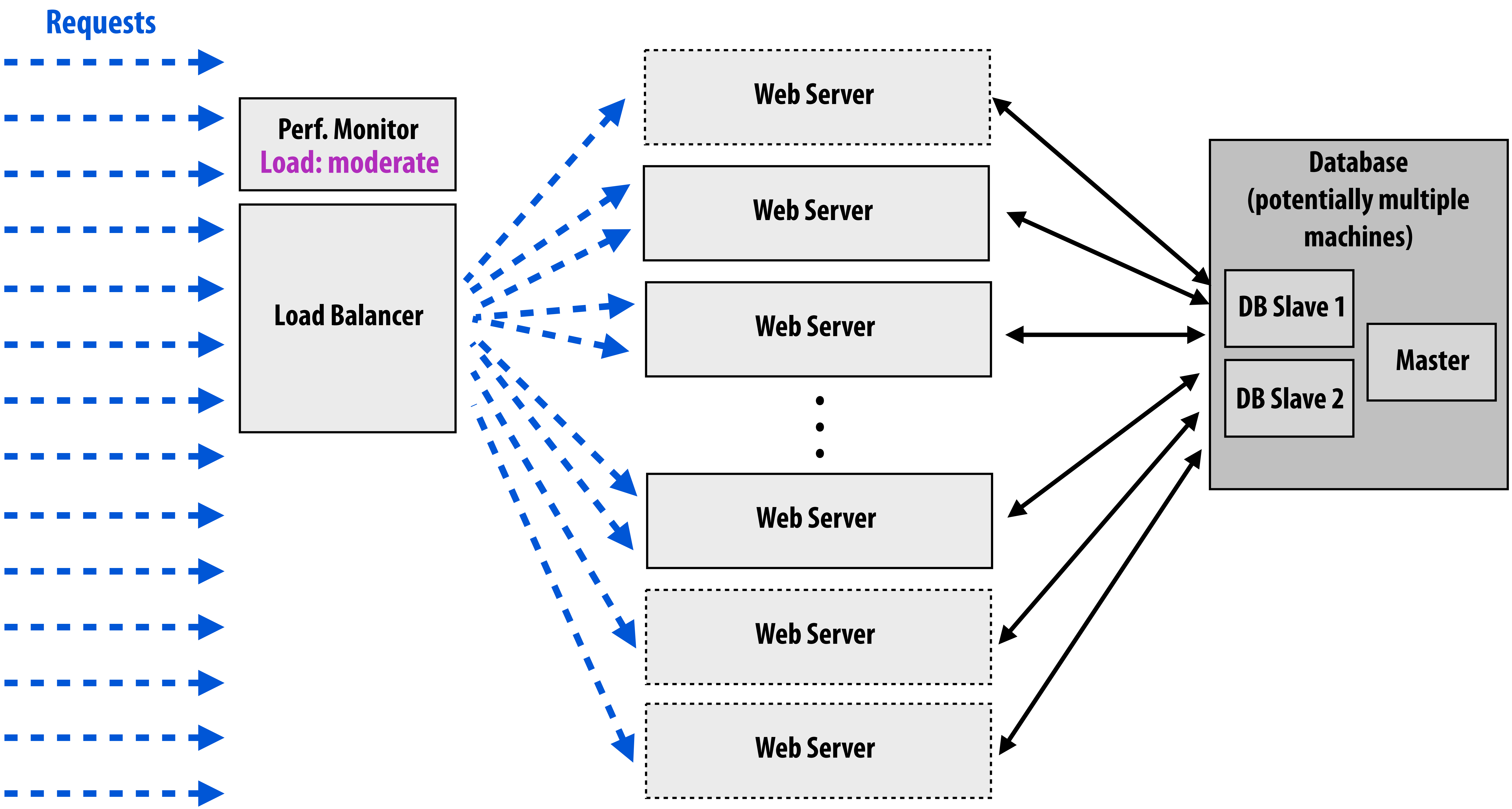
Heavily loaded servers = slow response times

- If requests arrive faster than site can service them, queue lengths will grow
- 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

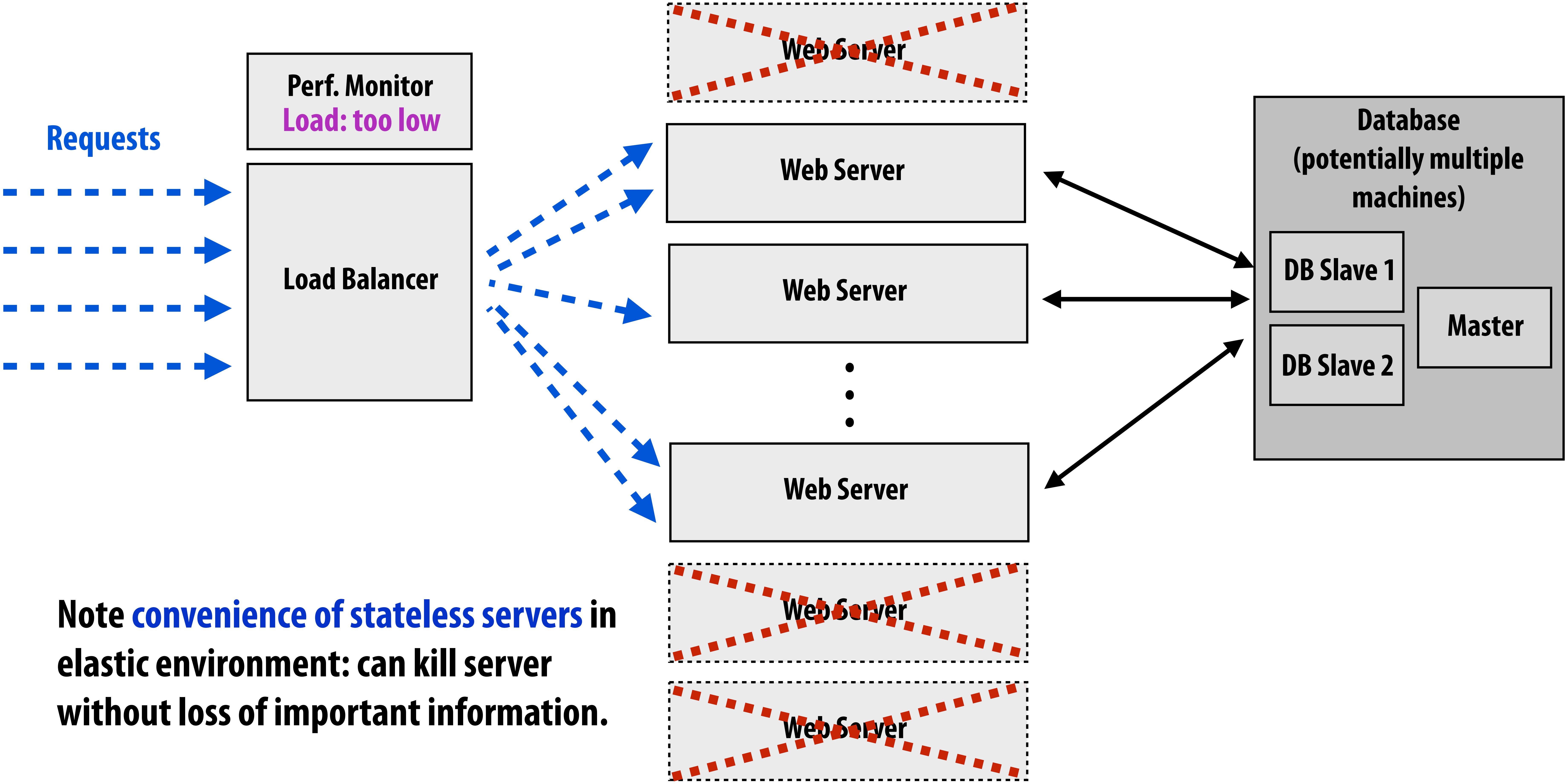
- Site performance monitor detects high load
- Instantiates new web server instances
- Informs load balancer about presence of new servers



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

@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

amazon web services

CloudWatch+Auto Scaling
Amazon Elastic Beanstalk

Google App Engine

RightScale

Engine Yard

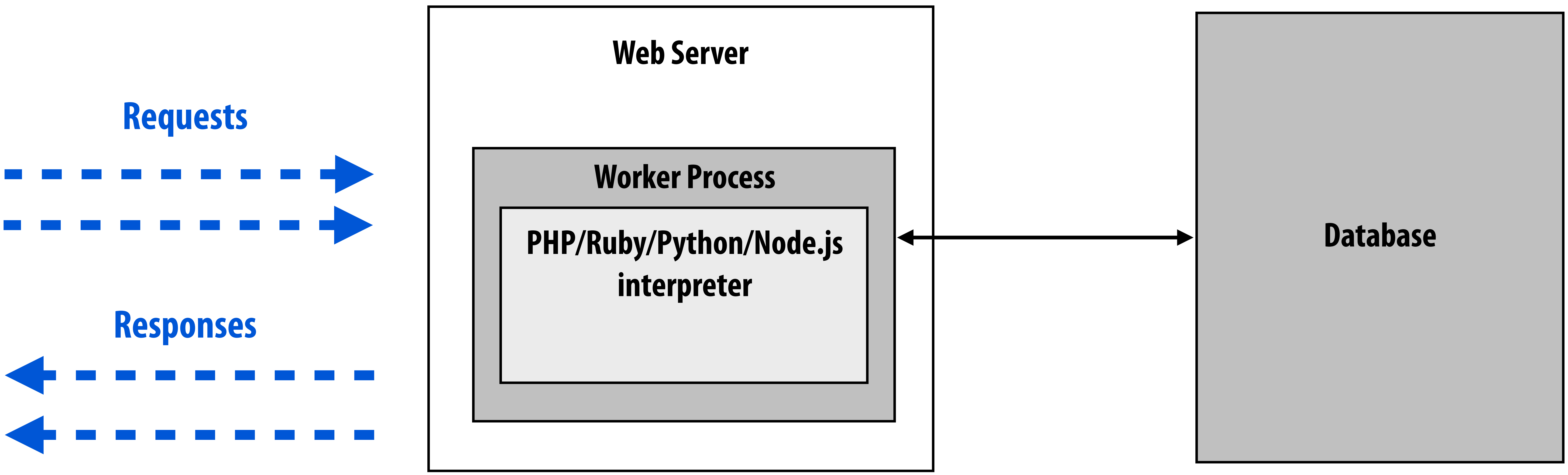
Google Cloud Platform

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

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

Now: reuse and locality

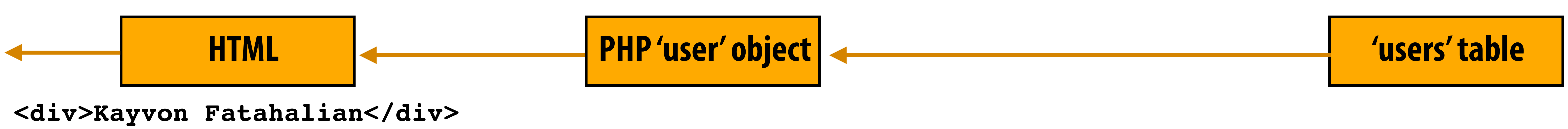
Recall: basic site configuration



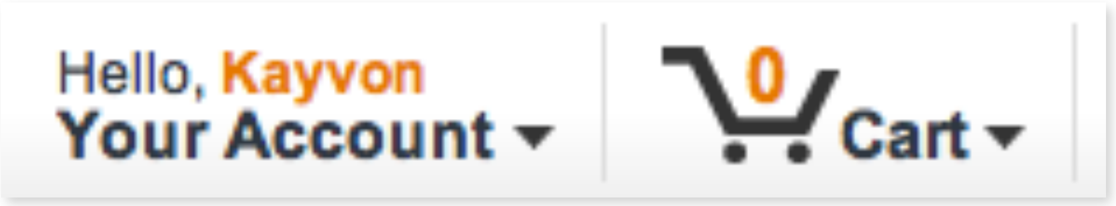
Example PHP Code

```
$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

HTML ← PHP ‘user’ object ← ‘users’ table

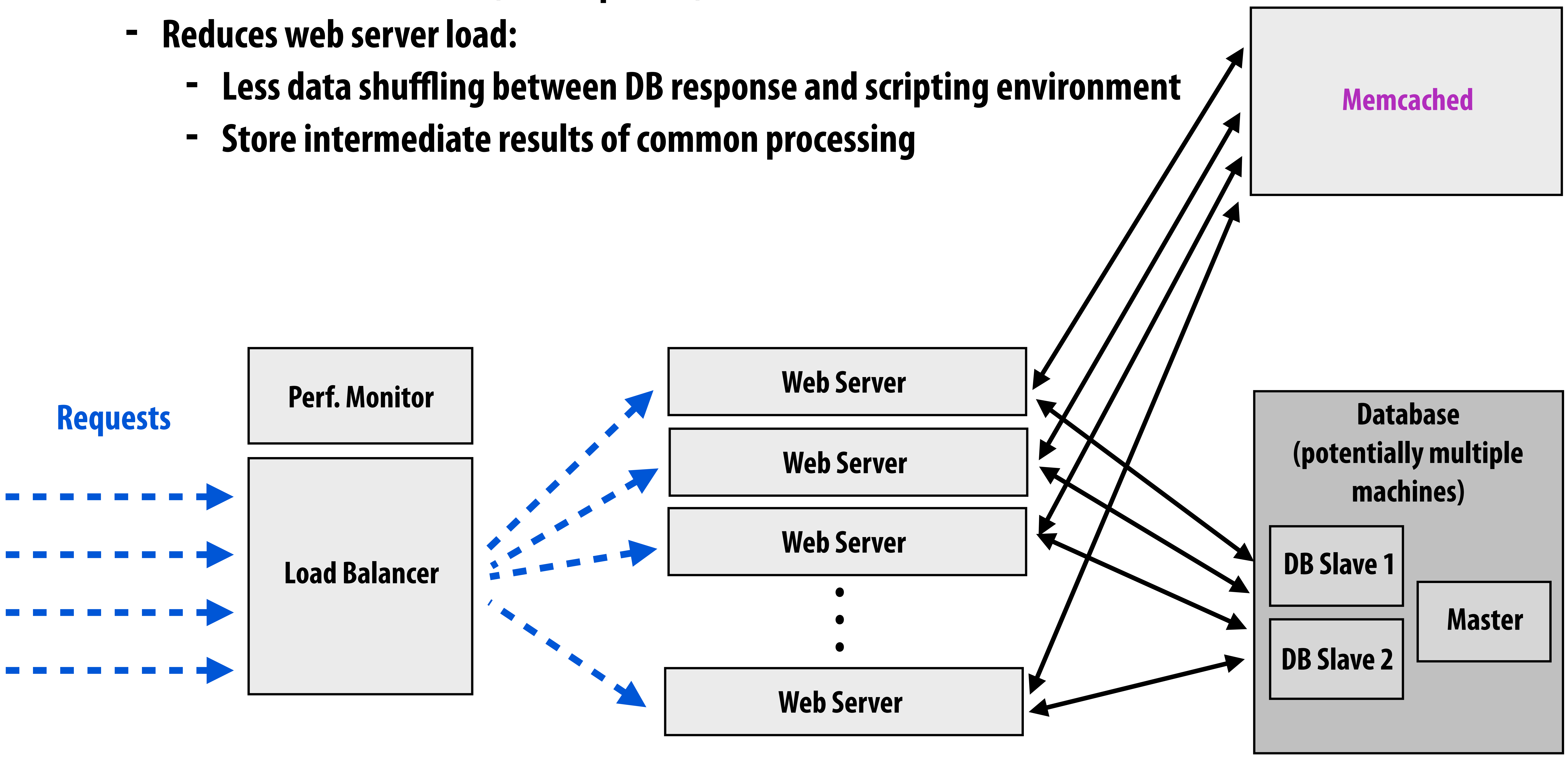
`<div>Kayvon Fatahalian</div>`

- **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

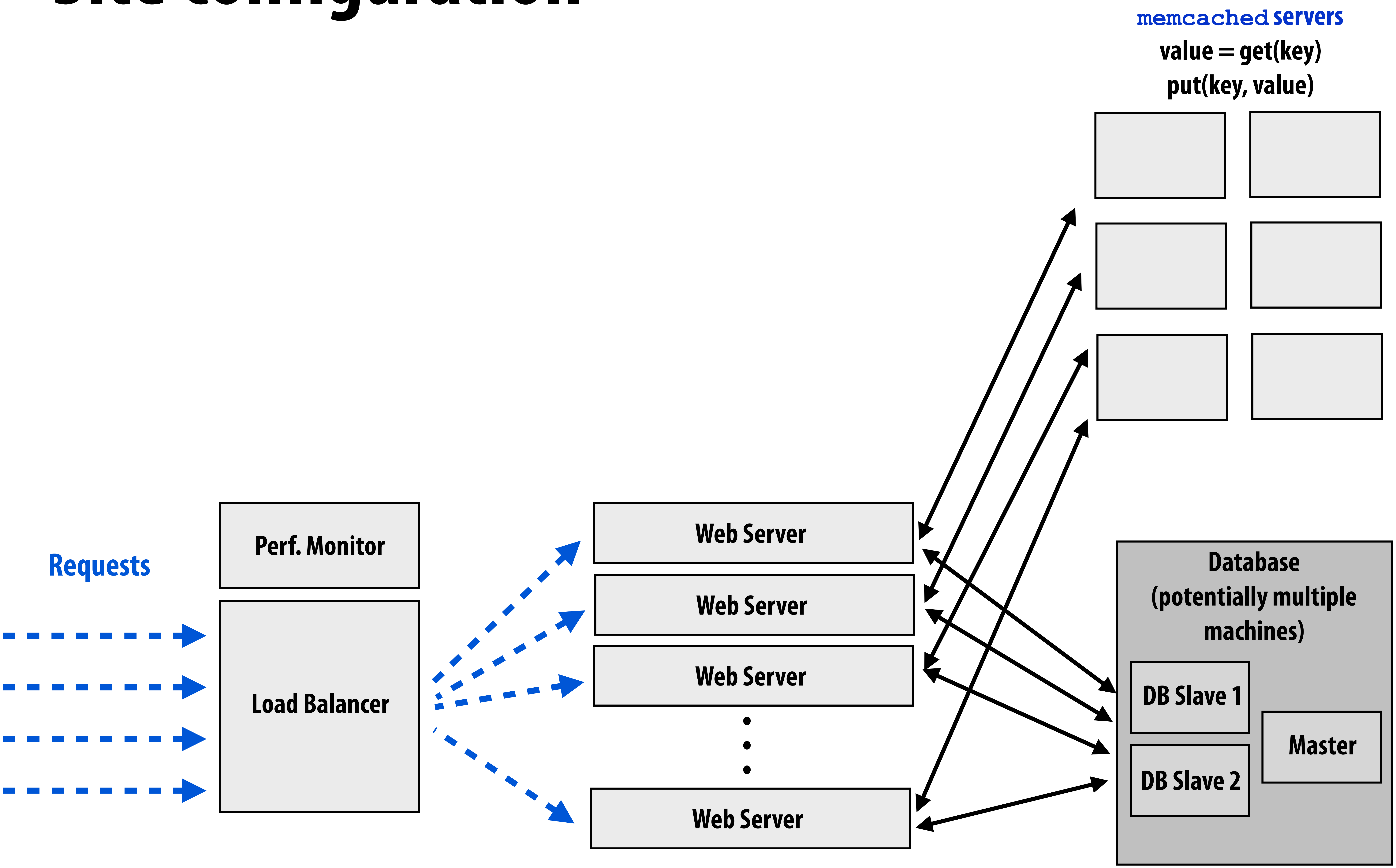


Caching example

```
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

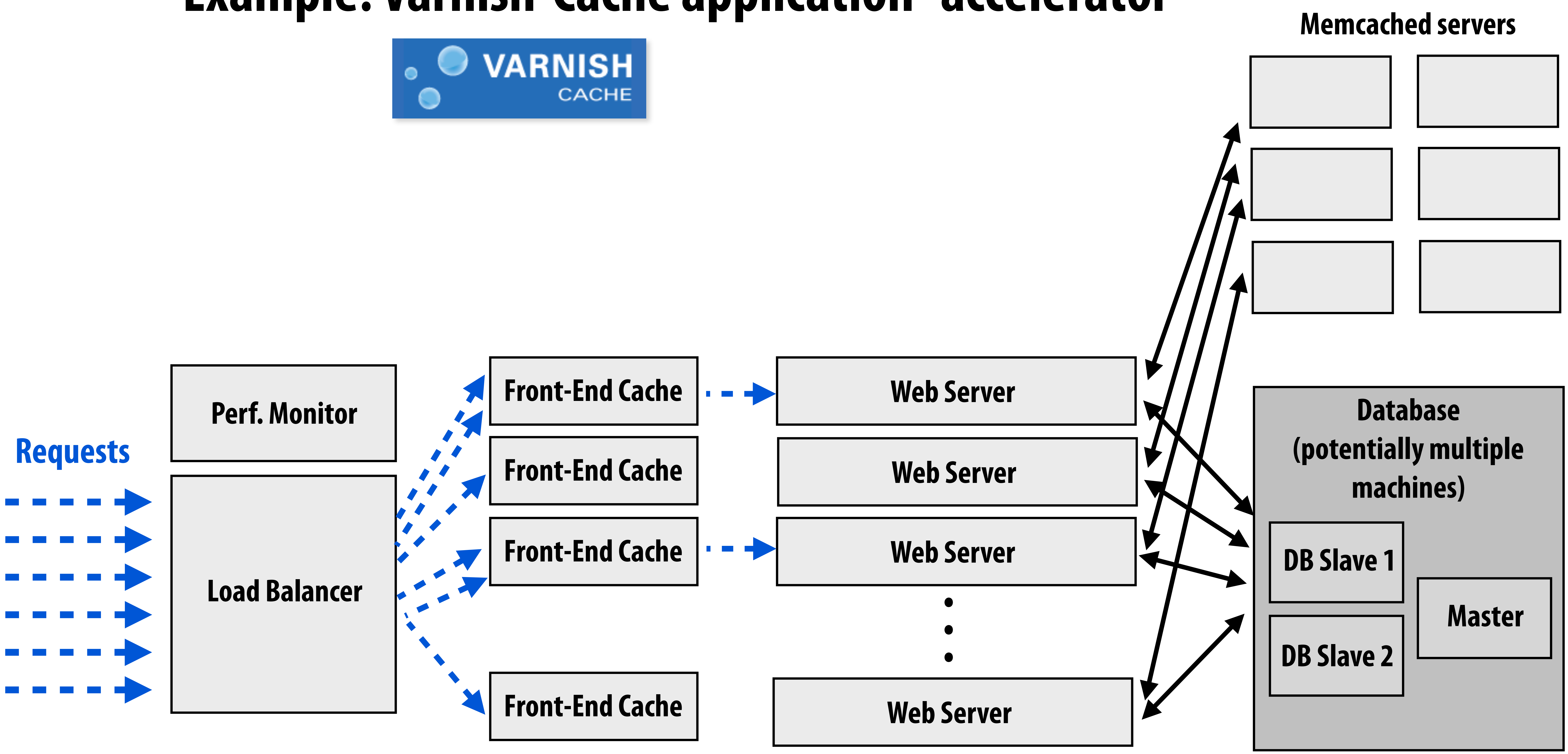
- Facebook, circa 2008
  - 800 memcached servers
  - 28 TB of cached data

- 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

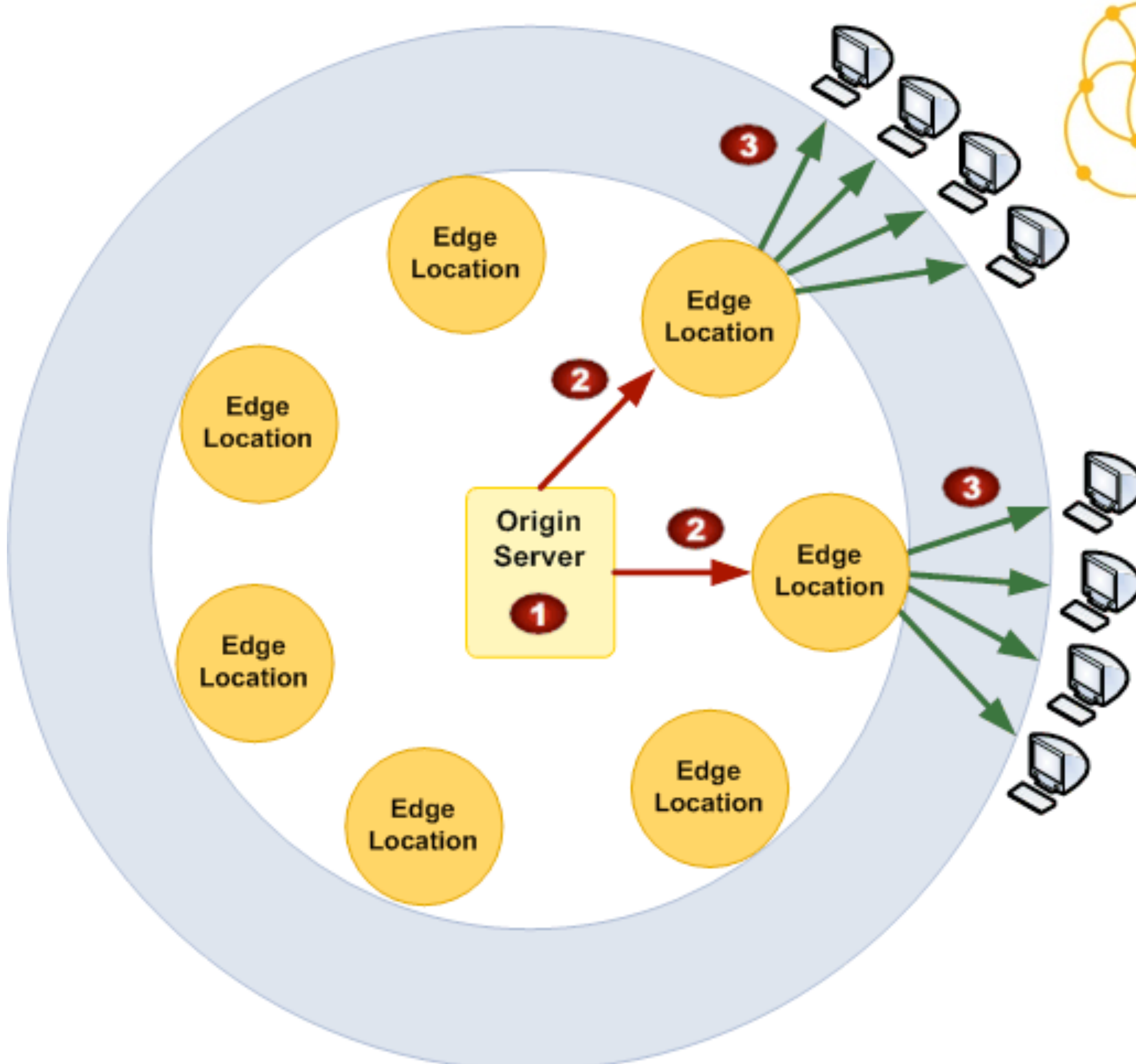
More caching

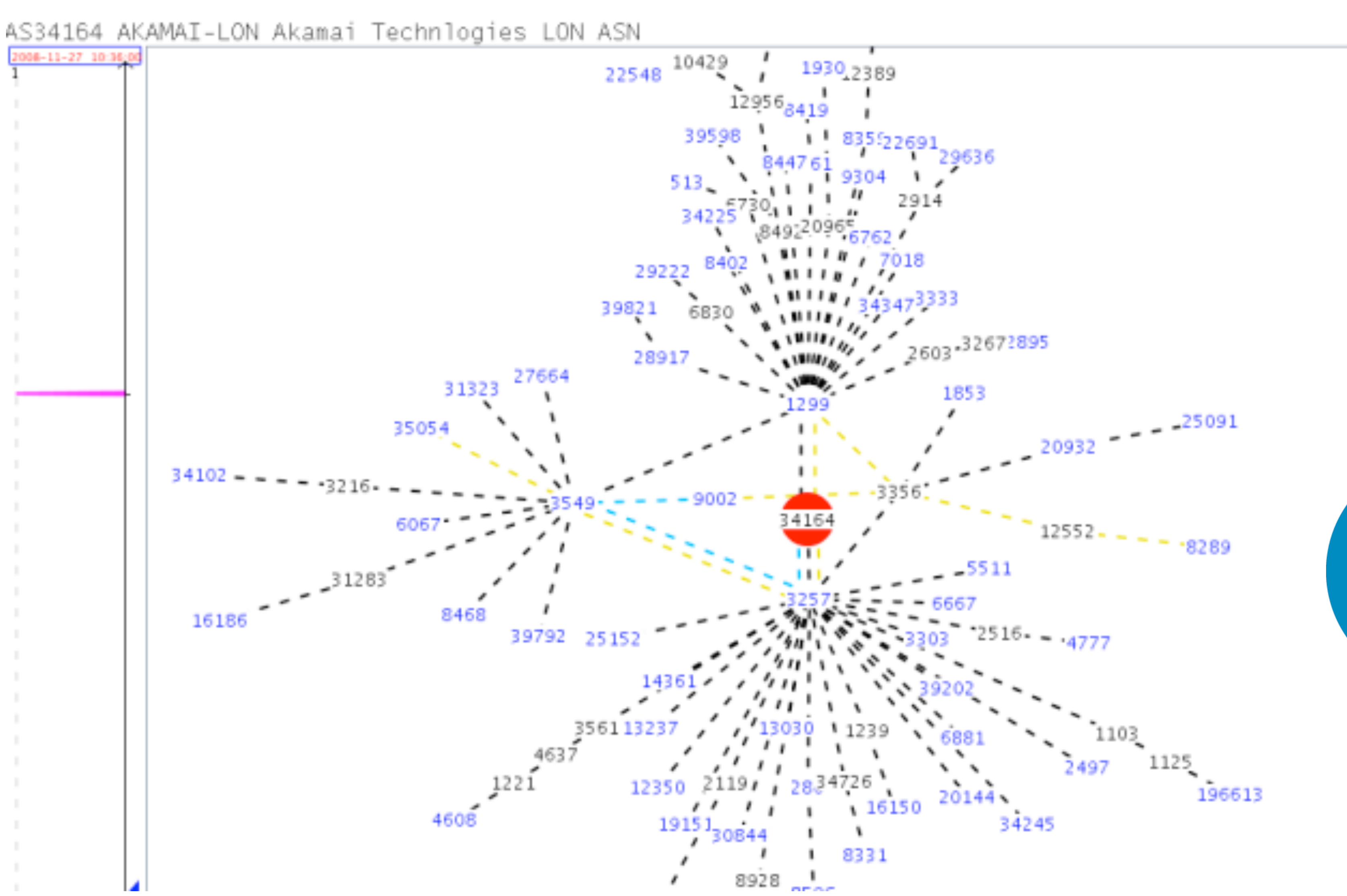
- **Cache web server responses** (e.g. entire pages, pieces of pages)
  - Reduce load on web servers
  - Example: Varnish-Cache application "accelerator"



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

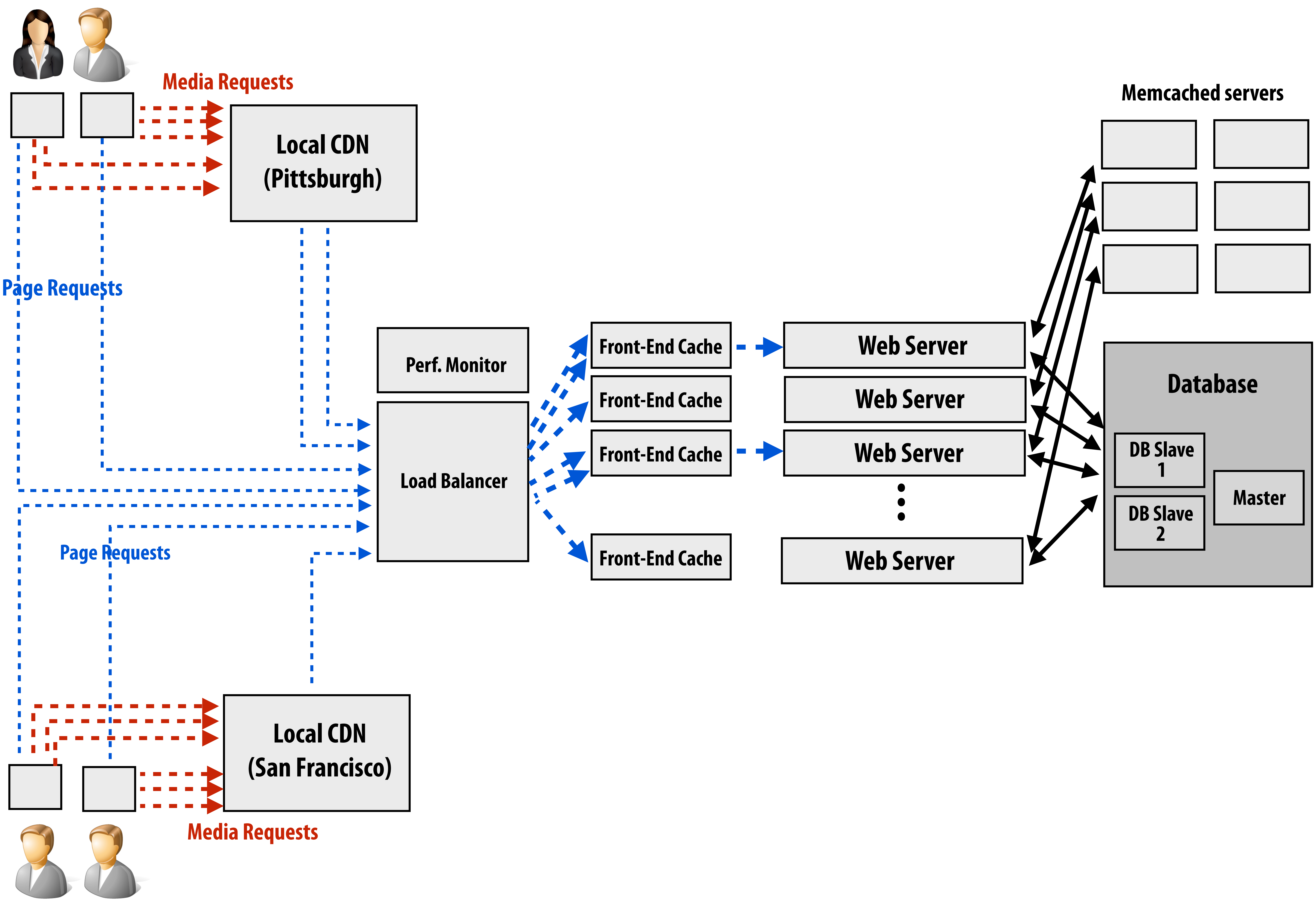
Source: http://www.telco2.net/blog/2008/11/amazon_cloudfront_yet_more_tra.html

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=10153516598728897&set=a.279790798896.141301.722973896&type=3&theater

Image source URL: (you can definitely see this photo… try it!)
https://scontent-iad3-1.xx.fbcdn.net/hphotos-xfl1/t31.0-8/12628370_10153516598728897_3170992092621097770_o.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
  - 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

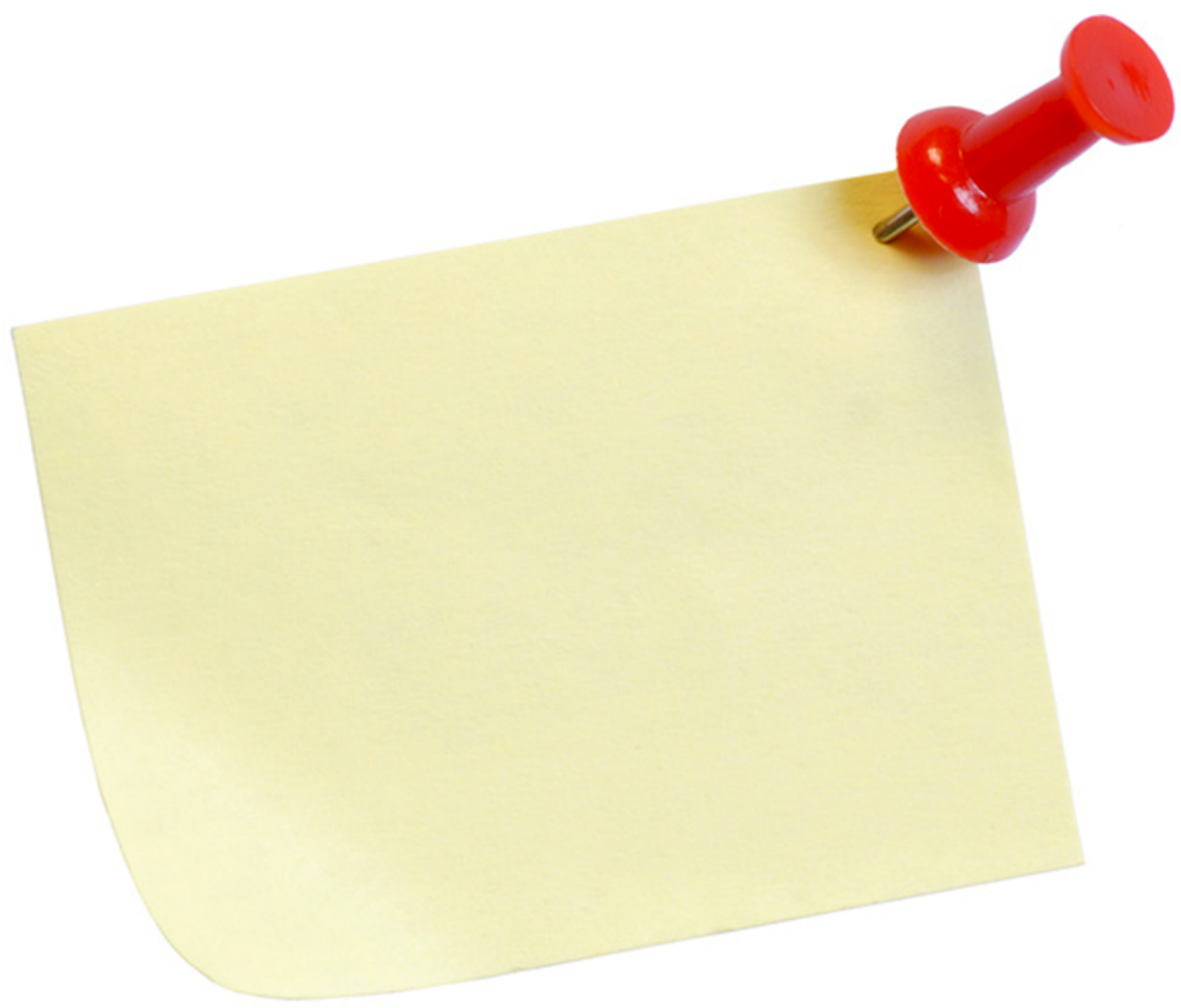
- Many great sites (and blogs) on the web to learn more:
  - www.highscalability.com has great case studies (see "All Time Favorites" section)
  - James Hamilton's blog: http://perspectives.mvdirona.com

Course so far review

(a more-or-less randomly selected collection of topics from previous lectures)

Exam details

- Online proctored exam on Gradescope
  - Login to Zoom with webcam turned on
- Open notes
- Covers all lecture material through Lecture 13 (Performance Measurement and Tuning)
- Typical question formats:
  - Short answer
  - Multiple choice with explanations



Throughput vs. latency

THROUGHPUT

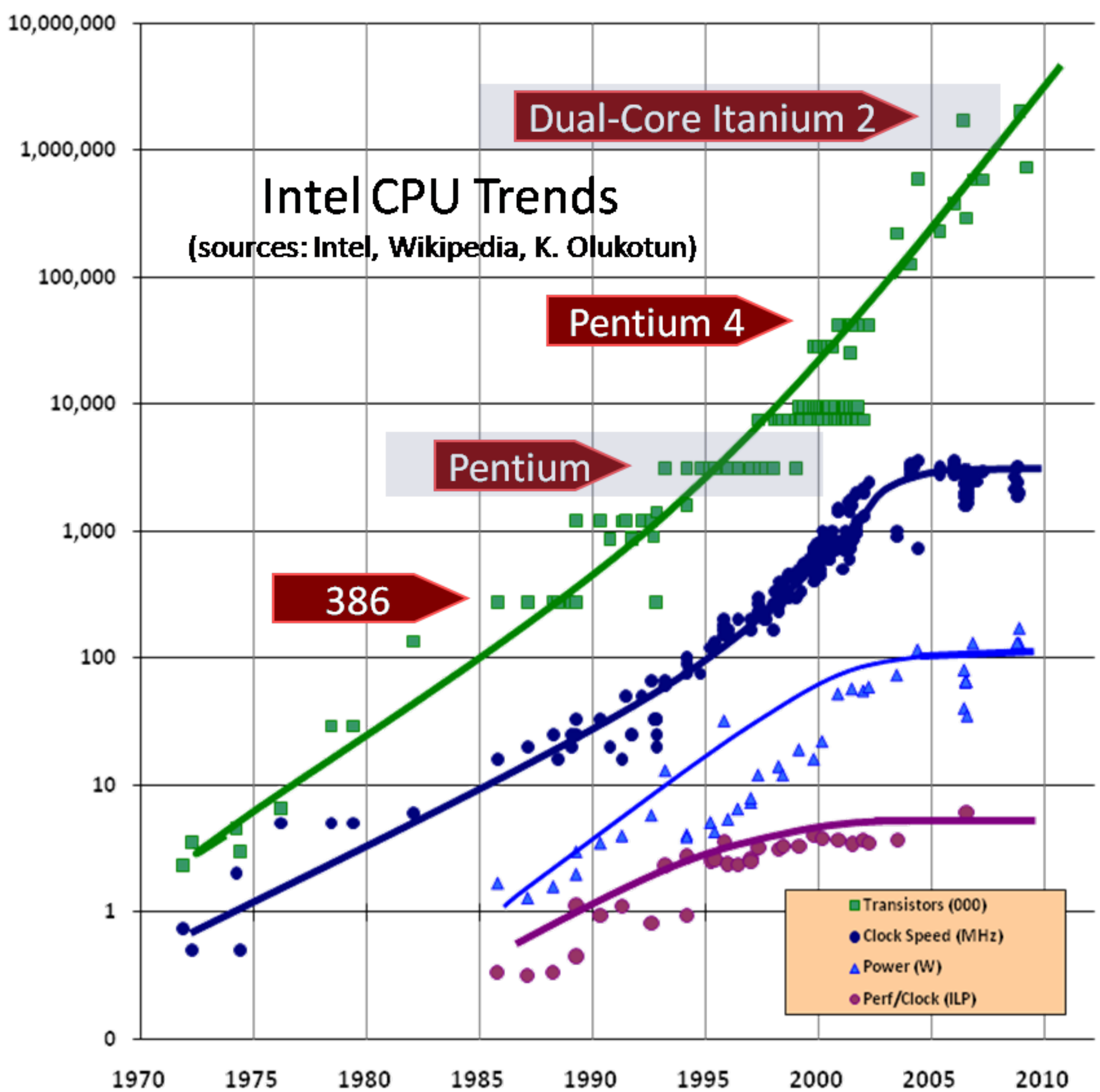
The rate at which work gets done.
- Operations per second
- Bytes per second (bandwidth)
- Tasks per hour

LATENCY

The amount of time for an operation to complete
- An instruction takes 4 clocks
- A cache miss takes 200 clocks to complete
- It takes 20 seconds for a program to complete

Ubiquitous parallelism

- What motivated the shift toward multi-core parallelism in modern processor design?
  - Inability to scale clock frequency due to power limits
  - Diminishing returns when trying to further exploit ILP



Is the new performance focus on throughput, or latency?

Techniques for exploiting independent operations in applications

What is it? What is the benefit?

1. **superscalar** execution	Processor executes multiple instructions per clock. Super-scalar execution exploits instruction level parallelism (ILP). When instructions in the same thread of control are independent they can be executed in parallel on a super-scalar processor.
2. **SIMD** execution	Processor executes the same instruction on multiple pieces of data at once (e.g., one operation on vector registers). The cost of fetching and decoding the instruction is amortized over many arithmetic operations.
3. **multi-core** execution	A chip contains multiple (mainly) independent processing cores, each capable of executing independent instruction streams.
4. **multi-threaded** execution	Processor maintains execution contexts (state: e.g, a PC, registers, virtual memory mappings) for multiple threads. Execution of thread instructions is interleaved on the core over time. Multi-threading reduces processor stalls by automatically switching to execute other threads when one thread is blocked waiting for a long-latency operation to complete.

Techniques for exploiting independent operations in applications

Who is responsible for mapping?

1. **superscalar** execution	Usually not a programmer responsibility: ILP automatically detected by processor hardware or by compiler (or both) (But manual loop unrolling by a programmer can help)
2. **SIMD** execution	In simple cases, data parallelism is automatically detected by the compiler, (e.g., assignment 1 saxpy). In practice, programmer explicitly describes SIMD execution using vector instructions or by specifying independent execution in a high-level language (e.g., ISPC gangs, CUDA)
3. **multi-core** execution	Programmer defines independent threads of control. e.g., pthreads, ISPC tasks, openMP #pragma
4. **multi-threaded** execution	Programmer defines independent threads of control. But programmer must create more threads than processing cores.

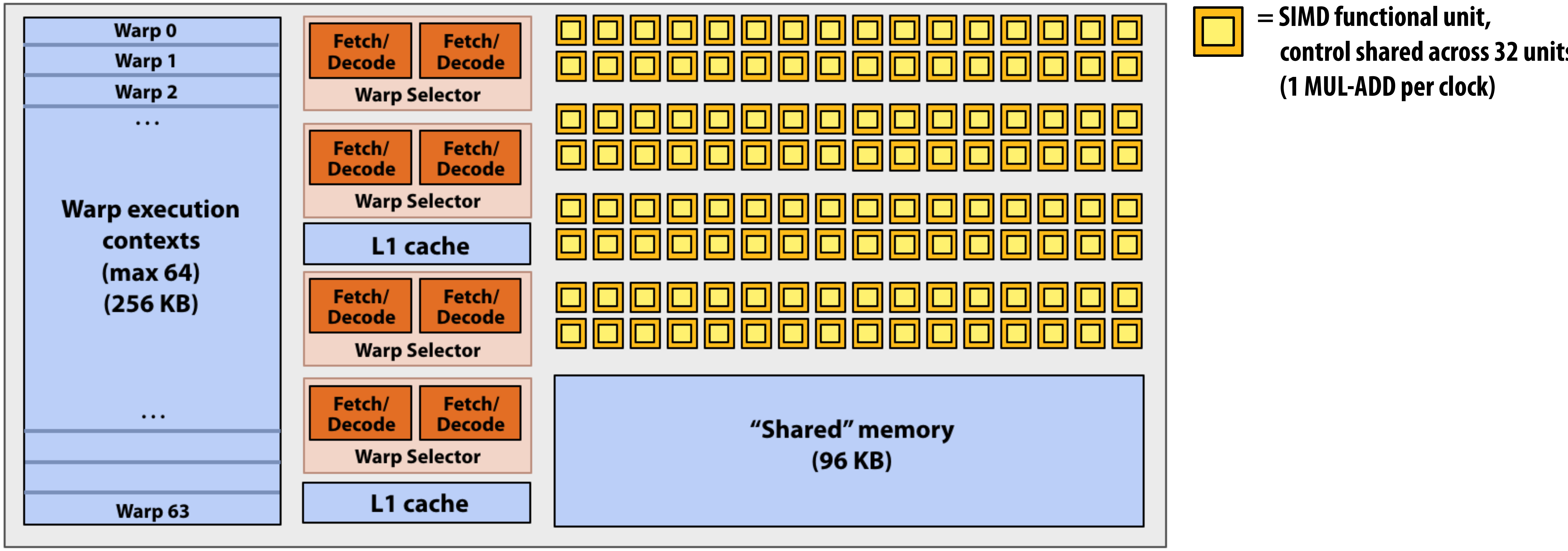
Frequently discussed processor examples

- **Intel Core i7 CPU**
  - 4 cores
  - Each core:
    - Supports 2 threads ("Hyper-Threading")
    - Can issue 8-wide SIMD instructions (AVX instructions) or 4-wide SIMD instructions (SSE)
    - Can execute multiple instructions per clock (superscalar)

- **NVIDIA GTX 980 GPU**
  - 16 "cores" (called SMM core by NVIDIA)
  - Each core:
    - Supports up to 64 warps (warp is a group of 32 "CUDA threads")
    - Issues 32-wide SIMD instructions (same instruction for all 32 "CUDA threads" in a warp)
    - Also capable of issuing multiple instructions per clock

- **Intel Xeon Phi**
  - 61 cores
  - Each core: supports 4 threads, issues 16-wide SIMD instructions

Multi-threaded, SIMD execution on GPU



- Describe how CUDA threads are mapped to the execution resources on this GTX 980 GPU?
  - e.g., describe how the processor executes instructions each clock

Decomposition: assignment 1, program 3

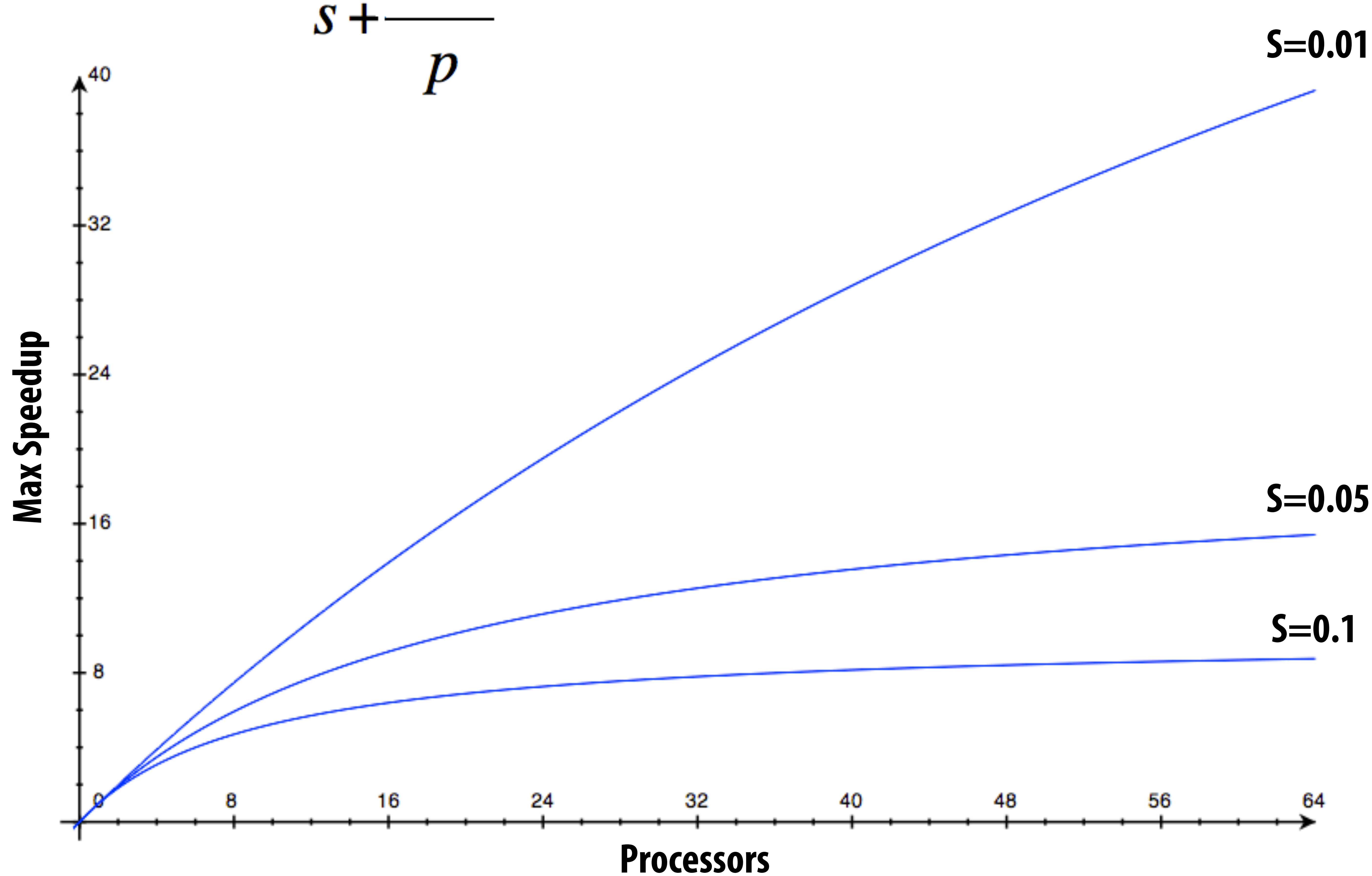
- **You used ISPC to parallelize the Mandelbrot generation**
- **You created a bunch of tasks. How many? Why?**

```
uniform int rowsPerTask = height / 2;

// create a bunch of tasks

launch[2] mandelbrot_ispc_task(
                x0, y0, x1, y1,
                width, height,
                rowsPerTask,
                maxIterations,
                output);
```

Amdahl's law

- Let S = the fraction of sequential execution that is inherently sequential
- Max speedup on P processors given by:

$$\text{speedup} \leq \frac{1}{s + \frac{1-s}{p}}$$



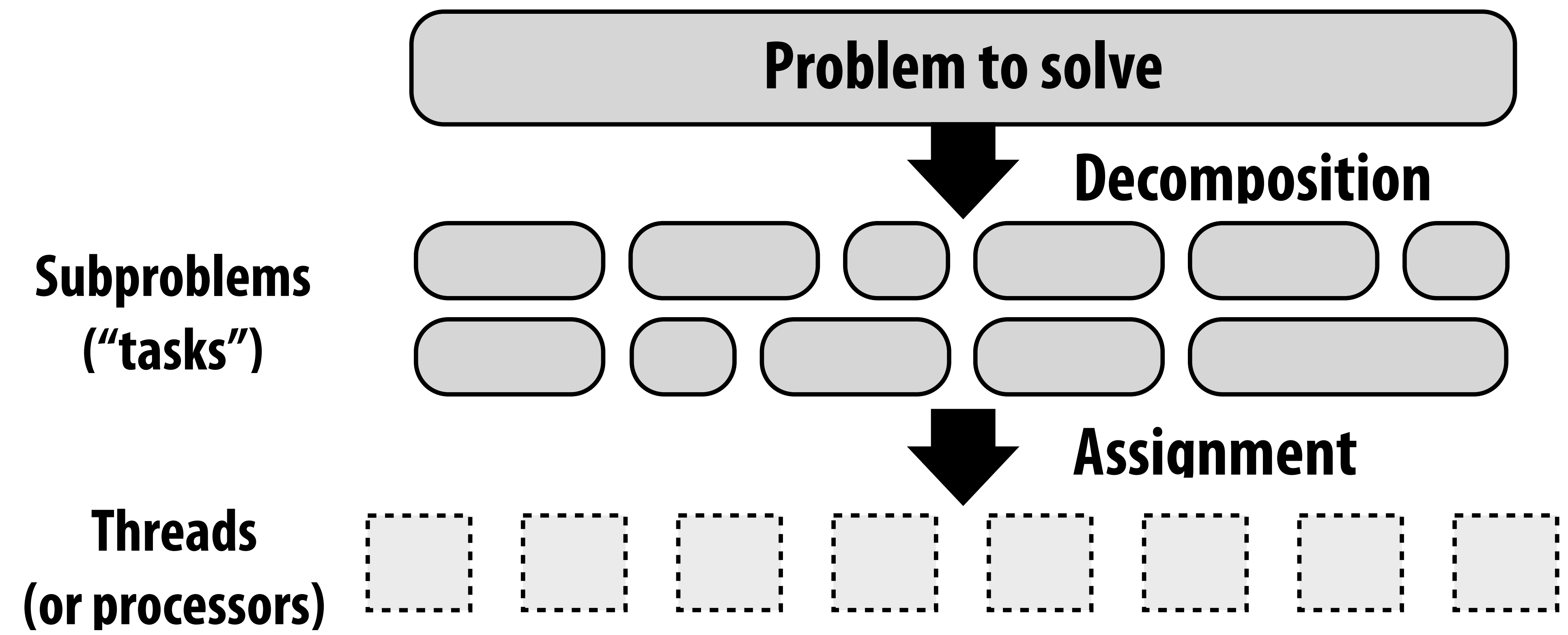
Thought experiment

- Your boss gives your team a piece of code for which 25% of the operations are inherently serial and instructs you to parallelize the application on a six-core machines in GHC 3000. He expects you to achieve 5x speedup on this application.

- Your friend shouts at your boss, “that is %#*$(%*!@ impossible”!
- Your boss shouts back, “I want employees with a can-do attitude! You haven’t thought hard enough.”

- Who is right?

Work assignment



STATIC ASSIGNMENT

Assignment of subproblems to processors is determined before (or right at the start) of execution. Assignment does not dependent on execution behavior.

Good: very low (almost none) run-time overhead
Bad: execution time of subproblems must be predictable (so programmer can statically balance load)

Examples: solver kernel, OCEAN, mandlebrot in asst 1, problem 1, ISPC foreach

DYNAMIC ASSIGNMENT

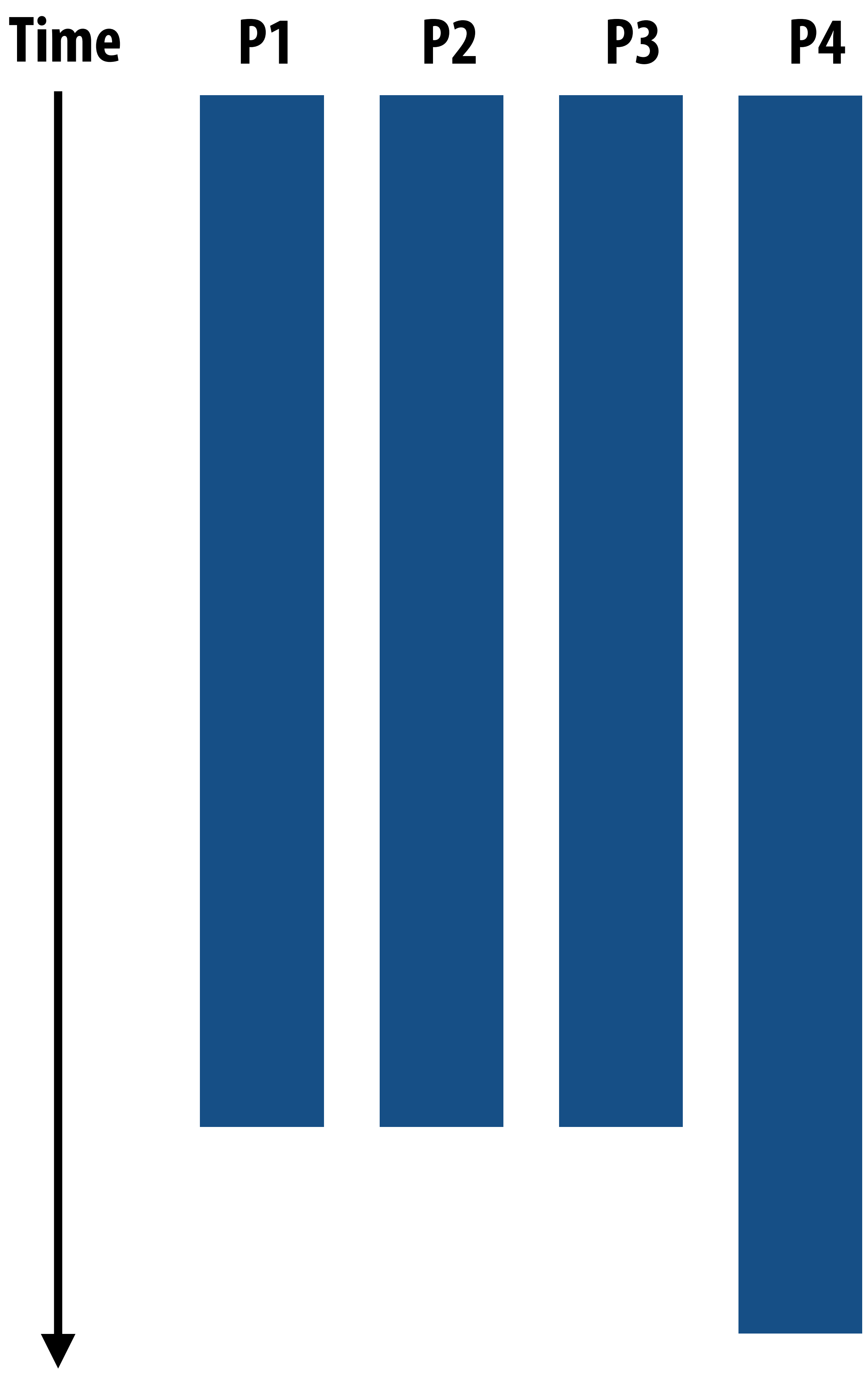
Assignment of subproblems to processors is determined as the program runs.

Good: can achieve balance load under unpredictable conditions
Bad: incurs runtime overhead to determine assignment

Examples: ISPC tasks, executing grid of CUDA thread blocks on GPU, assignment 3, shared work queue

Balancing the workload

Ideally all processors are computing all the time during program execution
(they are computing simultaneously, and they finish their portion of the work at the same time)



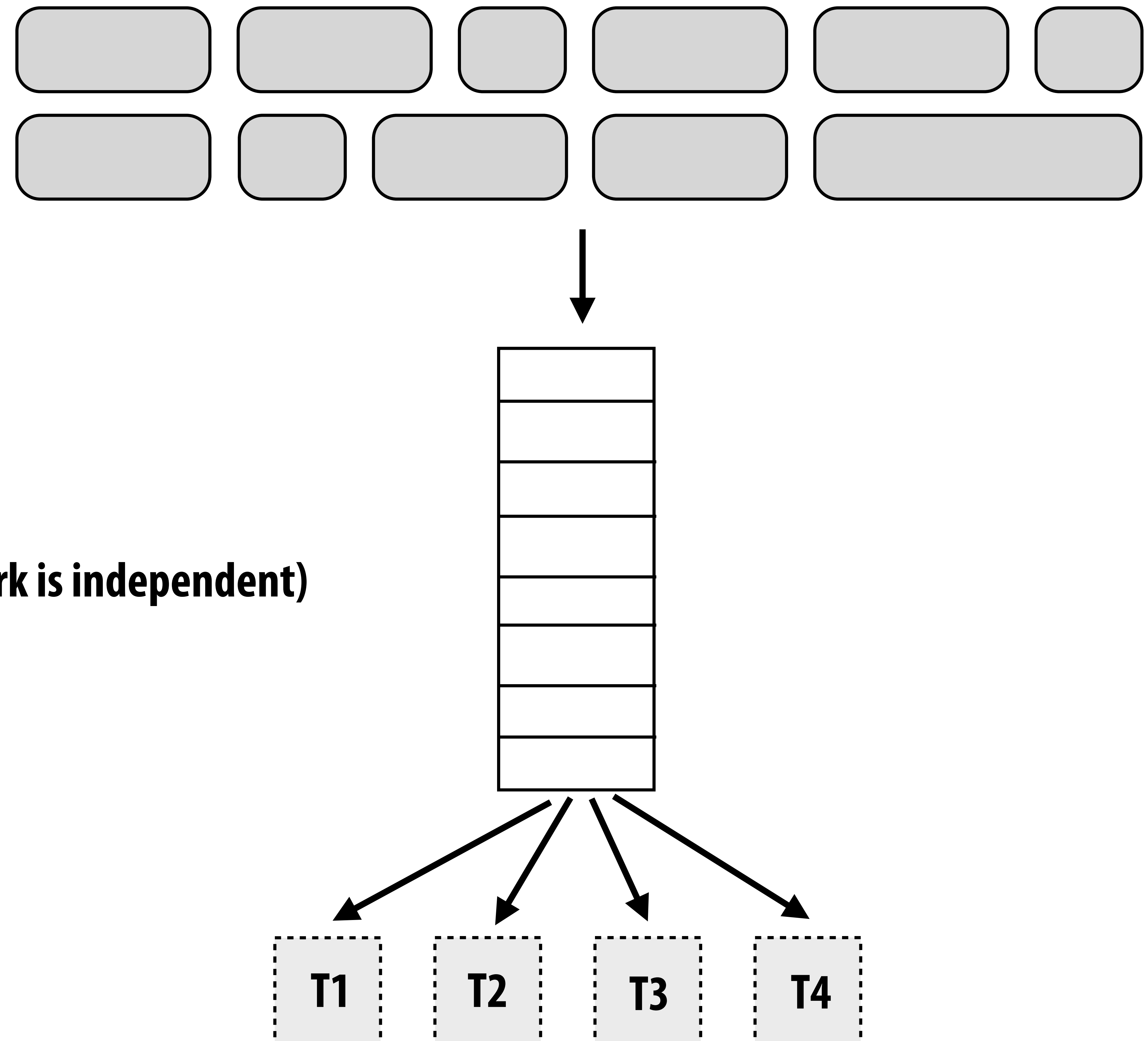
Load imbalance can significantly reduce overall speedup

Dynamic assignment using work queues

Sub-problems
(aka "tasks", "work")

Shared work queue: a list of work to do
(for now, let's assume each piece of work is independent)

Worker threads:
Pull data from work queue
Push new work to queue as it's created



Decomposition in assignment 2

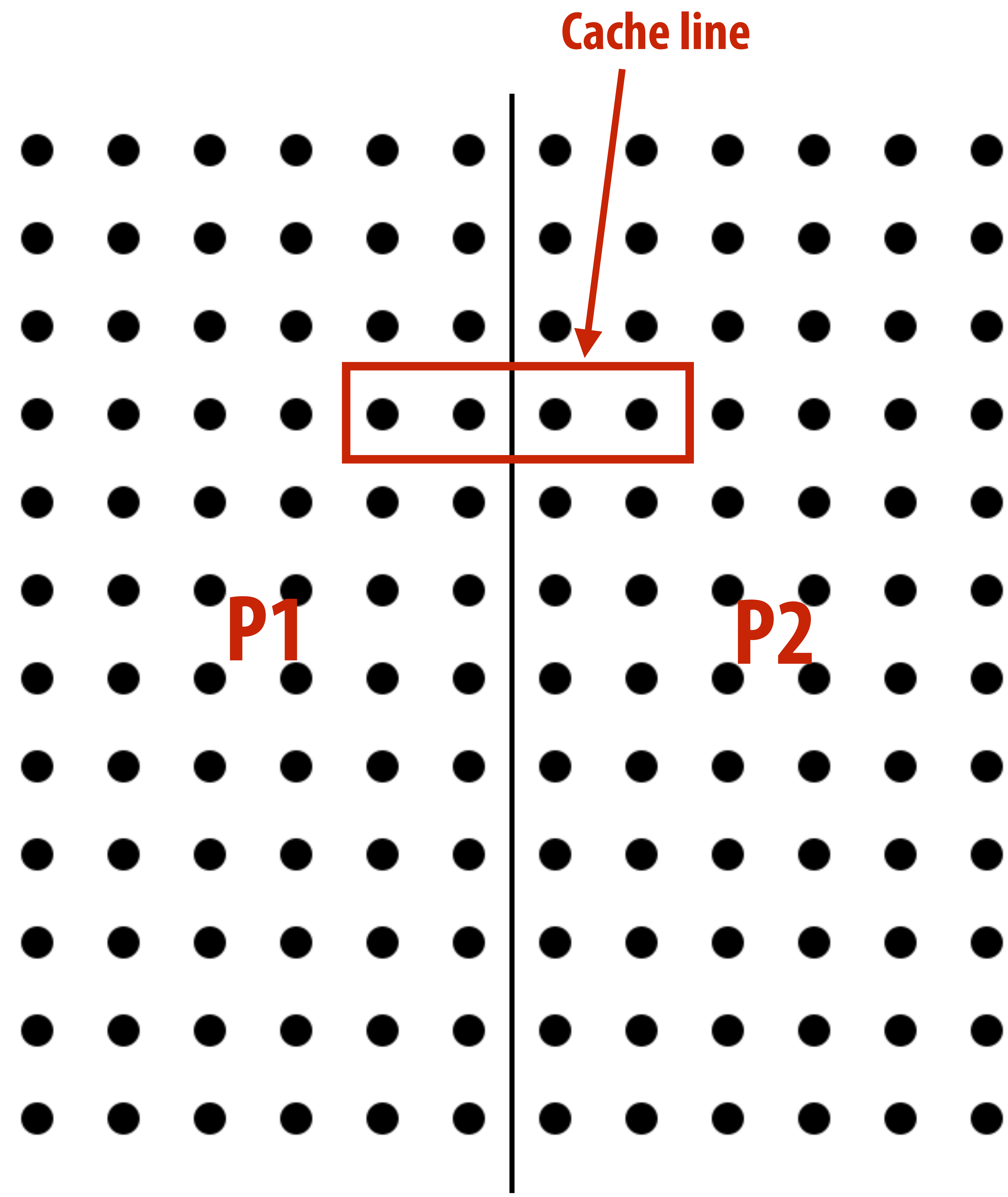
- Most solutions decomposed the problem in several ways
  - Decomposed screen into tiles ("task" per tile)
    - Decomposed tile into per circle "tasks"
    - Decomposed tile into per pixel "tasks"

Artifactual vs. inherent communication

INHERENT COMMUNICATION

ARTIFACTUAL COMMUNICATION

FALSE SHARING



Problem assignment as shown. Each processor reads/writes only from its local data.

Programming model abstractions

	Structure?	Communication?	Sync?
1. shared address space	Multiple processors sharing an address space.	Implicit: loads and stores to shared variables	Synchronization primitives such as locks and barriers
2. message passing	Multiple processors, each with own memory address space.	Explicit: send and receive messages	Build synchronization out of messages.
3. data-parallel	Rigid program structure: single logical thread containing `map(f, collection)` where "iterations" of the map can be executed concurrently	Typically not allowed within map except through special built-in primitives (like "reduce"). Comm implicit through loads and stores to address space	Implicit barrier at the beginning and end of the map.

Cache coherence

Why cache coherence?

Hand-wavy answer: would like shared memory to behave "intuitively" when two processors read and write to a shared variable. Reading a value after another processor writes to it should return the new value. (despite replication due to caches)

Requirements of a coherent address space

1. **A read by processor P to address X that follows a write by P to address X, should return the value of the write by P** ***(assuming no other processor wrote to X in between)***

2. **A read by a processor to address X that follows a write by another processor to X returns the written value... if the read and write are sufficiently separated in time** ***(assuming no other write to X occurs in between)***

3. **Writes to the same location are serialized; two writes to the same location by any two processors are seen in the same order by all processors.**
   (Example: if values 1 and then 2 are written to address X, no processor observes 2 before 1)

Condition 1: program order (as expected of a uniprocessor system)

Condition 2: write propagation: The news of the write has to eventually get to the other processors. Note that precisely <u>when</u> it is propagated is not defined by definition of coherence.

Condition 3: write serialization

Implementing cache coherence

Main idea of invalidation-based protocols: before writing to a cache line, obtain exclusive access to it

SNOOPING

Each cache broadcasts its cache misses to all other caches. Waits for other caches to react before continuing.

Good: simple, low latency
Bad: broadcast traffic limits scalability

DIRECTORIES

Information about location of cache line and number of shares is stored in a centralized location. On a miss, requesting cache queries the directory to find sharers and communicates with these nodes using point-to-point messages.

Good: coherence traffic scales with number of sharers, and number of sharers is usually low
Bad: higher complexity, overhead of directory storage, additional latency due to longer critical path

MSI state transition diagram