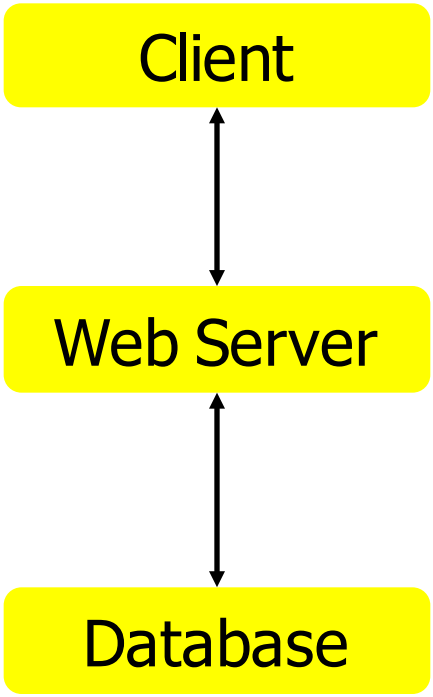
# Agenda

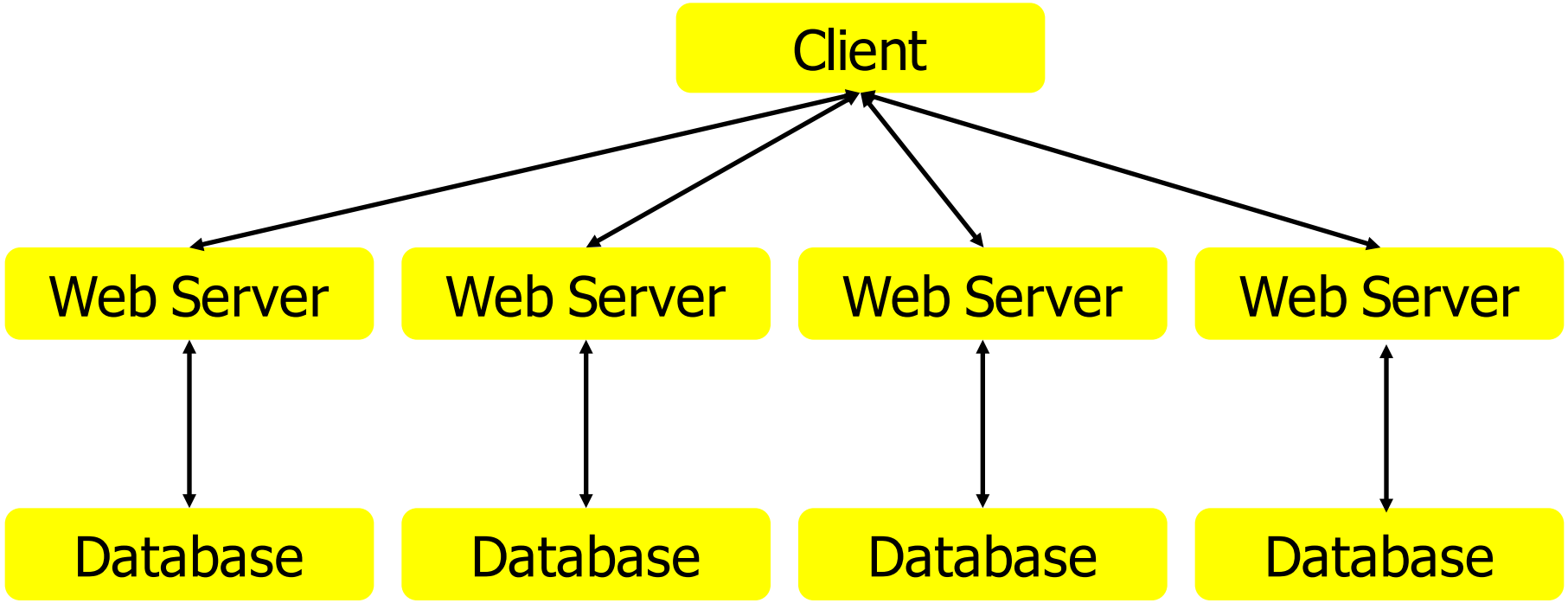
- Scaling
- Load Balancing
- Replication
- Caching
- NoSQL
- Datacenters and code

# Multiple Tiers



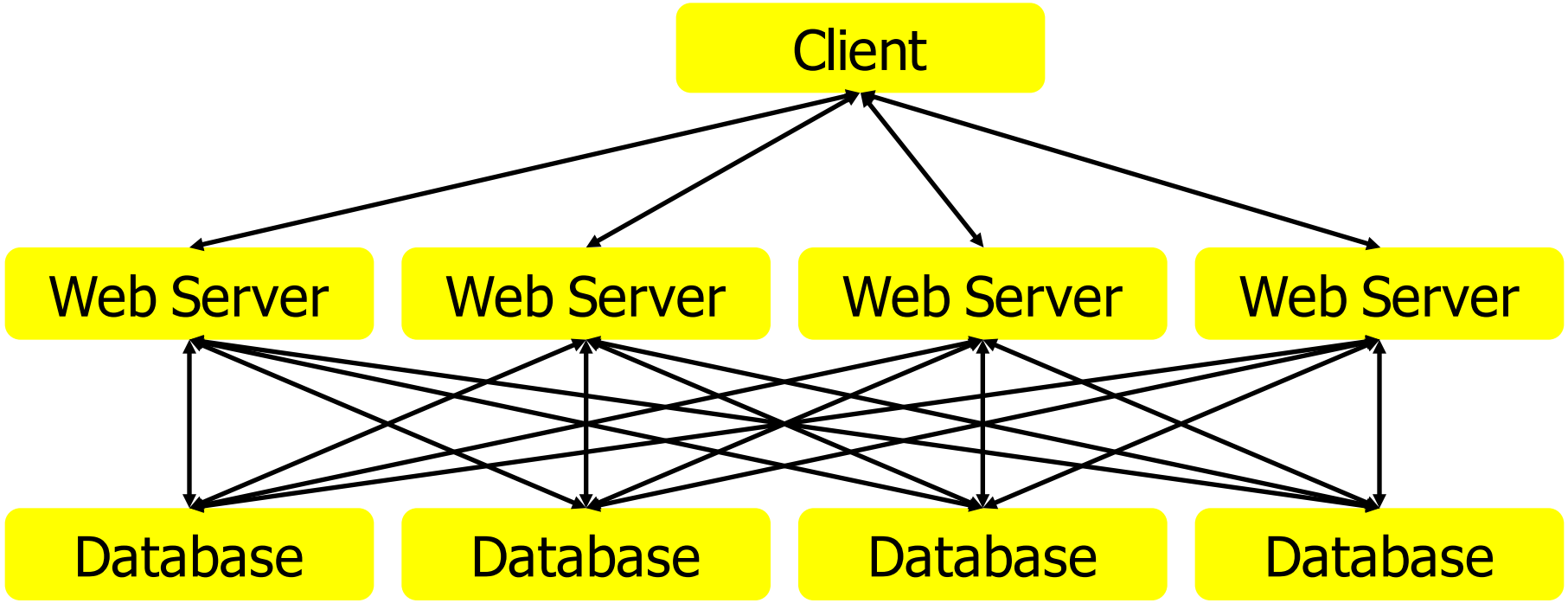

- We have covered making a web site
  - With one server and one database

# Replicate to Split Load




- How do we make a website for millions (or billions!!!) of users?
- Replicate to split load

# Network for Independence




- Scaling
- Network for independence

# Scaling

- Scaling the number of servers
- Scaling data
- Scaling datacenters
- Scaling code

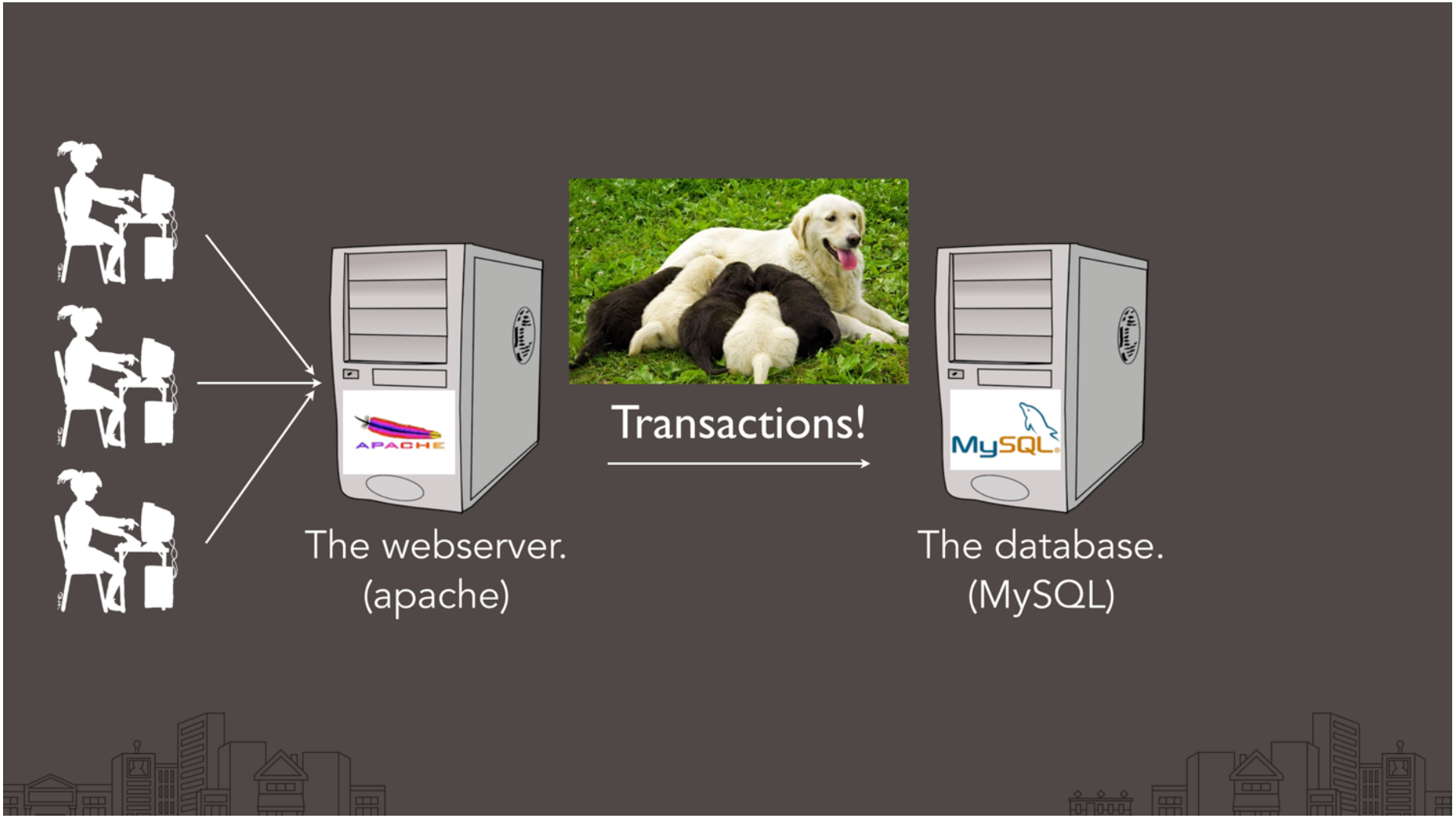
- More traffic means more servers and more data

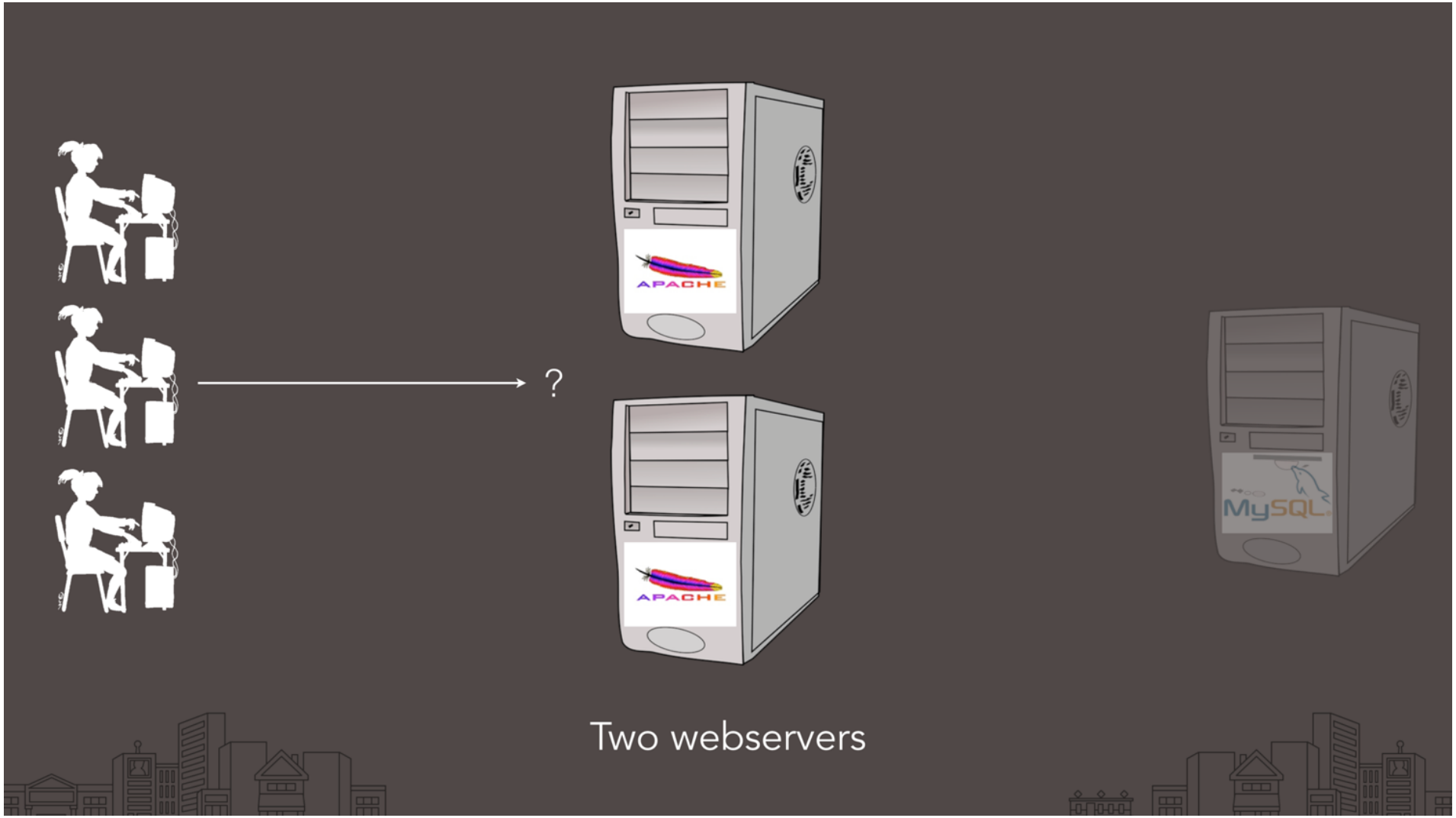
# From servers to cloud

- We started with a webserver


The webserver.
(apache)


The database.
(MySQL)


APACHE
Transactions!
MySQL
The webserver.
(apache)
The database.
(MySQL)


?
APACHE
APACHE
MySQL
Two webservers

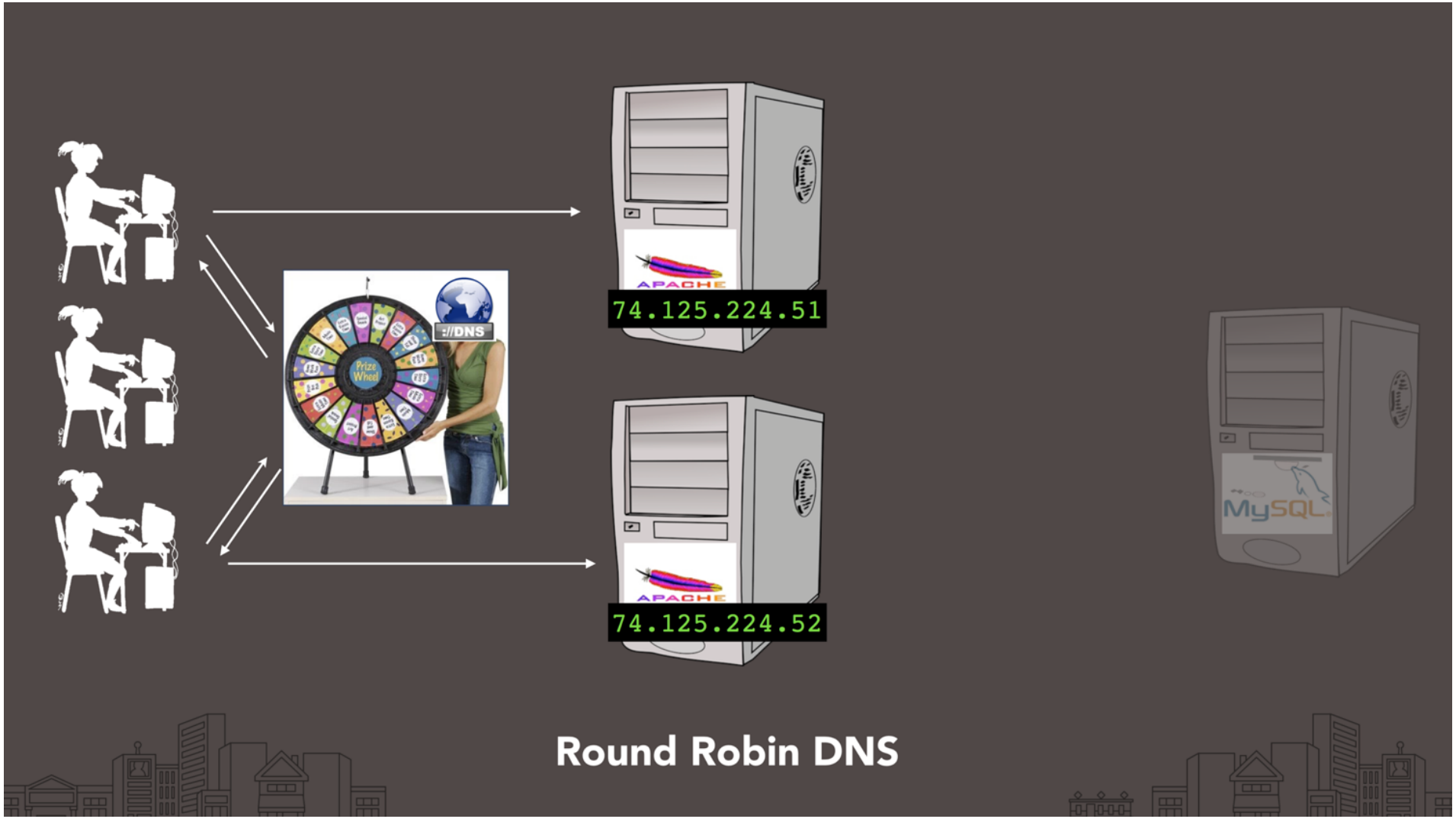
# Round Robin DNS

- Multiple IP address for one domain name
- DNS server responds to a DNS request with a *list of IP addresses*

```
$ host google.com
google.com has address 192.122.185.23
google.com has address 192.122.185.34
google.com has address 192.122.185.53
google.com has address 192.122.185.59
...
```

# Round Robin DNS

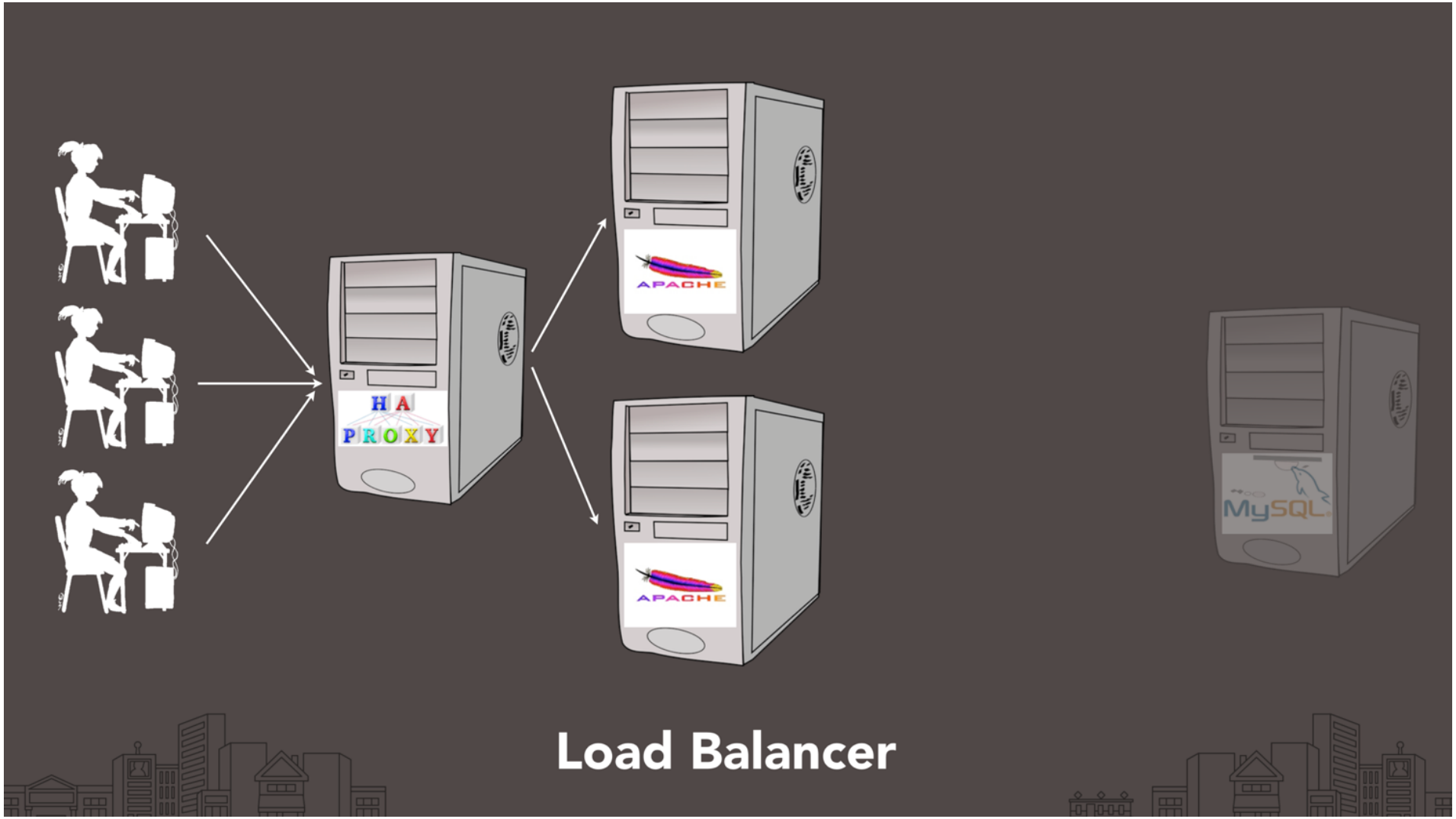
- Permute the list with each response
- No set method for client to select address
  - Usually just the first address
- Does not consider
  - transaction time
  - server load
  - network congestion
  - etc.
- No protection from server going down

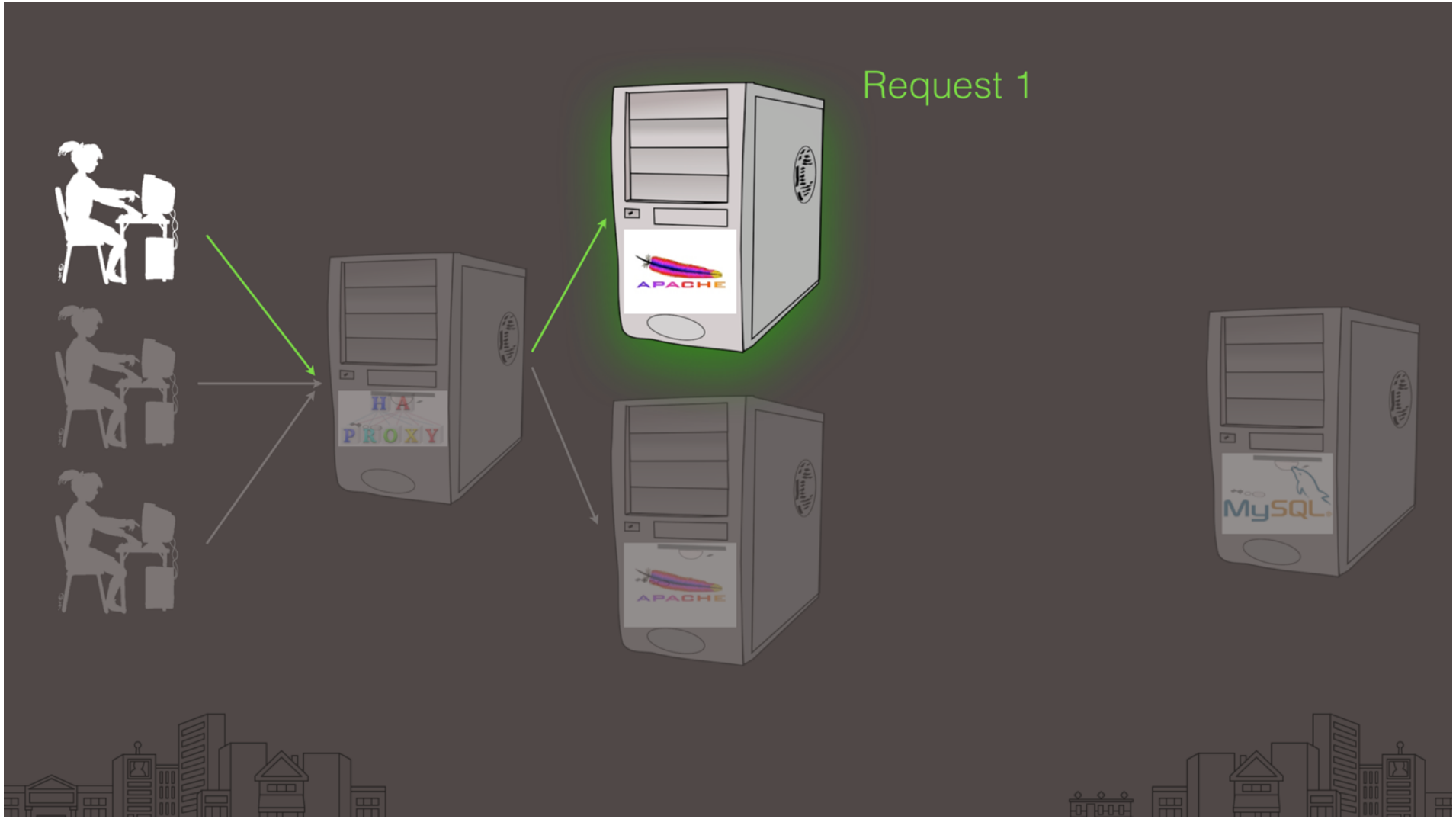


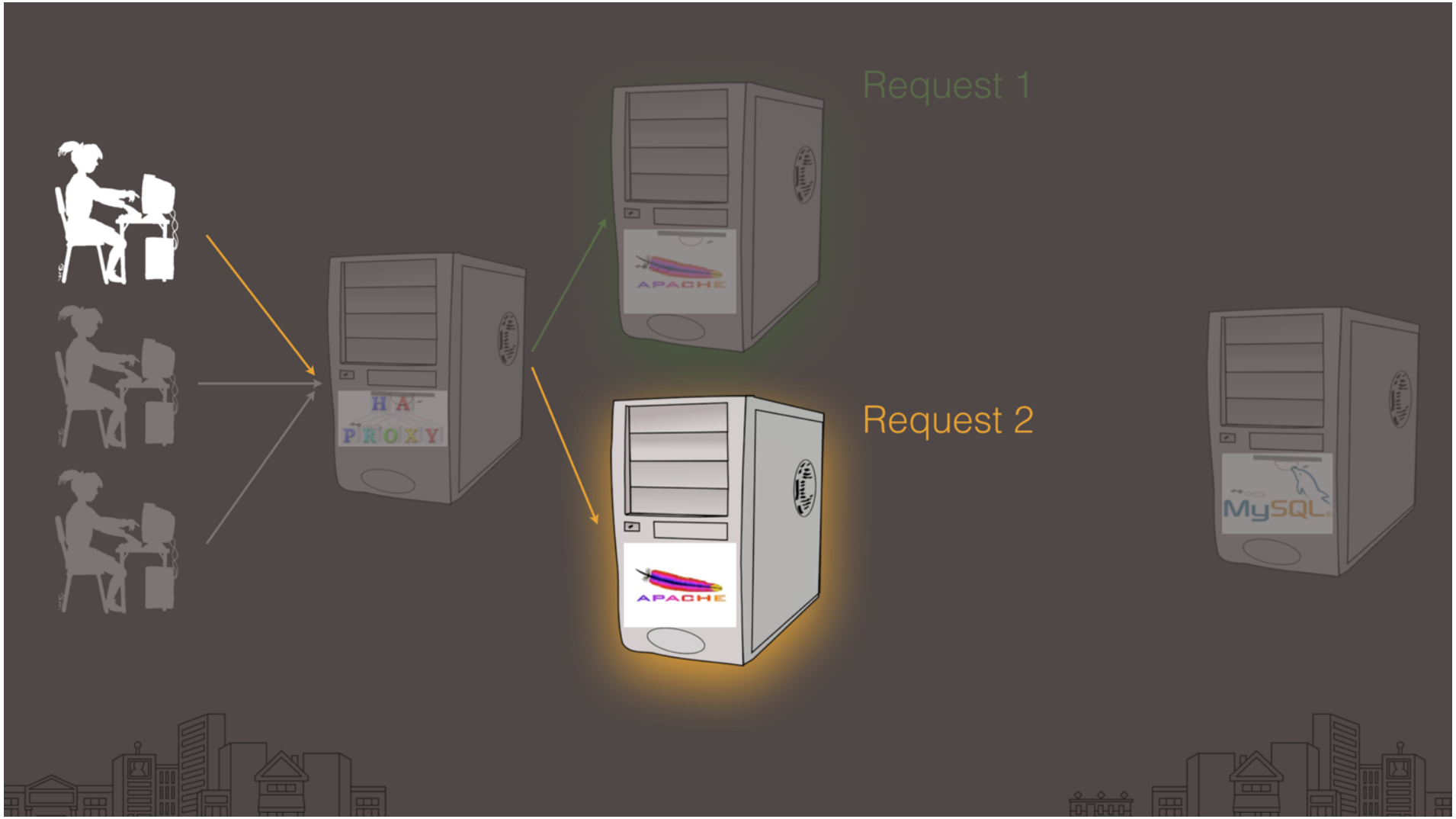

Round Robin DNS

# Proxies

- Another way to do it: proxy
- "Middleman" forwards requests to backend servers
- End user doesn't need to know about them
  - E.g., transparent to end user


HA
PROXY
APACHE
APACHE
MySQL
Load Balancer

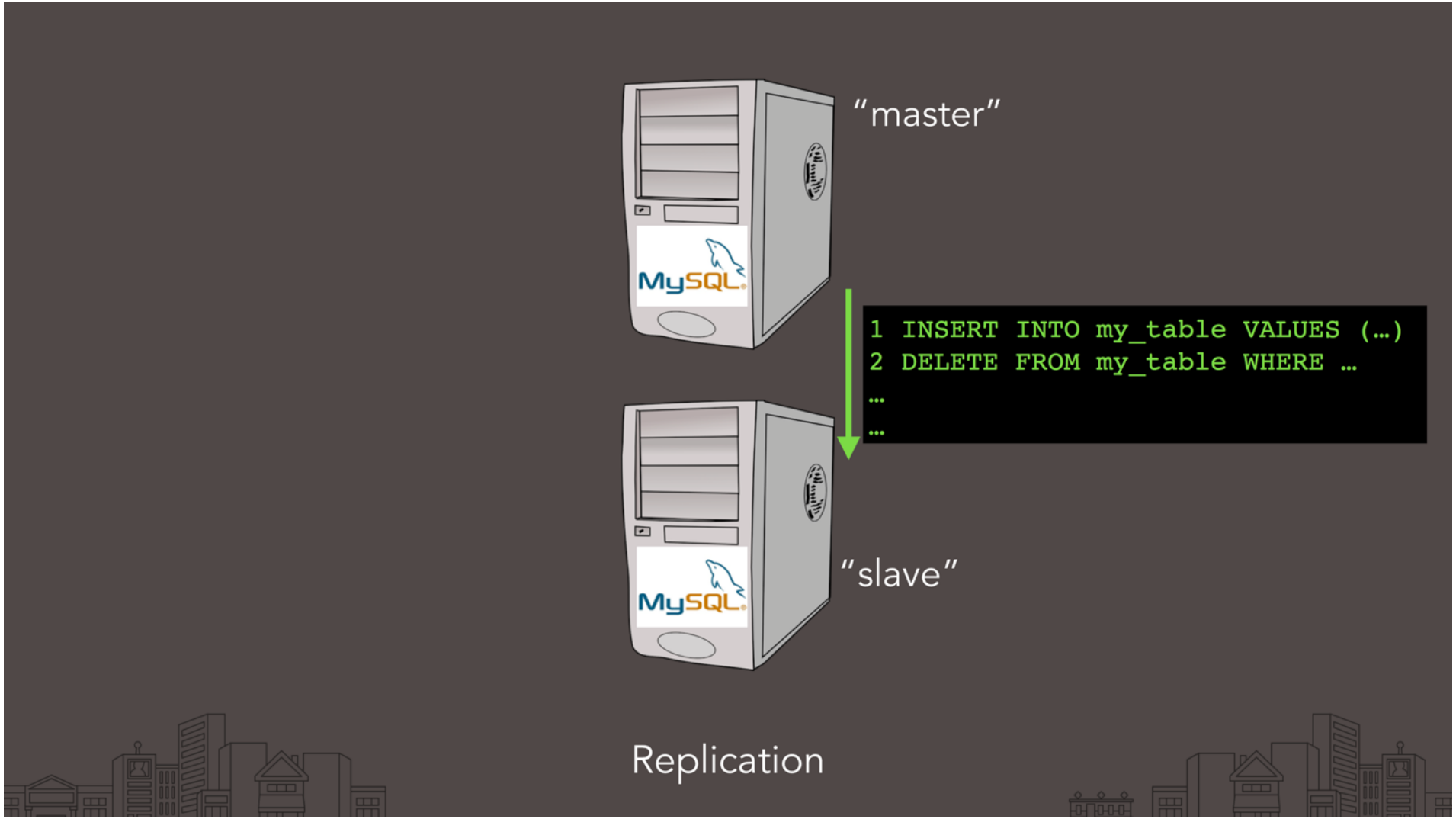

Request 1
HA
PROXY
APACHE
APACHE
MySQL

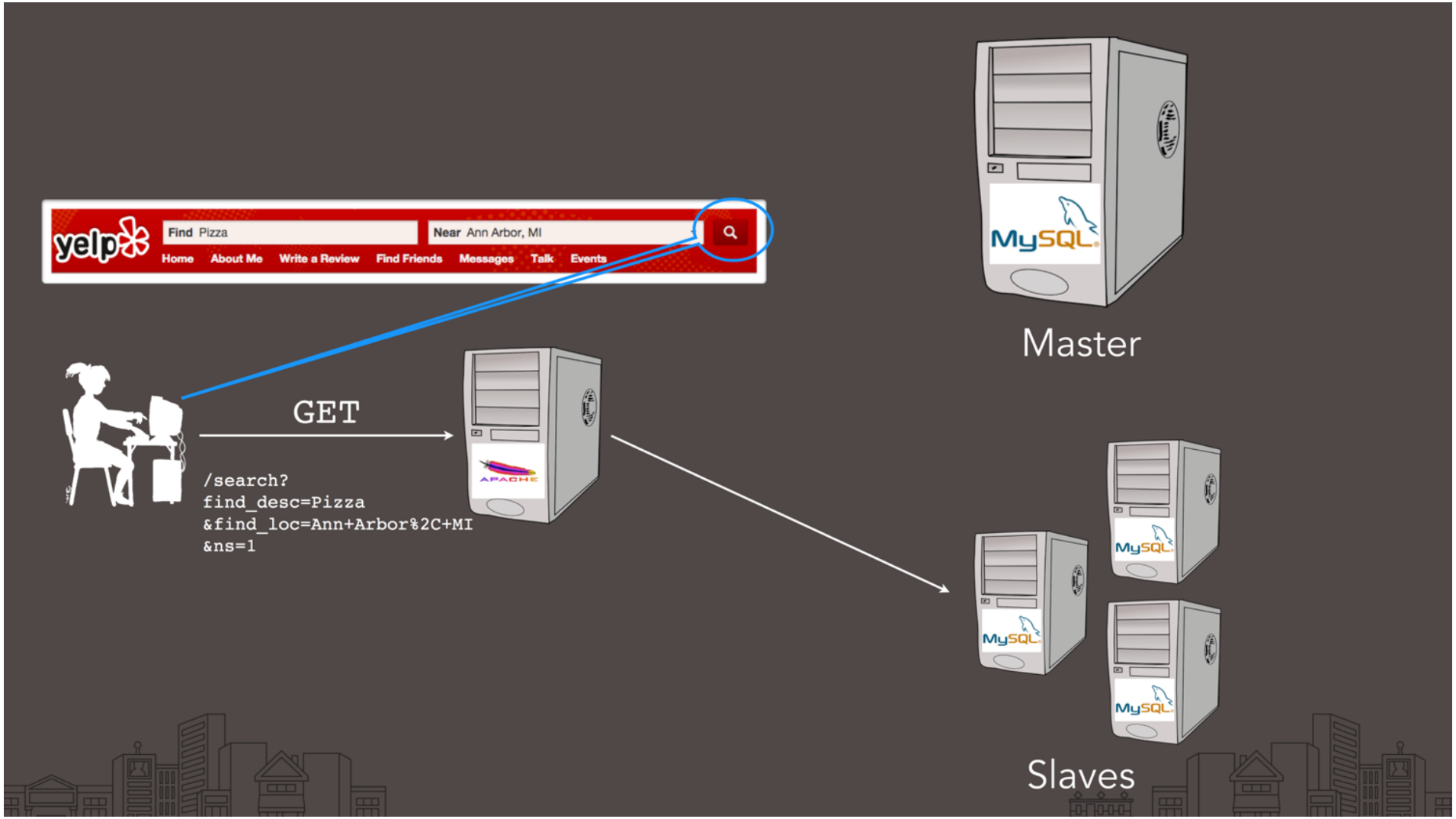

Request 1
HA PROXY
APACHE
Request 2
APACHE
MySQL

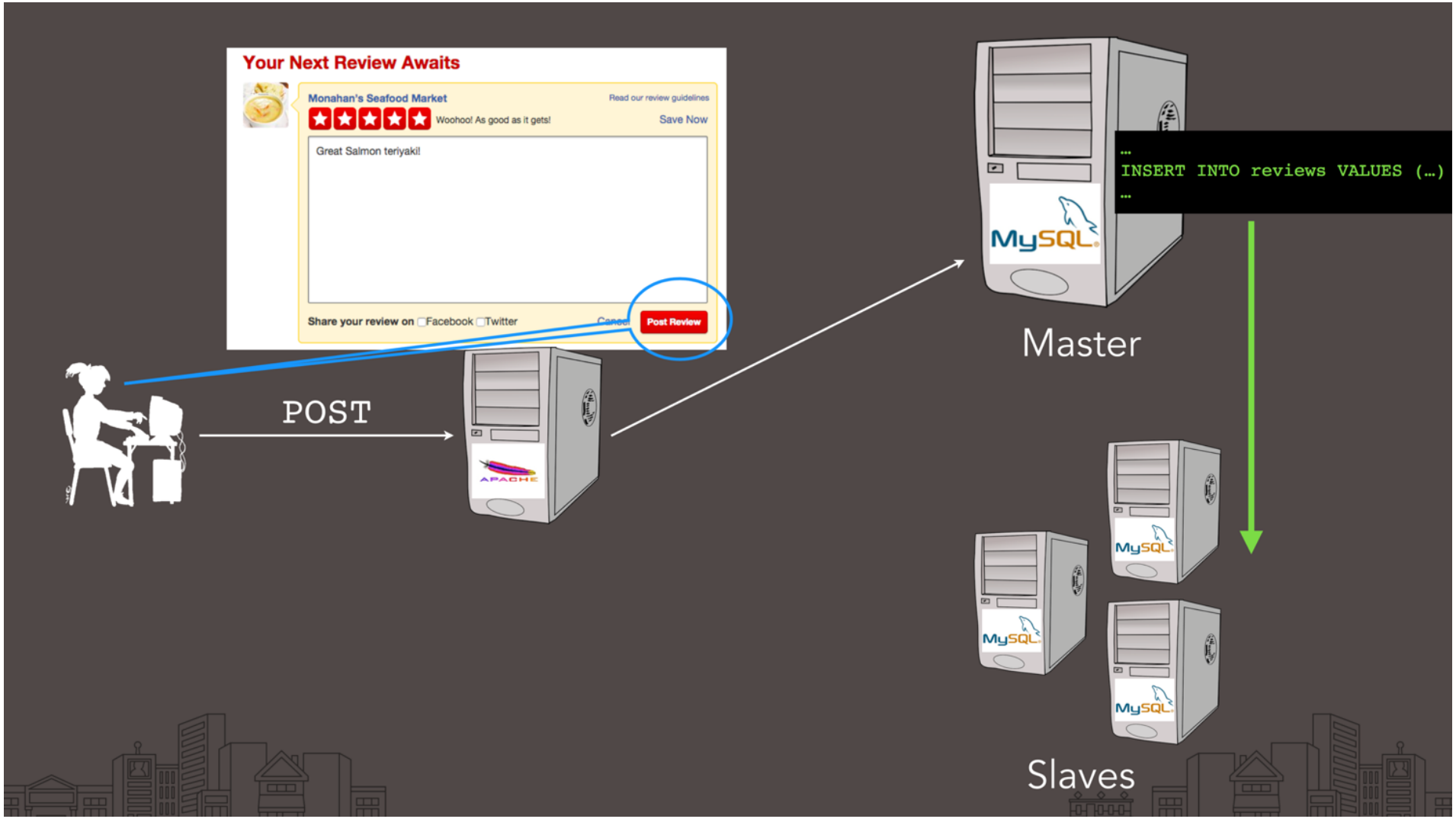
# DNS + Load balancing

- Round Robin DNS or Load balancing?
  - Large websites do both
- Round Robin DNS to multiple load balancers for redundancy
- Load balancers to backend web servers for scaling
- Web servers should be stateless
  - That's what the database is for


MySQL
MySQL
Eventually you'll need more database servers

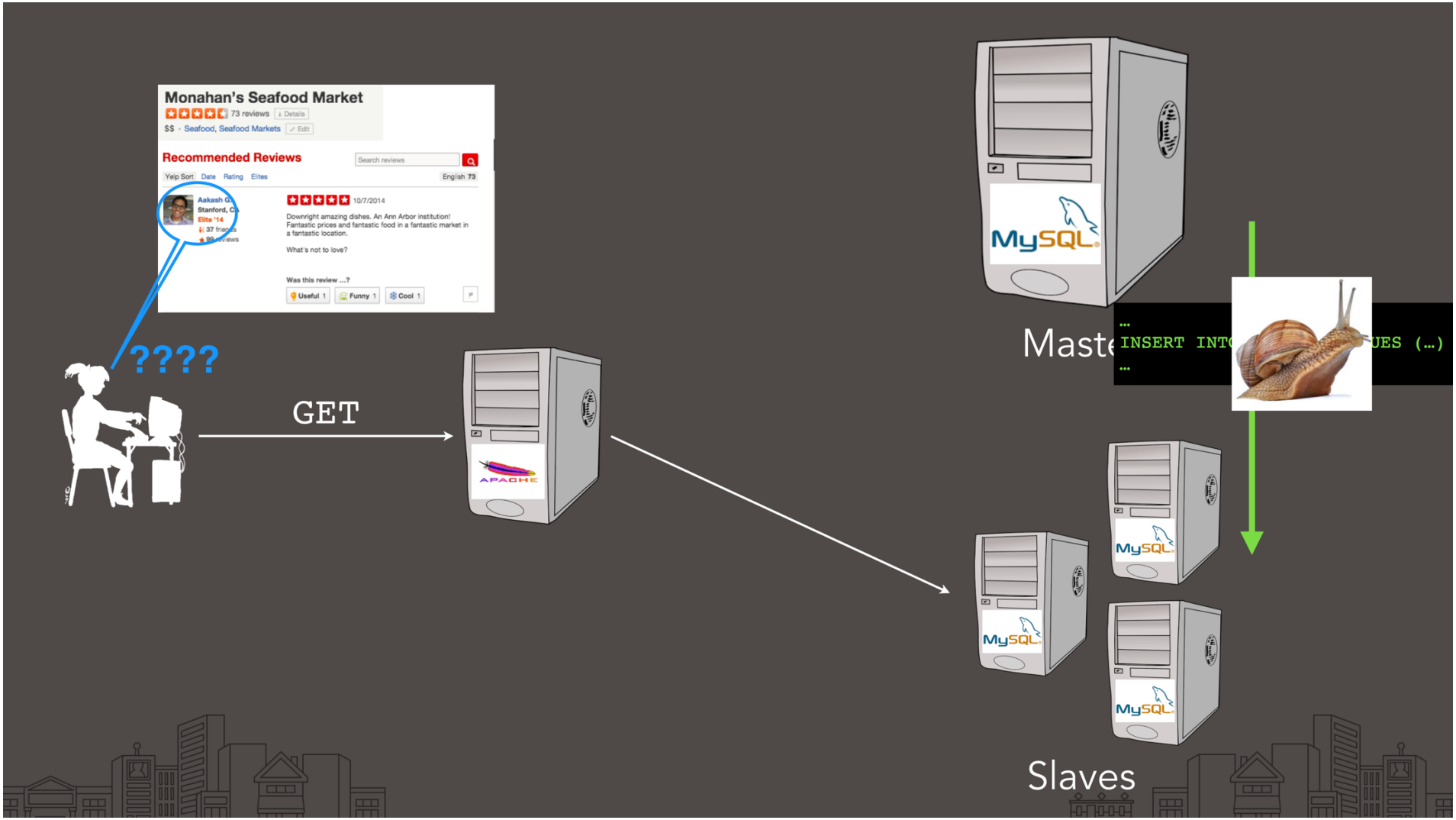

"master"
MySQL
1 INSERT INTO my_table VALUES (…)
2 DELETE FROM my_table WHERE …
…
…
MySQL
"slave"
Replication


yelp
Find Pizza
Near Ann Arbor, MI
Home
About Me
Write a Review
Find Friends
Messages
Talk
Events
GET
/search?
find_desc=Pizza
&find_loc=Ann+Arbor%2C+MI
&ns=1
APACHE
MySQL
Master
MySQL
MySQL
MySQL
Slaves


Your Next Review Awaits
Monahan's Seafood Market
Read our review guidelines
Woohoo! As good as it gets!
Save Now
Great Salmon teriyaki!
Share your review on Facebook Twitter
Post Review
POST
APACHE
MySQL
…
INSERT INTO reviews VALUES (…)
…
Master
Slaves

# Replicate Data

- Replicate the data so any database can handle any query
- Good for query workload
- Bad for updates
- Worry about inconsistent database if update hits some replicas only
- Can solve by having one master


Monahan's Seafood Market
73 reviews
$$ · Seafood, Seafood Markets
Recommended Reviews
Search reviews
Yelp Sort Date Rating Elites
English 73
10/7/2014
Downright amazing dishes. An Ann Arbor institution! Fantastic prices and fantastic food in a fantastic market in a fantastic location.
What's not to love?
Was this review ...?
Useful 1
Funny 1
Cool 1
????
GET
APACHE
MySQL
Maste
INSERT INT
UES (…)
Slaves

# Partition Data

- Partition data so each database server only keeps a fraction of the data
- Request has to be sent to the right database server(s)
- If partitioning basis is clean, and aligns with queries, can work well
- Can be administrative nightmare and/or computationally difficult if data from multiple partitions must be combined

# Wide Area Distribution

- Orthogonal to replication/partitioning
- Replicas called "mirrors"
- Good for disaster recovery
- Mirroring can even be used for web server
  - Exploit network proximity
- Partitioning can often be aligned geographically
  - E.g. account based on user geo-location
    - And not too much shared info across accounts

?


USA
West
East

# Caching

- Keeping a local copy of a resource
- Necessary for getting reasonable performance
  - From computers
  - From the Web
- Multiple things we cache in the web
  - URLs
  - DNS Resource Records
  - Web Page content

# Multiple Cache Points

- Multiple places we cache in the web
  - Browsers
  - Proxies
  - Servers ("front-side caching")
- Each independently decides what to keep in cache
- Cache space is limited
  - Choice is an optimization problem
  - Policy solved independently by every application

# Cache Invalidation

- Cached data can be stale
- Can have bad consequences
- Most content producers indicate how long something can be cached
  - And have to wait that long when they make a change to be sure its seen
    - Provided everyone follows the rules
- Some applications require stronger consistency
  - Must proactively invalidate cache

# NoSQL

- Rule #1 about NoSQL:
  - No one has any idea what it means
  - Except: someone somewhere hates SQL
- Common characteristics
  - Generally no joins
  - Optimized for simple, fast inserts/retrieval
  - Flexible data model, *relaxed consistency*

# NoSQL Review

- Systems we're pretty sure are NoSQL
  - Document dbs: MongoDB, CouchDB
  - Near-relational: Cassandra, BigTable
  - Key-value stores: Redis, Riak
  - XML stores

# CAP Theorem

- Consistency: all the nodes see the same data at the same time
- Availability: I can get my data at any time
- Partitioning: Part of the system can go down, and everything is still OK

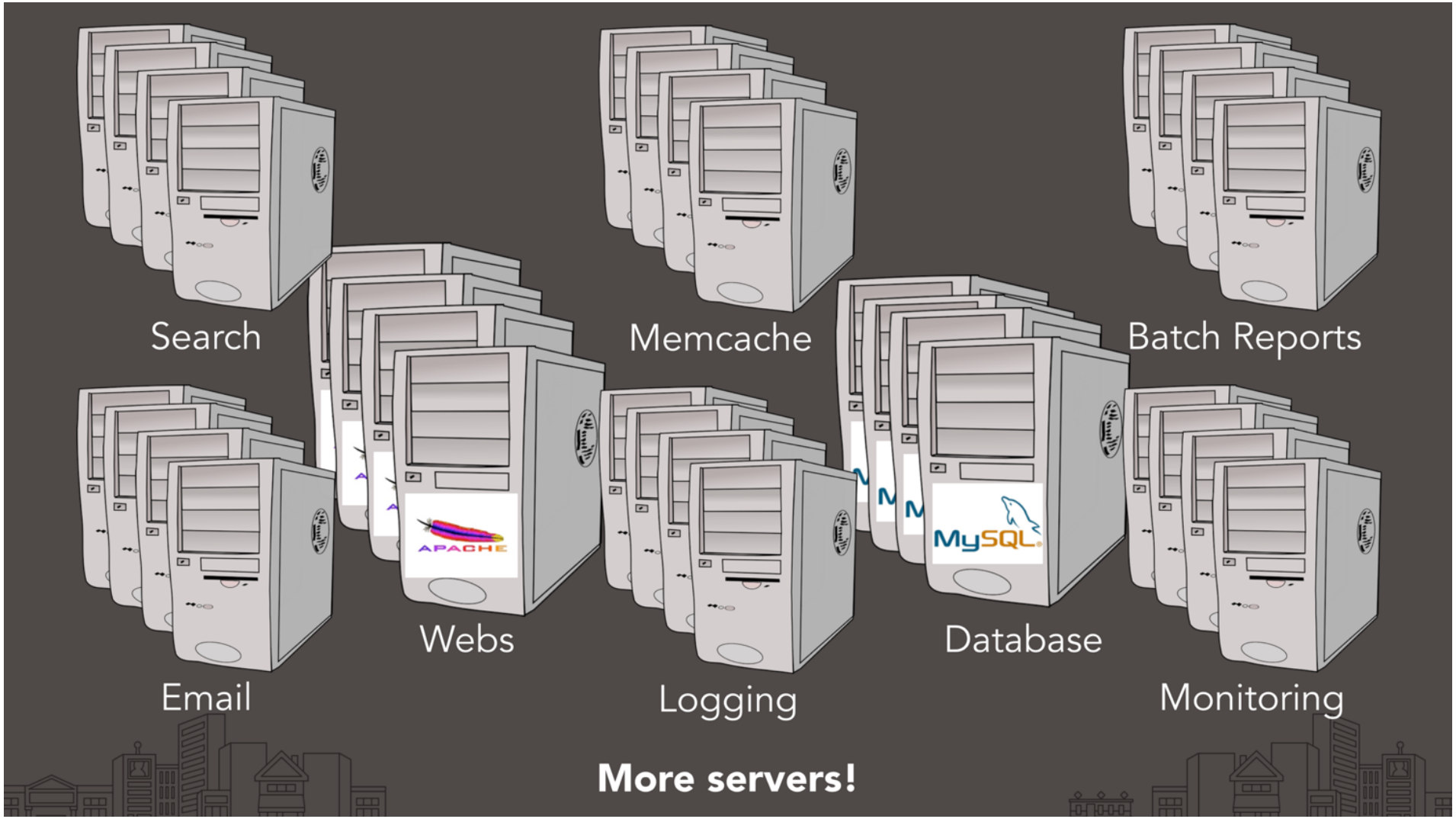
- *Pick two*

# ACID

- Guarantee that database transactions are processed reliably
- ACID: Atomicity, Consistency, Isolation, Durability
- Traditional databases (see EECS 484!)

# BASE

- When availability is more important than consistency, BASE
- BASE: Basically Available, Soft state, Eventually Consistent
- Semi-serious response to ACID, embodied in NoSQL


Search
Memcache
Batch Reports
Email
Webs
Logging
Database
Monitoring
APACHE
MySQL
More servers!

# Scaling datacenters

- How many servers? (estimates)
- Google: ~1M [2014]
- Microsoft: ~1M [2014]
- FB: ~200k [2012] no recent numbers
- Amazon: 2M [2014]

- One datacenter: ~50k – 100k *servers*

# Scaling datacenters

- Installation
- Configuration
- Monitoring
- Failure recovery

# Installation

- Rack and stack
- Power
- Cooling
- Network
- Physical security
- 1 M servers consumes 100's of megawatts of power
  - Roughly equivalent to a town of 10's of thousands of people
- ... or rent it all in the cloud

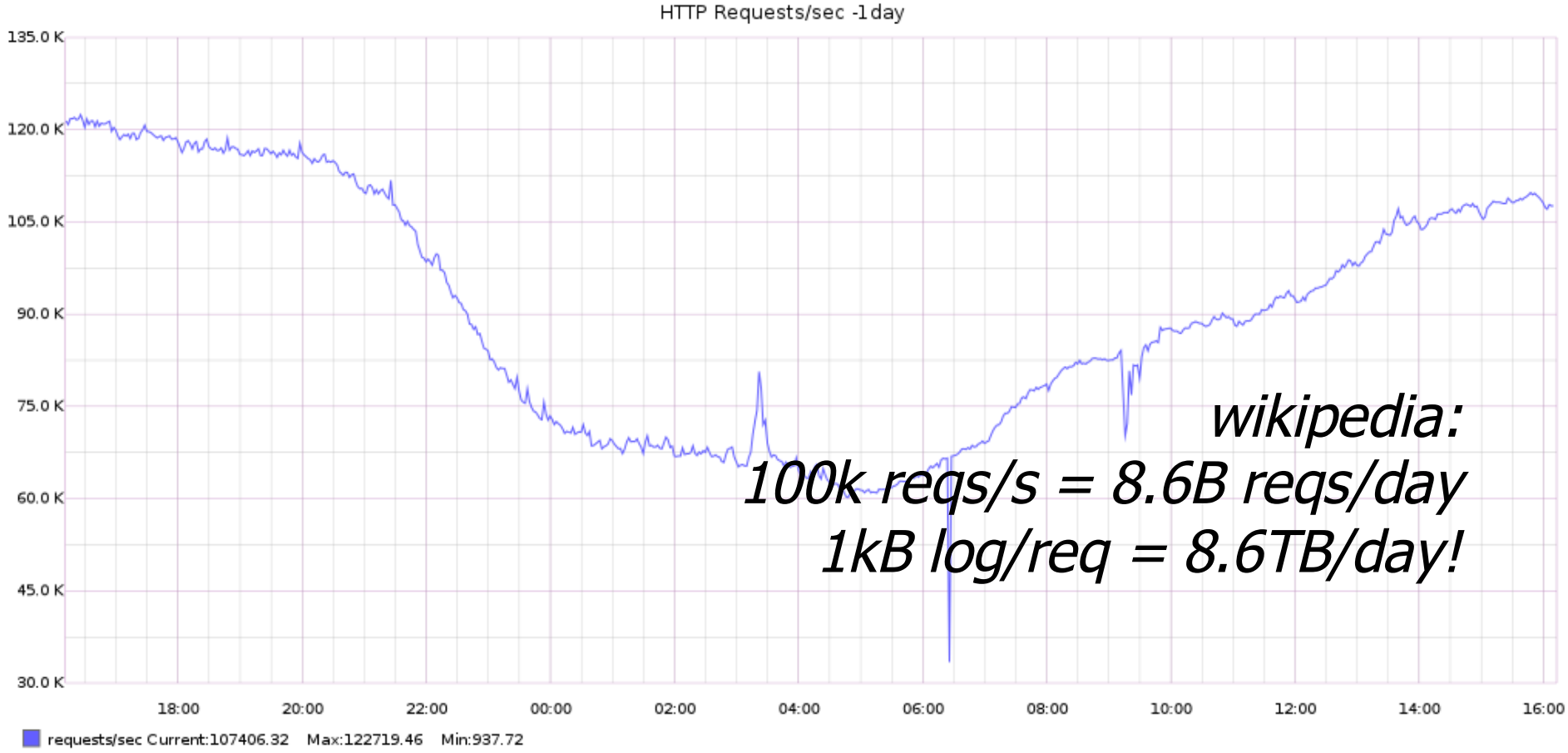
# Configuration

- How do you configure thousands of computers?
- Automate it!
  - Tools like Puppet, Chef
- Installs packages
- Runs daemons

# Monitoring

- Monitoring vs. alerting
  - What's happening vs. what do I care about?
  - What's actionable?
- For what do want to be woken at 3am?
- Signal-to-noise problem
  - Only page for important stuff!
- "Root cause" problem
  - Many alerts from each server for network outage

# Monitoring - logging




*wikipedia:*
*100k reqs/s = 8.6B reqs/day*
*1kB log/req = 8.6TB/day!*

https://gdash.wikimedia.org/dashboards/reqsum/

# Failure recovery

- More logging
- Backup data
- Redundant servers
- Test regularly (“canary server”)
- Restoring from backups part of regular workflow

# Multiple datacenters

- How to split up the work among multiple datacenters?
- **Horizontal sharding:** e.g., SF stuff in the SF datacenter, NY in east coast datacenter
- **Share nothing:** separate datacenters per product
- **Full replica:** copy everything
  - Redundancy for earthquakes, power outages, fires, etc.
  - Failover can be tricky

# Multiple datacenters

- How to connect users to right datacenter?
- GeoDNS: given an IP, find approximate location

# Avoiding down time

- Example: pushing a schema change that will occupy your database for 1 hr
- Option 1: take your site offline Wed 12am
- Option 2: "Read only mode" point all webs to a DB replica. Only allow GETs. Again, Wed night 12am
- Option 3: Online schema change: temporarily point webs at different DB server. Track changes. After schema change, apply changes. Switch back.

# Scaling code

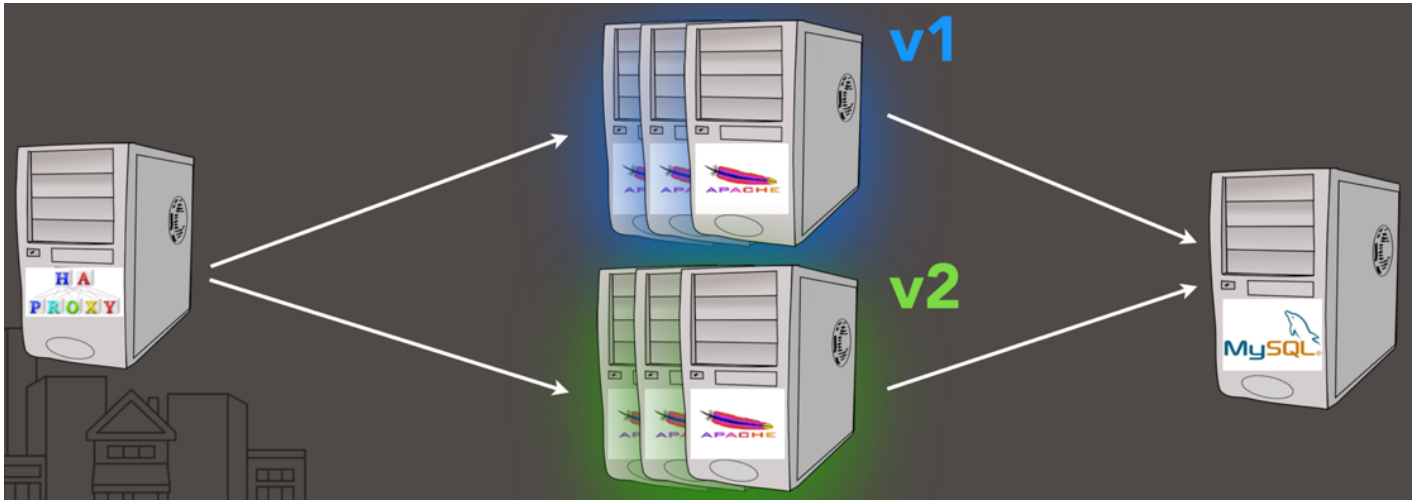
- The more similar your development environment is to production, the fewer surprises

# Scaling code

- Unit tests
  - Lots of small, fast tests
- Good abstractions
  - Boundary between components

# Deploying code

- **Rolling restarts:** restart each server, one at a time
- **Canarying:** update just one server and watch it carefully for problems
- **Blue/green development:** load balance between two separate clusters



# Thinking about scale up front

- Will it work at 10x traffic? 100x?
- How long will it take to build vs. how long will it last?
- How hard will it be to replace?
- Tradeoff: development time vs. lifetime of the system you're building
- Sometimes old code, poorly understood code, even **bad code** is OK
- Basic rule of thumb is: no design can long survive an order of magnitude increase in load