# Distributed Systems
# CS6421

## Scaling the Web

Prof. Tim Wood

# Practice / Projects

More of you should do projects!

You will learn more by trying to build something real!

If you want to get involved in research, this is your chance!

- I will be accepting students into a 3 credit Research course for the spring… but you need to do a cloud/NFV project and it needs to be done well!  Impress me!

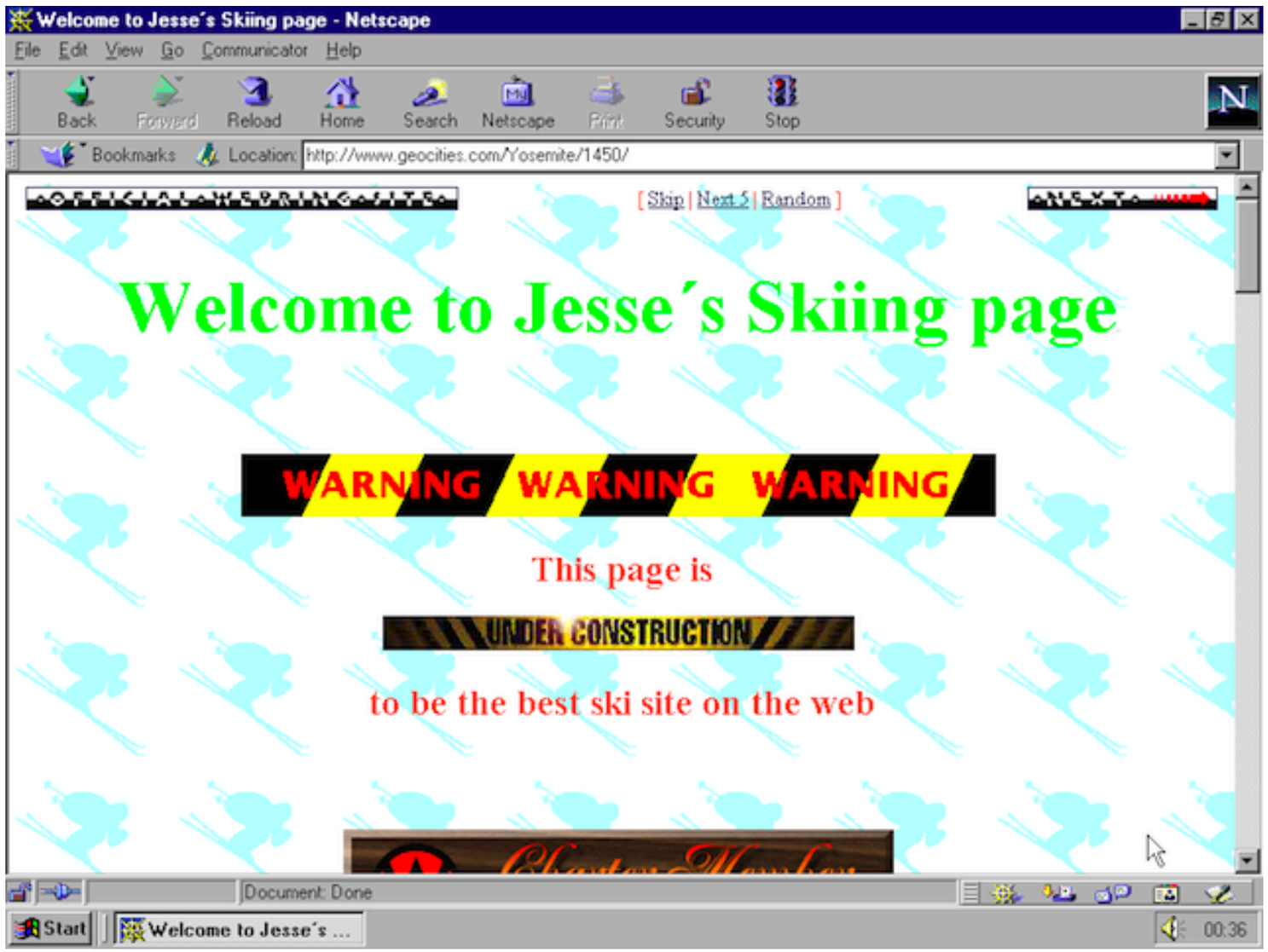
# Antique Web Servers

## Serve static content

- Read a file from disk and send it back to the client
- images, HTML

## Dynamic Content

- CGI Bin
- executes a program
- Not very safe or convenient for development…



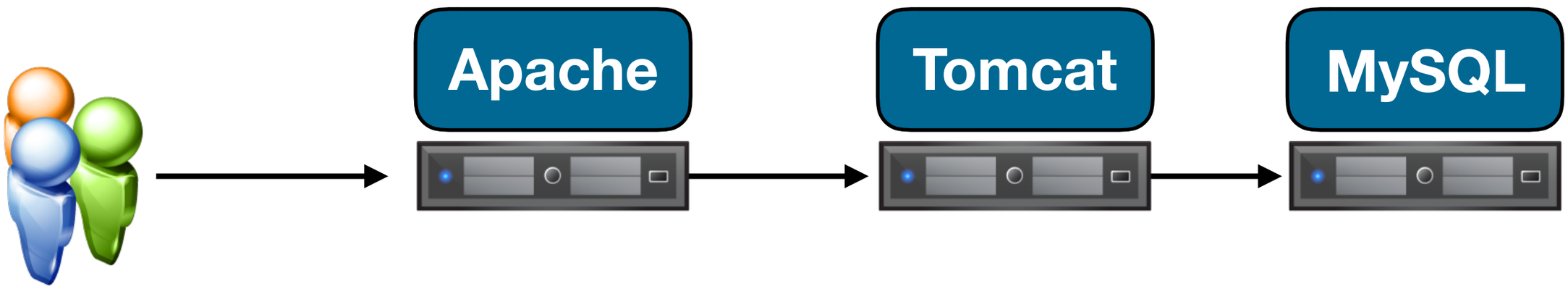
# 3-tier Web Applications

LAMP = Linux, Apache, MySQL, PHP

Separation of duties:

- Front-end web server for static content (Apache, lighttpd, nginx)
- Application tier for dynamic logic (PHP, Tomcat, node.js)
- Database back-end holds state (MySQL, MongoDB, Postgres)

Why divide up in this way?



# Stateful vs Stateless

The multi-tier architecture is based largely around whether a tier needs to worry about state

Front-end - totally **stateless**

- There is no data that must be maintained by the server to handle subsequent requests

Application tier - maintains **per-connection state**

- There is some temporary data related to each user, e.g., my shopping cart
- May not be critical for reliability - might just store in memory

Database tier - global state

- Maintains the global data that application tier might need
- Persists state and ensures it is consistent

# N-Tier Web Applications

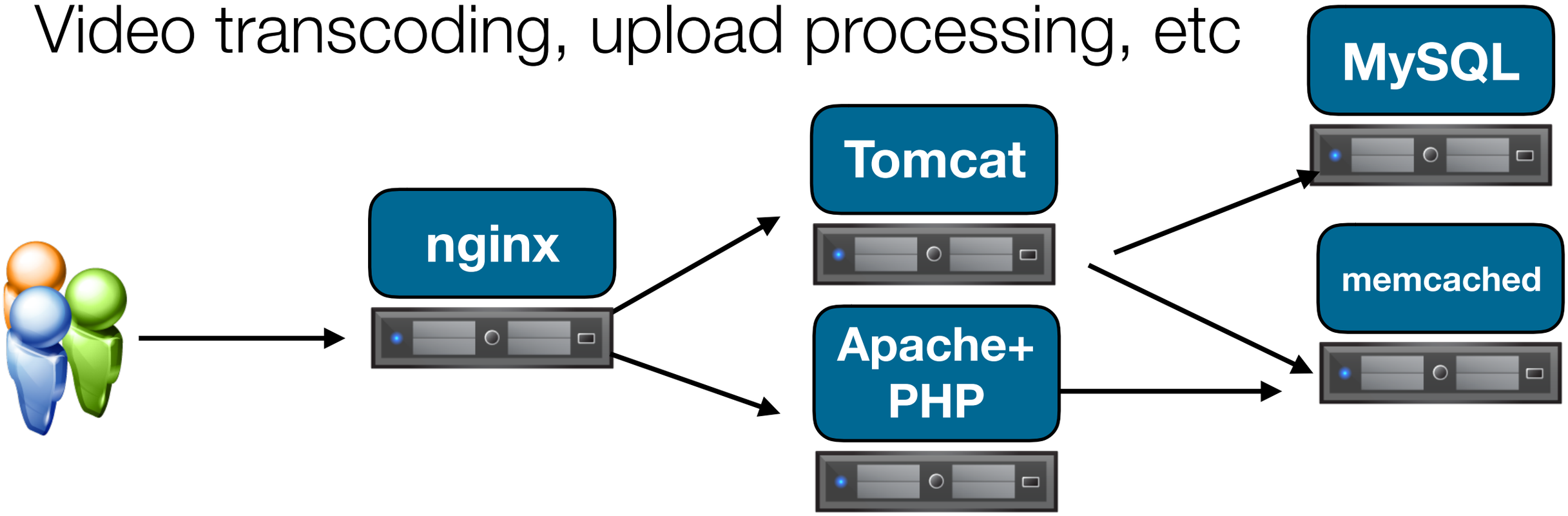
Sometimes 3 tiers isn't quite right

Database is often a bottleneck

- Add a cache! (stateful, but not persistent)

Authentication or other security services could be another tier

Video transcoding, upload processing, etc

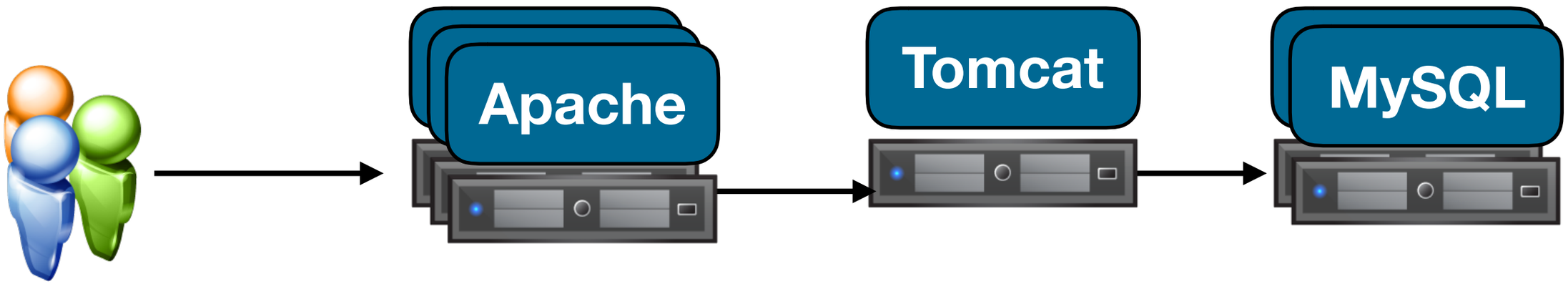


# Replicated N-Tier

Replicate the portions of the system that are likely to become overloaded

How easy to scale…?

- Apache serving static content
- Tomcat Java application managing user shopping carts
- MySQL cluster storing products and completed orders




Tune number of replicas based on demand at each tier

# Wikipedia: Big scale, cheap

5th busiest site in the world (according to alexa.com)

Runs on about **~ 1000** servers? (700 in 2012)

All open source software:

- PHP, MariaDB, Squid proxy, memcached, Ubuntu

Goals:

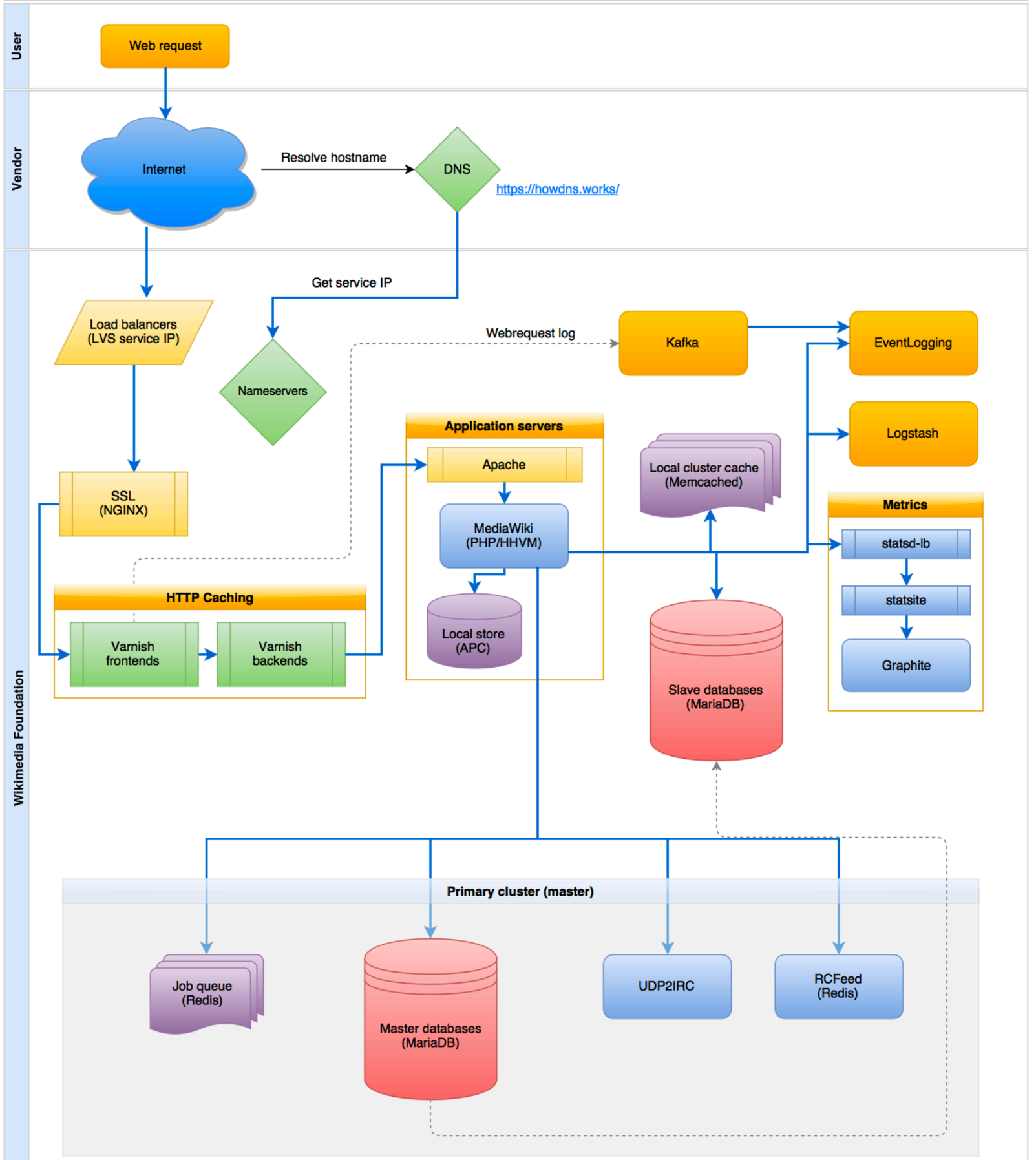
- Store lots of content (6TB of text data as of 2018)
- Make available worldwide
- Do this as cheaply as possible
- Relatively weak consistency guarantees

Stats: https://grafana.wikimedia.org

# Wikipedia

## Networking and application infrastructure

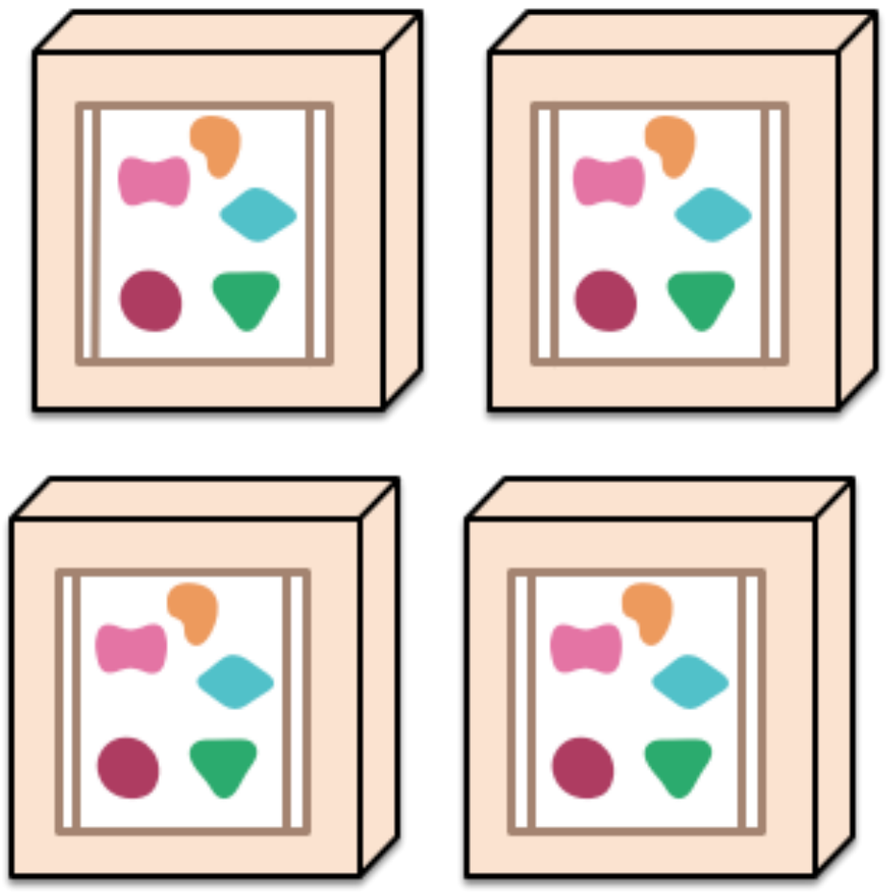
Revision: October 2015



# Application Tier

*A monolithic application puts all its functionality into a single process...*

*... and scales by replicating the monolith on multiple servers*



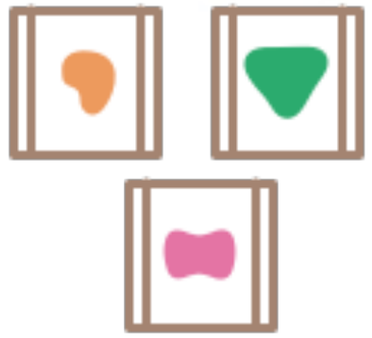
http://martinfowler.com/articles/microservices.html

**Problems with Monolithic approach?**

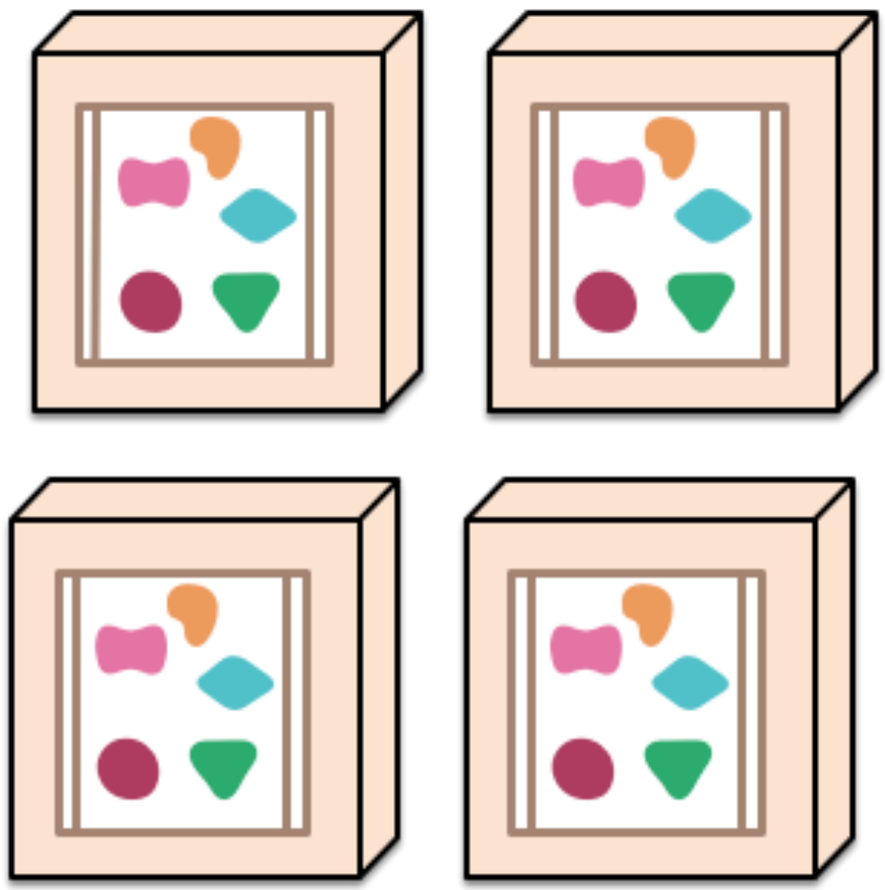
# Microservices

*A monolithic application puts all its functionality into a single process...*

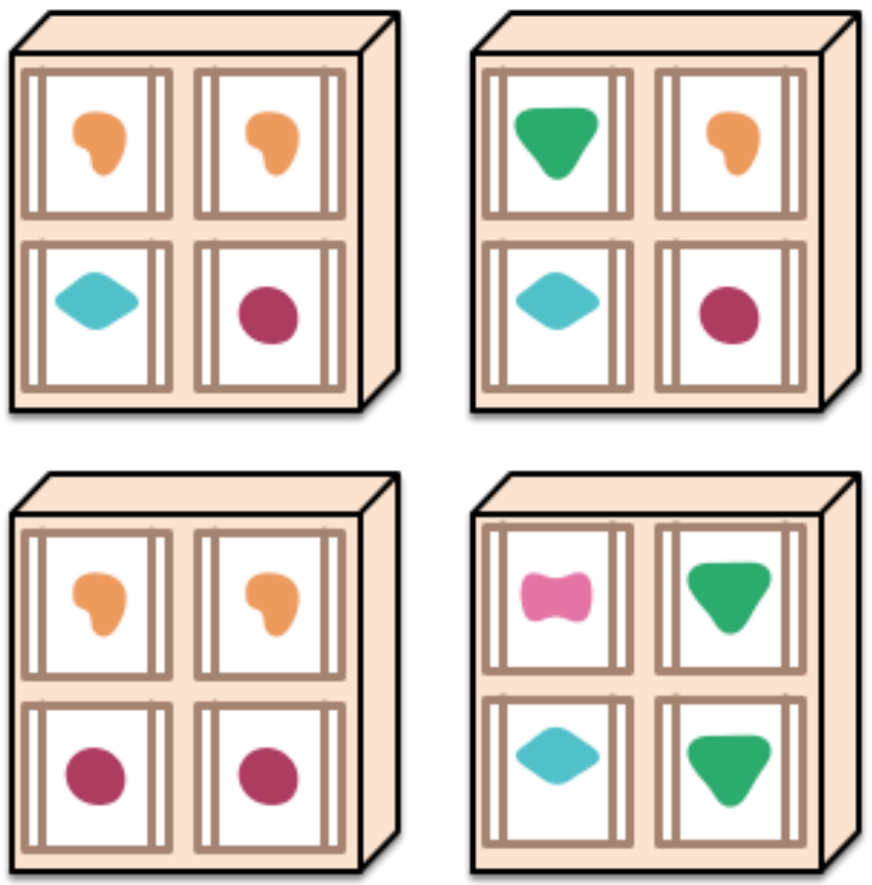
*A microservices architecture puts each element of functionality into a separate service...*



*... and scales by replicating the monolith on multiple servers*



*... and scales by distributing these services across servers, replicating as needed.*

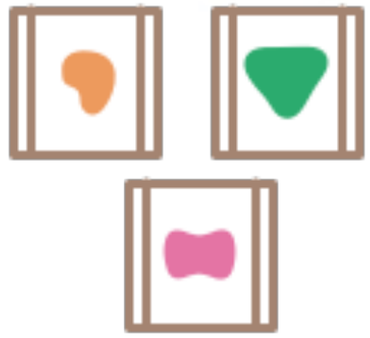


Read more: https://martinfowler.com/articles/microservices.html

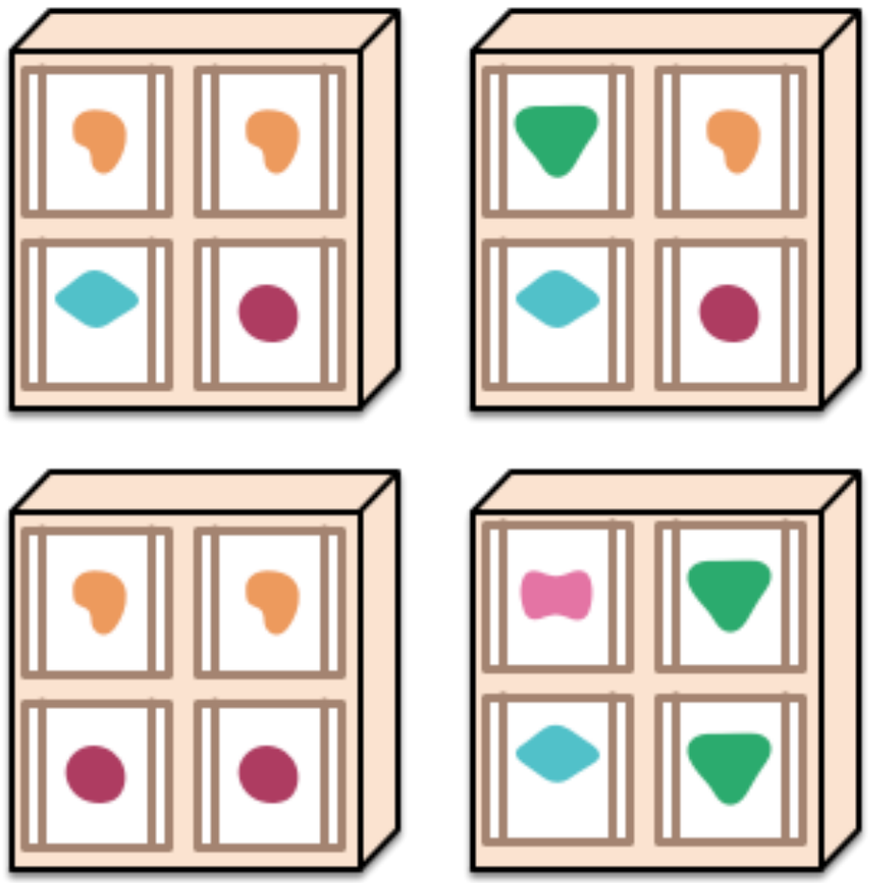
# Microservices

## Challenges with Microservices approach?

*A microservices architecture puts each element of functionality into a separate service...*



*... and scales by distributing these services across servers, replicating as needed.*



# Microservices Challenges

**Discovery**: how to find a service you want?

**Scalability**: how to replicate services for speed?

**Openness**: how to agree on a message protocol?

**Fault tolerance**: how to handle failed services?

All distributed systems face these challenges, microservices just increases the scale and diversity…

# Netflix

26th most popular website according to Alexa

Zero of their own servers

- All infrastructure is on AWS (2016-2018)
- Recently starting to build out their own Content Delivery Network

# Netflix

One of the first to really push microservices

- Known for their DevOps
- Fast paced, frequent updates, must always be available

700+ microservices

Deployed across 10,000s of VMs and containers

**Netflix ecosystem**

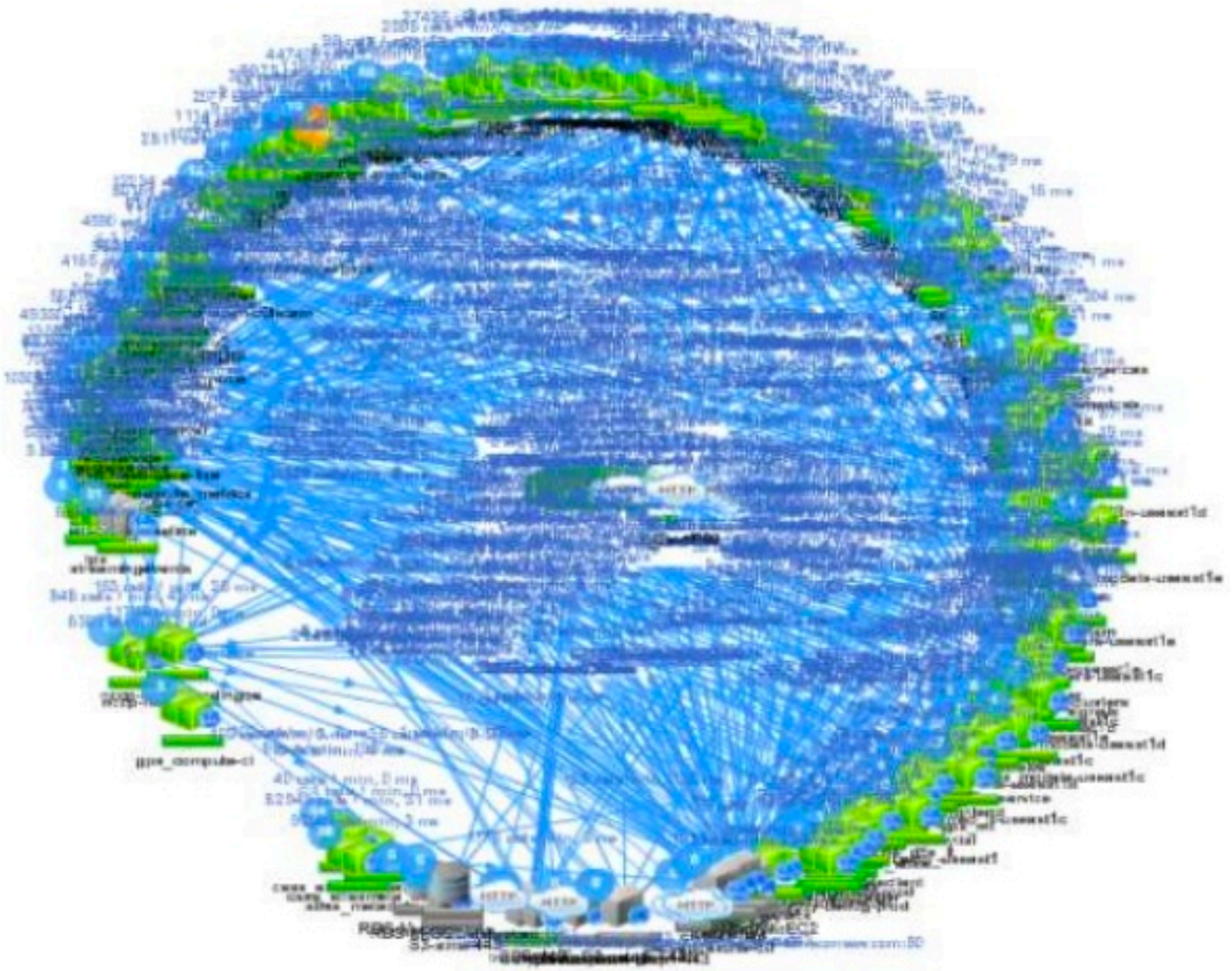
100s of microservices
1000s of daily production changes
10,000s of instances
100,000s of customer interactions per minute
1,000,000s of customers
1,000,000,000s of metrics
10,000,000,000 hours of streamed
**10s of operations engineers**

Netflix tech talk: https://www.youtube.com/watch?v=CZ3wIuvmHeM

# Netflix "Deathstar"

Microservice architecture results in a extremely distributed application

- Can be very difficult to manage and understand how it is working at scale



How to know if everything is working correctly?

# Netflix Chaos Monkey

Idea: If my system can handle failures, then I don't need to know exactly how all the pieces themselves interact!


Chaos Monkey:

- Randomly terminate VMs and containers in the production environment
- Ensure that the overall system keeps operating
- Run this 24/7

http://principlesofchaos.org/

Make failures the common case, not an unknown!

# Serverless Computing

Trendy architecture that improves the agility of microservices

What does “serverless” mean?

# Serverless Computing

Trendy architecture that improves the agility of microservices

What does “serverless” mean?

You still need a server!

BUT, your services will not always be running

Key idea: only instantiate a service when a user makes a request for that functionality

How will this work for stateful vs stateless services?

# Serverless Startup

AWS Lambda

- Define a stateless “function” to execute for each request
- A container will be instantiated to handle the first request
- The same container will be used until it times out or is killed

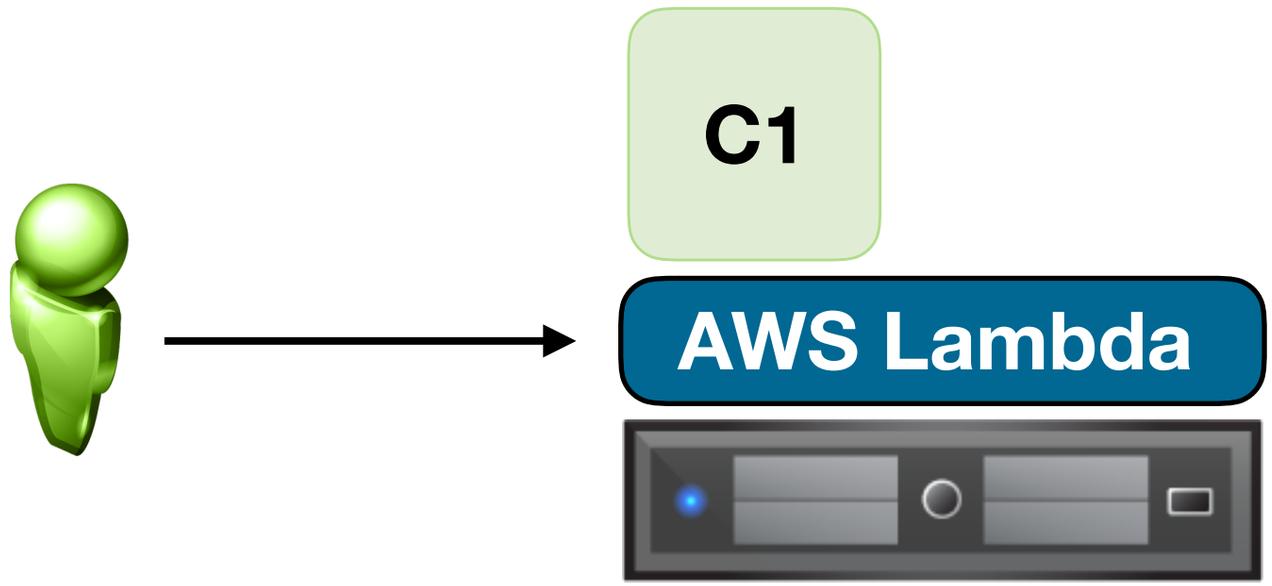
No workload means no resources being used!

# Serverless Startup

AWS Lambda

- Define a stateless “function” to execute for each request
- A container will be instantiated to handle the first request
- The same container will be used until it times out or is killed

Request arrives, start green container

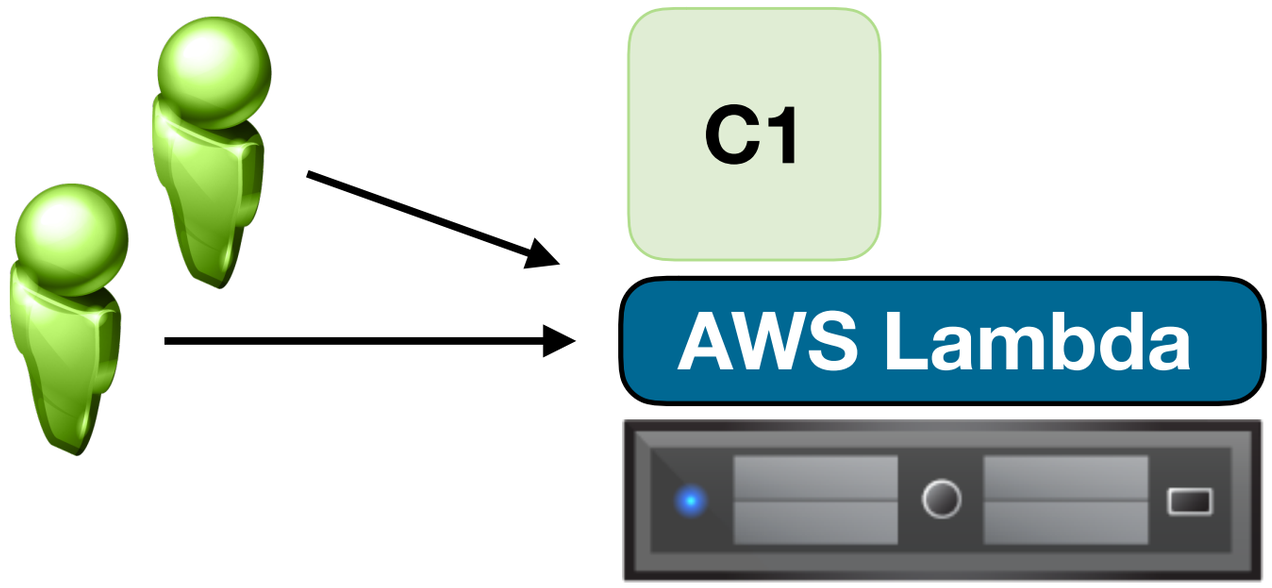


# Serverless Startup

AWS Lambda

- Define a stateless “function” to execute for each request
- A container will be instantiated to handle the first request
- The same container will be used until it times out or is killed

Reuse that container for subsequent requests

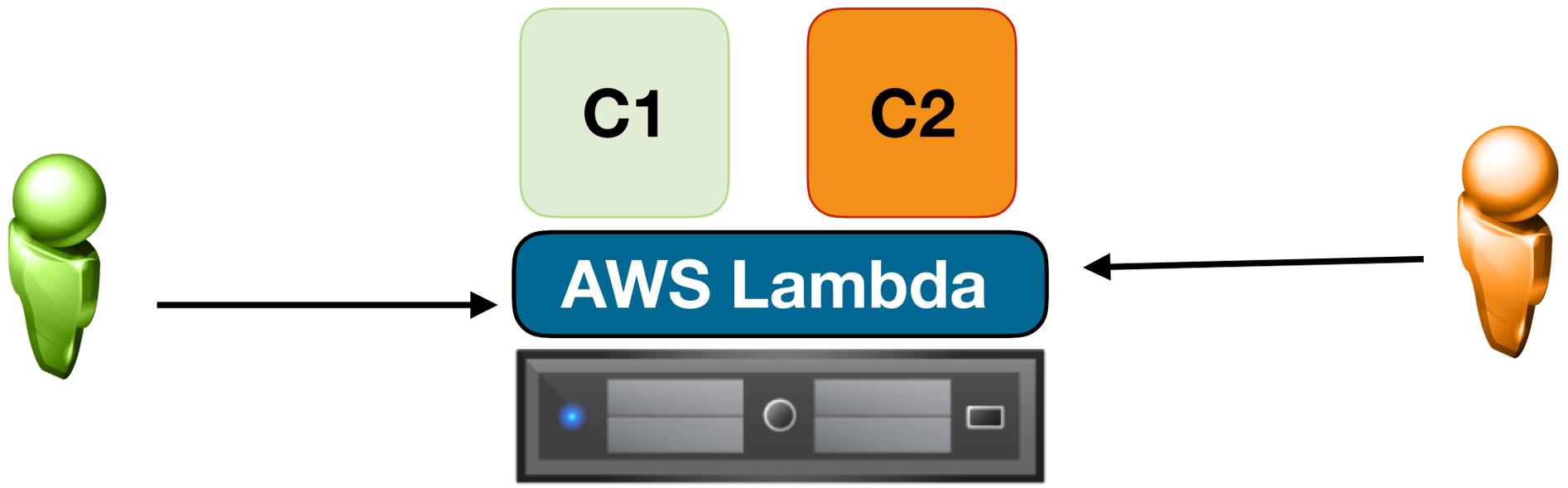


# Serverless Startup

AWS Lambda

- Define a stateless “function” to execute for each request
- A container will be instantiated to handle the first request
- The same container will be used until it times out or is killed

Start new container if user needs a different function

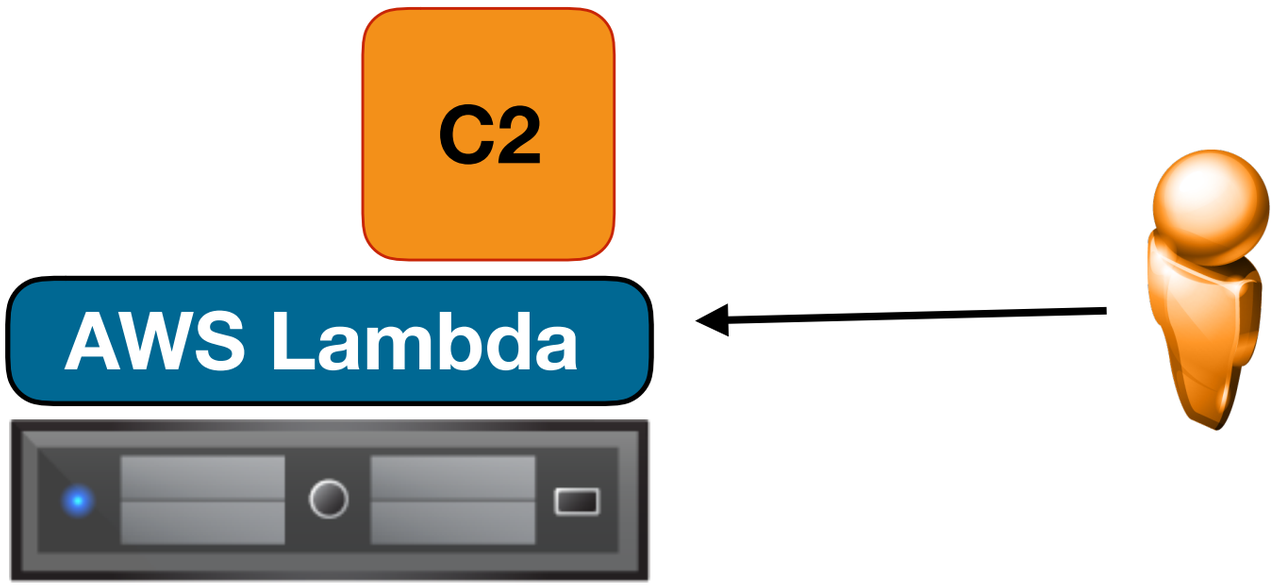


# Serverless Startup

AWS Lambda

- Define a stateless “function” to execute for each request
- A container will be instantiated to handle the first request
- The same container will be used until it times out or is killed

Clean up old containers once not in use



# Serverless Pros/Cons

Benefits:

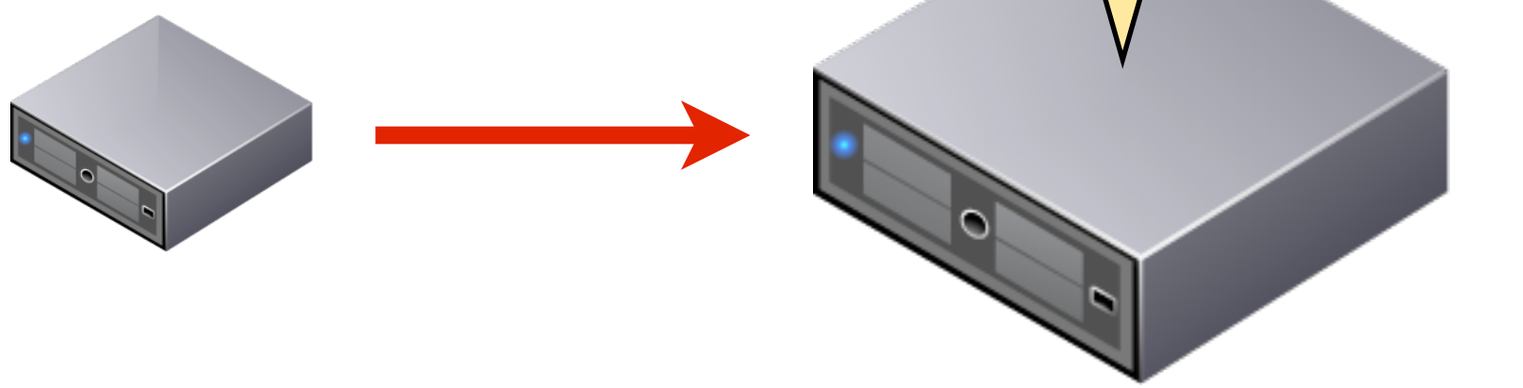
Drawbacks:

# Scaling

# Two ways to scale

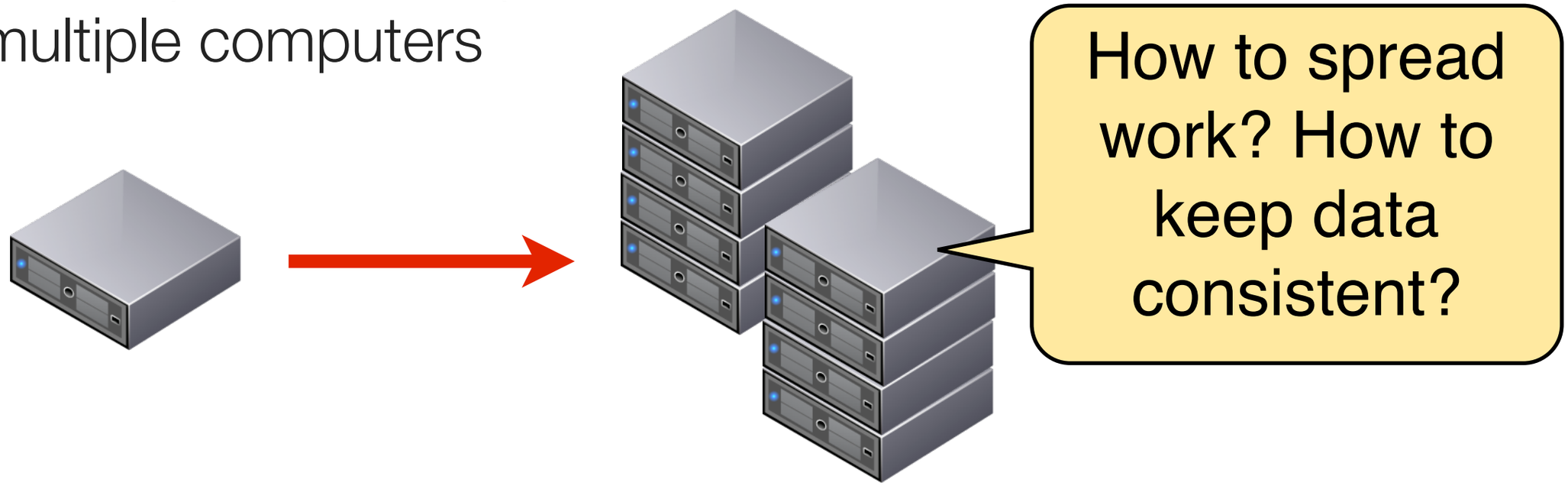
Scale UP (vertical)

- Buy a bigger computer

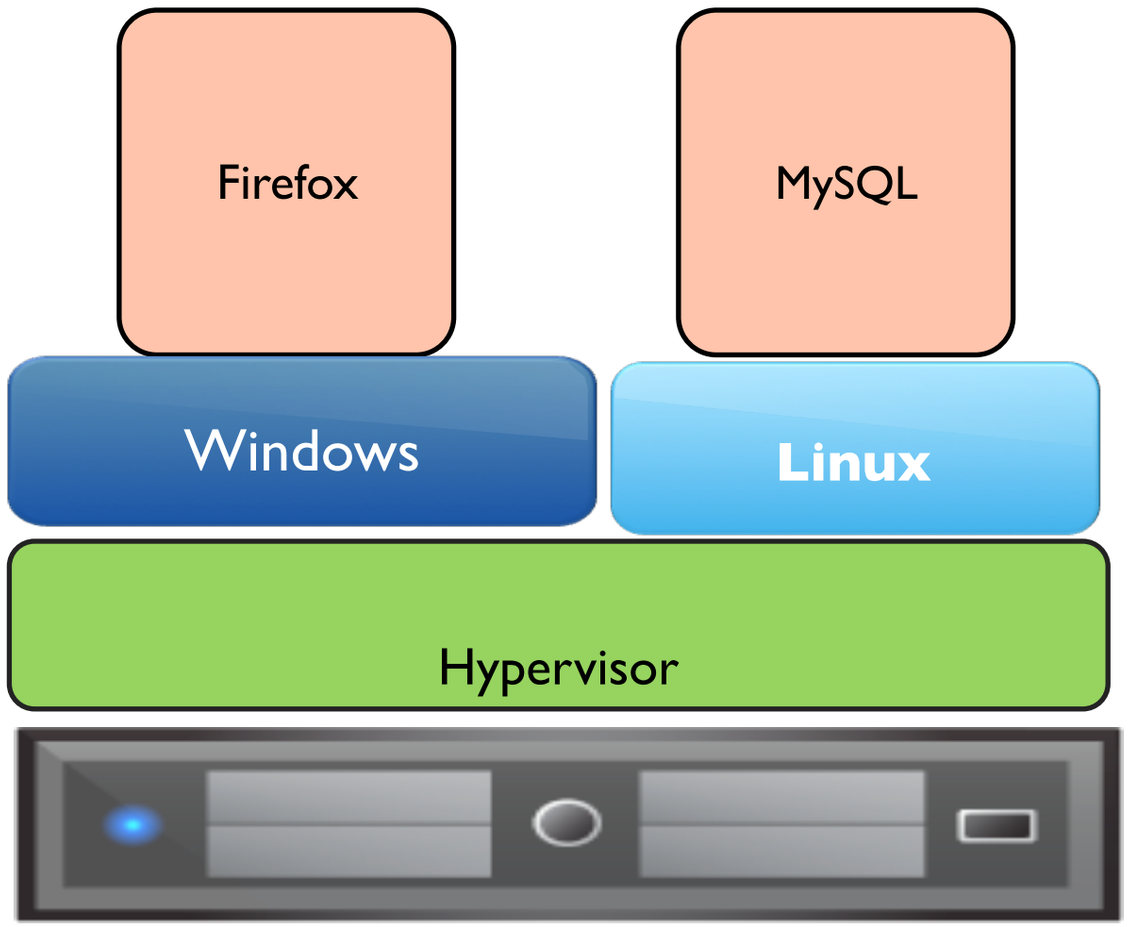



Scale OUT (horizontal)

- Buy multiple computers



# Does virtualization help?



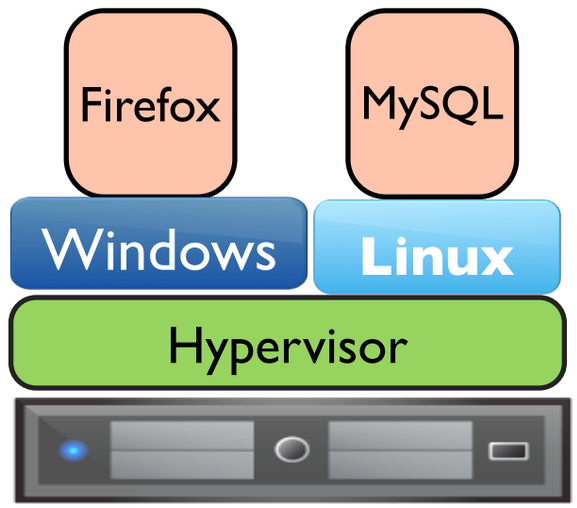
# Does virtualization help?

Not exactly...

Virtualization divides something big into smaller pieces

but still has features which can assist with scalability:

- Easy replication of VM images
- Dynamic resource management



# Replication

## Scale Out v1

# Biggest Challenge: Consistency

Replicating data makes it faster to access

**Computer science** or **computing science** (abbreviated **CS** or **compsci**) designates the scientific and mathematical approach in information technology and computing.

**Computer science** or **computing science** (abbreviated **CS** or **compsci**) designates the scientific and mathematical approach in information technology and computing.

**Computer science** or **computing science** (abbreviated **CS** or **compsci**) designates the scientific and mathematical approach in information technology and computing.

# Biggest Challenge: Consistency

Replicating data makes it faster to access

- But how to keep all copies of data consistent?

**Computer science** or **computing science** (abbreviated **CS** or **compsci**) designates the scientific and mathematical approach in information technology and computing.

**Computer science** is the art and science of doing awesome things with computers.

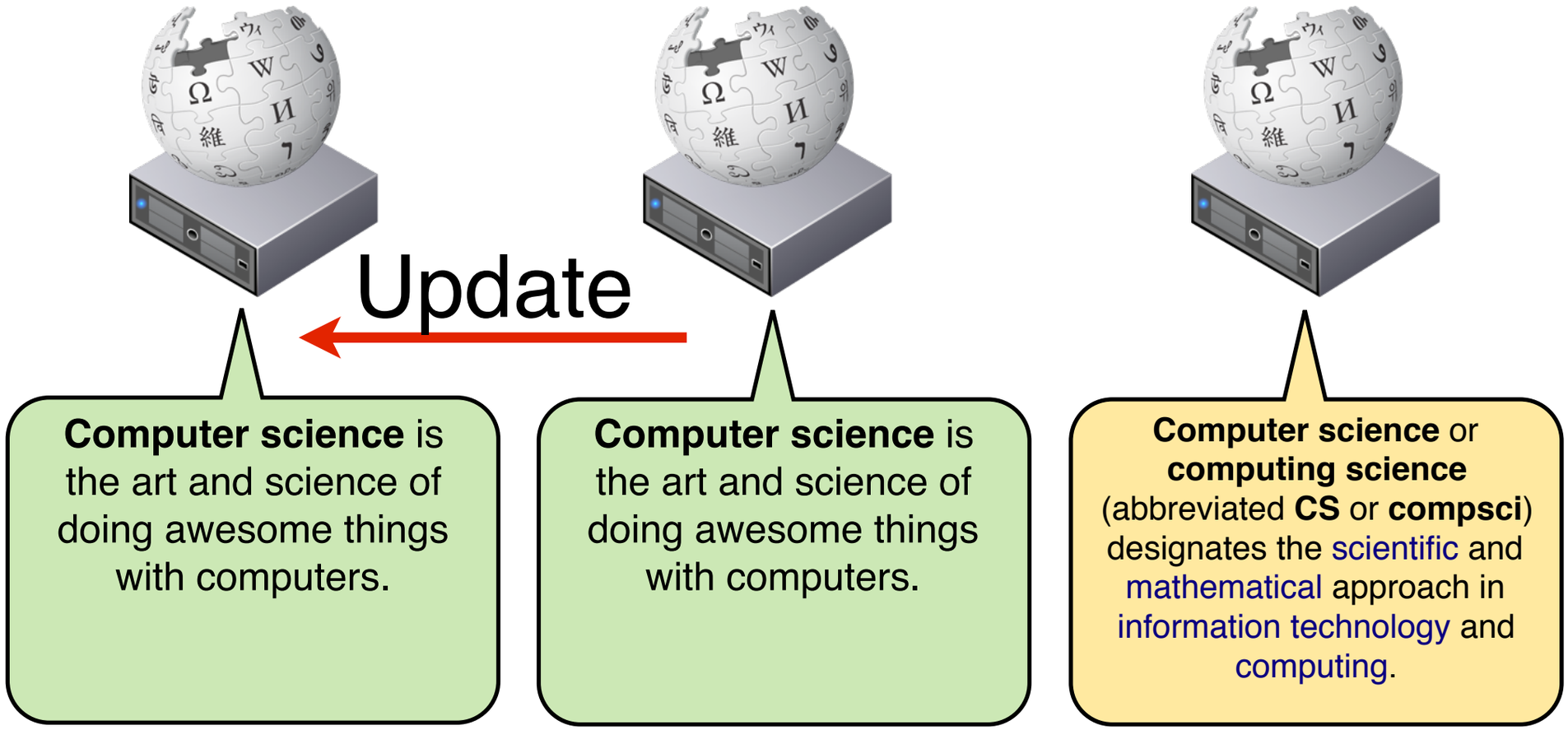
**Computer science** or **computing science** (abbreviated **CS** or **compsci**) designates the scientific and mathematical approach in information technology and computing.

Edit

# Biggest Challenge: Consistency

Replicating data makes it faster to access

- But how to keep all copies of data consistent?



# Biggest Challenge: Consistency

Replicating data makes it faster to access

- But how to keep all copies of data consistent?

**Computer science** is the art and science of doing awesome things with computers.

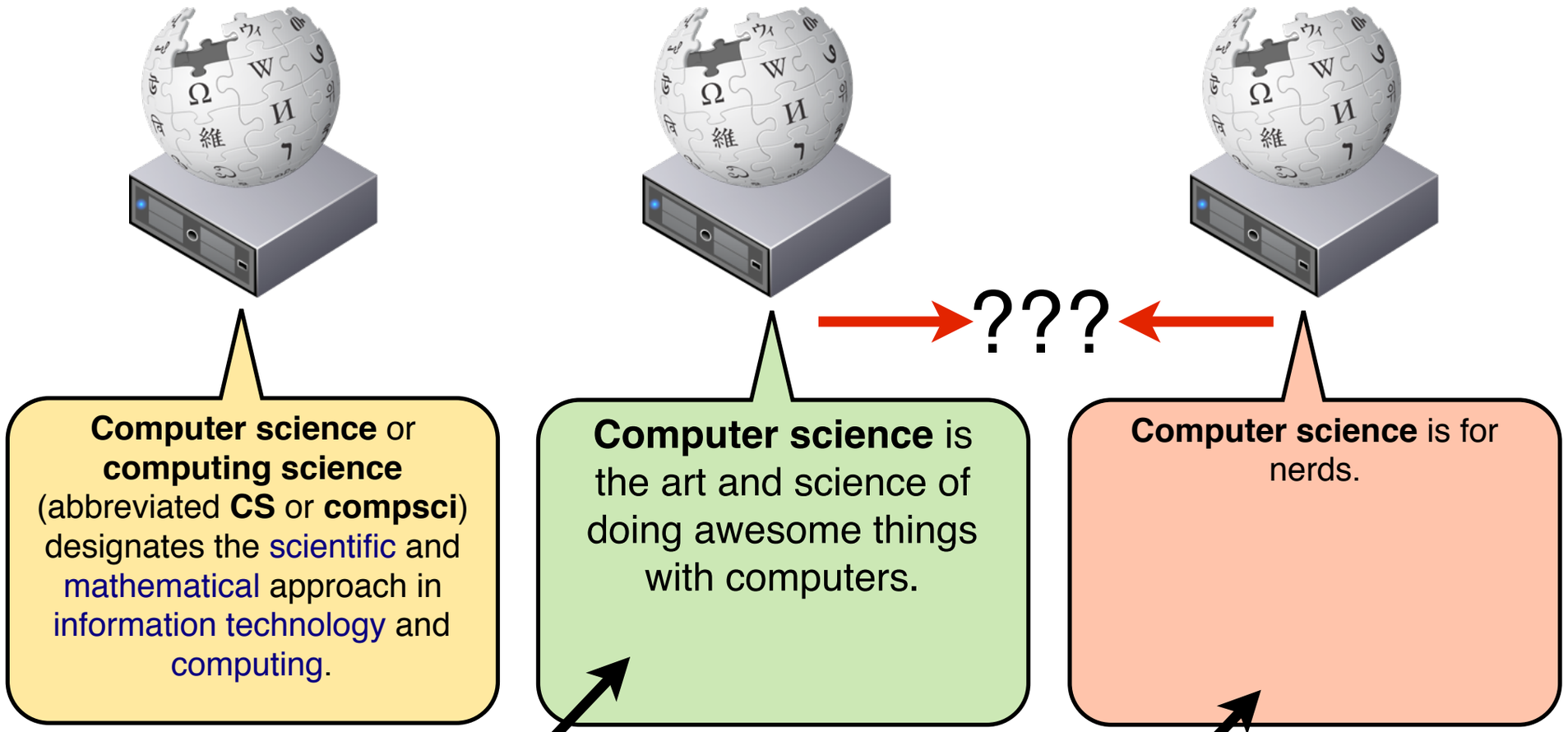
**Computer science** is the art and science of doing awesome things with computers.

**Computer science** or **computing science** (abbreviated **CS** or **compsci**) designates the scientific and mathematical approach in information technology and computing.


# Biggest Challenge: Consistency

Writes are even harder

- Would need time stamps or a consistent ordering
- Or, if writes are rare, just have a master coordinate



# Does it Matter?

A slightly out of date wikipedia page?

A post to your facebook profile?

1. Remove boss from friends list
2. Post "My boss is a moron, I want a new job!"

A change to a stock price in the NASDAQ exchange?

# Providing Consistency

We have already seen techniques that will help:

- Version vectors
- Distributed locking based on Lamport Clocks
- Election-based systems with a master/slave setup

There are many different types of **consistency**

- **Strict** - updates immediately available after a write
- **Sequential** - result of parallel updates needs to have the same effect as if they had been done sequentially
- **Causal** - updates that are casually related (e.g., where vector clocks can prove the -> relationship) are ordered sequentially, but others may not be

  … (several more) …
- **Eventual** - updates will converge so at some point reads to any replica will get the same result

# Partitioning

Scale Out v2