Announcements

- No questions asked ***two penalty free late days*** for P2 final and P3 (CHKPT and final)
  - You can use the late days for any reason: medical issues, stress, other projects, ...
  - No need to email us the reason – we trust you!
  - **No additional late days will be granted (unless a major emergency)**

- Submit P3 project partner form by tomorrow, Friday 11/12

15-440 Distributed Systems

Scaling Techniques

Scaling architectures

How to scale?

Assume: we have reached the limit of optimizing

(protocol design, caching, ...)

Still need to scale as clients grow.

How to add more resources?

Two fundamental approaches:

Scale Up (aka "vertical scaling")	Scale Out (aka "horizontal scaling")
add resources to a single node in the distributed system (e.g. more and faster CPUs/GPUs, more memory, more disks)	**add more nodes** to the distributed system

Scale Up or Scale Out?

Scale Up or Scale Out?

Scale Up
(aka "vertical scaling")

- **no application changes**
  huge win in terms of cost and time
  → IBM mainframes still a viable business
- **no new failure modes, latency concerns between nodes, etc.**
- **typically more expensive**
  bigger profit margins for IBM
- **hits limits sooner**

Scale Out
(aka "horizontal scaling")

- **application has to conform to scale out design**
  may involve total rewrite of application
- **more complex failure modes, latency concerns across nodes**
- **Scales better. Large Internet-based companies (e.g., Google, Facebook, Microsoft, Amazon,...) have been champions of this approach**

When to Scale Out?

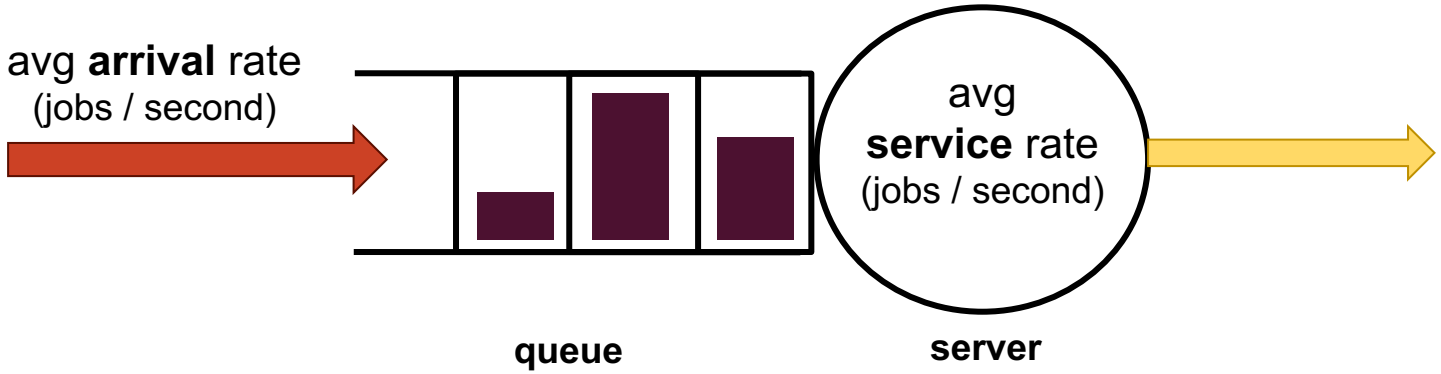
- How do we decide when a new node should be created?
- What happens on overload?

Response times (i.e., latency) get worse

Often very nonlinear

Latency Aspects of "Scale Up vs Scale Out"

Queueing time: a critical component of latency

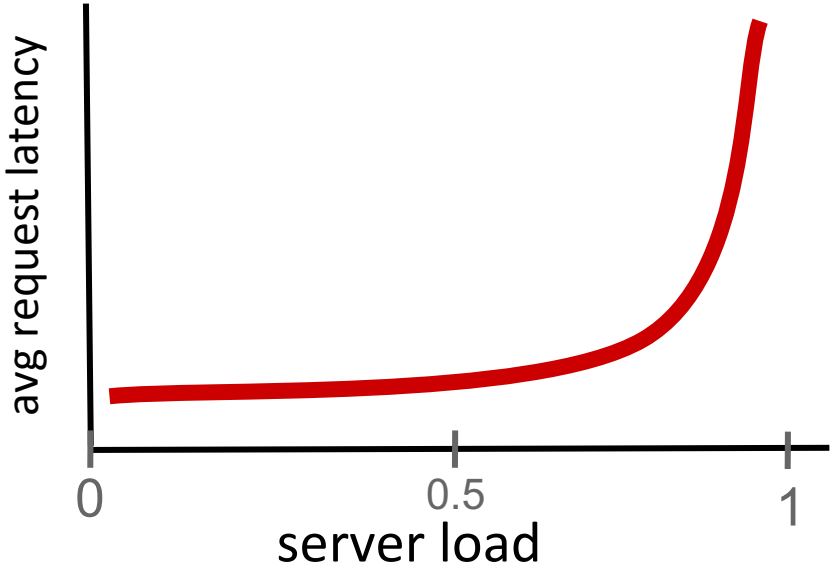


What is the avg server load (= fraction of time server is busy?)

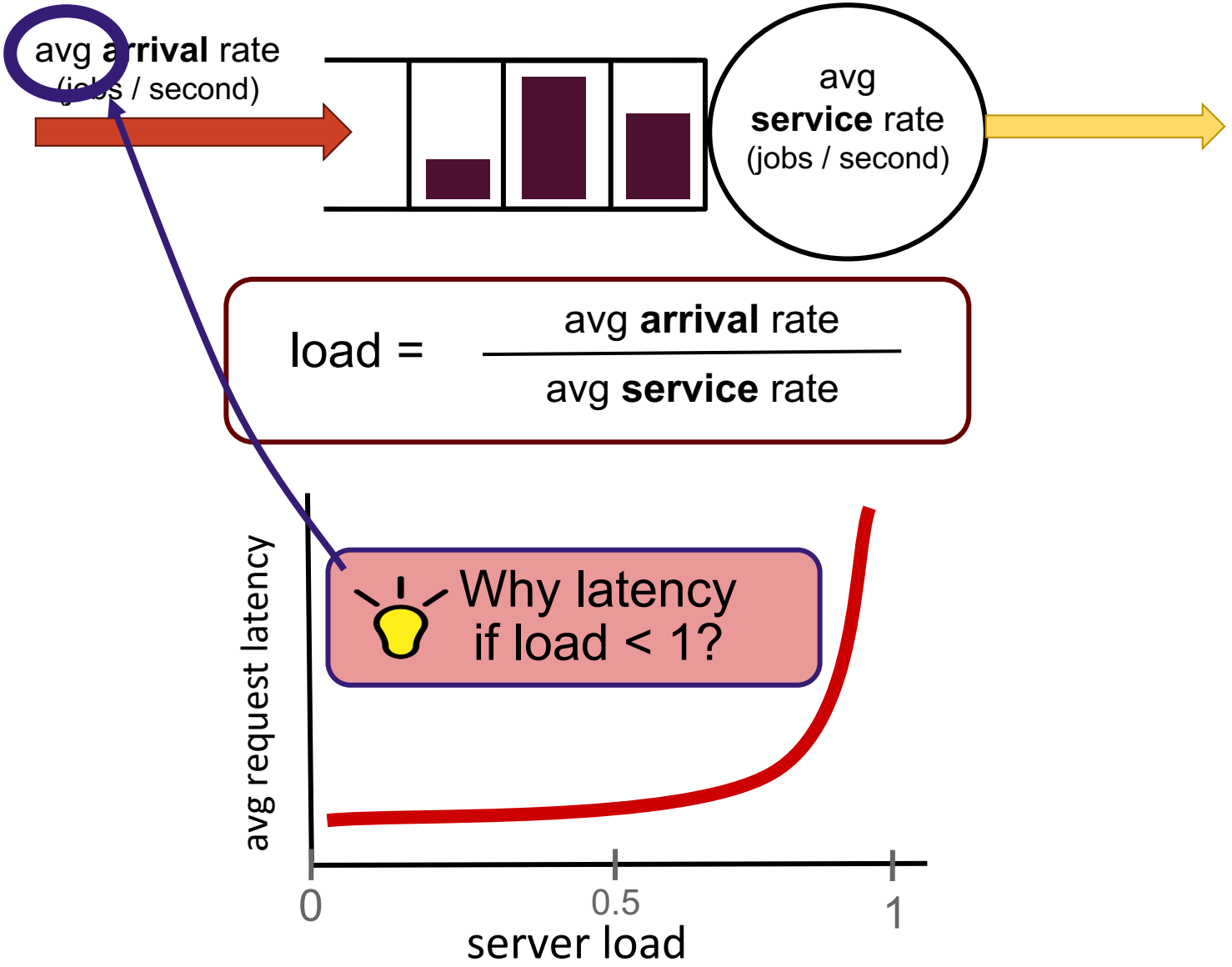
$$\text{load} = \frac{\text{avg arrival rate}}{\text{avg service rate}}$$

How Does Load Affect Latency?

$$\text{load} = \frac{\text{avg arrival rate}}{\text{avg service rate}}$$

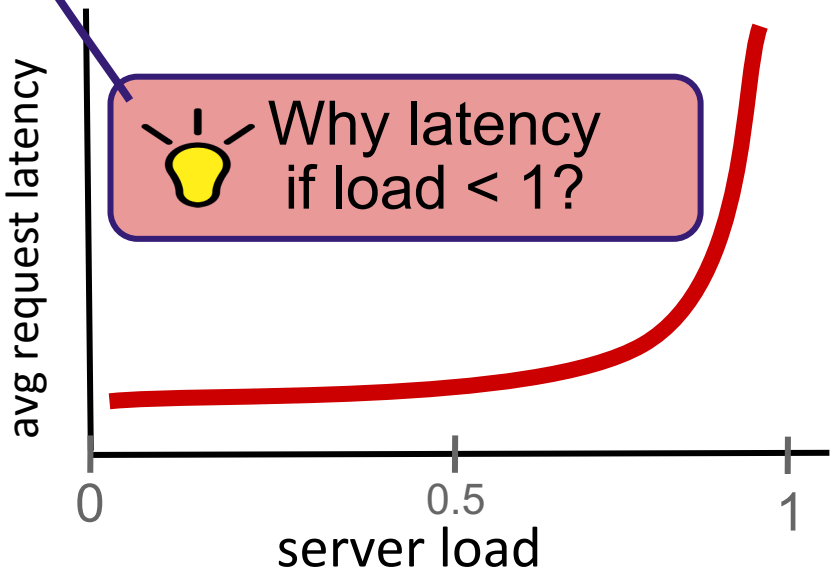


How Does Load Affect Latency?

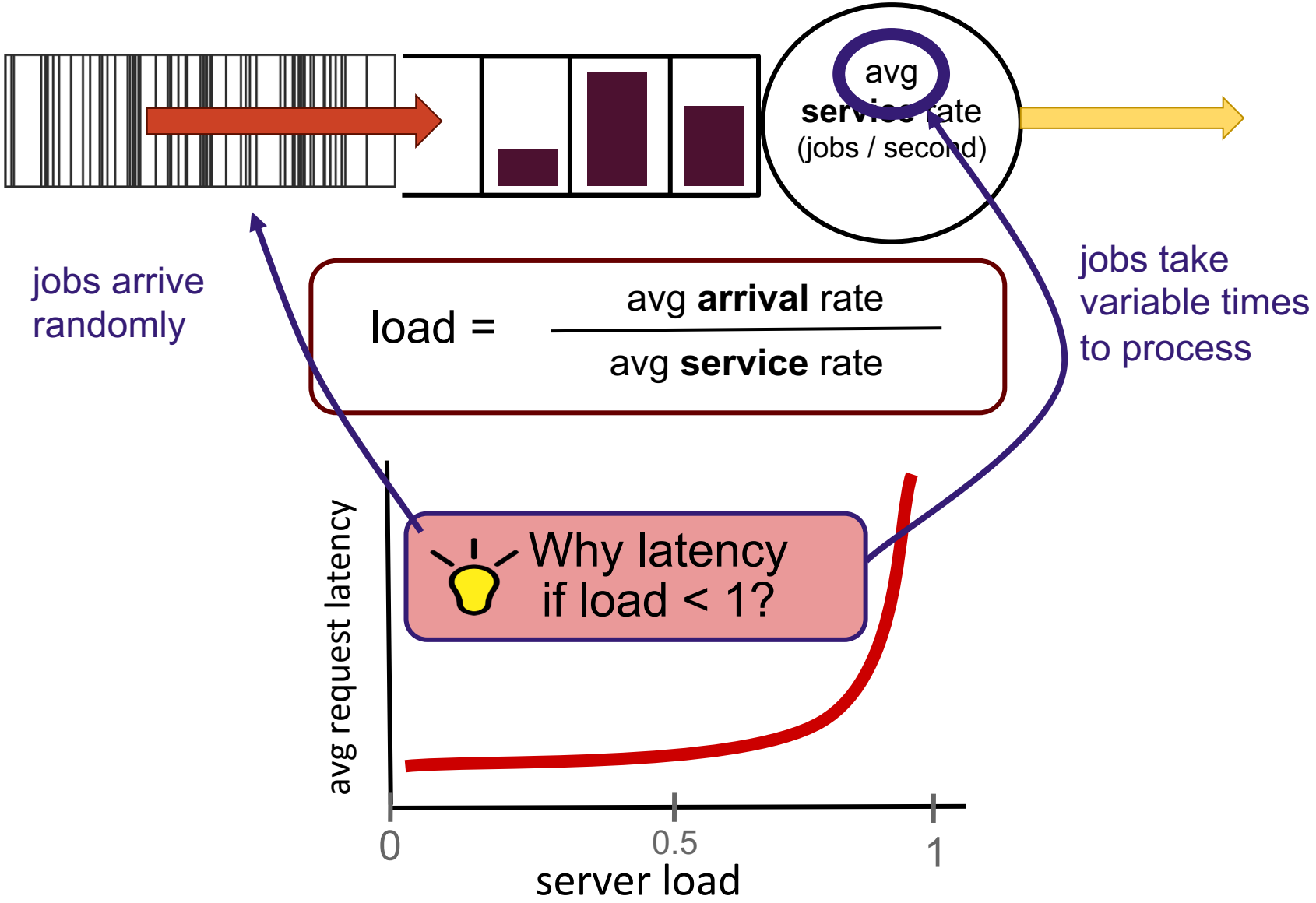


How Does Load Affect Latency?

$$\text{load} = \frac{\text{avg } \textbf{arrival} \text{ rate}}{\text{avg } \textbf{service} \text{ rate}}$$



How Does Load Affect Latency?

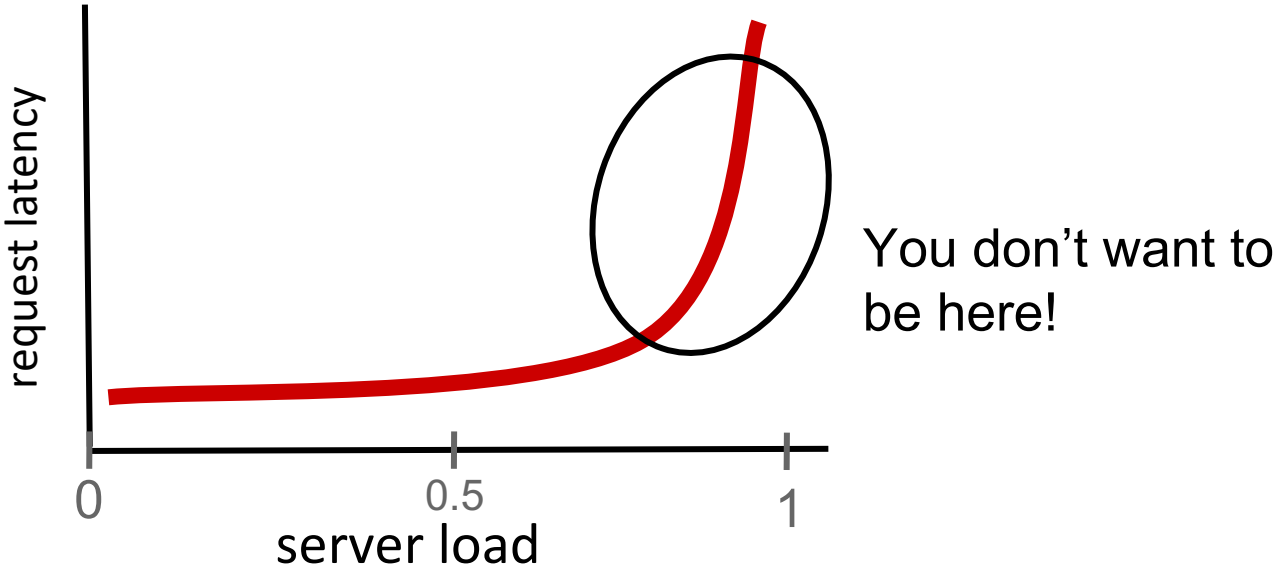


When to Scale Out?

- How do we decide when a new node should be created?
- What happens on overload?

Response times (i.e., latency) get worse

Often very nonlinear



When to Scale Out?

- Each node incurs some cost
  - CPU and memory overhead for VMs, processes and threads
  - overprovisioning can be expensive
  - billing can be excessive from cloud services like AWS

- Timing of scaling is crucial
  - Scaling out too late $\rightarrow$ long period of suboptimal response time
  - Scaling out too soon $\rightarrow$ greater overhead and underutilized resources

- How do we find the sweet spot?

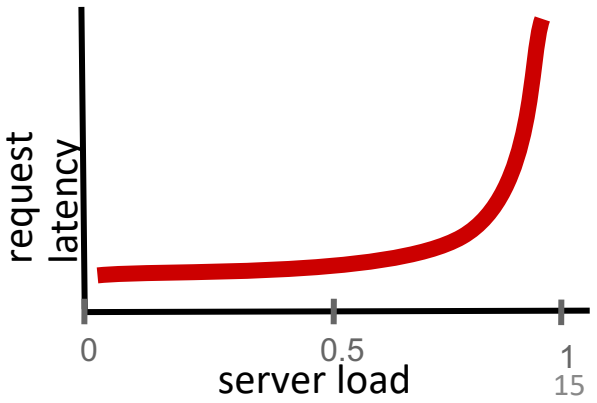
When to Scale Out?

A good heuristic: *queue length*

- As a server queue builds up, use a threshold to trigger scale out
- When load drops (e.g. empty queues for nodes) shrink scale
  - empty queues are the signal that shrinking might be in order
  - brief transients can muddy the picture

- *Hyteresis* essential to avoid wasteful oscillations
  - significant gap between upper and lower thresholds
  - size of gap determines extent of hysteresis

- Caution: often queue length also insufficient
  - Might have to resort to some degree of overprovisioning



How to Scale Out?

- load balancing front end typically does redirection

- how to partition work across nodes?
  - random assignment is one possibility
  - static partitioning of workspace (e.g. low order bits of user id) is another
  - content-based approaches also possible (one node handles A-F, another G-K, etc.)

Simple Case Study

Simple Web Service

Let's start with a simple web application

How might one scale it ?

This is just an illustrative example of how systems can be evolved to larger scales

Many other ways to do this as well

Starting point

Simple website for sharing pictures

Used by a few of your friends

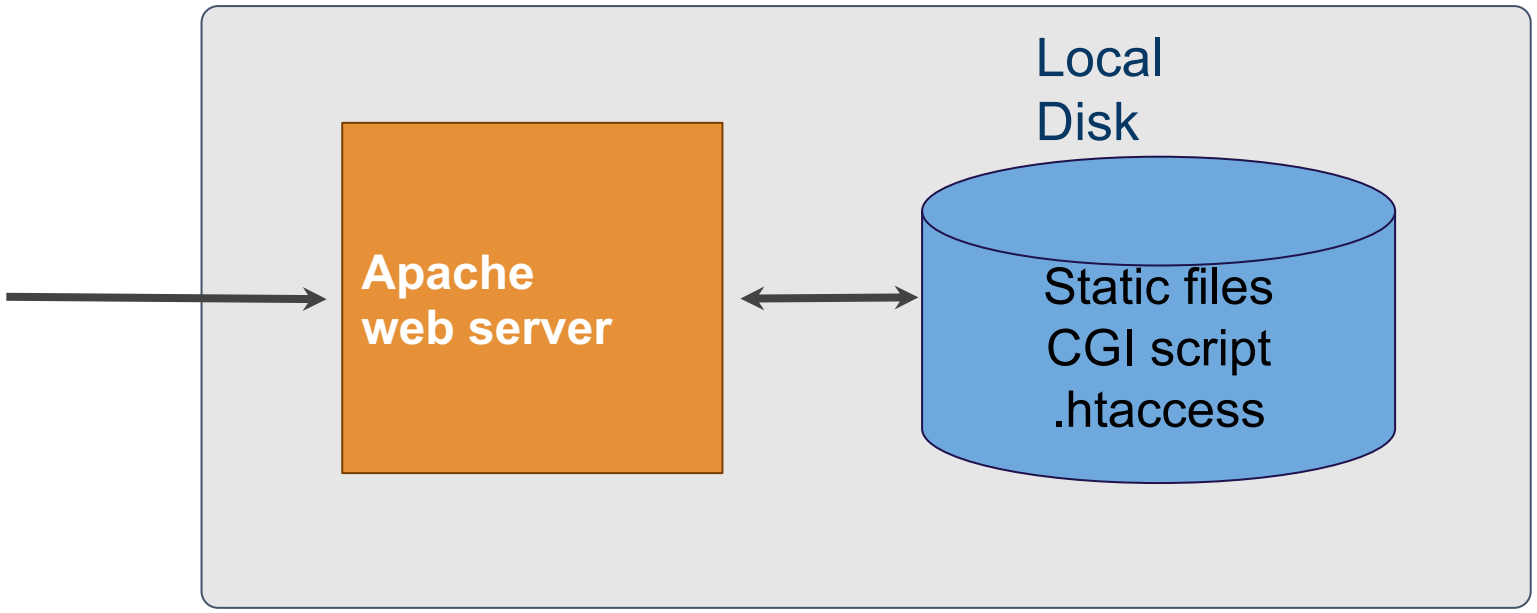
Simple implementation

- a few static pages
- a couple of CGI scripts for custom content
- .htaccess file for users/passwords
- simple Apache webserver

Running on small server

E.g., AWS EC2 “micro” instance, GCD “f1-micro” instance

Small Website



Issues with simple website

What if you included photos …

of your kitten?

How scalable is your website?

- Content
  - Easy to add user content, but limited storage
- Administrative
  - Painful to add, maintain user accounts
- Load
  - Limited to a fraction of a machine

Scaling to the next level

Want to scale to 10s - 100s of users

Use a database to store user accounts, settings

2-tier structure:

- Front end: web server, scripts
- Back end: DB, storage server

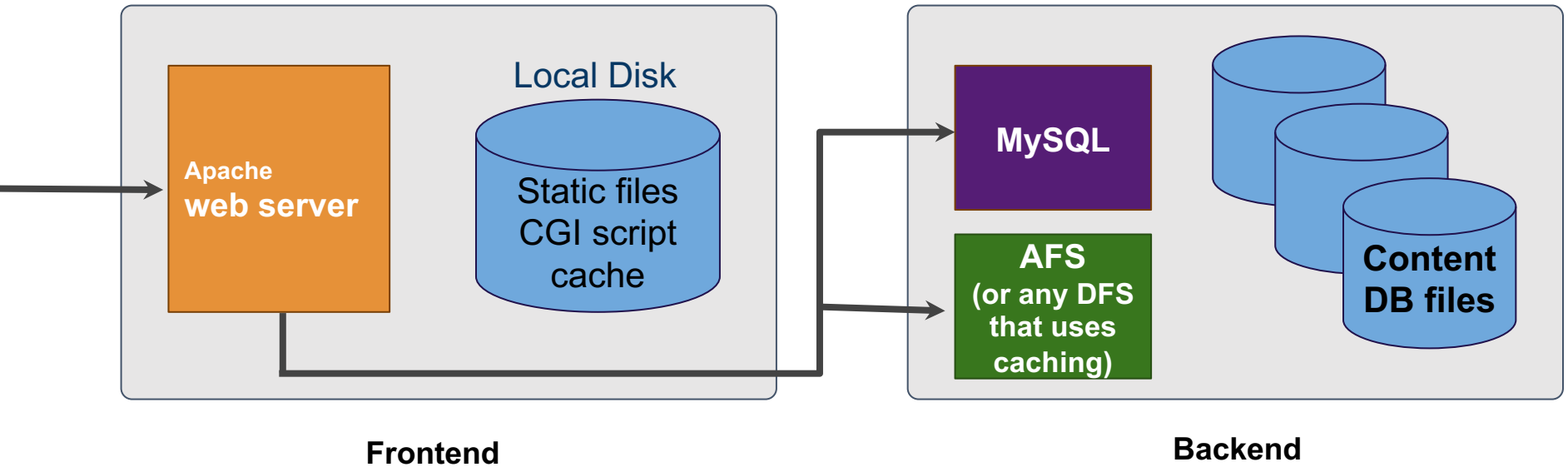
2-tier web site

DB improves administrative scaling

More storage for content scalability

Scaling up

- Frontend - more cores, RAM for running CGI
- Backend - more disk for storing content



Growing bigger

Webserver likely to be bottleneck

- Dynamic content will consume cpu cycles

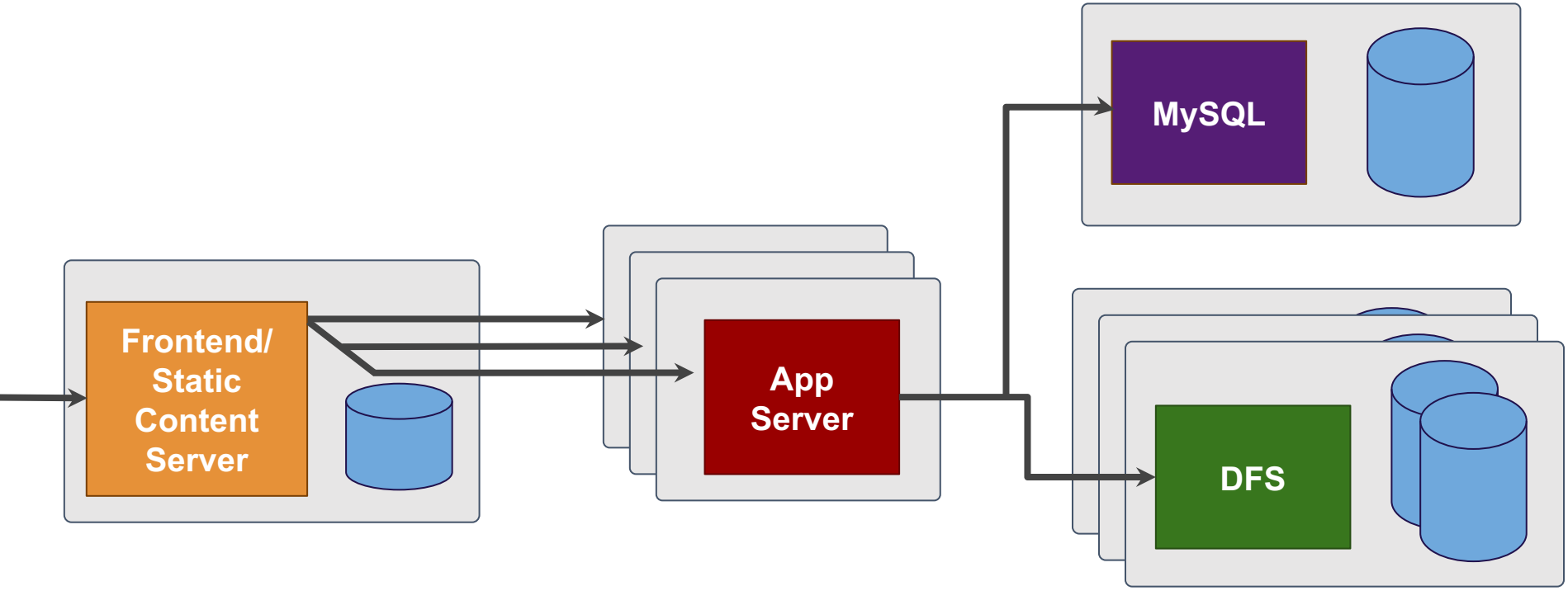
Need to scale out

- Introduce middle tier for application servers
- Scale out storage

3-tier web site

Great for scaling up features

Can add lots of application processing power, storage for content



Scaling out the front end

Frontend servers likely to be bottleneck

- Connection termination
- TLS termination
- Open to probing from the Internet

Need multiple frontend webservers

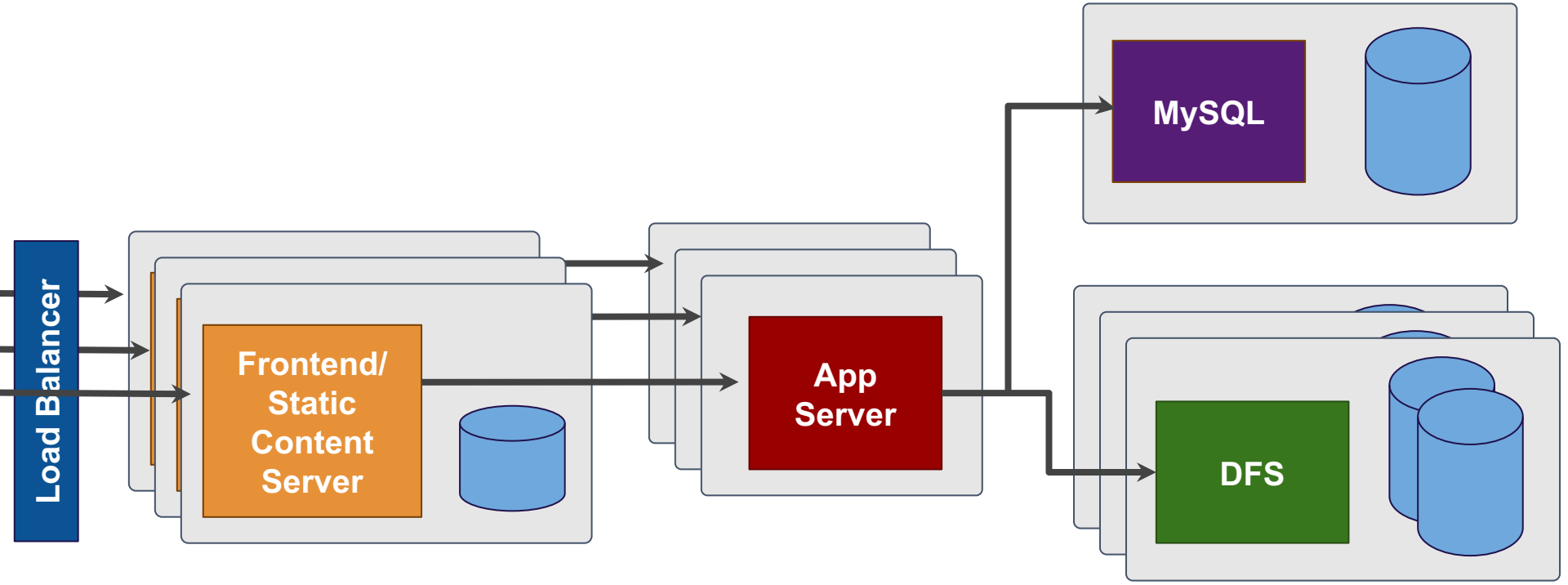
How to direct traffic to the right ones?

Solution: load-balancing switch

- "Layer 7" – application layer in network stack
- HW that understands http, sessions,...
- Map Single IP maps to multiple servers

Large web site

Now, can handle 1000's of concurrent users



What bottlenecks remain?

What to do about DB?

Databases excel at ensuring correctness during concurrent operations, maintaining persistent state

Databases notoriously difficult to scale out

- Critical transactional operations work best on single beefy machine

However, not all data / operations need such stringent consistency semantics

Option 1: Save DB for critical things (e.g., money), something more scalable for rest

Option 2: Use DB as master store, but have some form of cache in front of it

Alternative data stores

Key-value stores – provide simple interface for storing key-value pairs

Memcache – RAM-only storage layer, used as a cache for DB or disk-based KV store

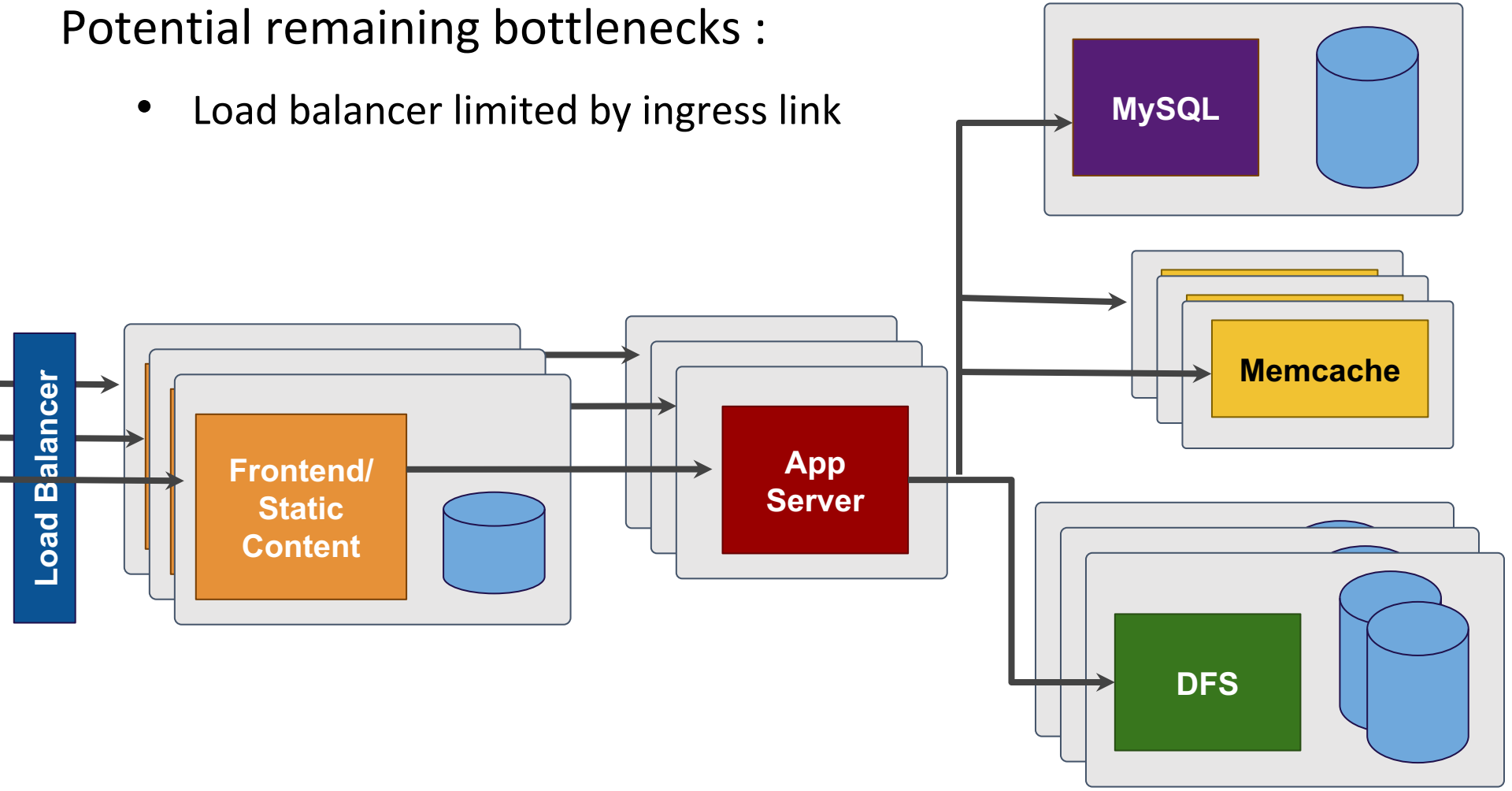
In-memory DBs – sacrifice durability for performance

Highly-scaled web service

All components scaled out

Potential remaining bottlenecks :

- Load balancer limited by ingress link



Can we go bigger?

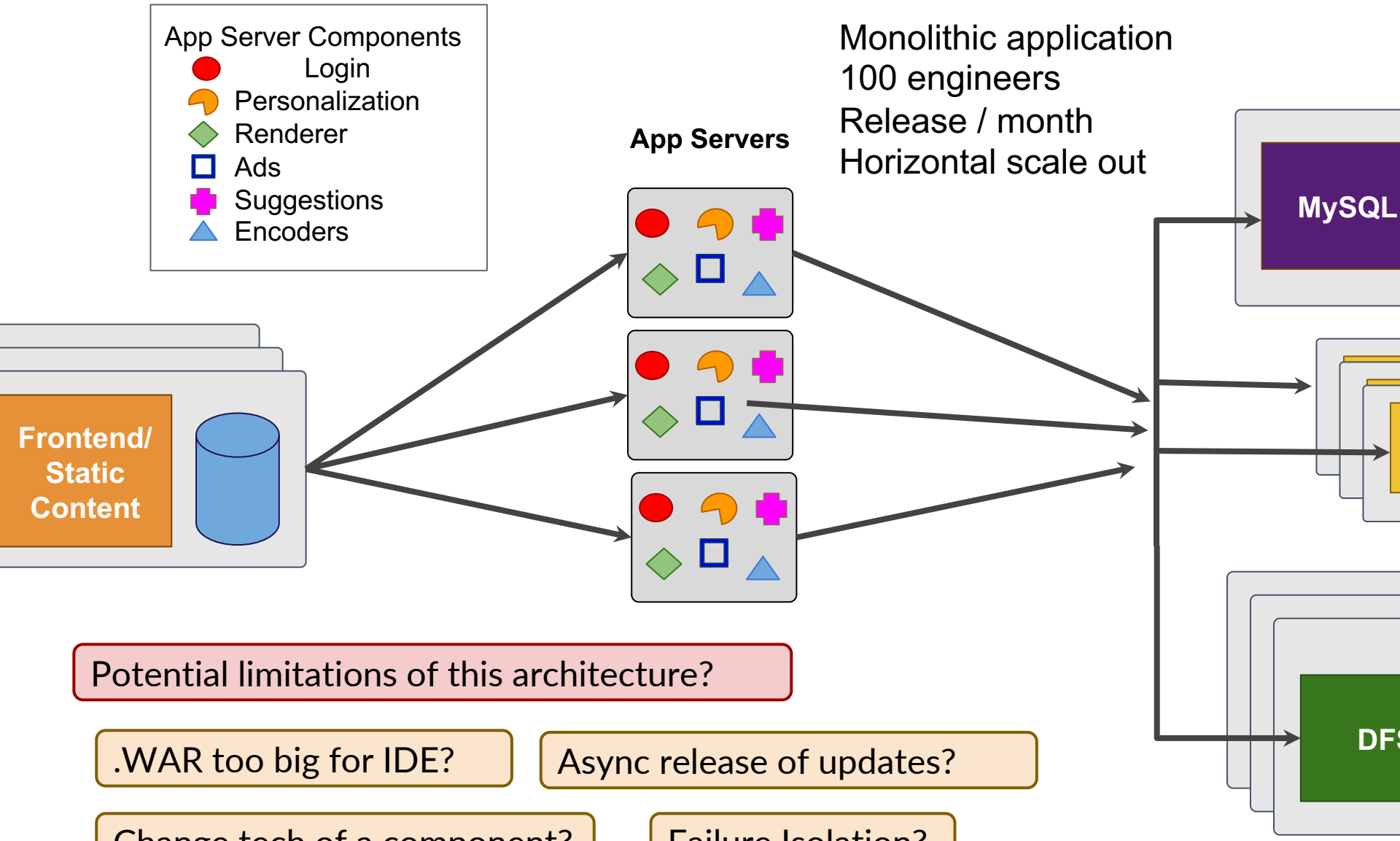
Georeplication

- Deploy to multiple sites around the globe
- Each site is a large-scale web service
- How do we direct users to the right site?
  - DNS tricks – mysite.com resolves to different IP addresses depending on where you are
  - Can be randomized to help with load balancing

Introduces more challenges: Data consistency across sites

- Can get close to one-copy semantics at a single site
- How long does it take to propagate changes globally?
- What if users move?

Monolithic Architecture



Potential limitations of this architecture?

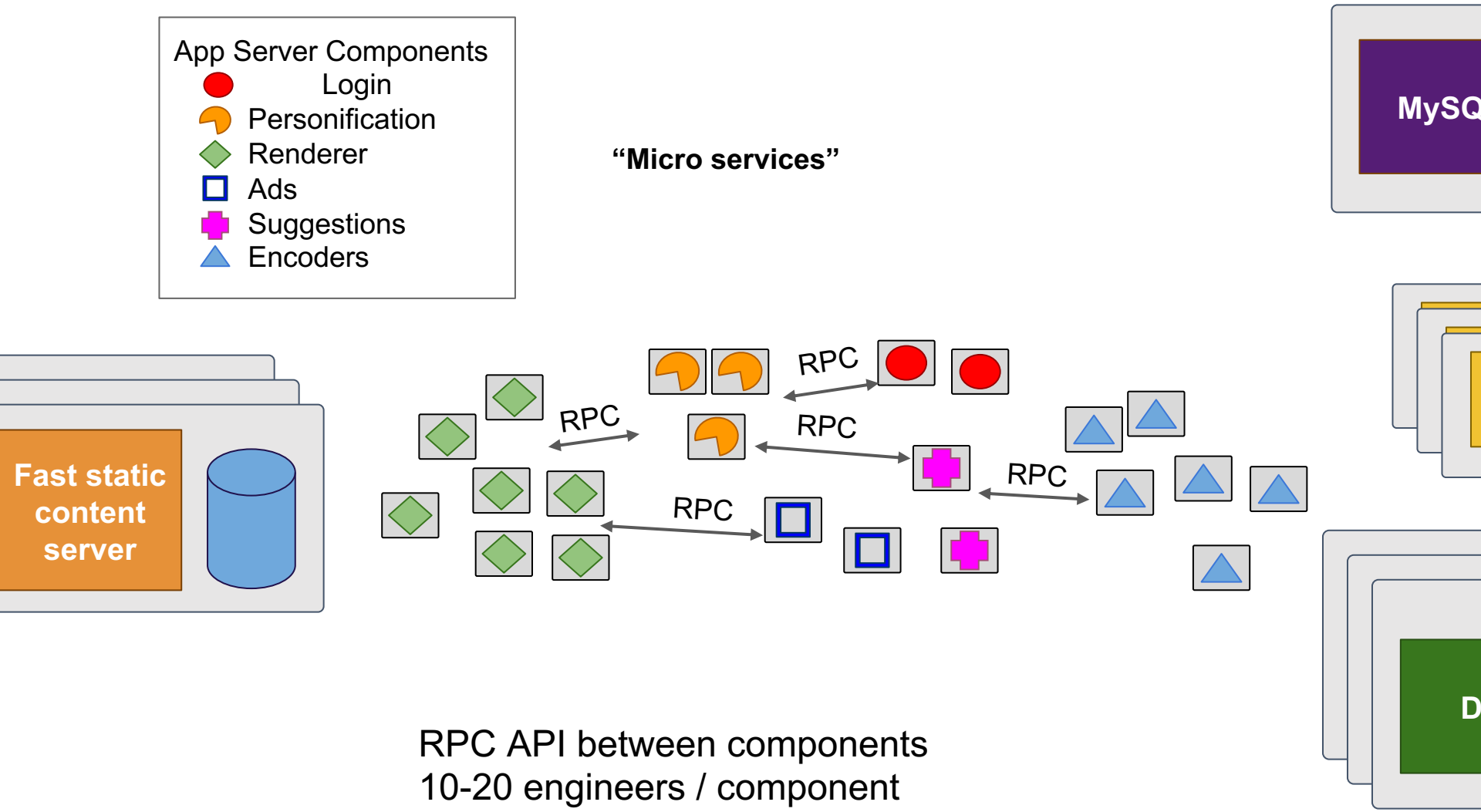
.WAR too big for IDE?

Async release of updates?

Change tech of a component?

Failure Isolation?

Micro-Service Architecture



RPC API between components
10-20 engineers / component
Components release and scale independently

Reliability?

Lots of components → What happens to reliability?

- Failures more likely!

However, most components can be stateless

- Simply restart any that fail

Storage layer: Filesystem uses redundancy to protect against failures

DB: Replicated DB with “hot spare”; logging