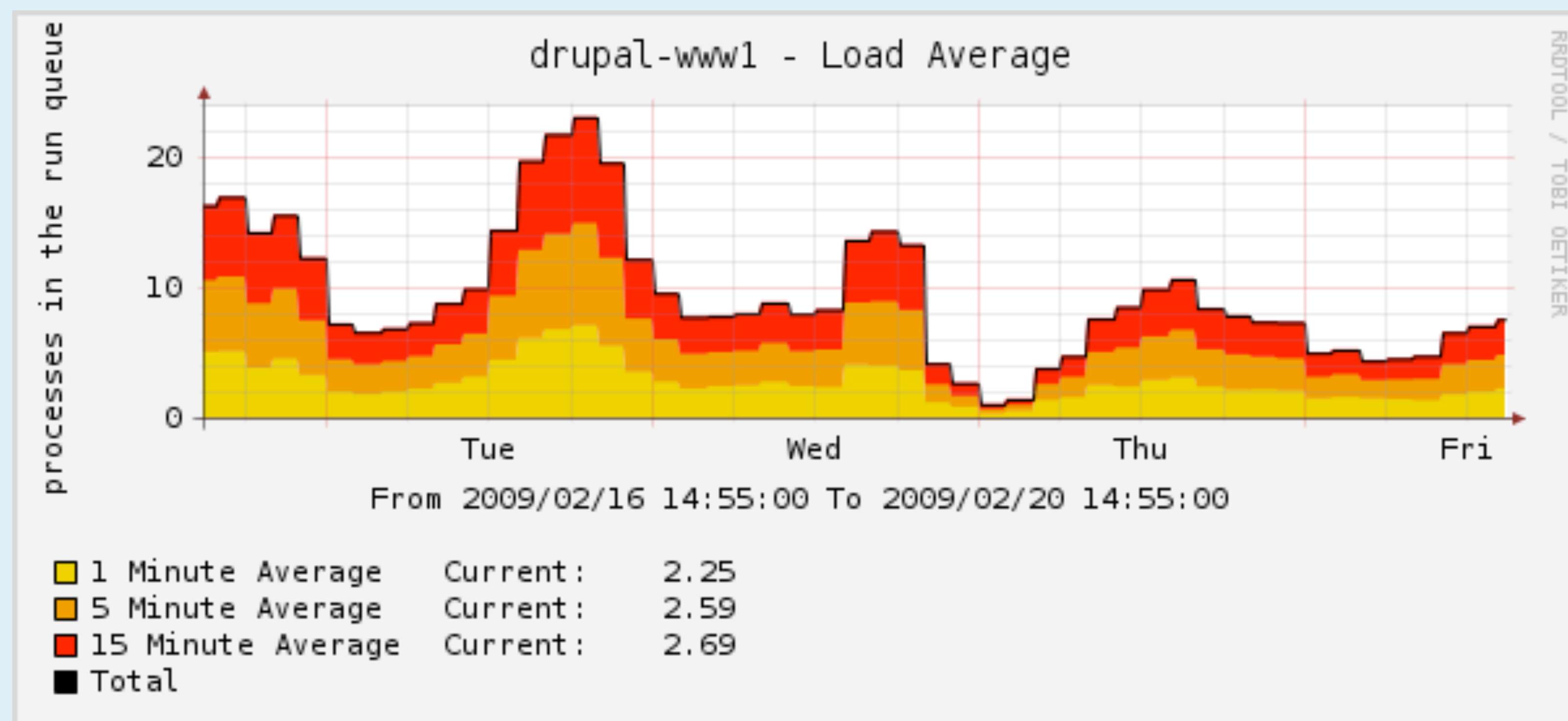
Building Infrastructure You Can Scale, Monitor, and Maintain

This is not a presentation about queries, indexes, table engines, or my other standard fare.

Designing Physical Architecture

Predicting peak traffic

Traffic over the day can be highly irregular. To plan for peak loads, design as if all traffic were as heavy as the peak hour of load in a typical month -- and then plan for some growth.



Analyzing hit distribution

100%
Static Content
30%
Dynamic Pages
70%
Anonymous
50%
Human
40%
Web Crawler
10%
No Special Treatment
3%
"Pay Wall" Bypass
7%
Authenticated
20%

Throughput vs. Delivery Methods

	Green (Static)	Yellow (Dynamic, Cacheable)	Red (Dynamic)
Content Delivery Network	●●●●●●●●●●●	✖ [2]	✖
Reverse Proxy Cache	●●●●●●●● 1000 req/s	●●●●●●●●	✖
Drupal + Page Cache + memcached	●●● [1]	●●●	✖
Drupal + Page Cache	●●● [1]	●●	✖
Drupal	●●● [1]	●	● 10 req/s

More dots = More throughput

[1] Delivered by Apache without Drupal
[2] Some actually can do this.

Objective

Deliver hits using the fastest, most scalable method available.

Layering: Less Traffic at Each Step

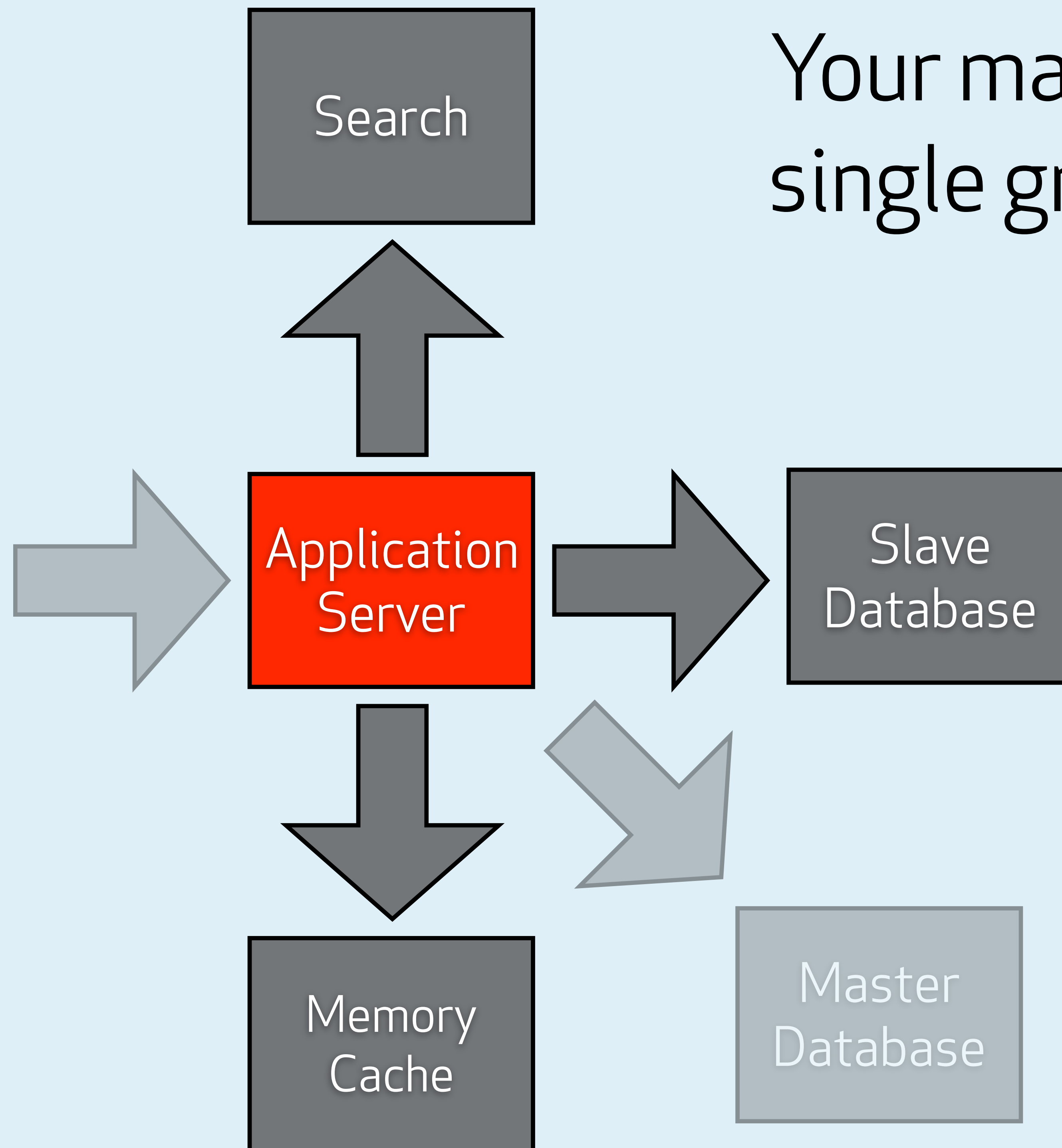
Your Datacenter

Traffic → CDN

Traffic → Load Balancer → Reverse Proxy Cache → Application Server → Database

DNS Round Robin

Offload from the master database



Your master database is the single greatest limitation on scalability.

Tools to use

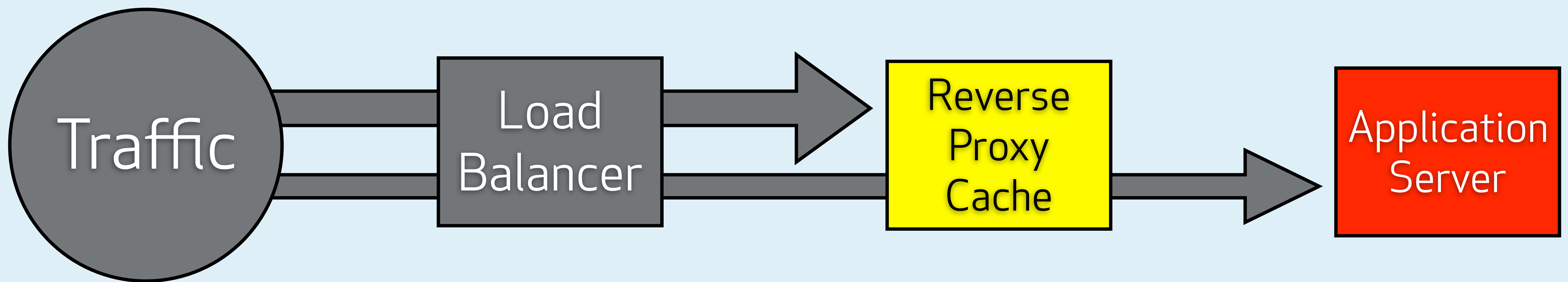
Apache Solr for search. (Acquia offers hosting of this now.)

Squid or **Varnish** for reverse proxy caching.

Any third-party service for **CDN**.

Do the math

All non-CDN traffic travels through your load balancers and reverse proxy caches. Even traffic passed through to application servers **must run through the initial layers**.



What hit rate is each layer getting?
How many servers share the load?

Get a management/monitoring box

(maybe two or three and have them specialized or redundant)

Load Balancer

Database

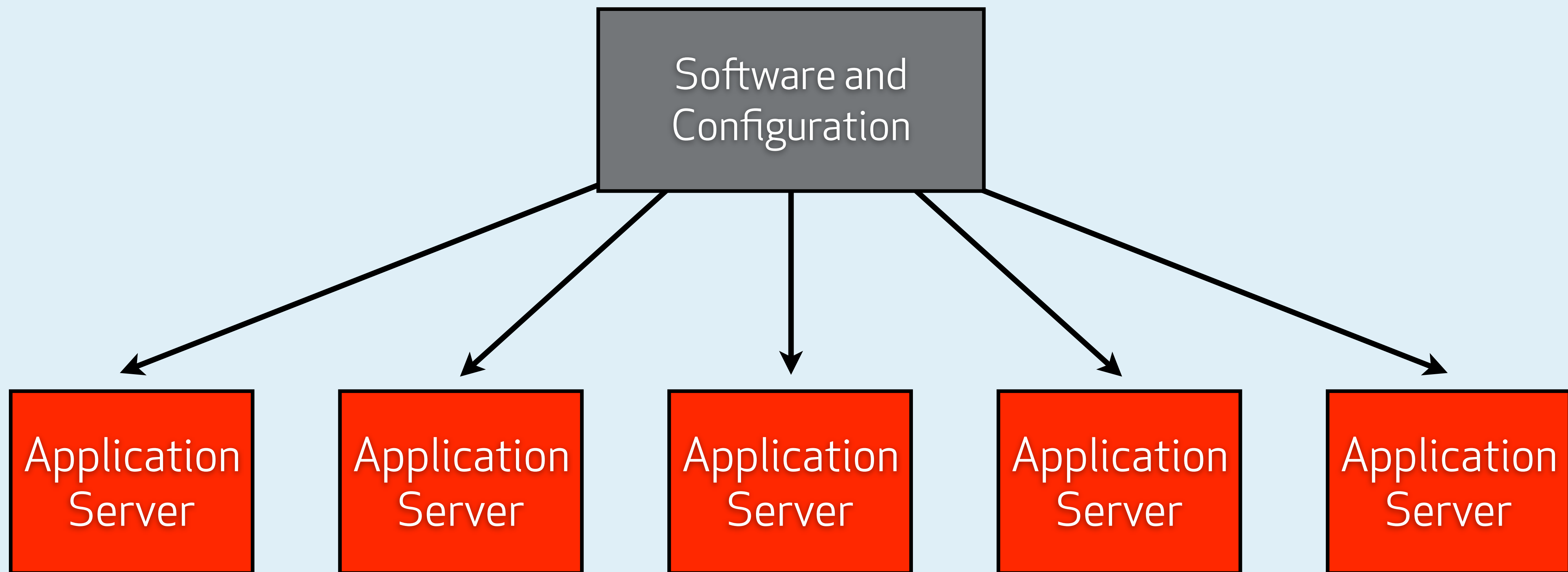
Management

Application Server

Reverse Proxy Cache

Managing the Cluster

The problem



Objectives:

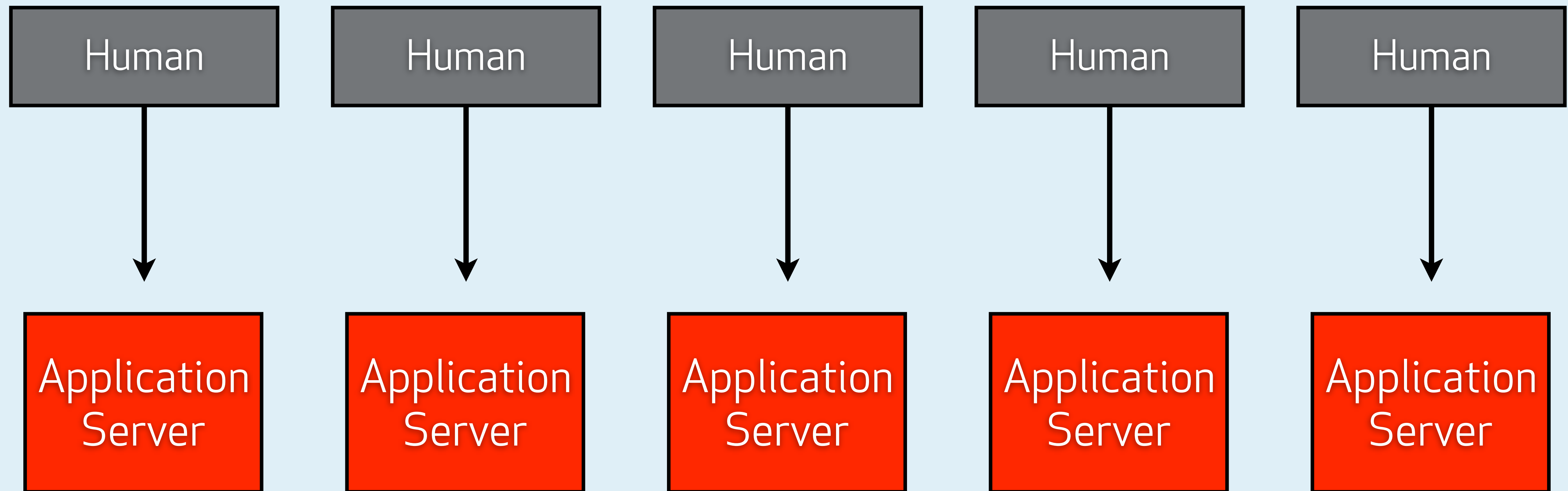
Fast, atomic deployment and rollback

Minimize single points of failure and contention

Restart services

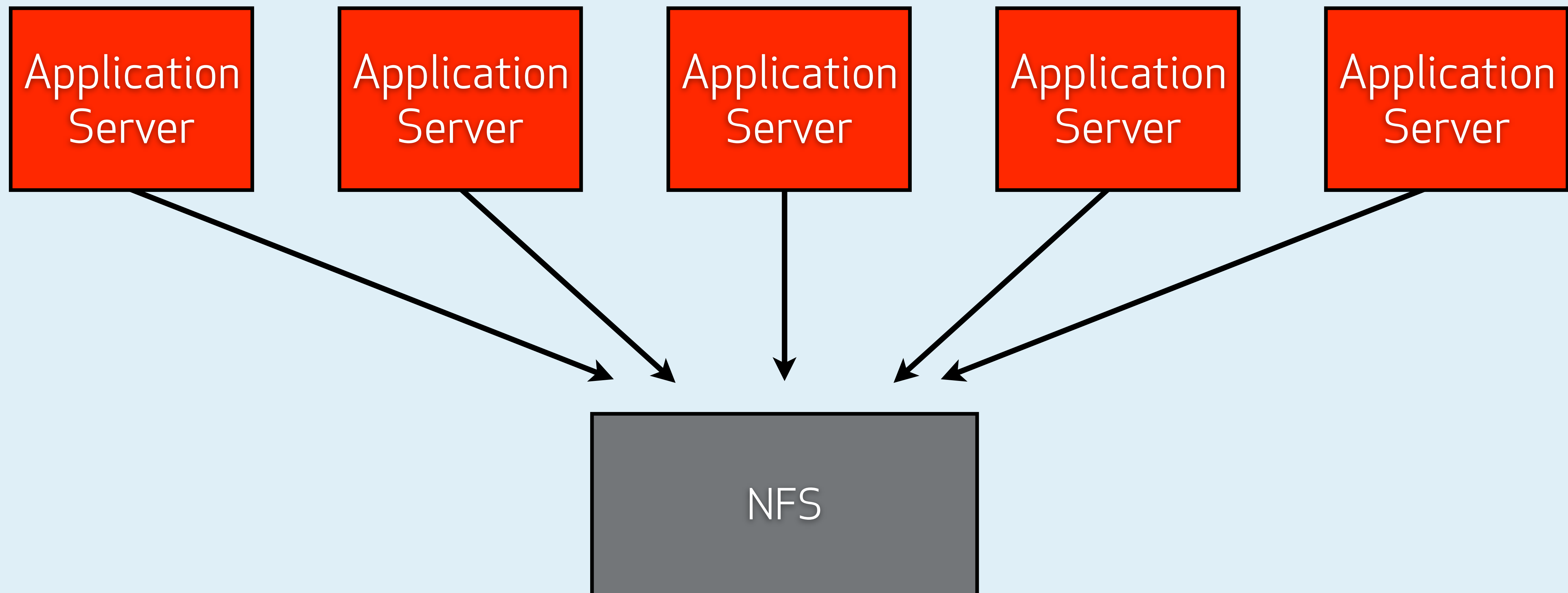
Integrate with version control systems

Manual updates and deployment



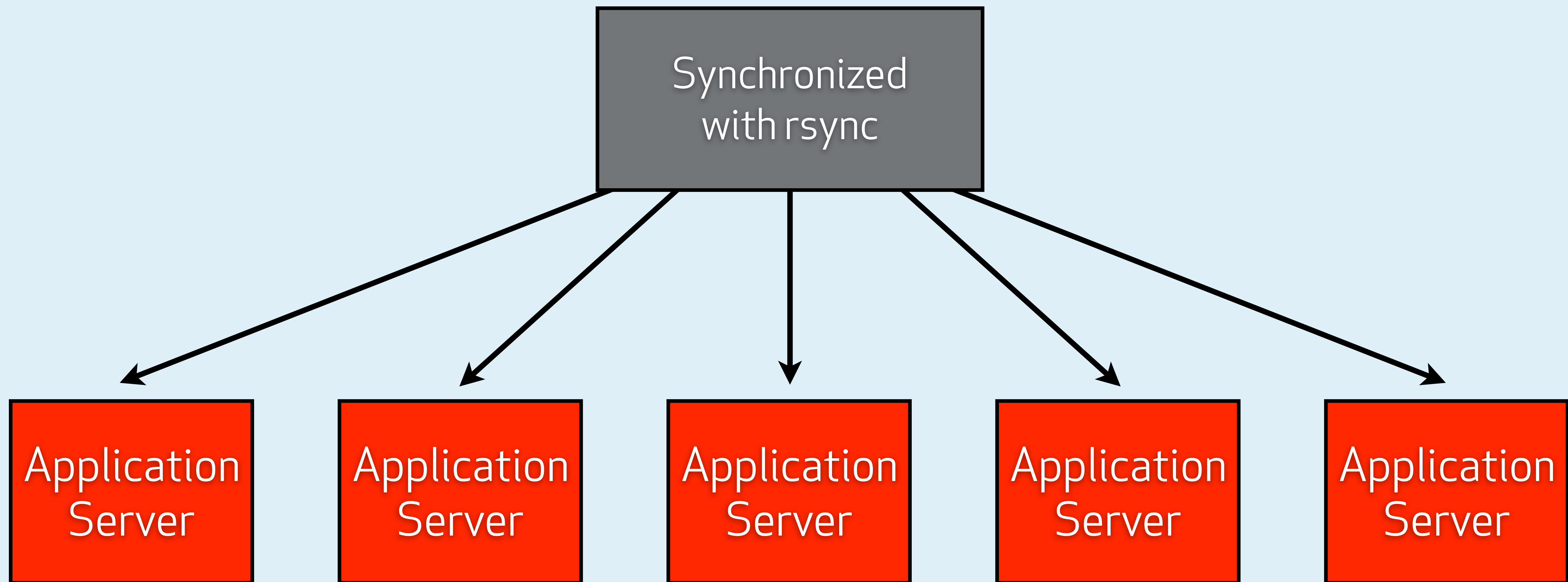
Why not: slow deployment, non-atomic/difficult rollbacks

Shared storage



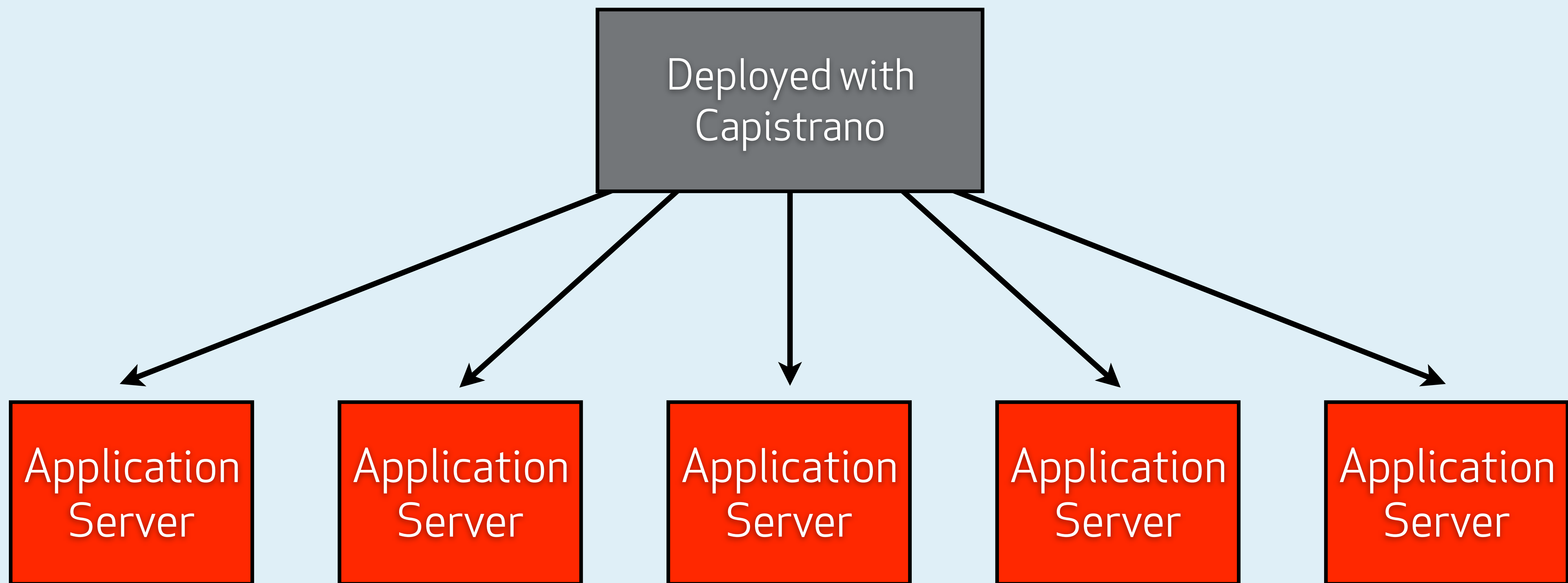
Why not: single point of contention and failure

rsync



Why not: non-atomic, does not manage services

Capistrano



Capistrano provides near-atomic deployment, service restarts, automated rollback, test automation, and version control integration (tagged releases).

Multistage deployment

Deployments
can be **staged**.

cap **staging** deploy
cap **production** deploy

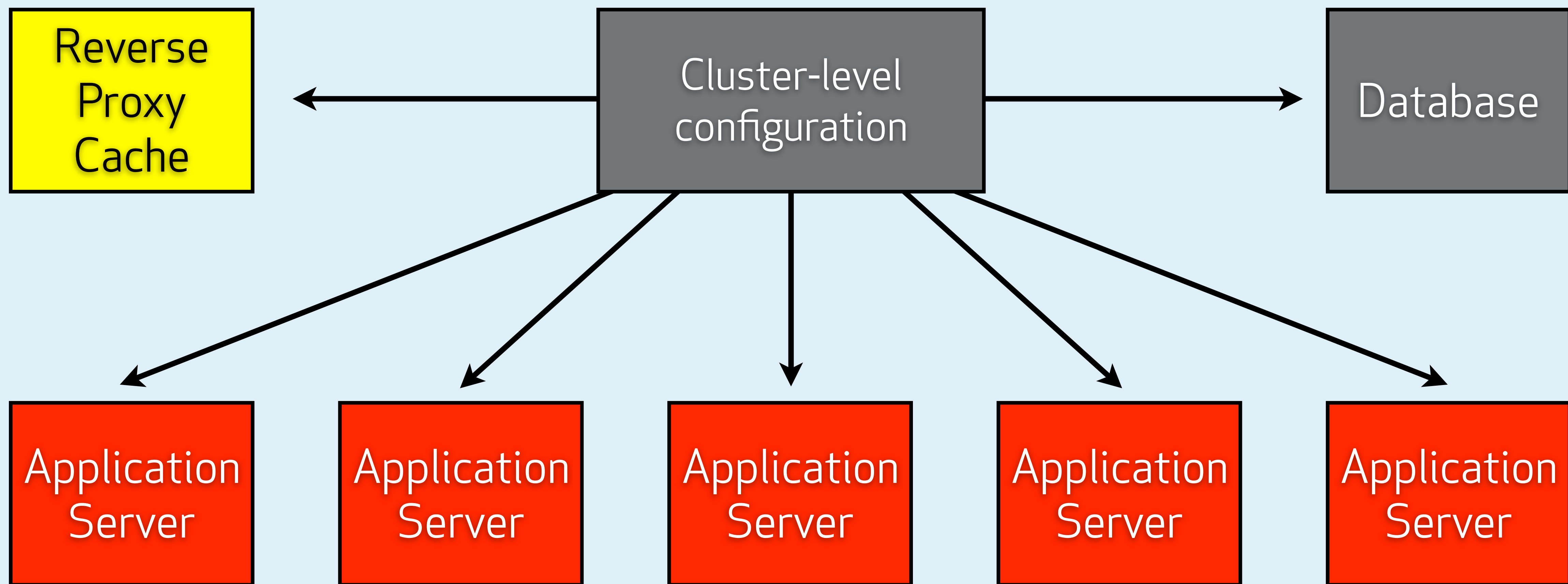
Deployed with Capistrano → Development Integration

Deployed with Capistrano → Staging

Deployed with Capistrano → Application Server, Application Server, Application Server, Application Server, Application Server

But your application isn't the only thing to manage.

Beneath the application



Cluster management applies to package management, updates, and software configuration.

cfengine and **bcfg2** are popular cluster-level system configuration tools.

System configuration management

Deploys and updates **packages**, cluster-wide or selectively.

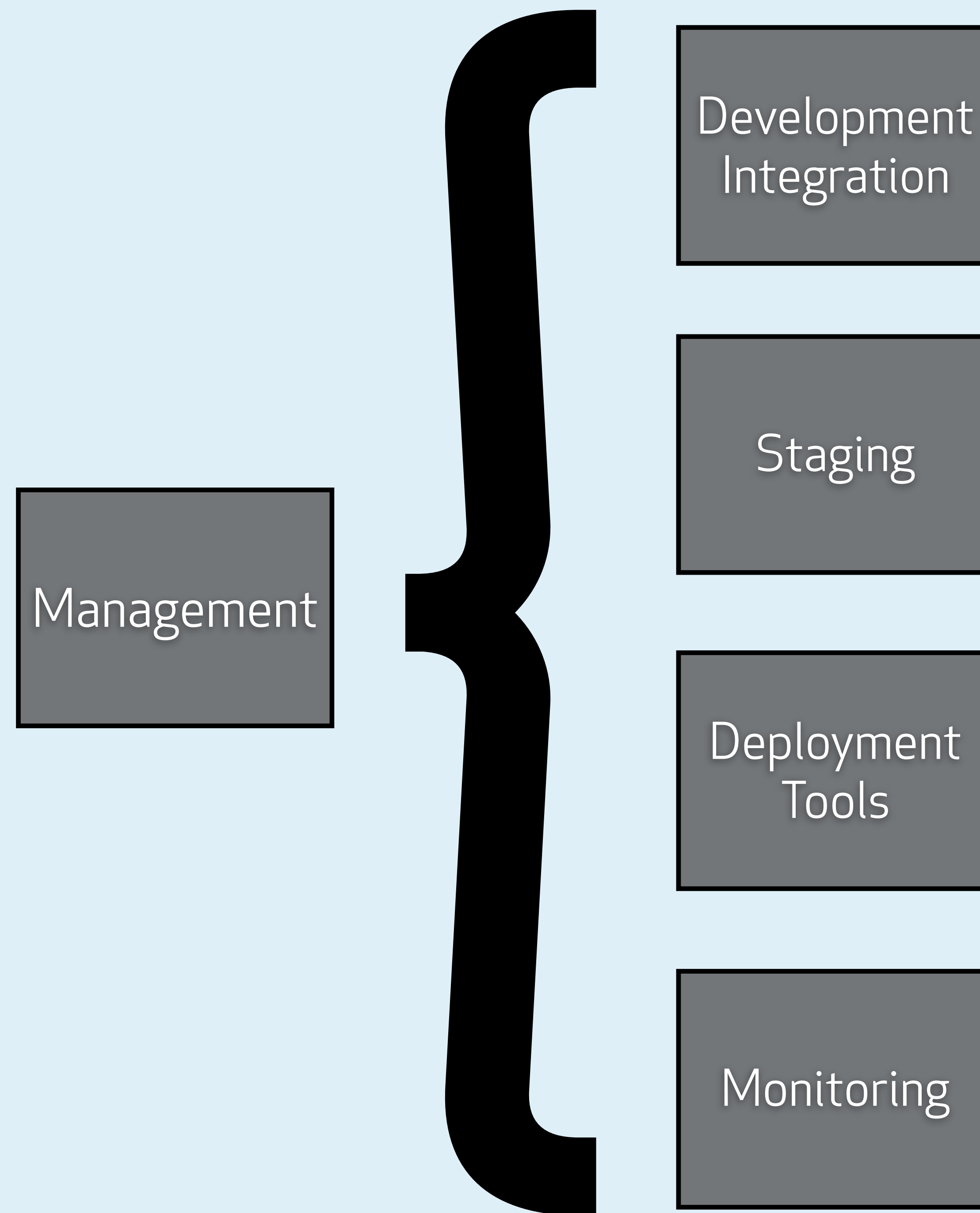
Manages arbitrary text configuration files

Analyzes inconsistent configurations (and converges them)

Manages device classes (app. servers, database servers, etc.)

Allows confident configuration **testing** on a staging server.

All on the management box



Monitoring

Types of monitoring

Failure	Capacity/Load
Analyzing Downtime	Analyzing Trends
Viewing Failover	Predicting Load
Troubleshooting	Checking Results of Configuration and Software Changes
Notification	

Everyone needs both.

What to use

Failure/Uptime	Capacity/Load
Nagios	Cacti
Hyperic	Munin

Nagios

Highly **recommended**.

Used by Four Kitchens and Tag1 Consulting for client work, Drupal.org, **Wikipedia**, etc.

Easy to install on CentOS 5 using **EPEL** packages.

Easy to install **nrpe** agents to monitor diverse services.

Can **notify** administrators on failure.

We use this on Drupal.org.

Hyperic

I haven't used this much, but it's fairly **popular**.

More difficult to set up than Nagios.

Cacti

Highly **annoying** to set up.

One instance generally collects all statistics.
(No "agents" on the systems being monitored.)

Provides **flexible** graphs that can be customized on demand.

Optimized database for perpetual statistics collection.

We use this on Drupal.org and for client sites.

Munin

Fairly **easy** to set up.

One instance generally collects all statistics.
(No "agents" on the systems being monitored.)

Provides **static** graphs that cannot be customized.

Cluster Problems

Cache/session coherency

Systems that run properly on single boxes may **lose coherency** when run on a networked cluster.

Some caches, like APC's object cache, have **no ability** to handle network-level coherency. (APC's opcode cache is safe to use on clusters.)

memcached, if misconfigured, can hash values **inconsistently** across the cluster, resulting in different servers using different memcached instances for the same keys.

Session coherency can be helped with load balancer **affinity**.

Cache regeneration races

Downside to network cache coherency: **synched expiration**

Hard to solve

All servers regenerating the item.

Old Cached Item

New Cached Item

Expiration

Time

Broken replication

MySQL slave servers get out of synch, fall further behind

No means of automated recovery

Only solvable with good **monitoring** and recovery procedures

Can automate removal from use, but requires cluster management tools

Server failure

Load balancers can remove broken or overloaded application reverse proxy caches.

Reverse proxy caches like Varnish can automatically use only functional application servers.

Cluster management tools like heartbeat2 can manage service IPs on MySQL servers to automate failover.

Conclusion: Each layer intelligently monitors and uses the servers beneath it.