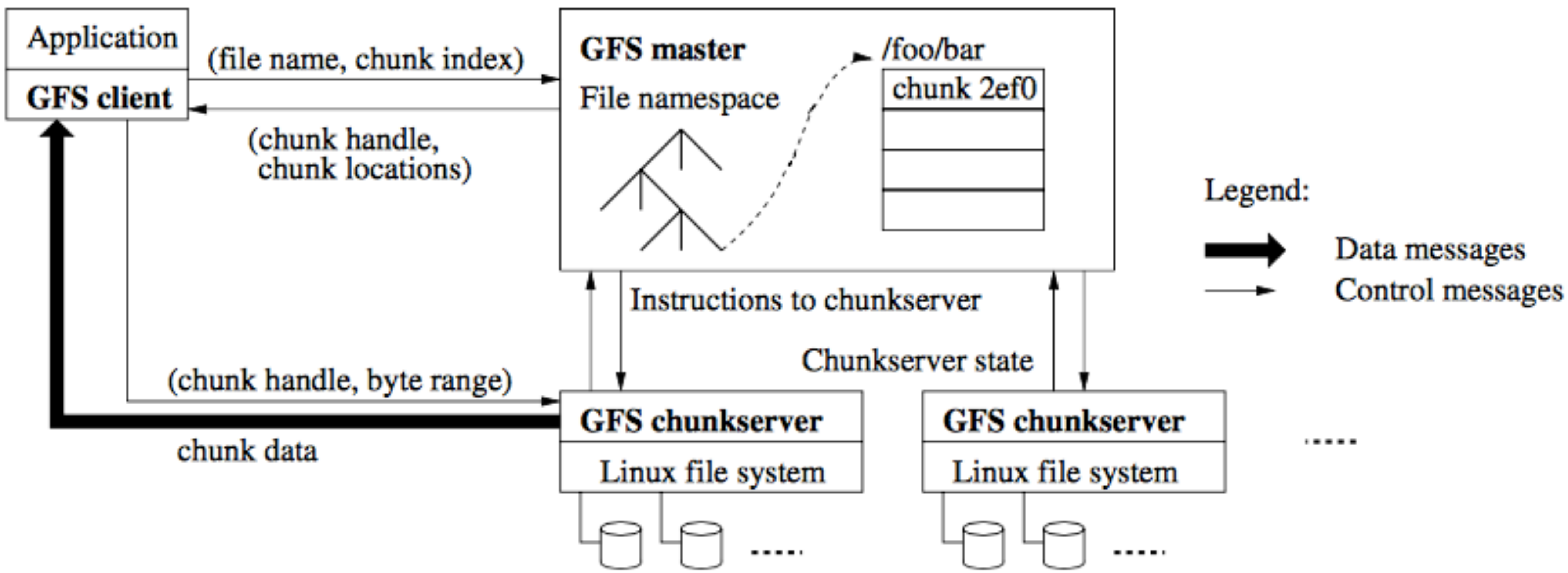
Distributed Architectures & Microservices

CS 475, Spring 2018
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

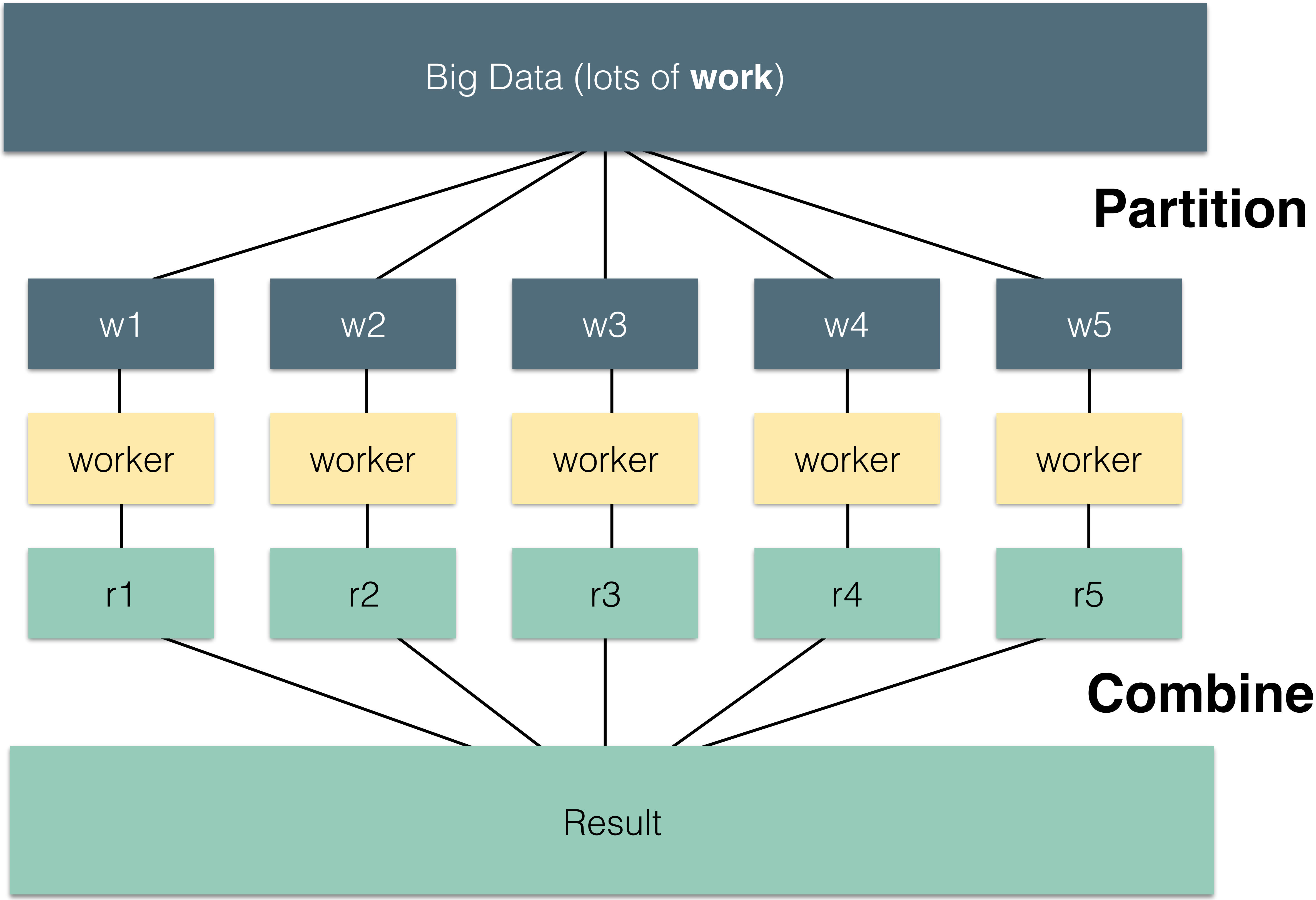
GFS Architecture



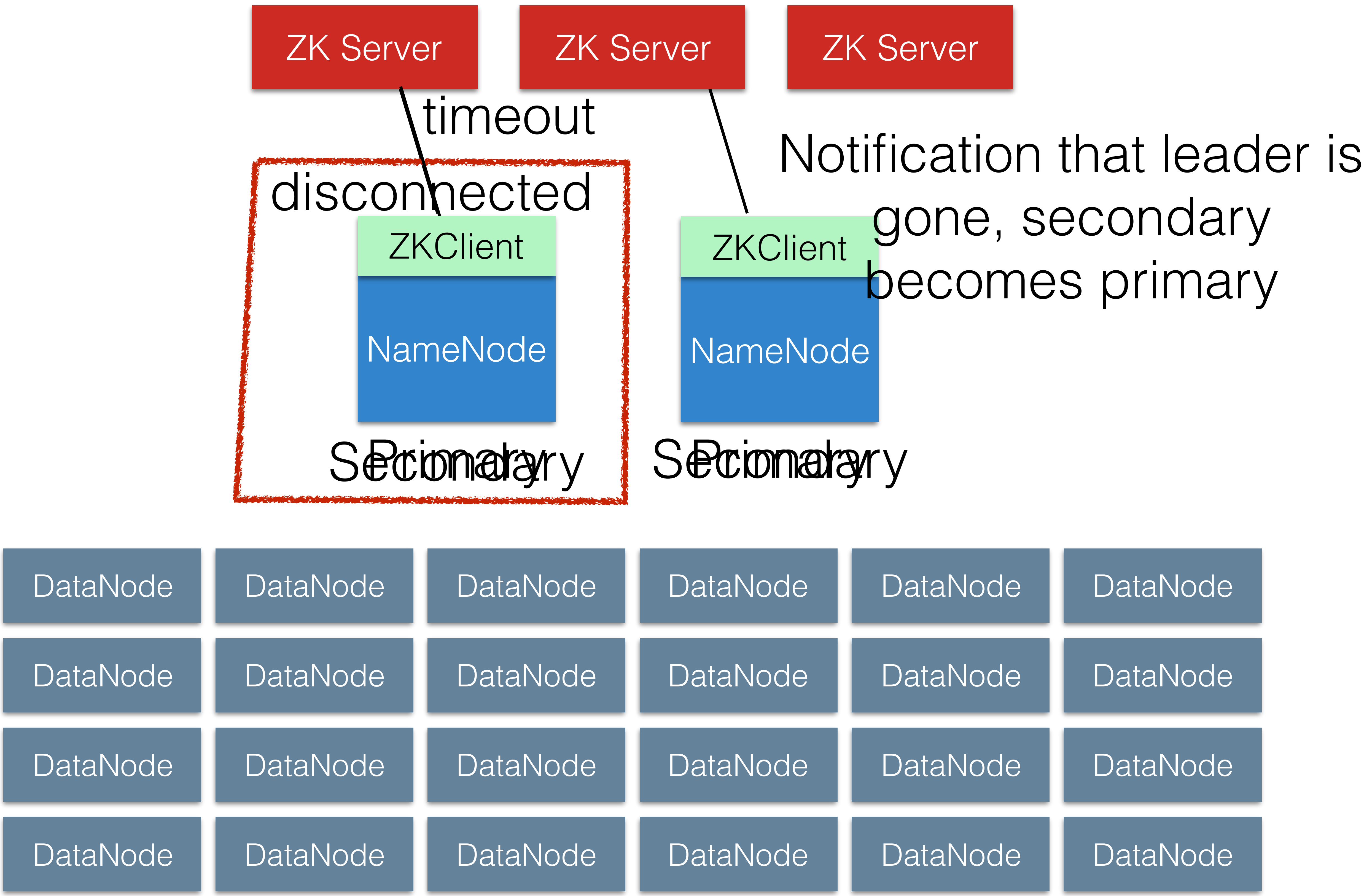
GFS Summary

- Limitations:
  - Master is a huge bottleneck
  - Recovery of master is slow
- Lots of success at Google
- Performance isn't great for all apps
- Consistency needs to be managed by apps
- Replaced in 2010 by Google's Colossus system - eliminates master

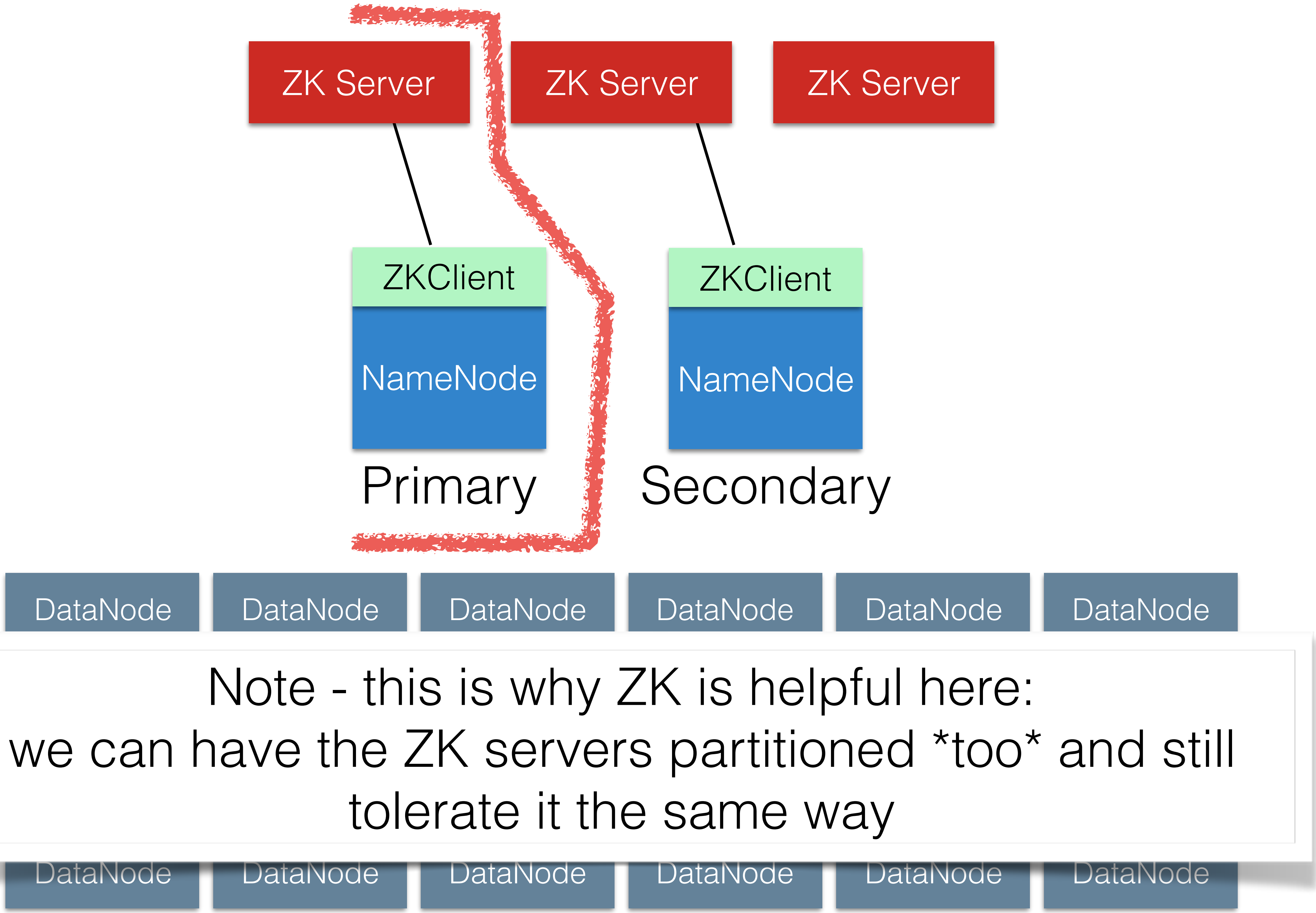
MapReduce: Divide & Conquer



Hadoop + ZooKeeper



Hadoop + ZooKeeper



Note - this is why ZK is helpful here:
we can have the ZK servers partitioned *too* and still tolerate it the same way

Announcements

- Form a team and get started on the project!
  - http://jonbell.net/gmu-cs-475-spring-2018/final-project/
  - AutoLab available soon
- Today:
  - Distributed system architectures

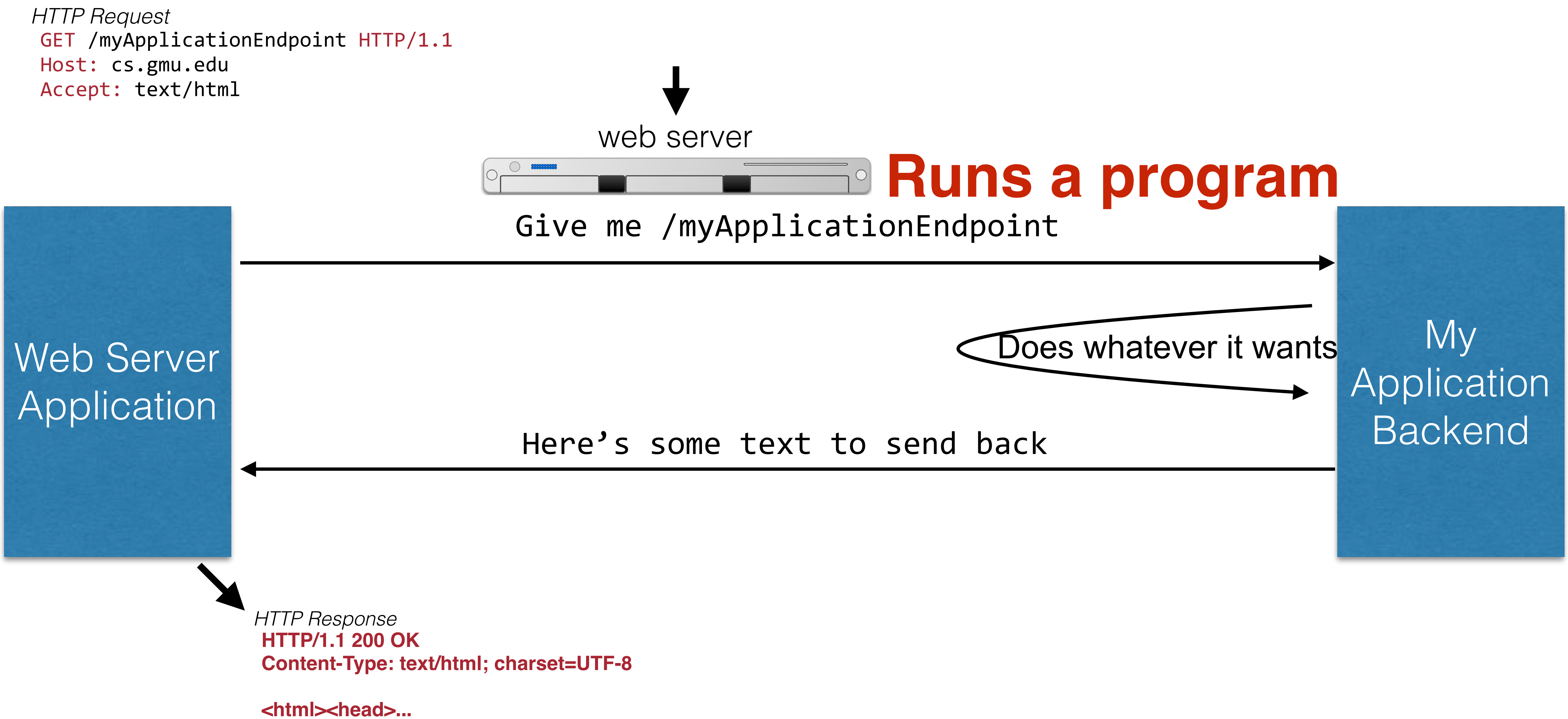
Distributed Systems Abstractions

- Goal: find some way of making our distributed system look like a single system
- Never achievable in practice
- BUT if we can come up with some model of how the world might behave, we can come up with some generic solutions that work pretty well
  - And hopefully we can understand how they can go wrong

Abstractions & Architectures

- We can design *architectures* that embody some systems model, providing some framework code to make it easier to get some task done
- Case study example: web architectures
- Assumptions:
  - "one" server, many clients
  - Synchronous communication
  - Client is unlikely to be partitioned from a subset of servers; likely some subset of servers are partitioned from other servers
  - Client is mostly stateless

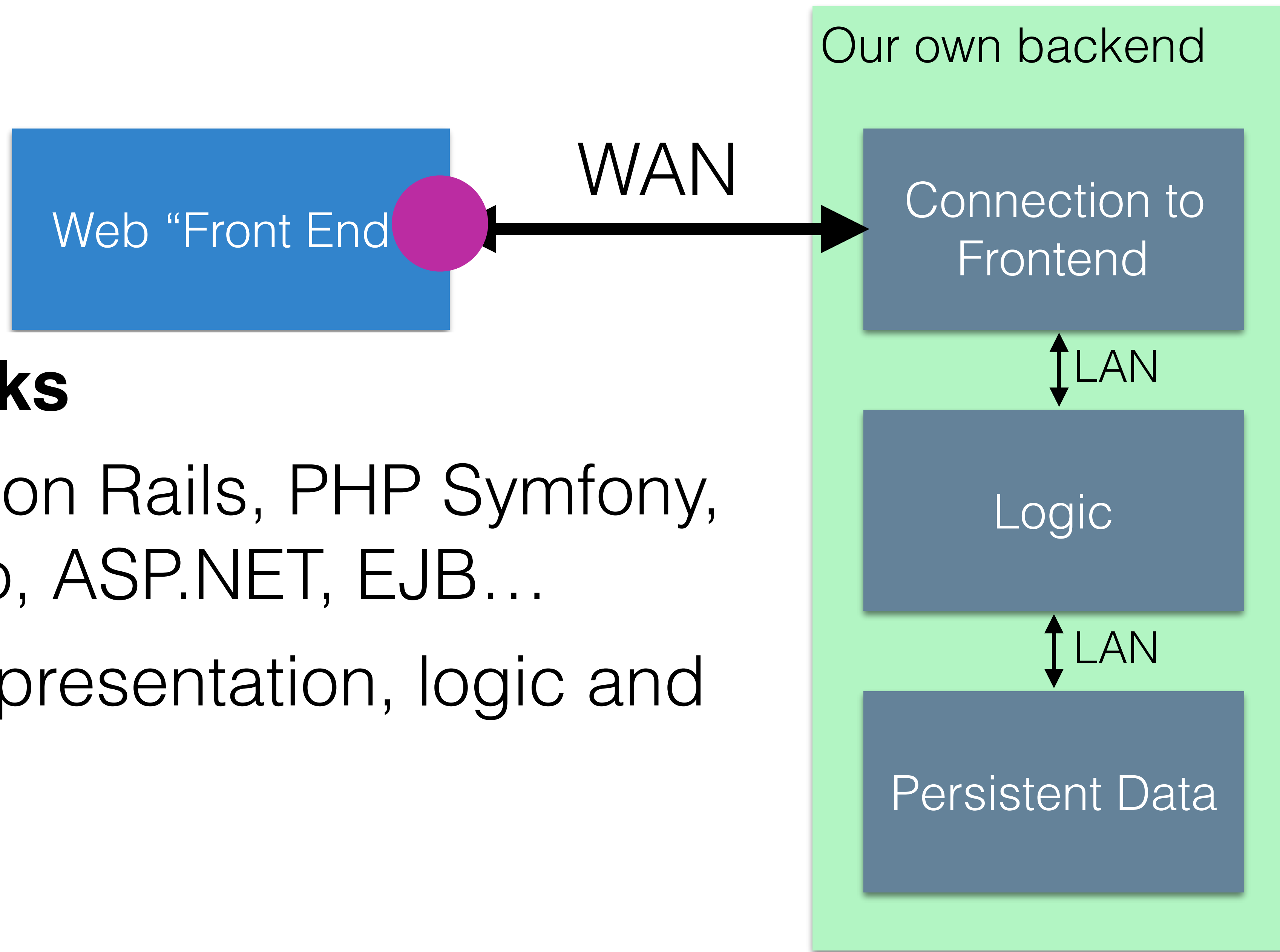
The good old days of web apps



Brief history of Backend Development

- In the beginning, you wrote whatever you wanted using whatever language you wanted and whatever framework you wanted
- Then… PHP and ASP
  - Languages “designed” for writing backends
  - Encouraged spaghetti code
  - A lot of the web was built on this
- A whole lot of other languages were also springing up in the 90’s…
  - Ruby, Python, JSP

Backend Frameworks

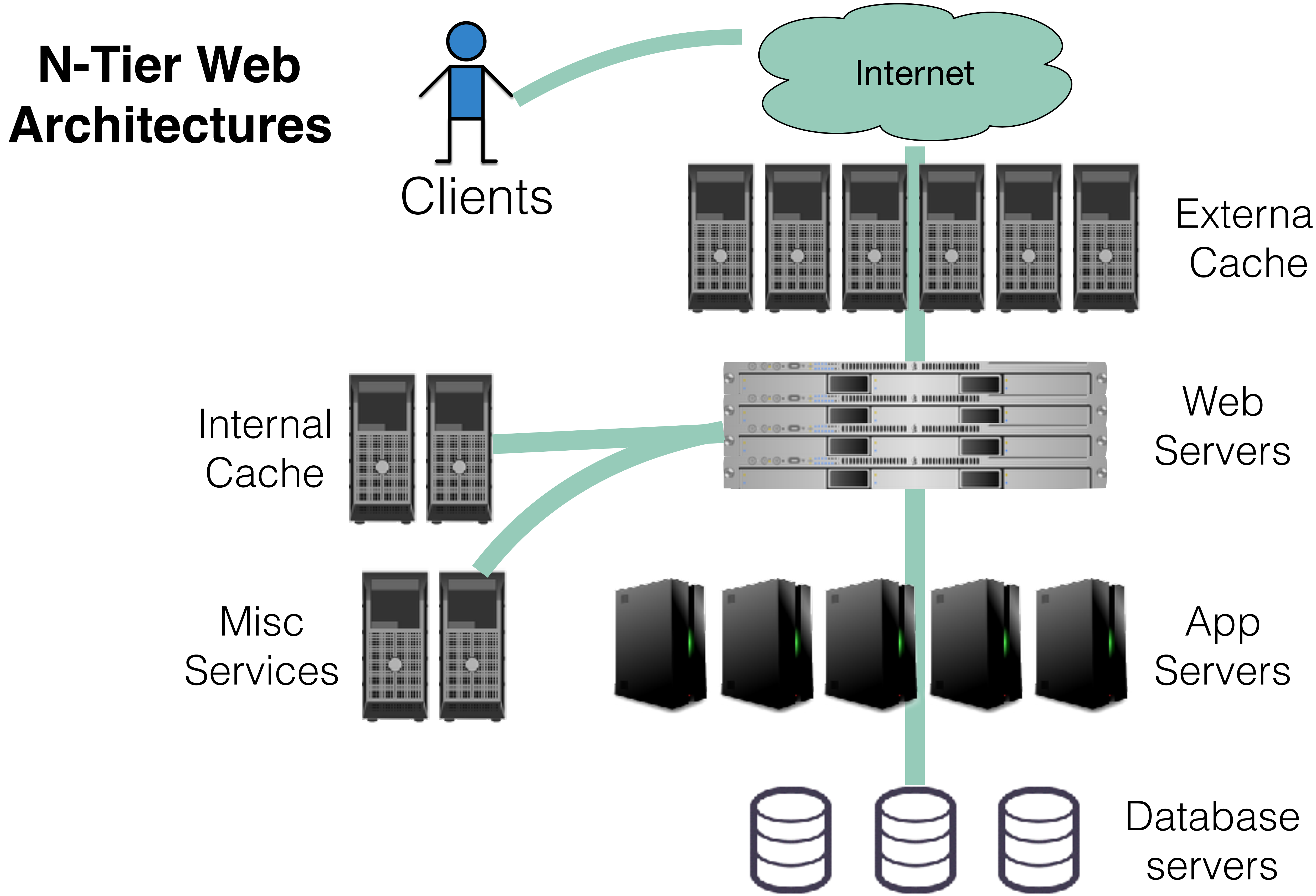


- Then: **frameworks**
  - SailsJS, Ruby on Rails, PHP Symfony, Python Django, ASP.NET, EJB…
- MVC - separate presentation, logic and persistence

Scaling web architectures up

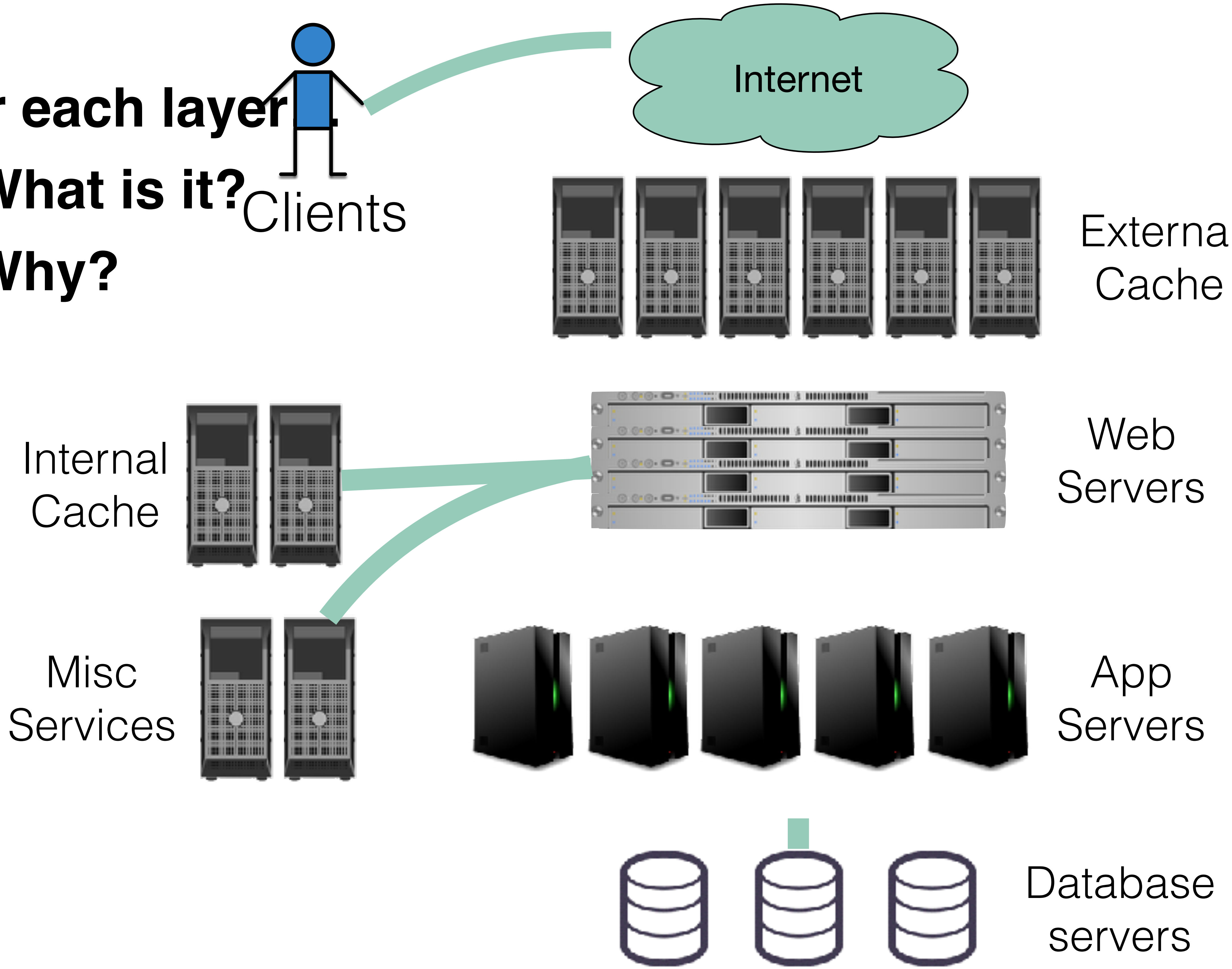
- What happens when we have to use this approach to run, say… Facebook?
- Tons of dynamic content that needs to be updated, petabytes of static content (like pictures), users physically located all over, lots of stuff to keep track of, where do we start?

Real Architectures



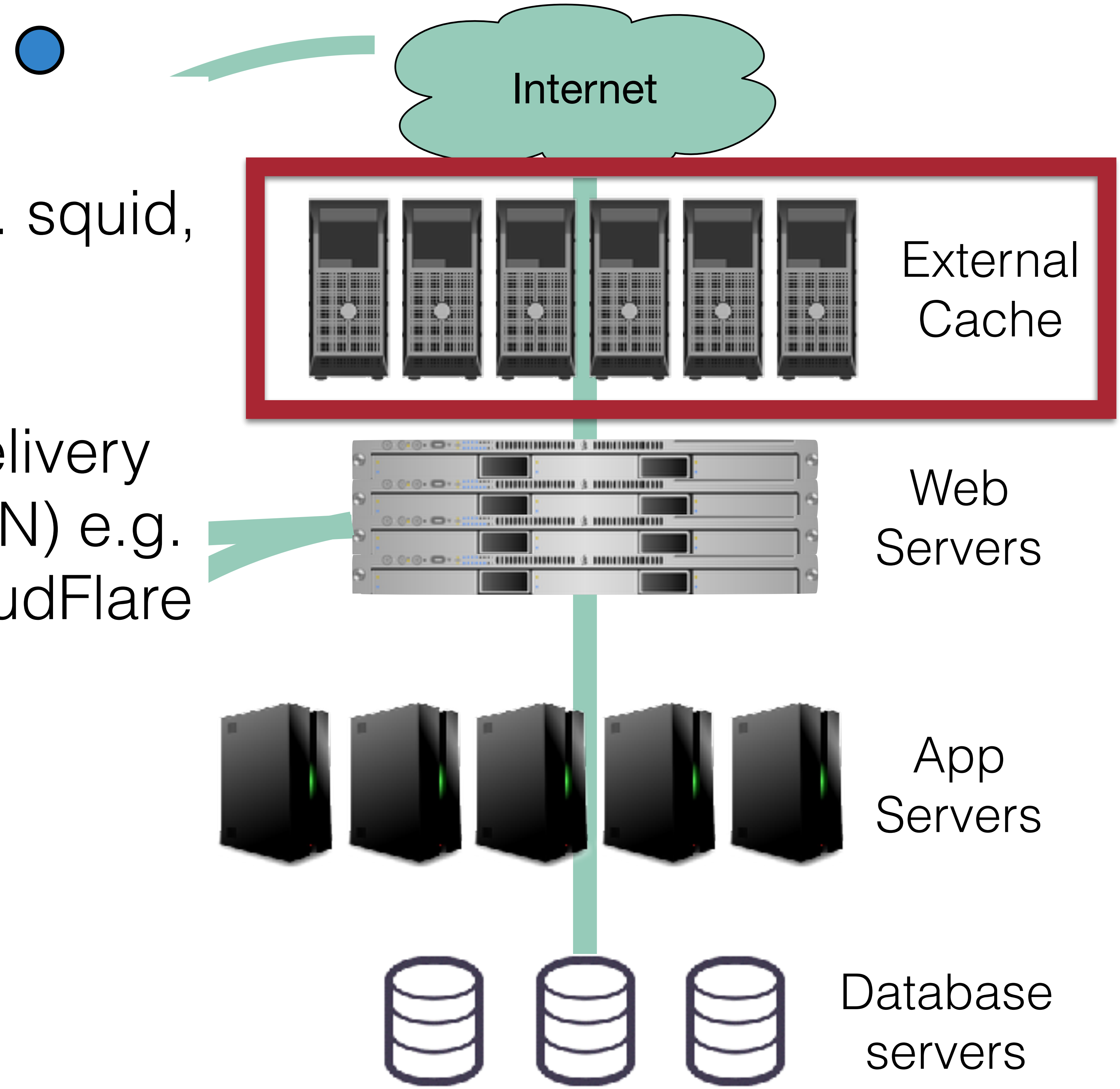
Real Architectures

- **For each layer:**
  - **What is it?**
  - **Why?**



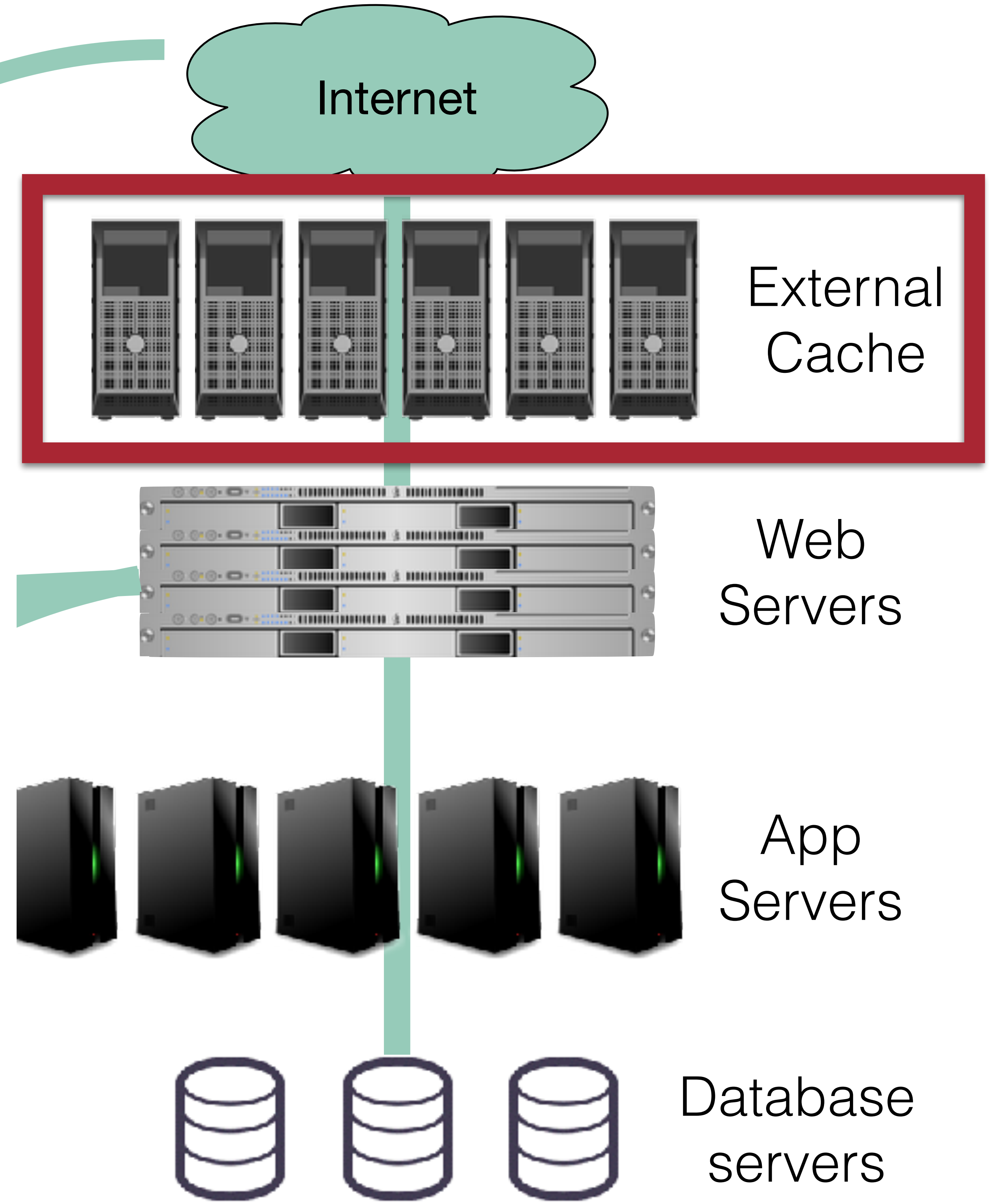
External cache

- What is it?
  - A proxy (e.g. squid, apache mod_proxy)
  - A content delivery network (CDN) e.g. Akamai, CloudFlare



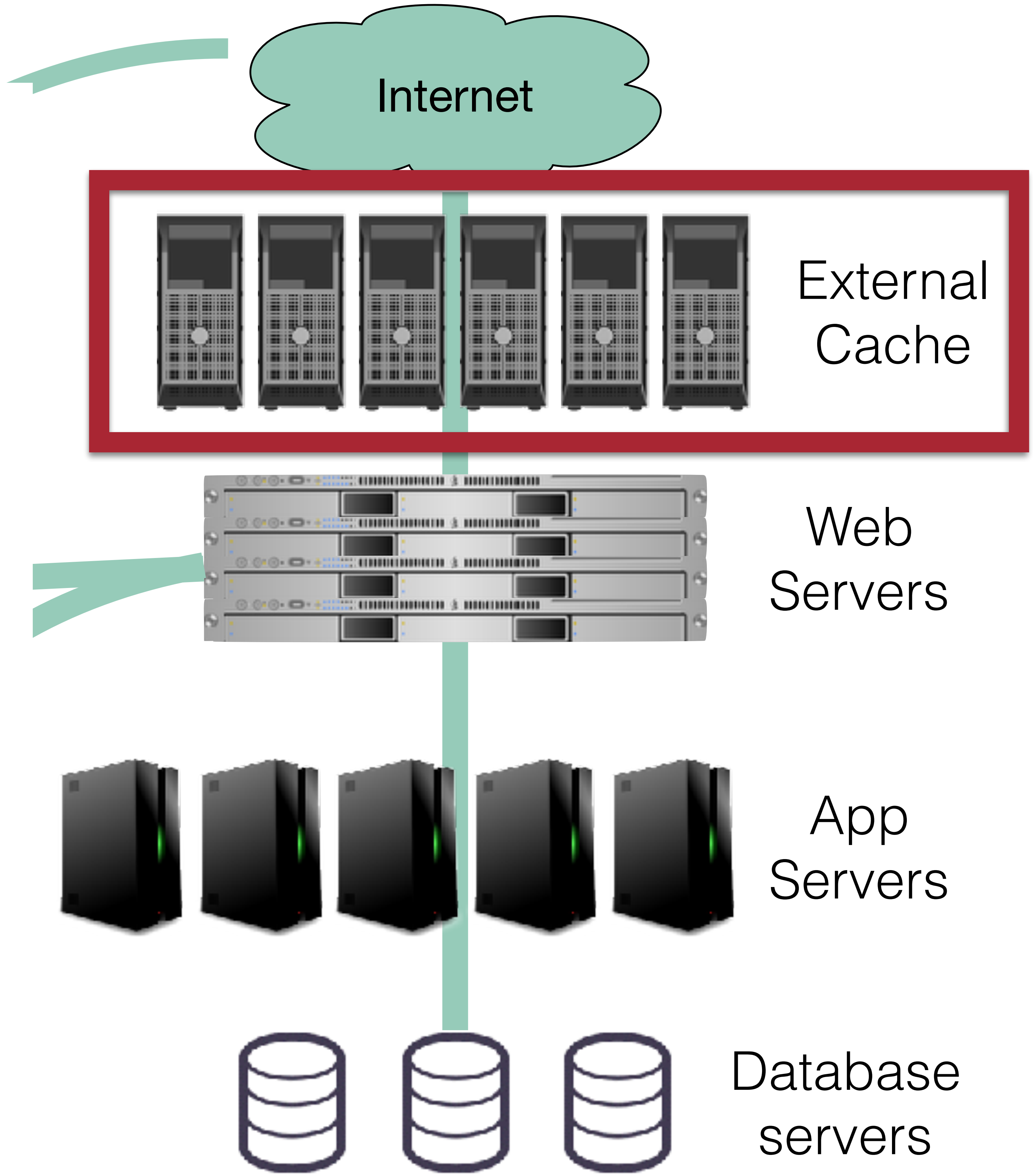
External cache

- What is it for?
    - Caches outbound data
        - Images, CSS, XML, HTML, pictures, videos, anything static (some stuff dynamic maybe)
    - DoS defense
    - Decrease latency - might be close to the user



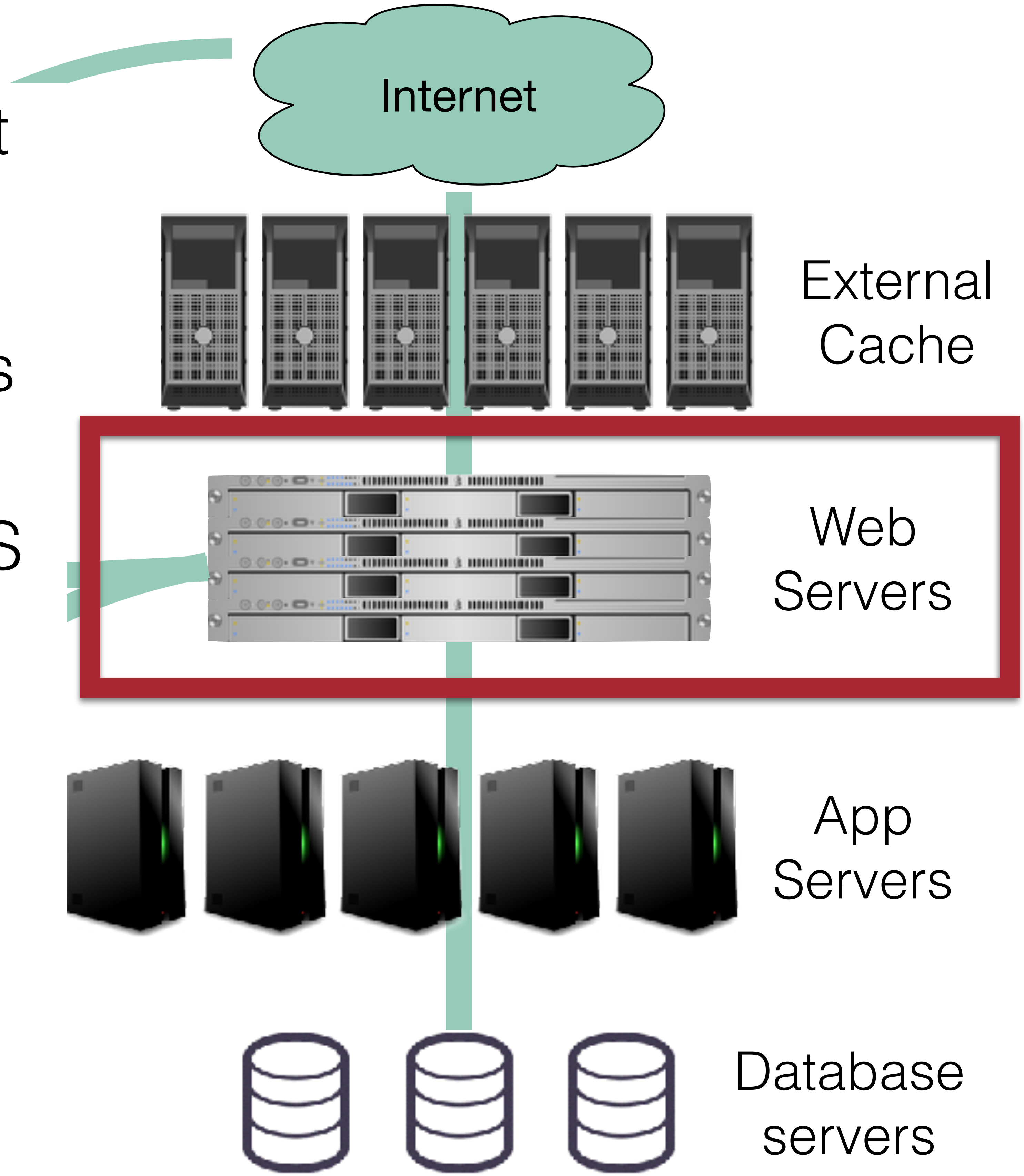
External cache

- What is it made of?
- Tons of RAM, fast network, physically located all over
- No need for much CPU



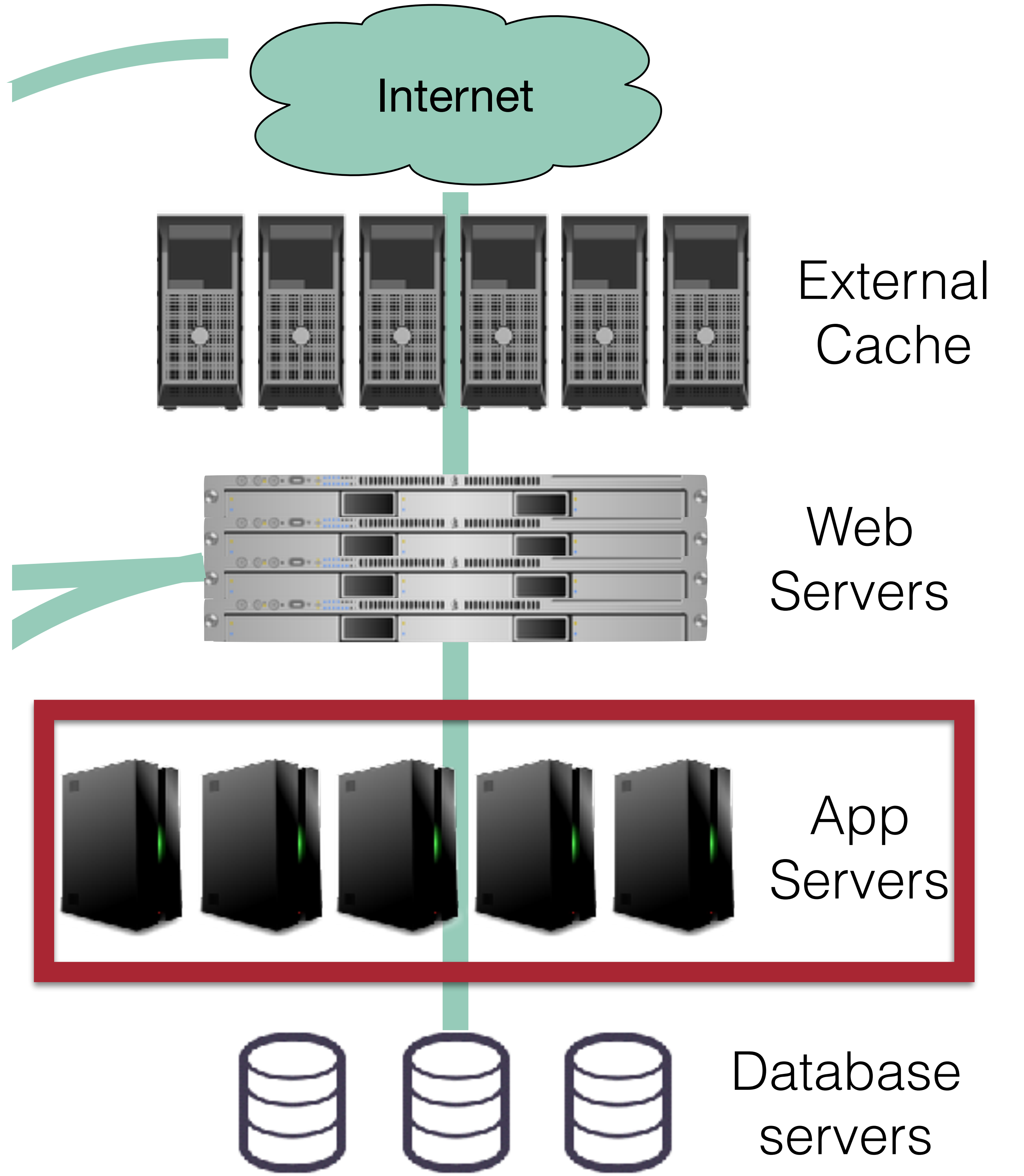
Front-end Tier

- Serves static content from disk, generates dynamic content by dispatching requests to app tier
- Speaks HTTP, HTTPS



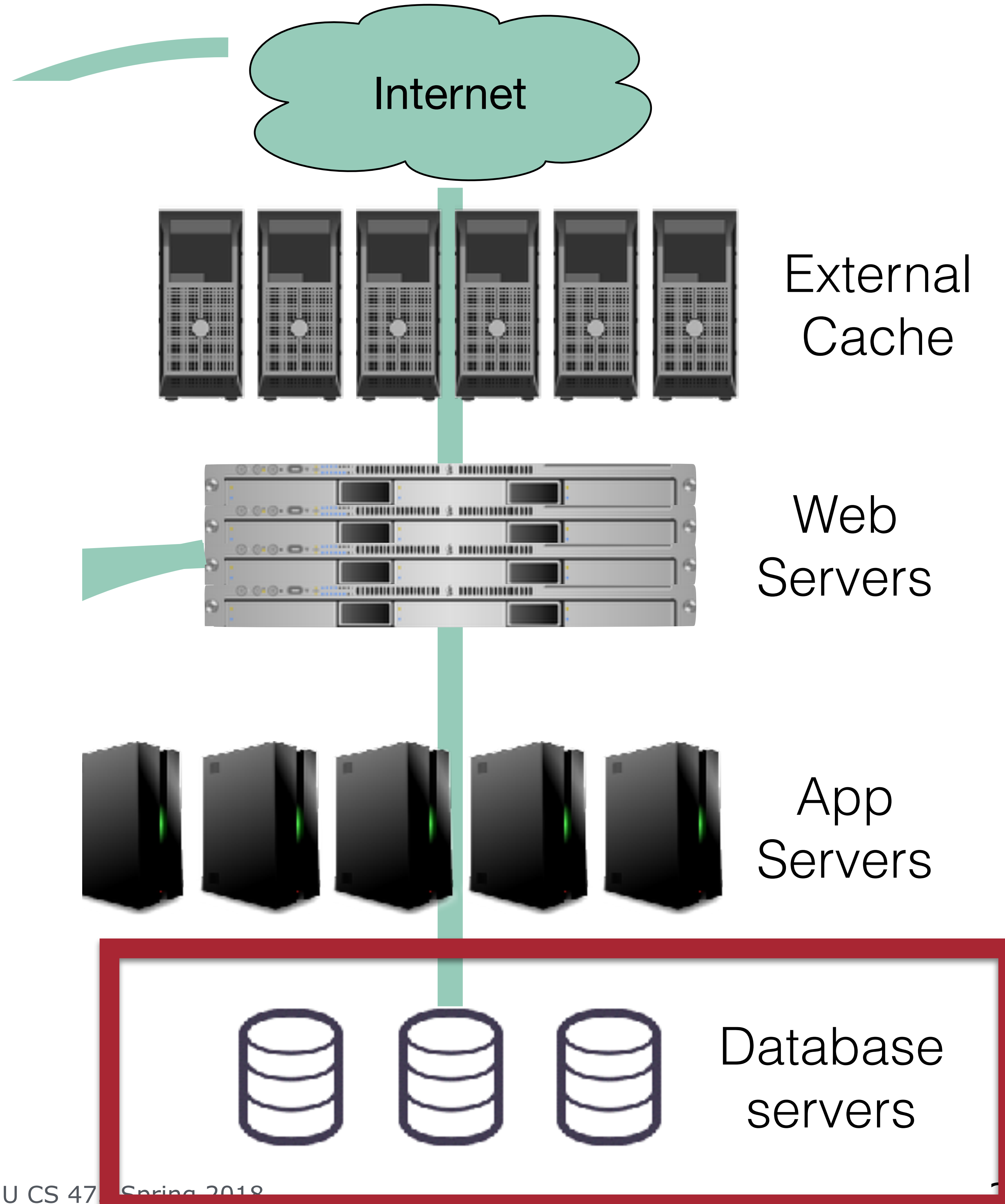
Application Server Tier

- Serves dynamic pages
- Provides internal services
  - E.g. search, shopping cart, account management
- Talks to web tier over..
  - RPC, REST, CORBA, RMI, SOAP, XMLRPC… whatever
- More CPU-bound than any other tier



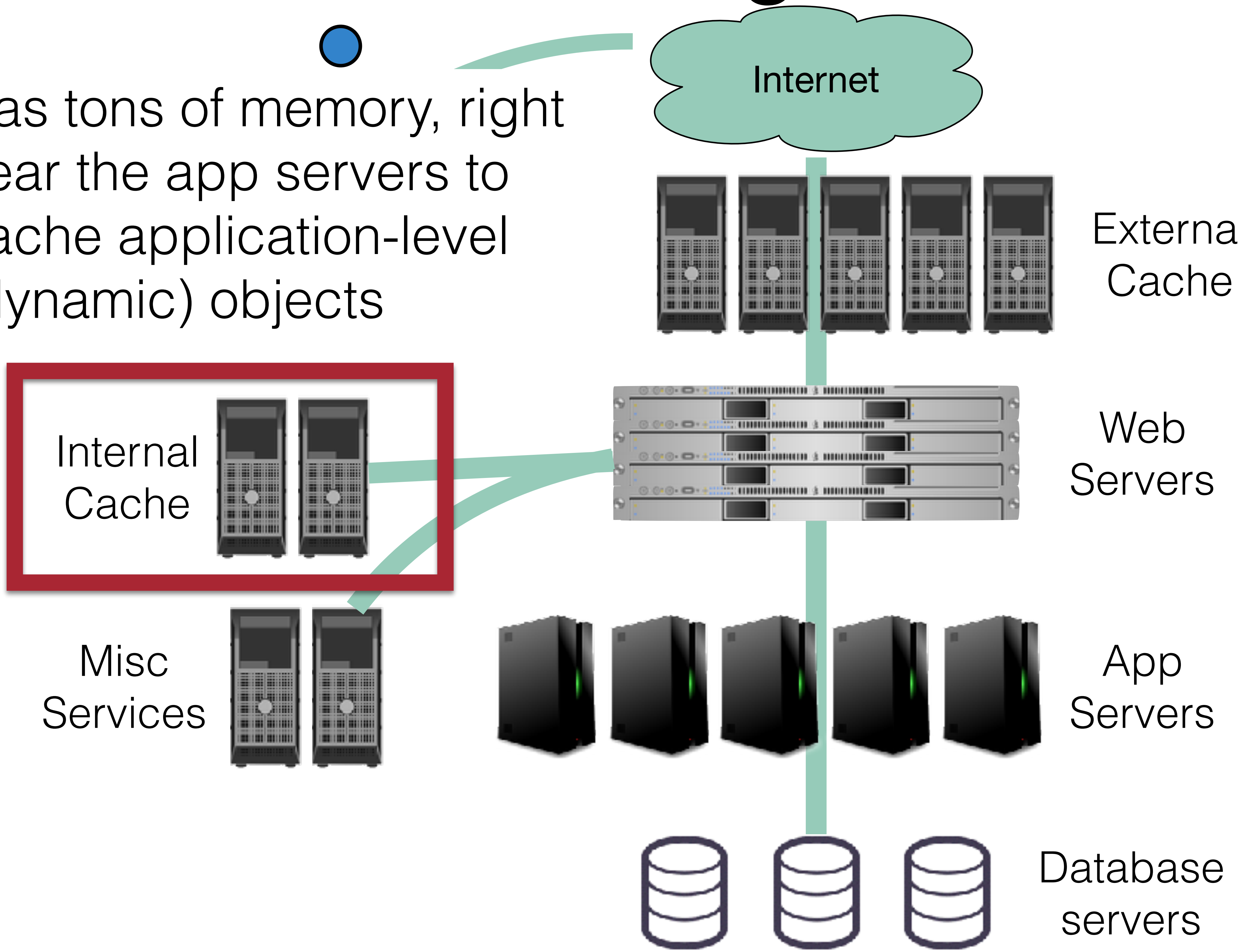
Database Tier

- Relational or non-relational DB
  - PostgreSQL, MySQL, Mongo, Cassandra, etc
- Most storage



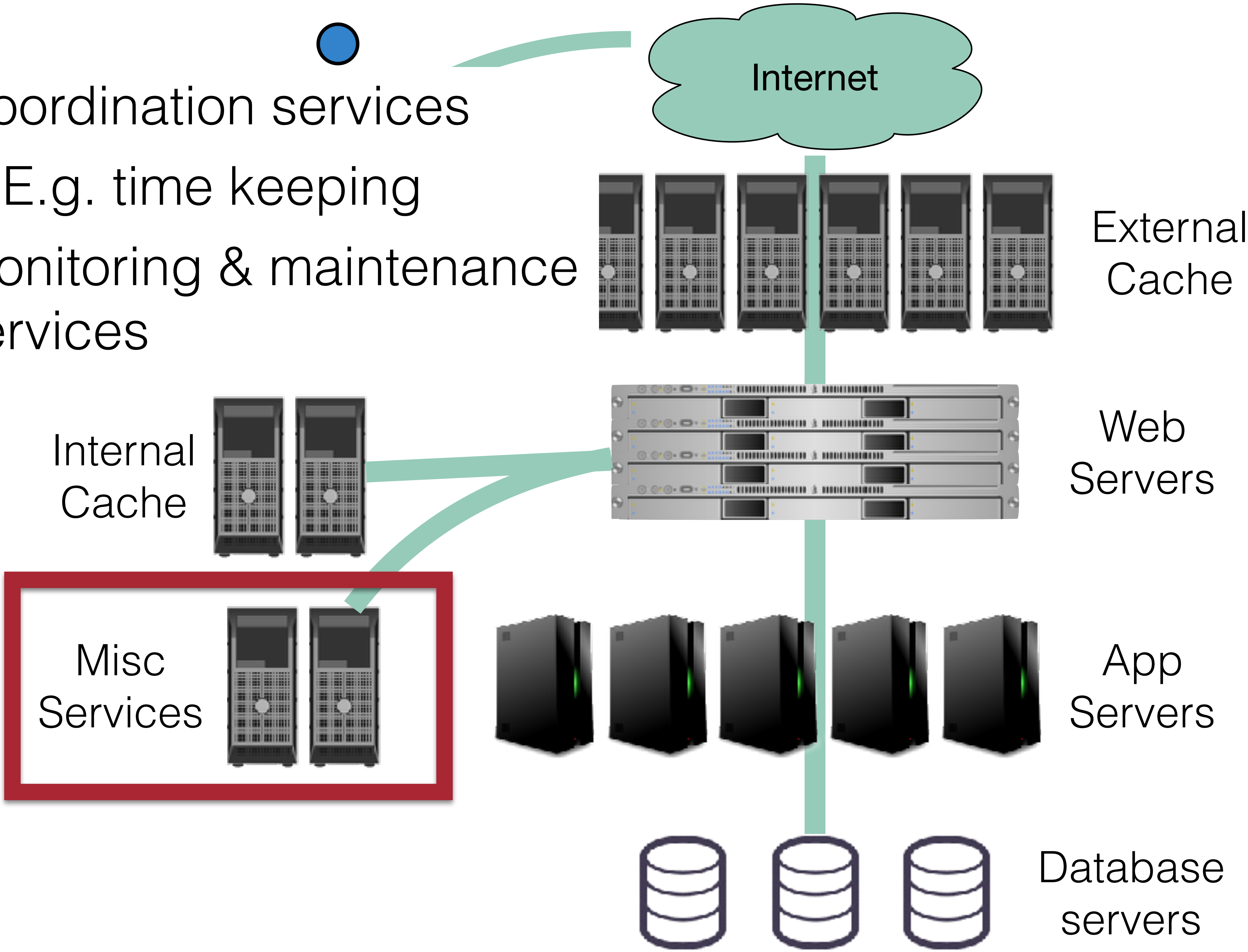
Internal Caching Tier

- Has tons of memory, right near the app servers to cache application-level (dynamic) objects



Internal Services Tier

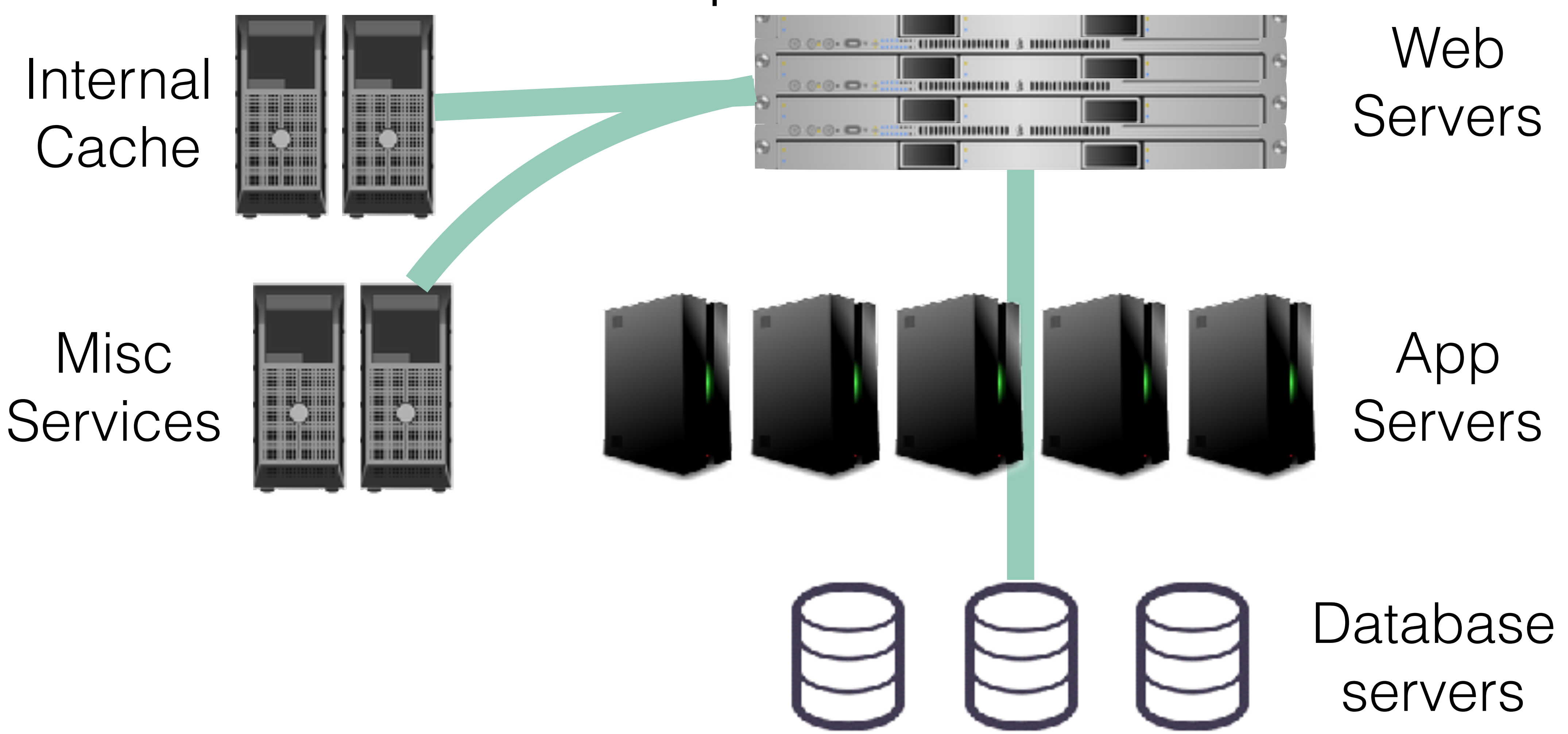
- Coordination services
  - E.g. time keeping
- Monitoring & maintenance services



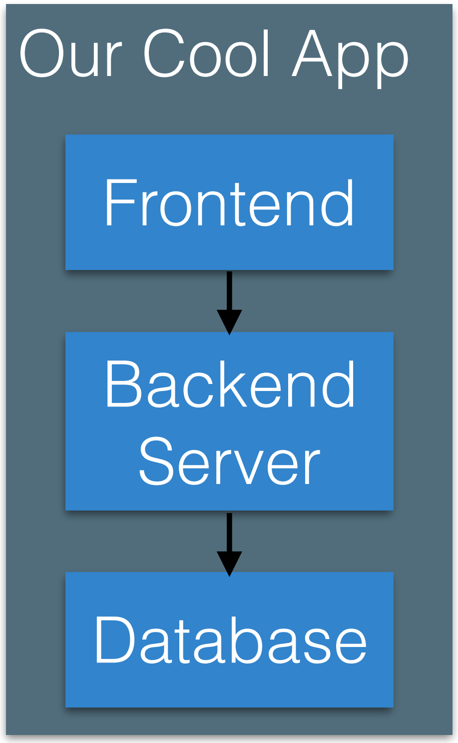
Real Architectures

N-Tier Web Architectures

Separate out responsibilities with abstractions: each tier cares about a different aspect of getting the client their response



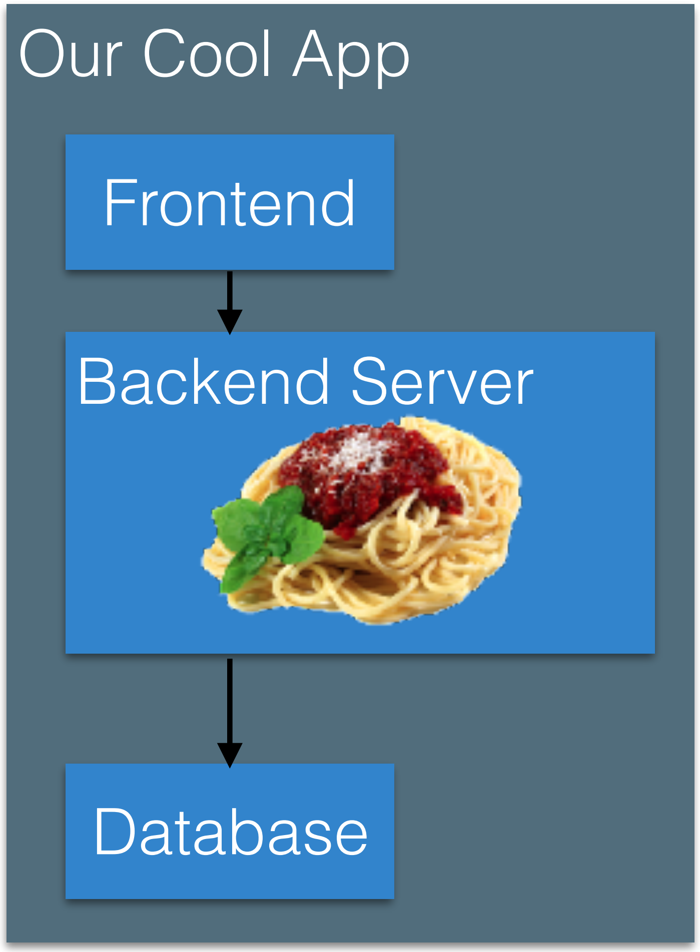
How do we build big apps?



Basic todo app

What happens when we want to add more functionality to our backend?

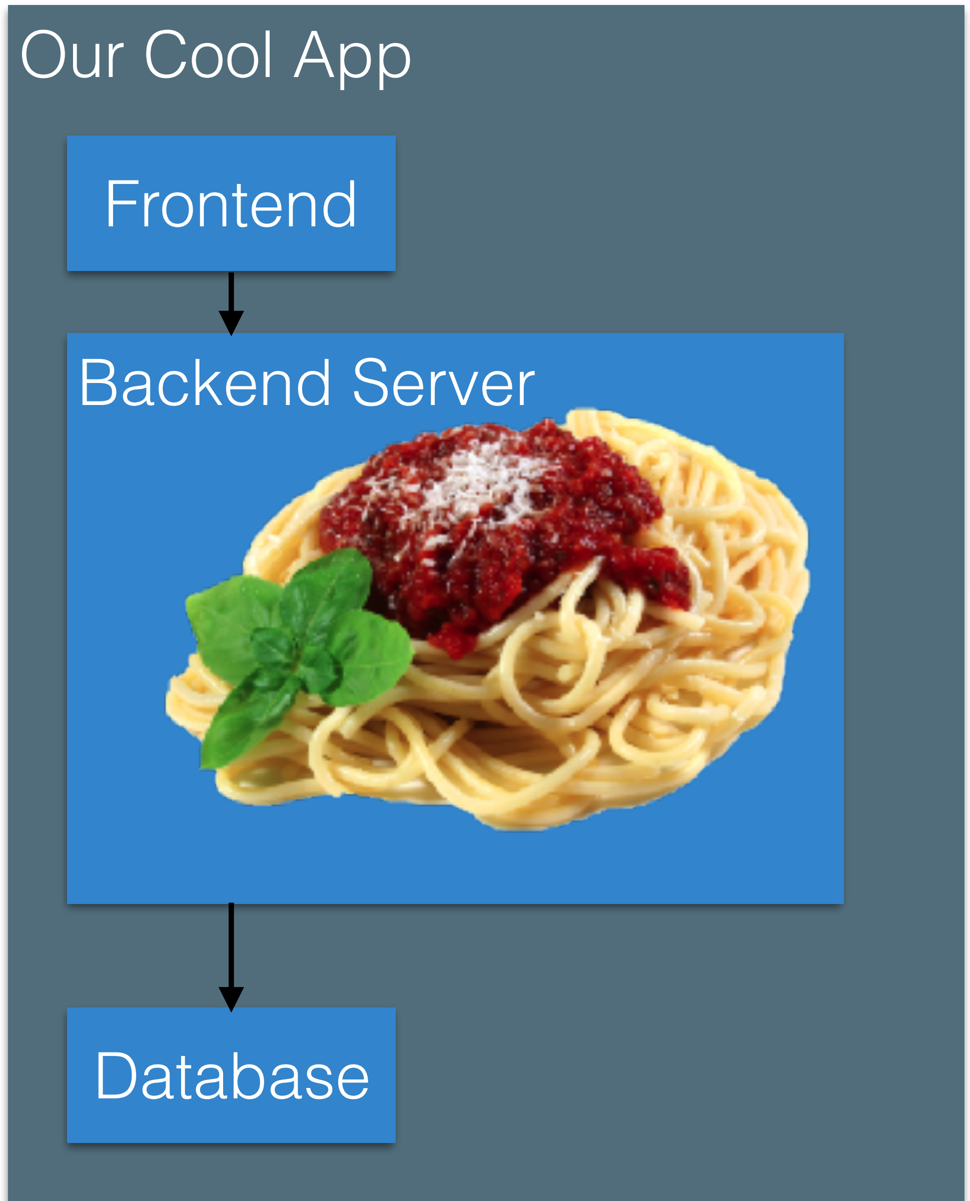
How do we build big apps?



Basic todo app with new feature to email todo reminders

What happens when we add more functionality?

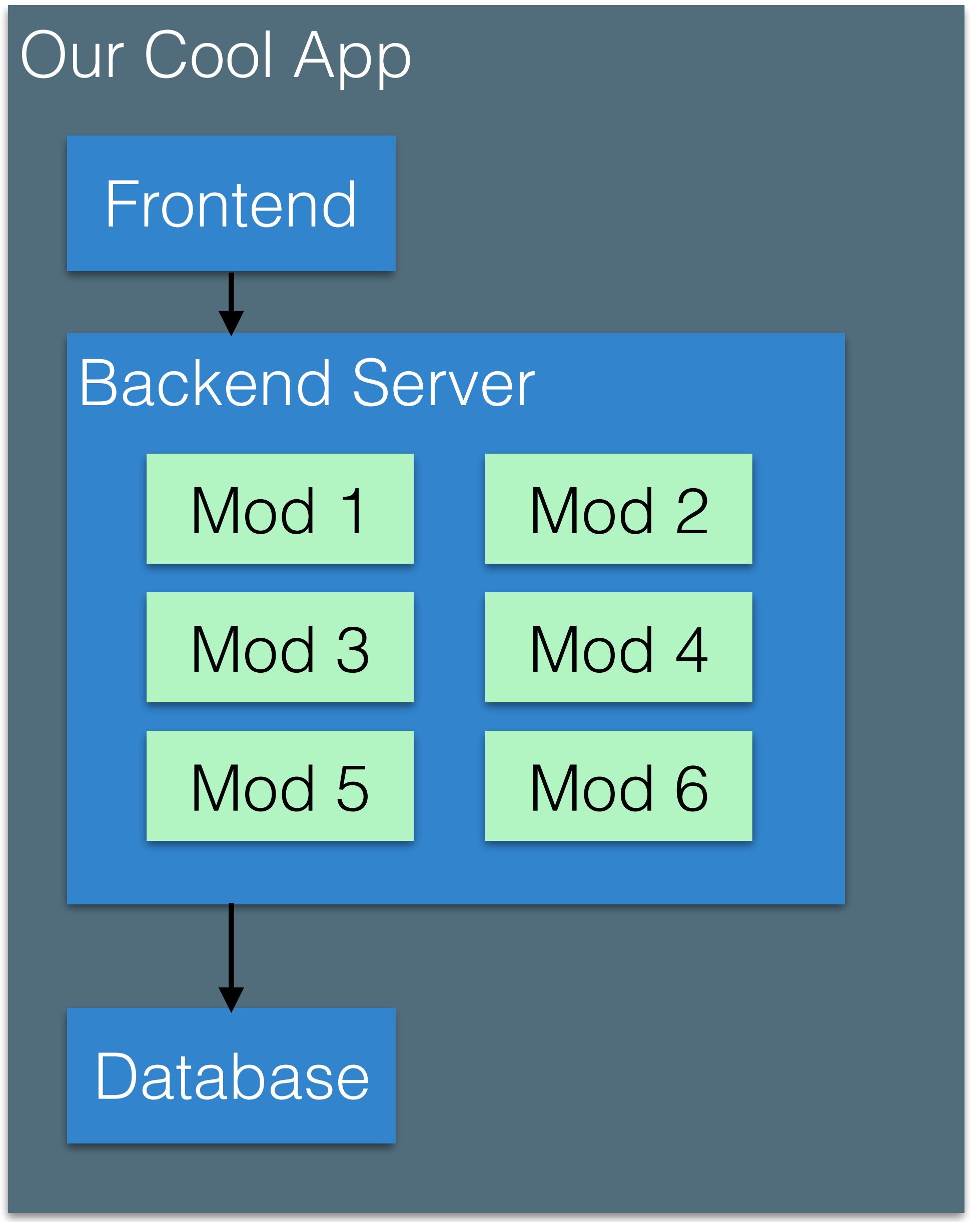
How do we build big apps?



Basic todo app with new feature to email todo reminders PLUS something to find events on Facebook and create Todos for them

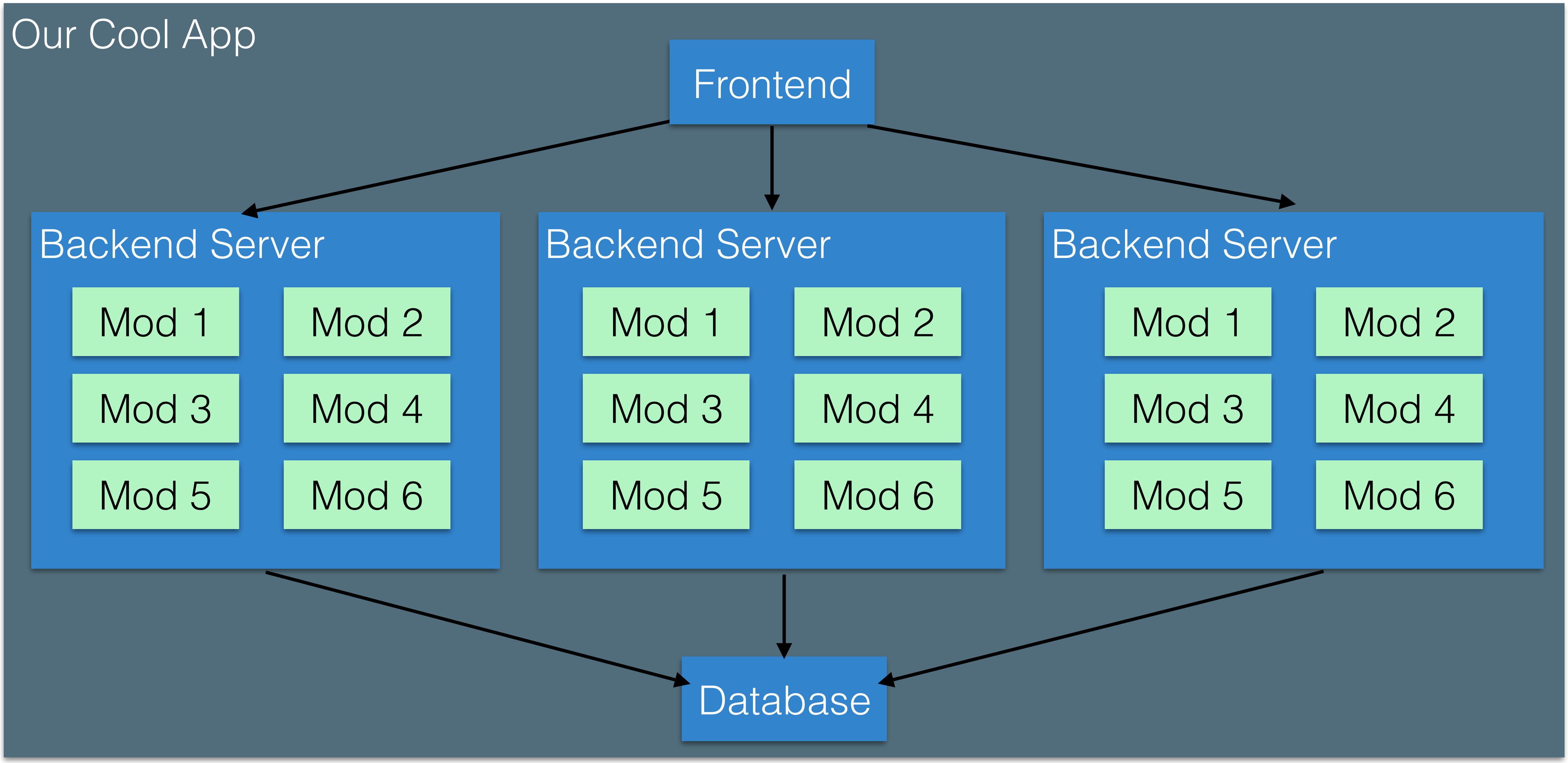
But we’re smart, and learned about modules, so our backend isn’t total spaghetti but rather…

How do we build big apps?



Sweet: Our backend is not an unorganized mess, but instead just modules. Now how do we scale it? Run multiple backends?

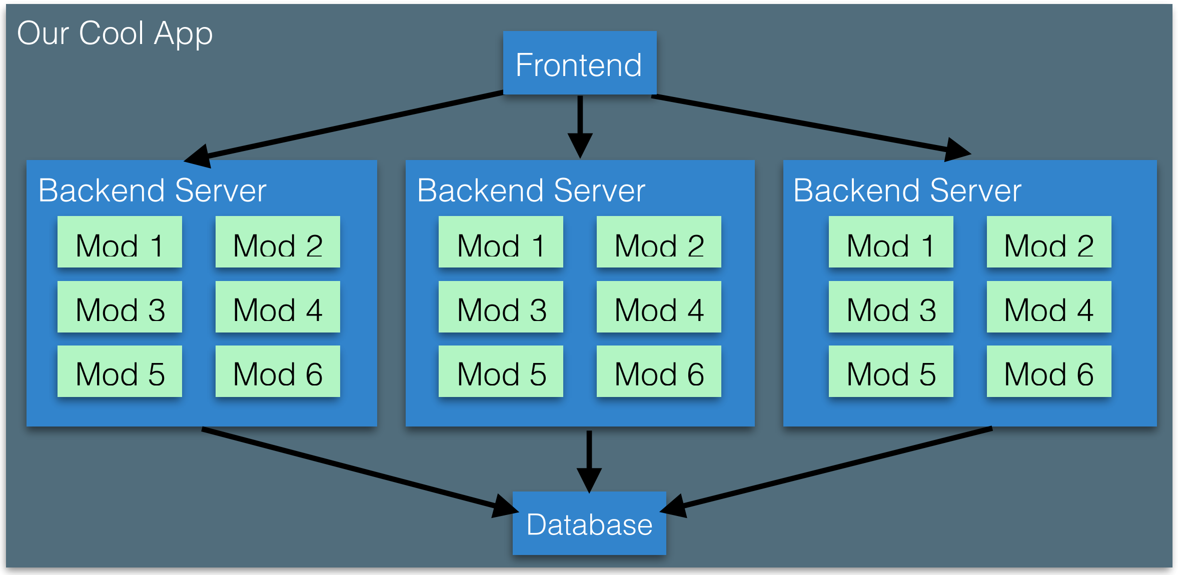
Now how do we scale it?



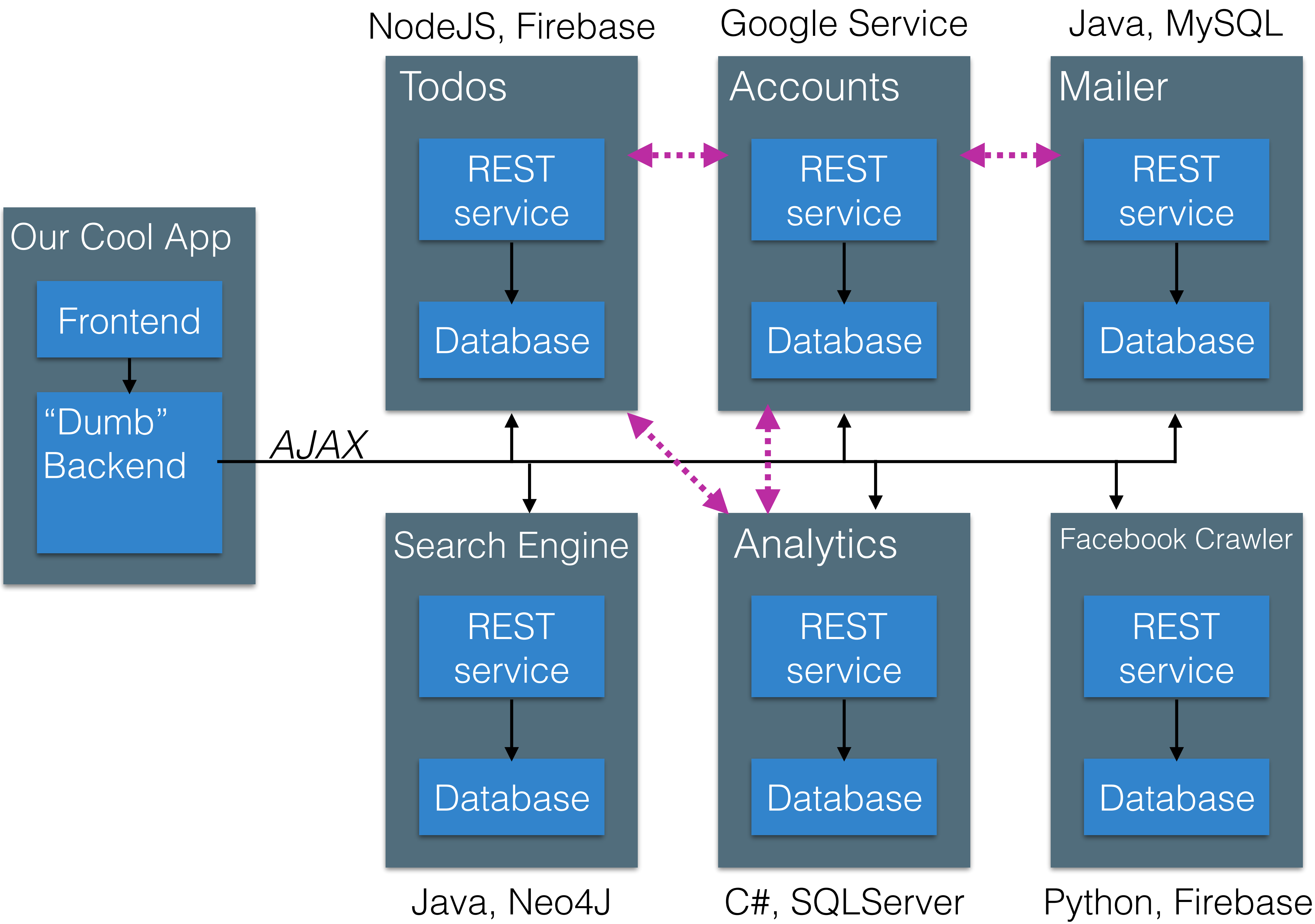
We run multiple copies of the backend, each with each of the modules

What's wrong with this picture?

- This is called the “monolithic” app
- If we need 100 servers…
- Each server will have to run EACH module
- What if we need more of some modules than others?
- How do we update individual modules?
- Do all modules need to use the same DB and language, runtime etc?

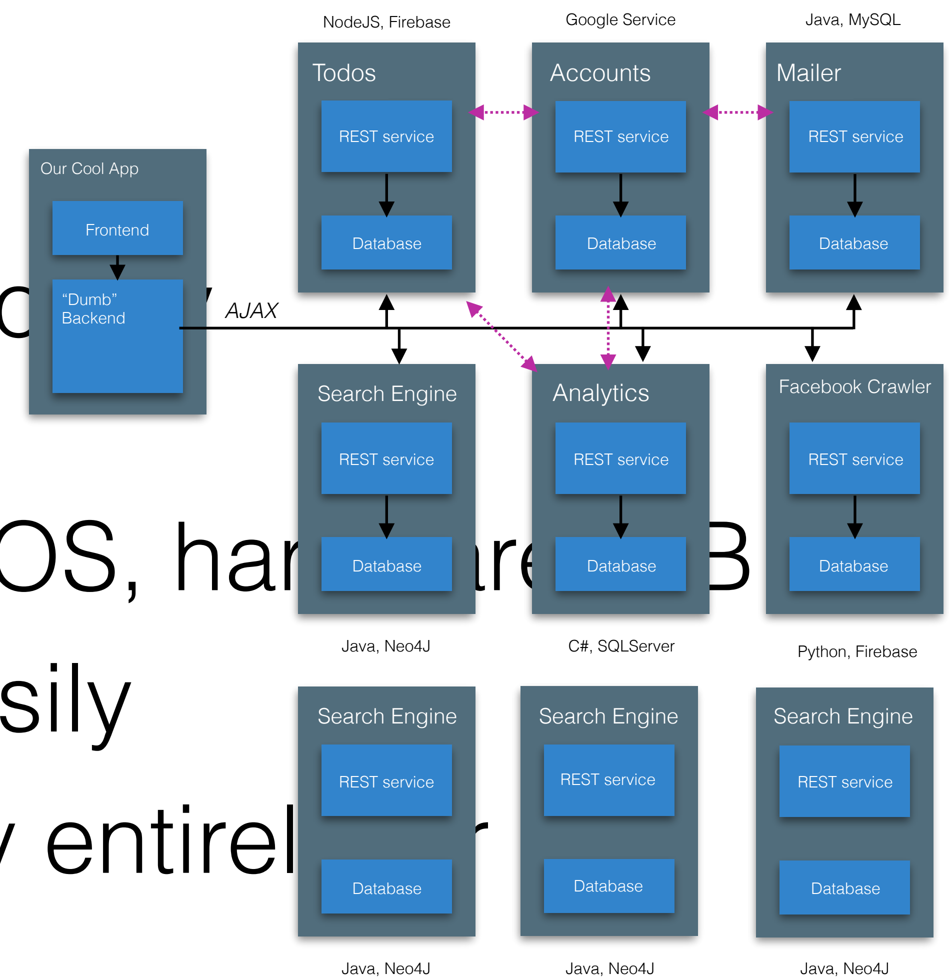


Microservices



What's good about this picture?

- Spaghetti is contained
- Components can be developed to independently
  - Different languages, runtimes, OS, har re B
- Components can be replaced easily
  - Could even change technology entirel r legacy service)
- Can scale individual components at different rates
  - Components may require different levels of resources



Requirements for successful microservices

- 1 component = 1 service
- 1 business use case = 1 component
- Smart endpoints, dumb pipes
- Decentralized governance
- Decentralized data management
- Infrastructure automation
- Design for failure
- Evolutionary design

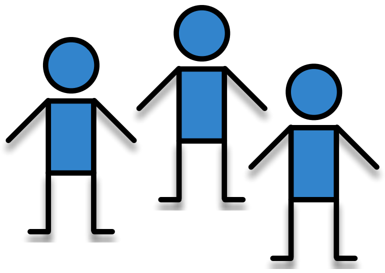
How big is a component?

- Metaphor: Building a stereo system
- Components are independently replaceable
- Components are independently updatable
- This means that they can be also independently developed, tested, etc
- Components can be built as:
  - Library (e.g. module)
  - Service (e.g. web service)

Components as Libraries or Services?

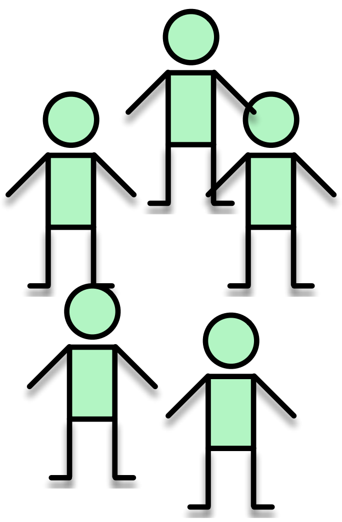
- Microservices says 1 service per component
- This means that we can:
  - Develop them independently
  - Upgrade the independently
  - Have ZERO coupling between components, aside from their shared interface

Organization around business capabilities



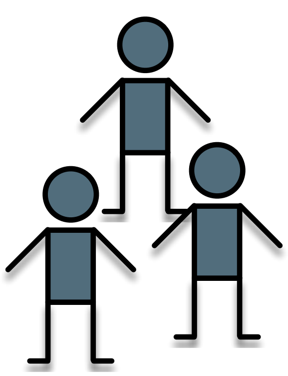
Frontend

Orders, shipping, catalog



Backend

Orders, shipping, catalog



Database

Orders, shipping, catalog

Classic teams:
1 team per "tier"

Organization around business capabilities

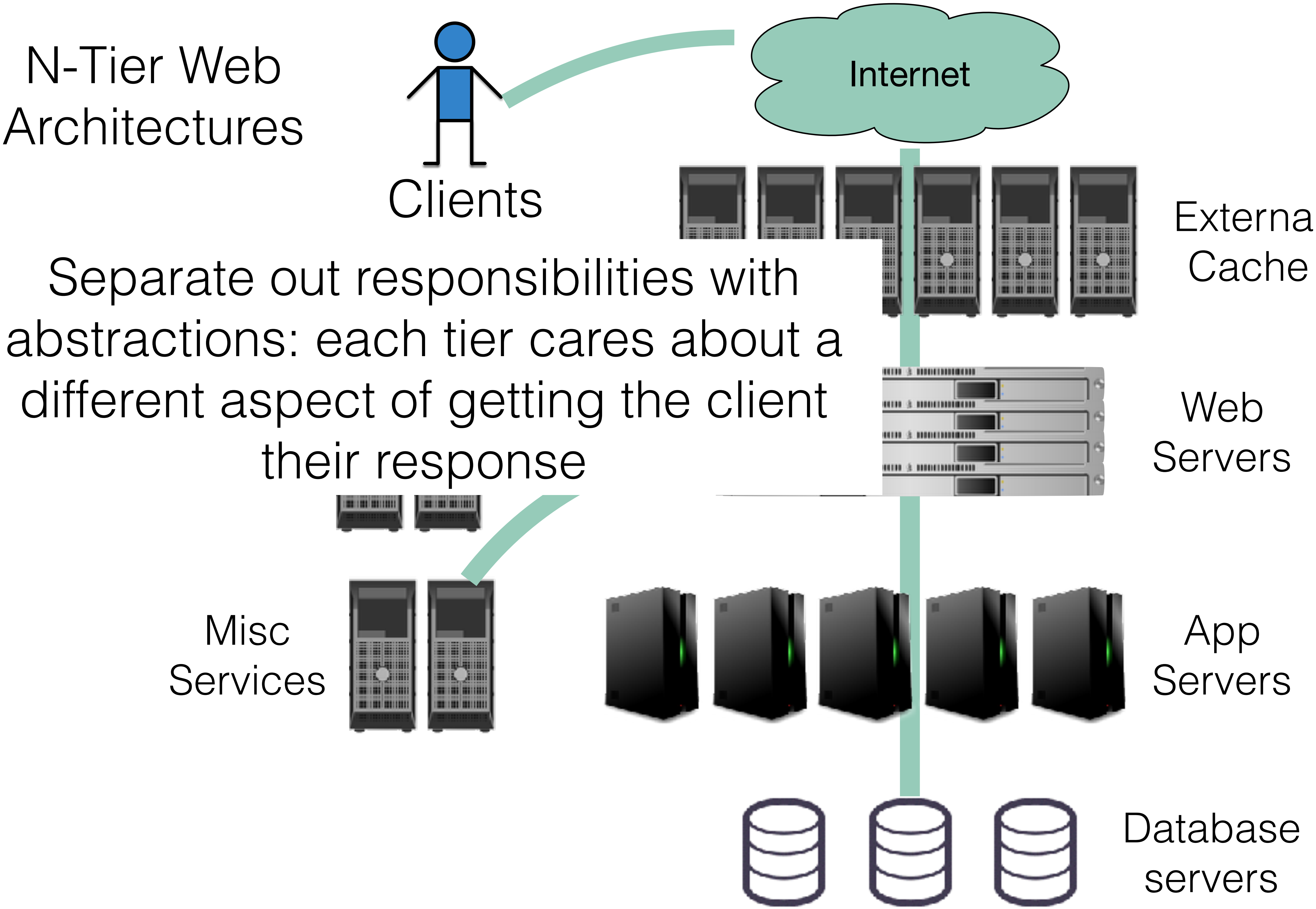
Example: Amazon

Teams can focus on one business task
And be responsible directly to users

“Full Stack”

“2 pizza teams”

Real Architectures



Separate out responsibilities with abstractions: each tier cares about a different aspect of getting the client their response

Abstracting the tiers

- Take, for instance, this *internal cache*
- Can we build one really good internal cache, and use it for all of our problems?
- What is a reasonable model for the cache?
  - Partition: yes (get more RAM to use from other servers)
  - Replicate: NO (don't care about crash-failures)
  - Consistency: Problem shouldn't arise (aside from figuring out keys)

How much more can we abstract our system?

- At its most basic… what does a program in a distributed system look like?
  - It runs concurrently on multiple nodes
  - Those nodes are connected by some network (which surely isn't perfectly reliable)
  - There is no shared memory or clock
- So…
  - Knowledge can be localized to a node
  - Nodes can fail/recover independently
  - Messages can be delayed or lost
  - Clocks are not necessarily synchronized -> hard to identify global order

Back to reality

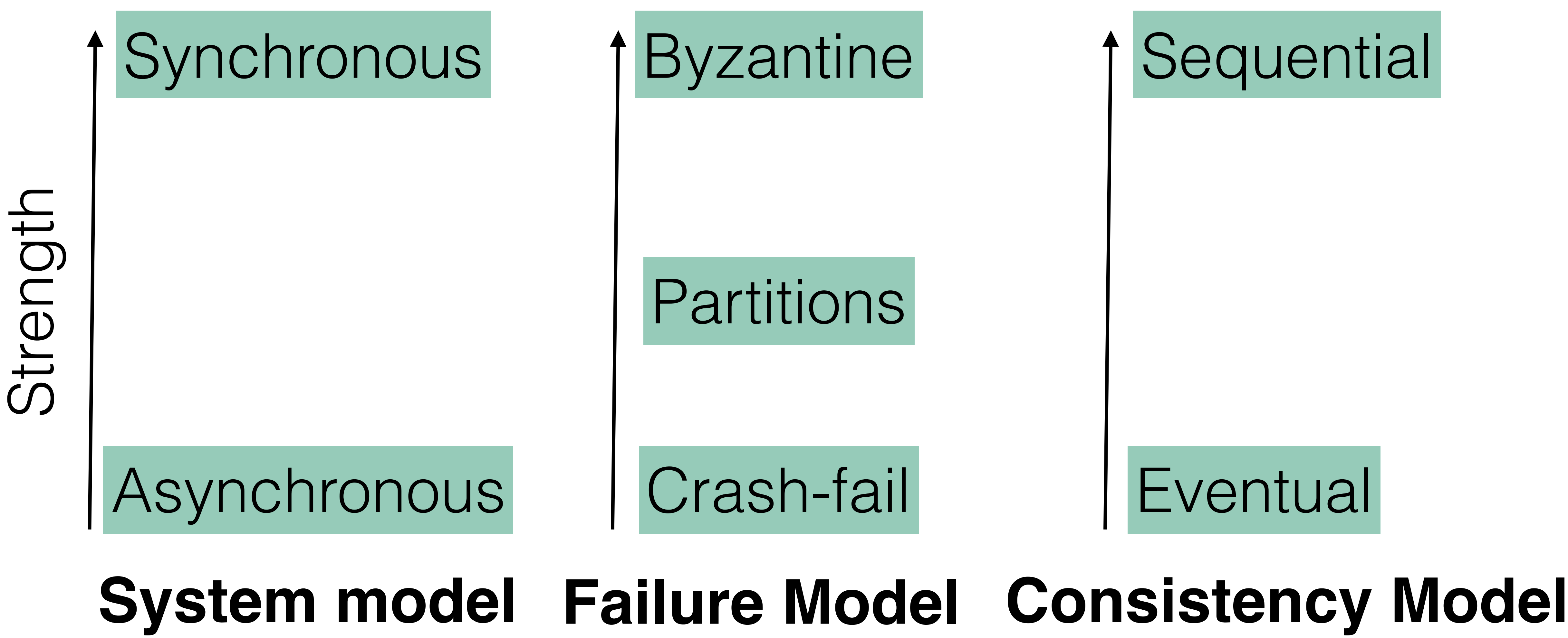
- That's a little TOO abstract - given that system, how can we define a good way to build one?
- In practice, we need to make assumptions about:
  - Node capabilities, and how they fail
  - Communication links, and how they fail
  - Properties of the overall system (e.g. assumptions about time and order)

Designing and Building Distributed Systems

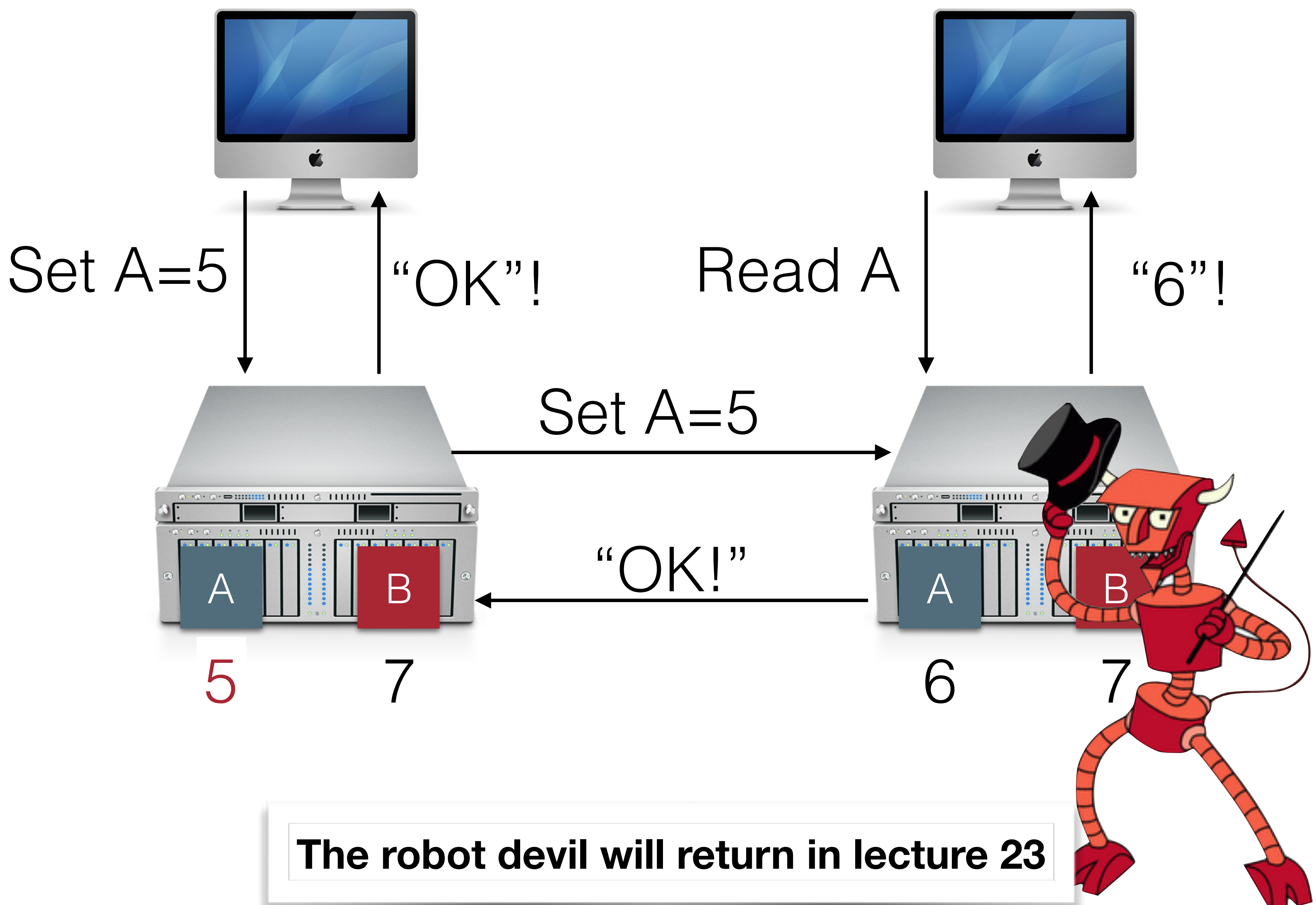
To help design our algorithms and systems, we tend to leverage abstractions and models to make assumptions

Generally: Stronger assumptions -> worse performance

Weaker assumptions -> more complicated



Byzantine Failures



The robot devil will return in lecture 23

Timing & Ordering Assumptions

- No matter what, there will be *some* latency between nodes processing the same thing
- What model do we assume though?
- Synchronous
  - Processes execute in lock-step
  - We (the designers) have a known upper bound on message transmission delay
  - Each process (somehow) maintains an accurate clock
- Asynchronous
  - Opposite - processes can run out of order, network arbitrarily delayed

Modeling network transmissions

- Assuming how long it can take a message to be delivered helps us figure out what a failure is
- Assume (for instance), messages are always delivered (and never lost) within 1 sec of being sent
- Now, if no response received after 2 sec, we know remote host failed
- Typically NOT reasonable assumptions