Scalability & Availability

Load Balancing, Modern Web Servers, and Project 2

CS110L
Feb 28, 2022

Logistics (reminder)

- Project 2 released; due on the last day of class.
  - This is the last assignment! You're in the home stretch!!
  - Encouraged to work with others (groups of 2-3) and/or discuss generally
  - Let me know how I can help!
    - My office hours are "by appointment"; please don't hesitate to reach out.
- Today: project 2 intro: what's a load balancer and how does it fit in to large-scale web content delivery systems?
- Posted on Canvas: optional content on channels & event driven programming
- Wednesday & Monday: time to work on project 2 (Weds is remote — Zoom link on Canvas if you want to chat; Monday I'll be here in-person.)
- Next Wednesday (last class): **guest talk from Ryan!**

Background Knowledge: Networking from 110

Assuming basic background: intro to networking at the 110 level

- Client/server model:
  - Two “roles” in a communication. Server “listens” for and responds to requests from clients.
- IP addresses: how routers in a network know where to send data
- DNS: domain name system — maps domain names to IP addresses
- Port numbers: distinguish multiple different networked programs on the same machine by assigning them unique port numbers
  - Assigned port numbers for common applications
  - Server program runs on machine; will be assigned a port number. Client hoping to connect to the server sends a request to the (IP address, port).

Lectures 20 and on from this year: https://web.stanford.edu/class/cs110/

Assuming basic background: intro to networking at the 110 level

- Sockets: Open connection to program on another machine => get back a "socket descriptor" (a file descriptor that refers to a socket).
  - Write to the socket descriptor to send data to the peer
  - Read from the socket descriptor to receive data from the peer
  - Note: C++ and Rust have abstractions around sockets — e.g., "socket streams" — to make it easier to write data to a socket.
- Protocols: a "shared", agreed-upon format in which to send data
  - E.g., HTTP 1.1 headers and bodies (data) have a particular format that both clients and servers know how to interpret
  - HTTP is the predominant protocol for web requests/responses

Lectures 20 and on from this year: https://web.stanford.edu/class/cs110/

Review: simple server setup from CS110

- Server is running service on known port number (e.g., HTTP on port 80 or HTTPS on port 443)
- Client looks up server's IP address using DNS
- Client connects to server's IP:port over the network
- Client and server each create a file descriptor (referring to a network socket) for communication with each other
- Client sends HTTP(S) requests to the server; server replies with HTTP(S) responses

Review: simple server setup from CS110

- *CS110: “Networking is like a function call & return.”*
  - *“Client sends requests — this is like a function call — then gets back responses — like return values from a function.”*

Scalability & Availability

Properties of large systems

- **Scalability**: How well can the system grow as demands increase over time?
  - An unscalable system will not be able to grow to meet demand no matter how many resources you throw at it
- **Availability**: How well is the system able to stay available and avoid downtime?
  - Becomes increasingly challenging as a system scales
  - If an server is available 99.99% of the time (down only 0.88 hours/year), a system not engineered for fault tolerance relying on 1,000 servers will be available 99.99% ^ 1000 = 90.48% of the time (down 834 hours/year)
- (There are many more properties we will not talk about today!)

Chaos engineering

- To design reliable networked systems, you must assume any part of the system can fail
- But in a complex system, it's hard to predict all failure modes
- Hard to learn how a system will fail until it fails
- *Testing: intentionally induce failure!*
  - *(in a controlled environment, where we can fix problems quickly, instead of having unexpected disasters at 3am)*
  - *Netflix philosophy of Chaos Engineering: "the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production."*

Simple server setup from CS110

- Is it scalable? (Why/why not?)
- Individual computers aren't scalable
  - Becomes exponentially more expensive as you try to upgrade performance
  - Much cheaper if we could use two machines with commodity performance than one machine with 2x performance
  - Internet traffic has grown far faster than hardware has increased in power. Hardware can't keep up even if our wallets could
- *Scale out, not up!*

Simple server setup from CS110

- Is it available? (Why/why not?)
- Hardly.
  - Server could get overloaded and run out of resources (memory, CPU time, file descriptors, etc)
  - Server could fail (system crashes, hardware fails, dog eats power cable, network outage, etc)

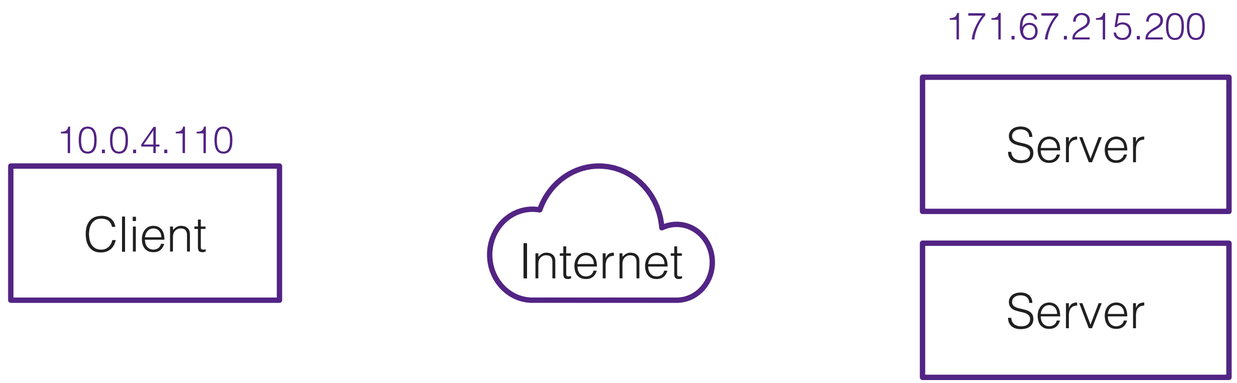
Distributed systems

- We want to distribute a system's functionality over a large number of servers to achieve scalability and availability
- These servers talk to each other using networking to collaborate on whatever problem we are trying to solve (e.g., delivering web content to clients)
- *Distributed systems go WAY beyond web servers, but I'm focusing on web servers here :)*

Scaling out

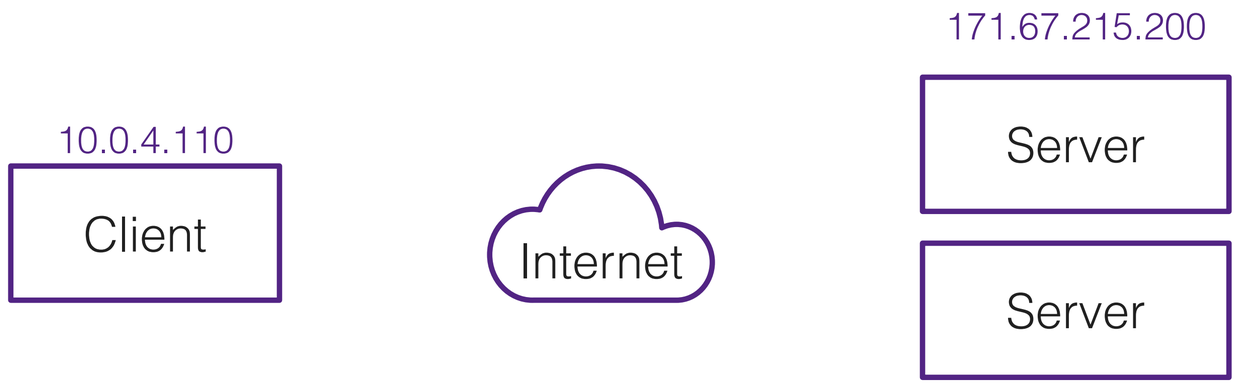
How can we design our system to make use of multiple servers?

Scaling out



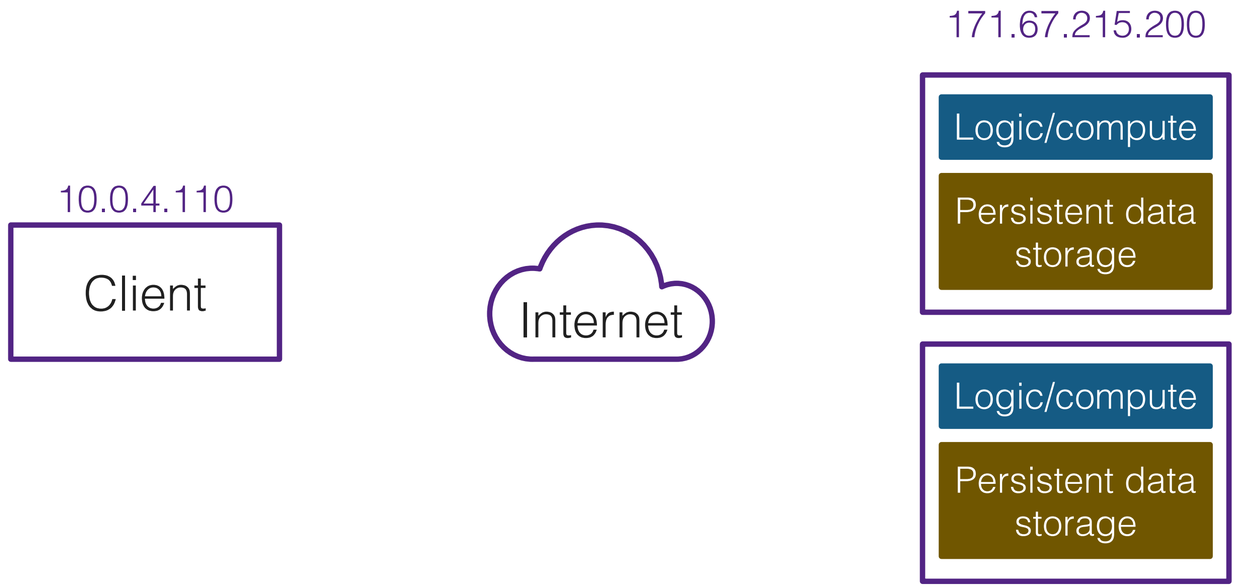
How can we design our system to make use of multiple servers?

Scaling out



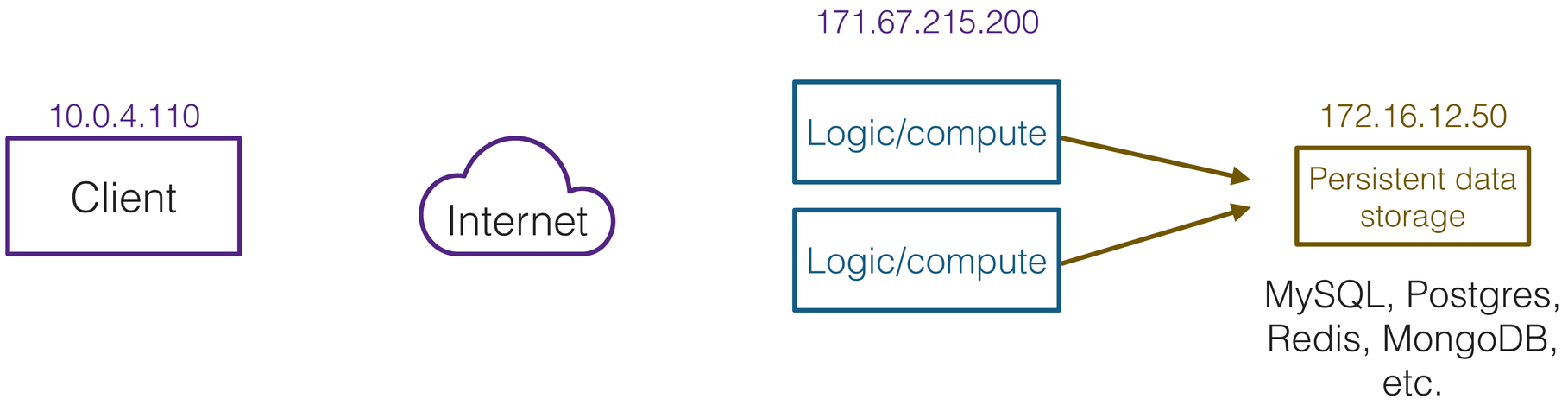
Simply duplicating our current setup won’t work.

Scaling out



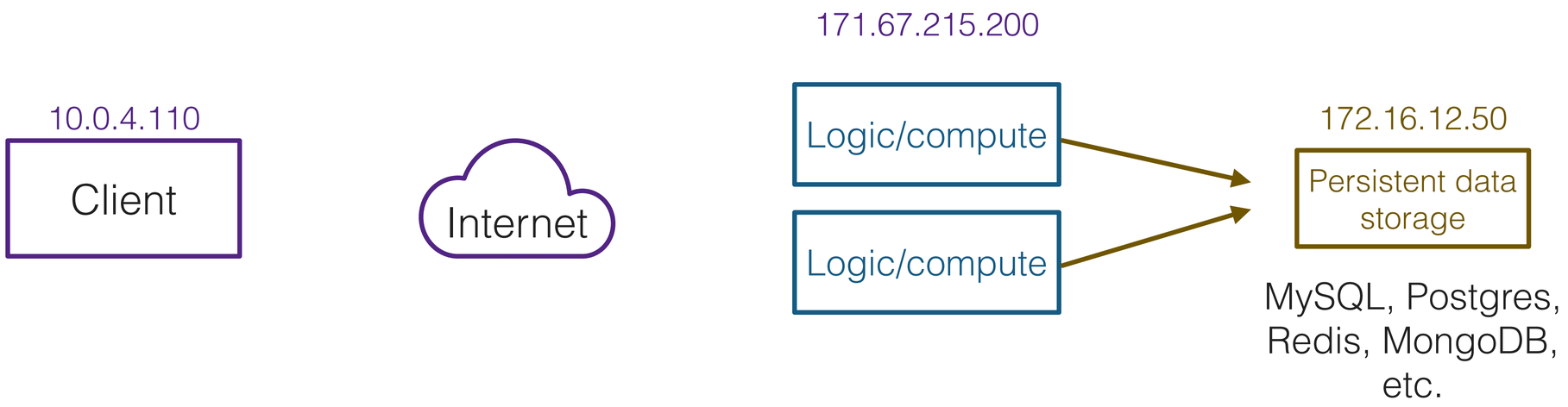
Simply duplicating our current setup won't work.
The duplicate servers would need to synchronize their data storage.
This is a very hard problem…

Scaling out



Simply duplicating our current setup won't work.
The duplicate servers would need to synchronize their data storage.
This is a very hard problem… that is already solved by databases!

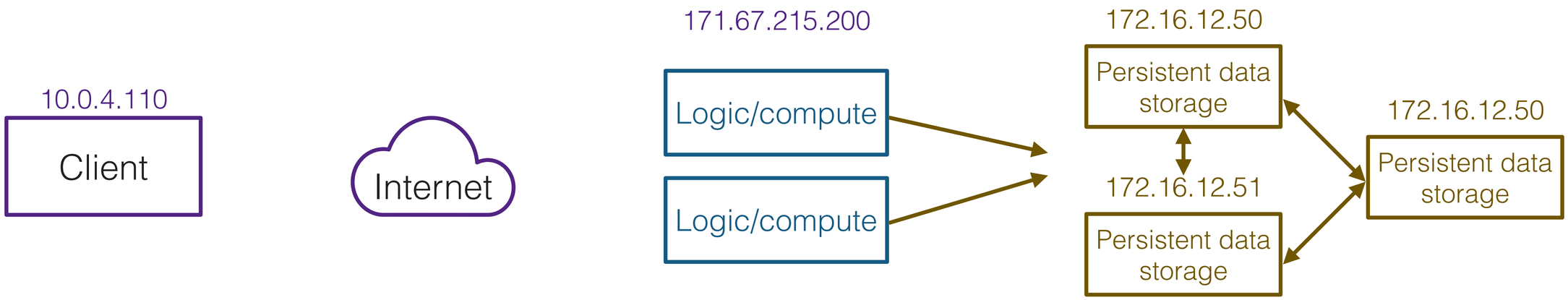
Scaling out



These database systems come with mechanisms to scale to multiple servers for reliability and performance

Scaling out

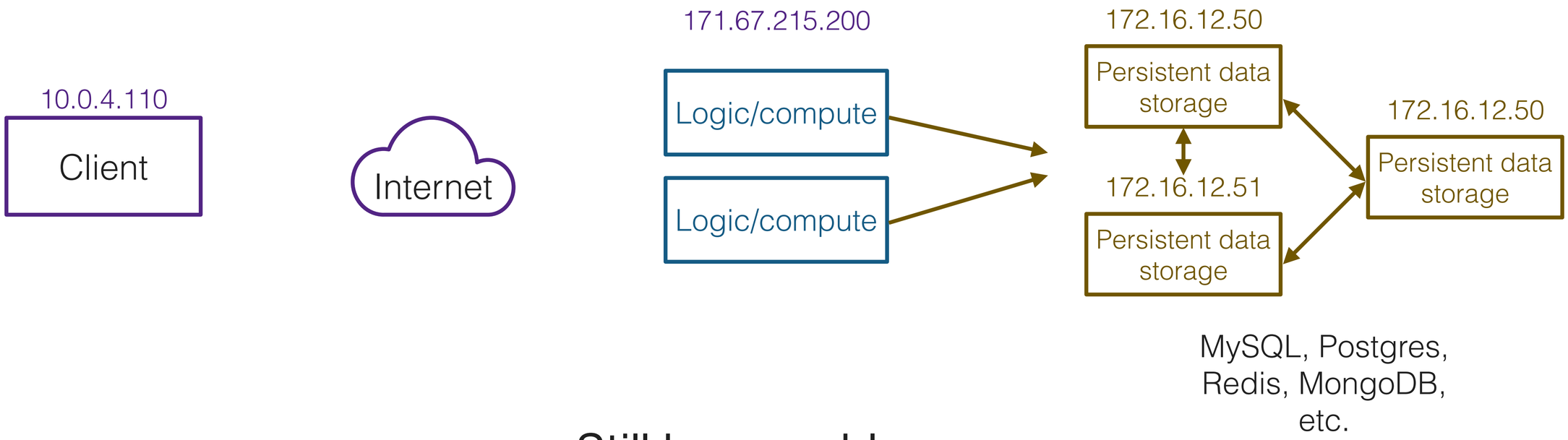
Take CS 245, CS 244B, CS149!



MySQL, Postgres,
Redis, MongoDB,
etc.

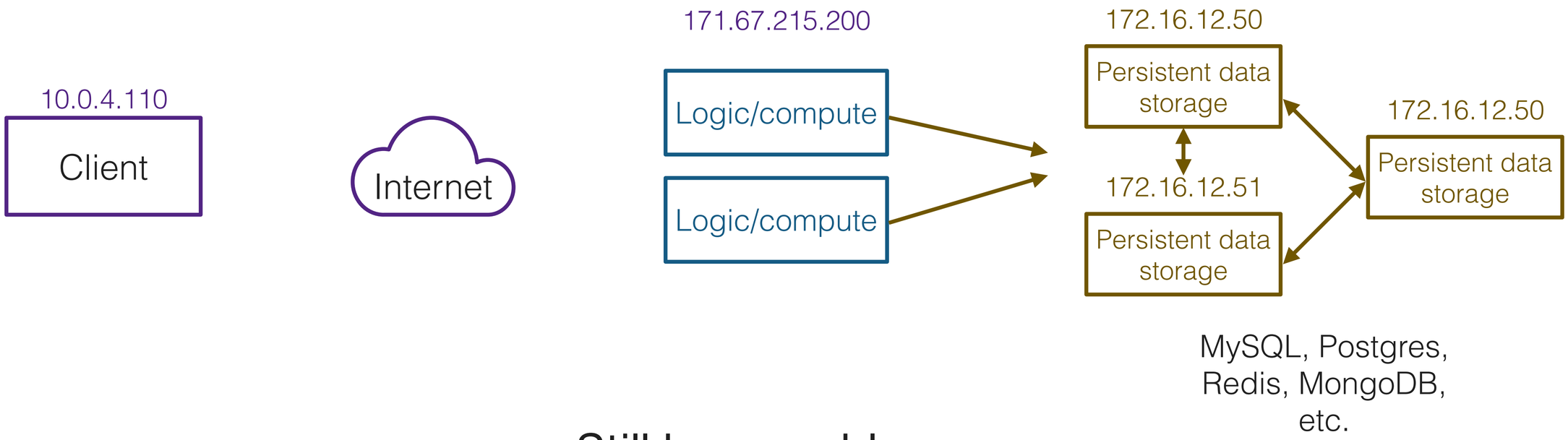
These database systems come with mechanisms to scale to multiple servers for reliability and performance

Scaling out



Still have problems:
Multiple servers, but only one IP

Scaling out

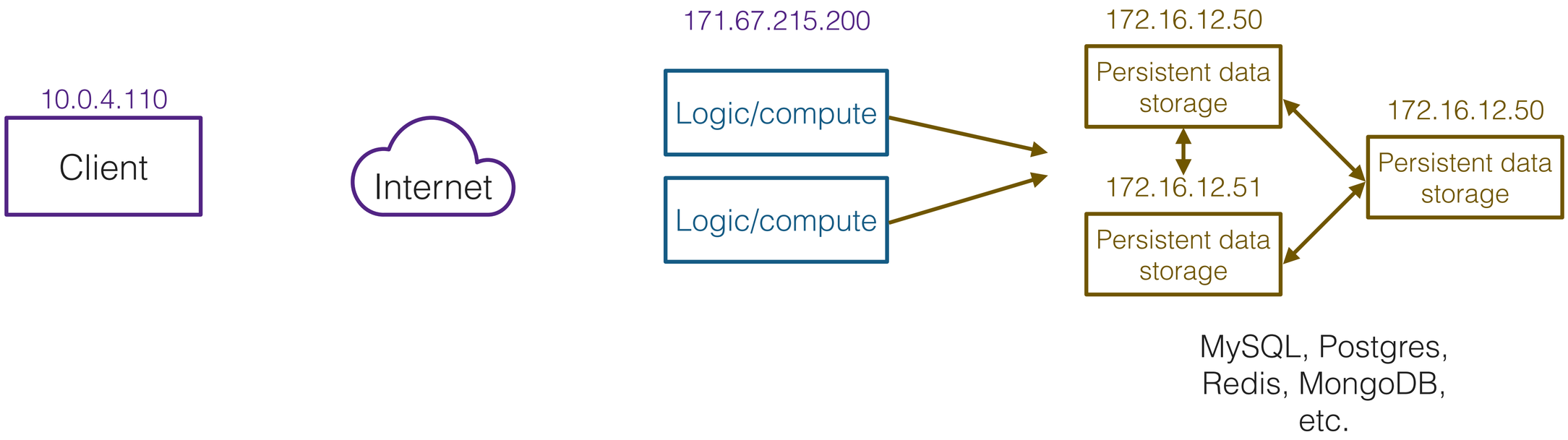


Still have problems:

Even if there were multiple IPs, how would every client know which server to connect to? What if a server gets overloaded, or goes down? How would clients get that information *quickly*?

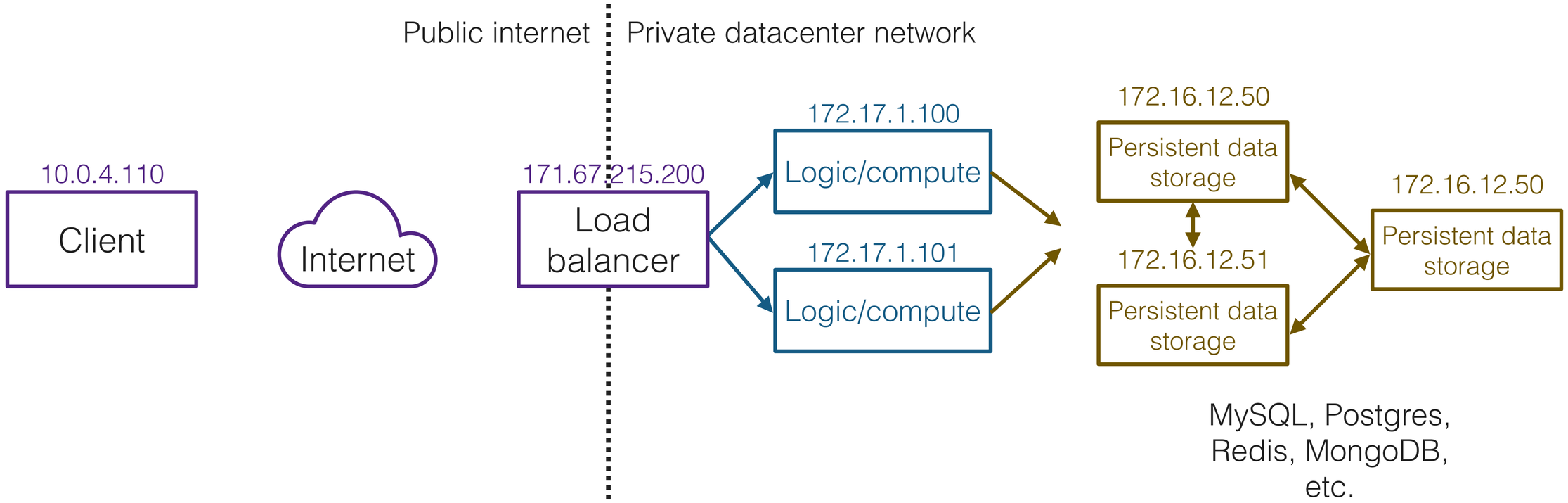
Load Balancers

Scaling out



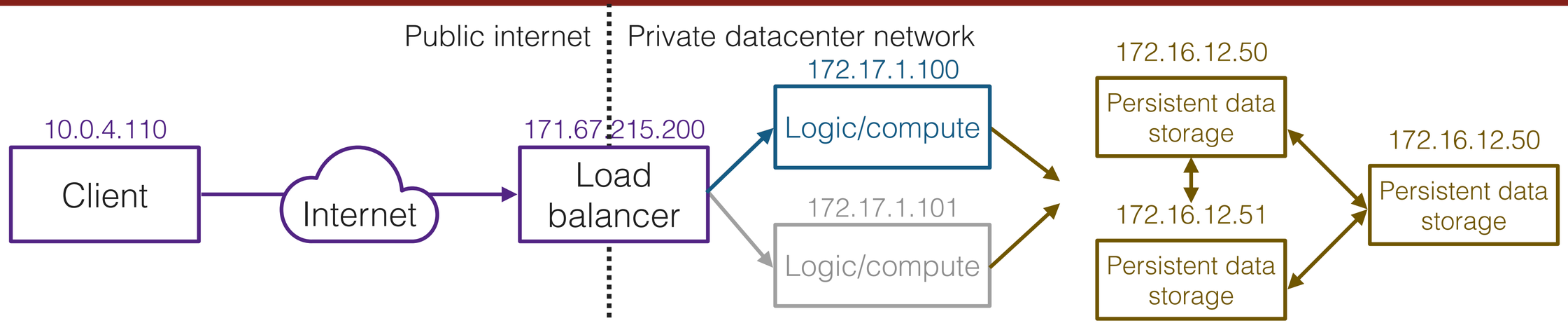
Load balancers: Distribute traffic across compute nodes
…and provide the client with the abstraction (illusion) of a single server

Scaling out



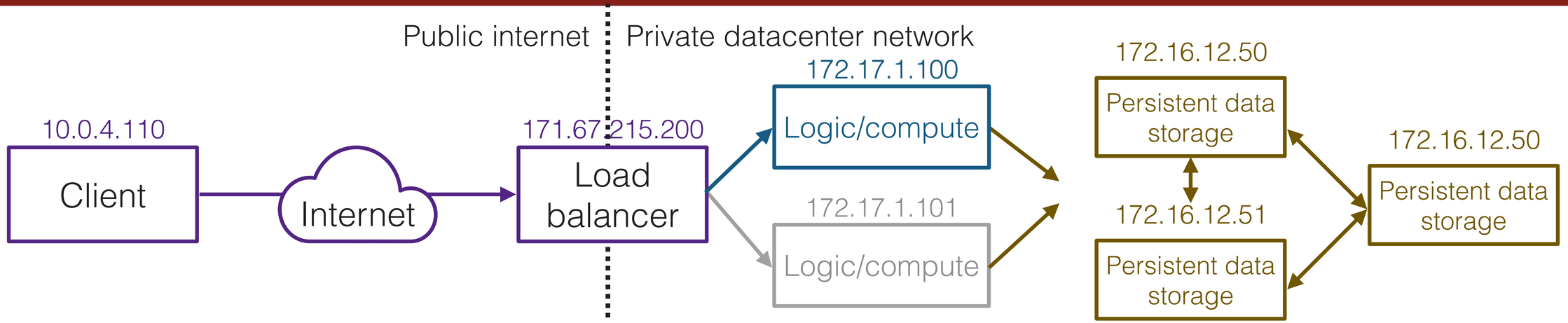
Load balancers: Distribute traffic across compute nodes

Load balancers



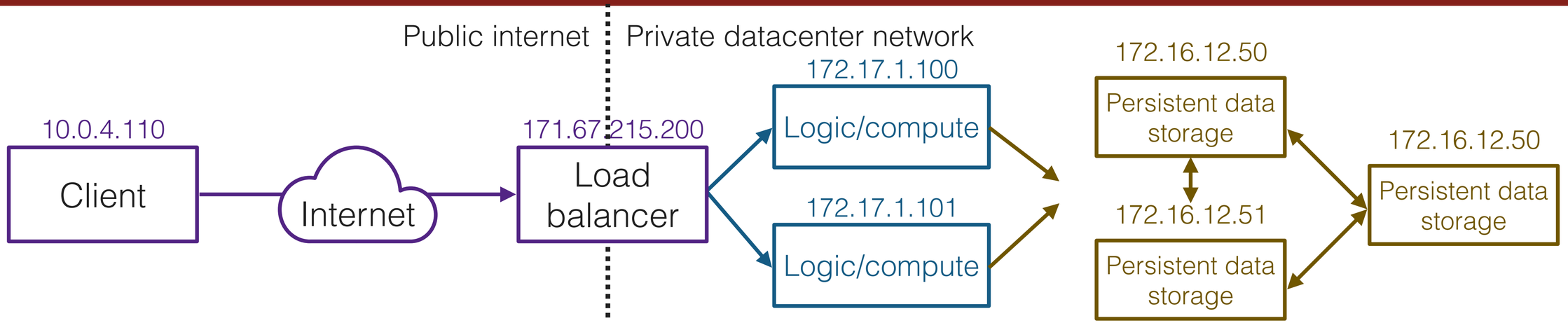
- When a client opens a connection to the load balancer, the load balancer selects a compute node and opens a connection to that compute node
  - Any traffic the client sends is relayed to the compute node. Any traffic the compute node sends is proxied back to the client
  - There are a variety of strategies for selecting the compute node (e.g. random selection, picking the one with the lowest load, round-robin, etc)

Load balancers



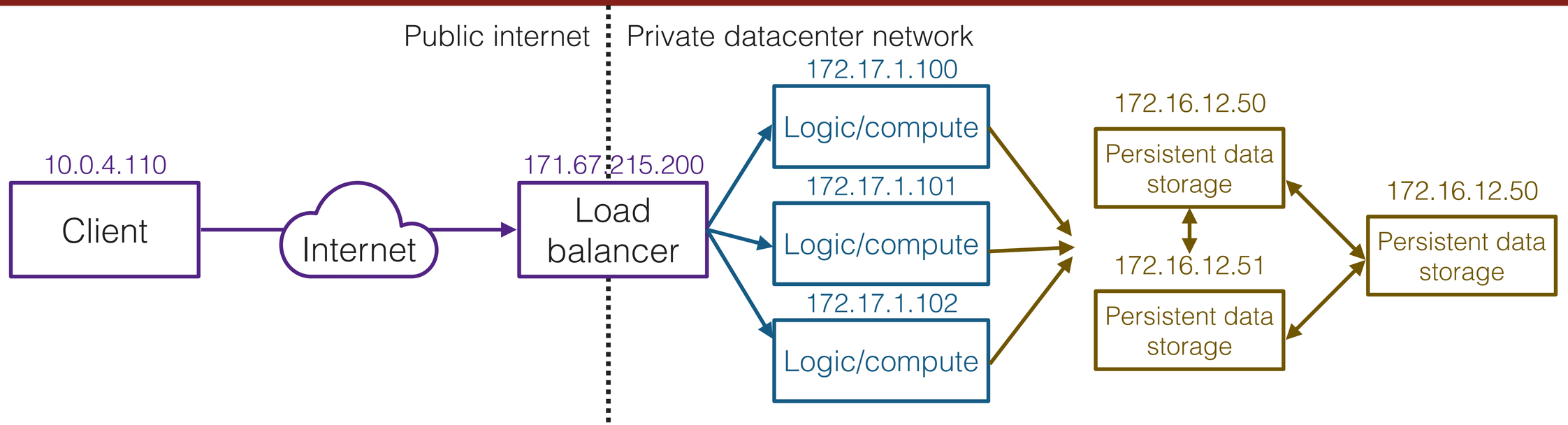
- The load balancer doesn't do anything else; anything resource-intensive is offloaded to the compute nodes. Consequently, load balancers can handle a large number of concurrent connections

Load balancers



- Scalability: If many clients are connecting, we can add more compute nodes

Load balancers



- Scalability: If many clients are connecting, we can add more compute nodes
- Availability: If one of the compute nodes fails, load balancer will detect that it isn't able to contact that server, and it can stop relaying traffic there
- Client never needs to know that our infrastructure is changing!
- **This is a huge improvement over the single server model!**

We can’t stop there

Aside: load balancing your load balancers

- We can't stop there
  - What if one load balancer fails?
  - What if a datacenter goes down?
  - If you're designing a networked system meant to serve content to users across the world, you probably can't just do this from one location?
  - What if you're a huge website? E.g., YouTube currently accounts for 15% of all internet traffic (source)... there's no way a single machine can handle that much traffic passing through it.
- There are many solutions to this, and these are huge topics (take CS249i!)
  - Two main sets of strategies: (1) use DNS; and (2) use routing.

Aside: load balancing your load balancers

- DNS based strategy 1: return multiple IP addresses for a given site, shuffling the order.
  - Client chooses the first one, moving down the list if IPs are unreachable.
  - *Can be simple and effective, but not very intelligent, and can introduce latency (e.g., client may need to "try" multiple IPs, or client may choose an IP that's overloaded). Exacerbated by caching in DNS infrastructure.*
- DNS based strategy 2: geographic routing: DNS server returns the IP address that it thinks is closest to the client, based on the client's IP.
  - *Problem: geographic guesses aren't always reliable*

Aside: load balancing your load balancers

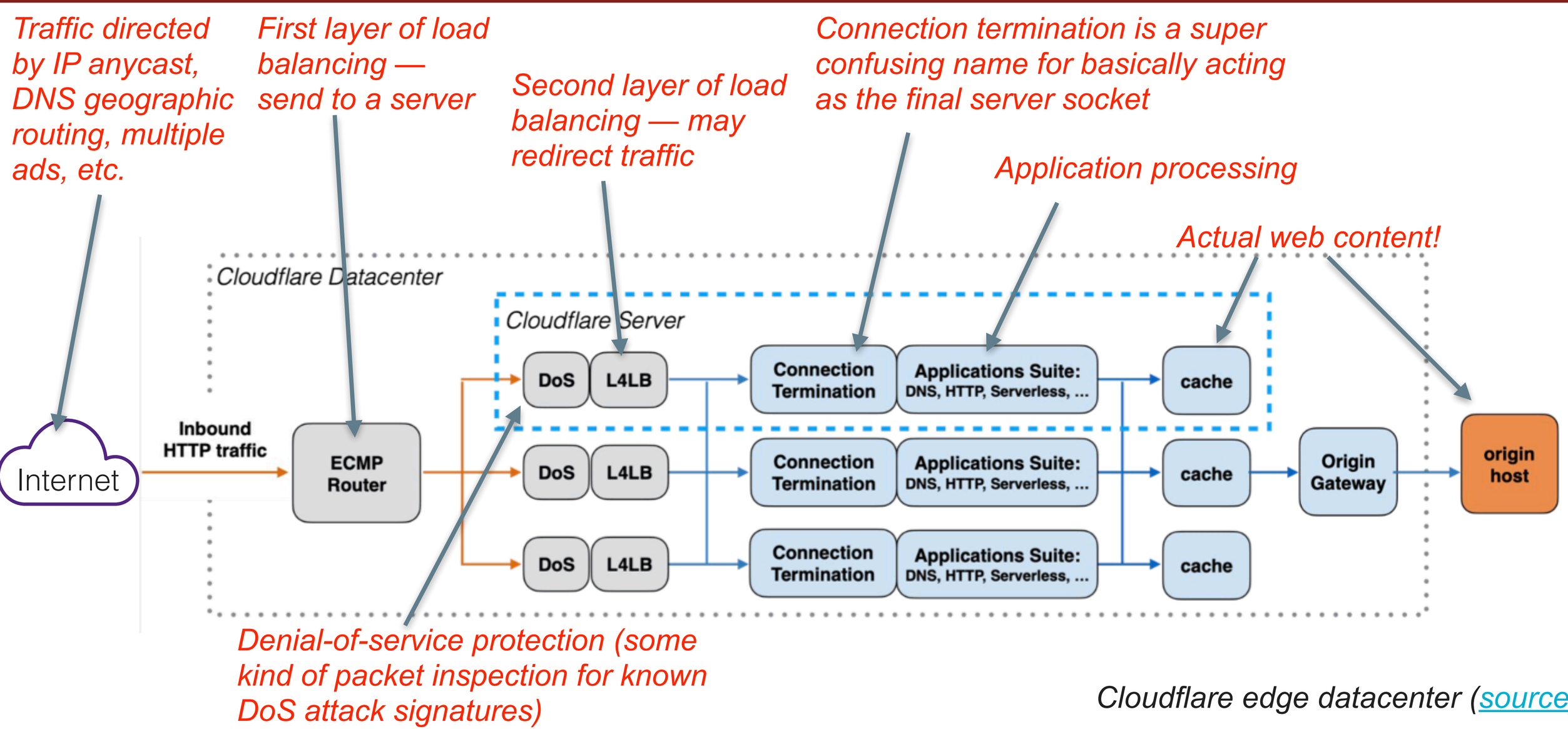
- Use routing: IP anycast
- **Background:**
- The Internet is split up into different *autonomous systems* —networks that represent a single domain of administrative control. Autonomous systems “announce” to the Internet what IP addresses they own.
- Routers figure out where to forward traffic based on these announcement.
- If routers receive multiple announcements for the same IP, they’ll save the one that seems closest (shortest path).

Aside: load balancing your load balancers

- IP Anycast: multiple servers announce to the Internet (via BGP) that they “own” a particular IP.
- Routers in other networks will save whatever one they think is closest in their routing table.
- When a router receives a packet destined for that IP, they’ll forward it according to the rules in their routing table.
- Across the world, routers might forward traffic to the same IP addresses to completely different places!

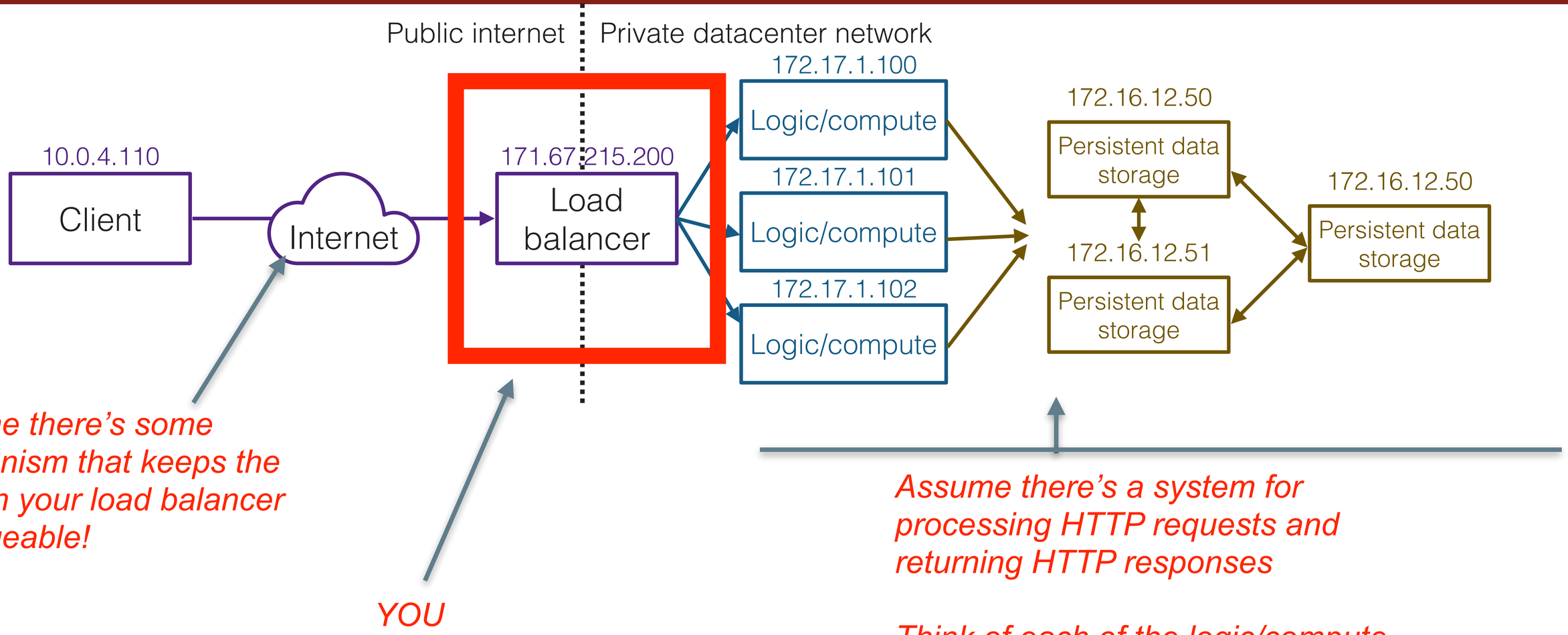
Recap: modern web servers & what you're building

Recap: modern web servers

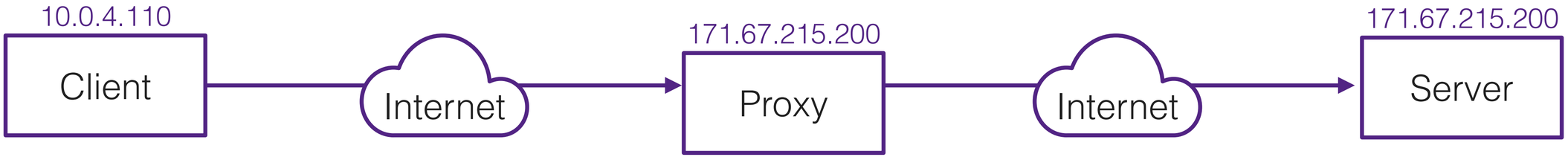


Cloudflare edge datacenter (source).

You're building one part of a huge system

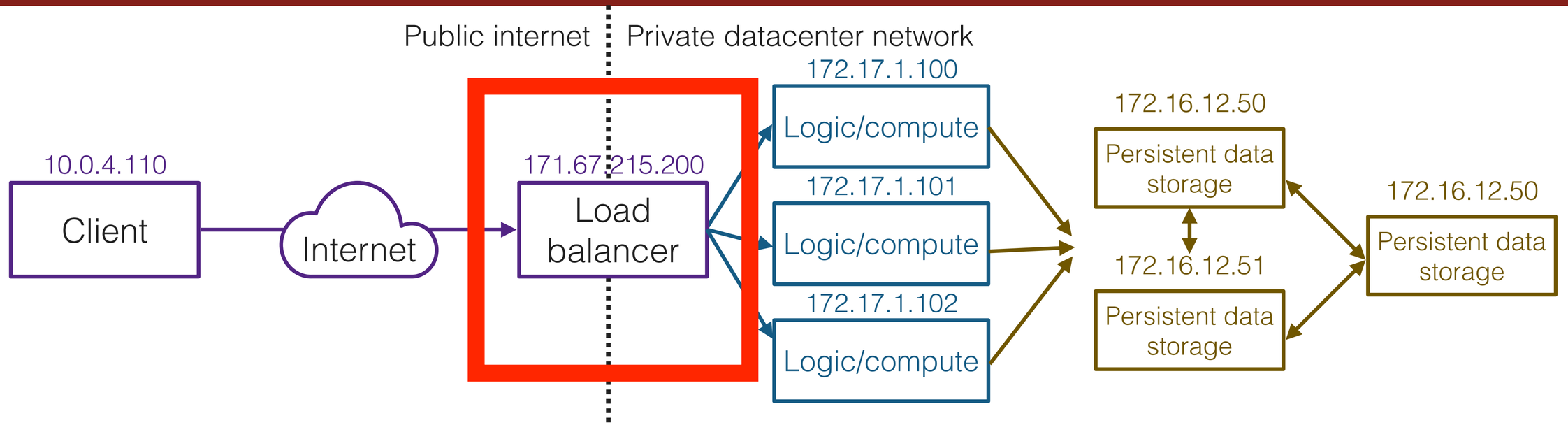


Note: you'll build a *client proxy* in CS110



- Client is configured so that all Internet traffic goes through the proxy
- Proxy receives connection setup requests & HTTP requests from the client, forwards them to the server, waits for responses from the server, and forwards those responses back to the client
- Proxy acts (kind of) as both a client and a server:
  - "Server": listens for and accepts incoming connections from the client
  - "Client": opens connection to server & sends HTTP requests

A load balancer is like a server-side proxy:



- Datacenter configured so that all traffic destined for a particular site goes through the load balancer
- Load balancer receives connection setup requests & HTTP requests from the client, forwards them to an “upstream server”, waits for responses, and forwards responses back to the client
- Load balancer acts (kind of) as both a client and a server:
  - “Server”: listens for and accepts incoming connections from the client
  - “Client”: opens connection to server & forwards those requests
- Instead of forwarding to some client-specified destination, *chooses* a destination.

Implementation milestones

Note: keeping this high-level

- This project is meant to be more open-ended than project 1
  - Part of this project is translating the *required specifications* into an actual *implementation.*
  - (This is a skill that all of us are still building, and it's no easier in an ~~annoying~~ unfamiliar programming language.)
- So: focusing on *what you're implementing*, not how you should do it.
- I highly recommend working together & discussing your challenges & implementation ideas, and please let me know how I can help!

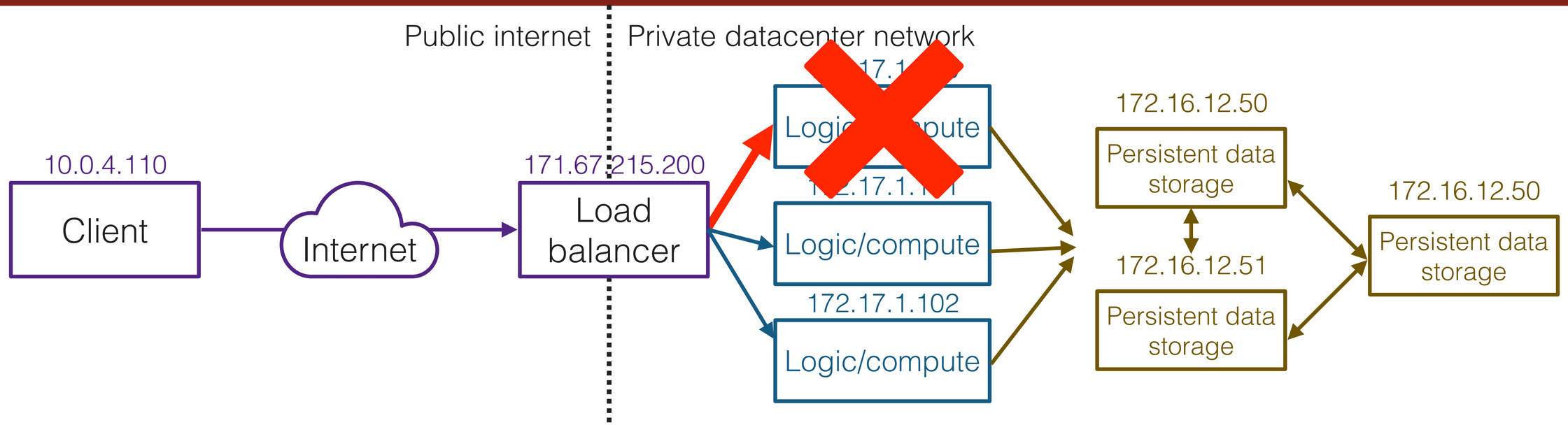
What you’re given

- Infrastructure for processing & sending HTTP requests and responses
  - Should look pretty similar to CS110 proxy assignment
- A proxy (again, should look pretty similar to CS110)
  - Creates a server socket to begin listening for connections
  - When connection comes in, creates a “client socket stream”: a data stream wrapping a socket, which can be used to write data to and read data from the client.
  - “Proxies” the connection (forwards client requests & server responses)

Milestone 1: Failover with Passive Health Checks

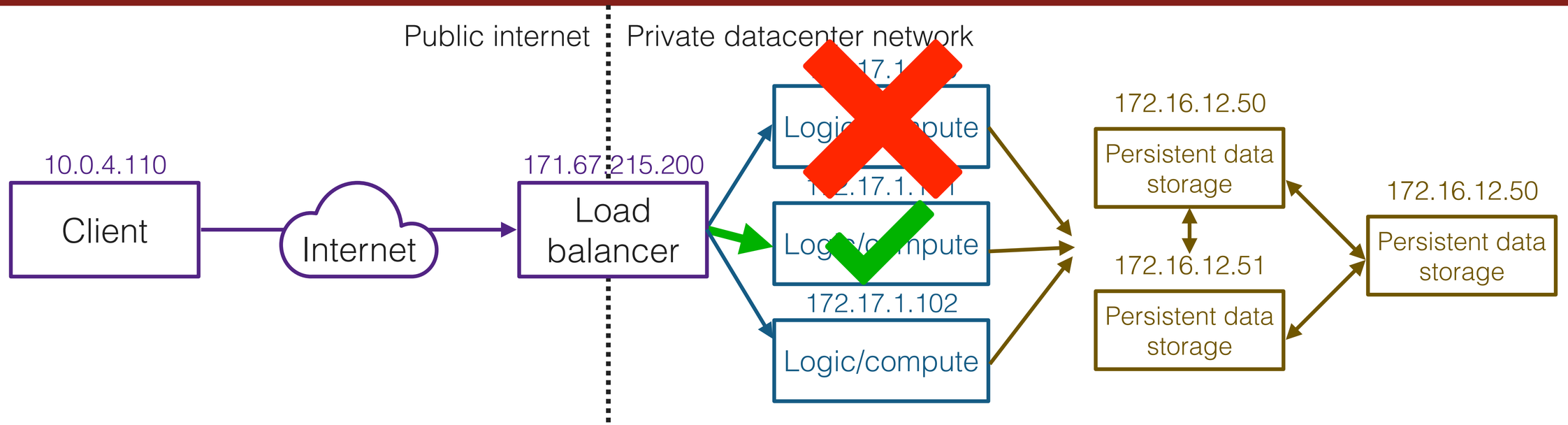
- In the starter code:
  - Load balancer connects to a random upstream server
  - If something fails, load balancer sends an error to the client
- We can do better!
  - Load balancer is part of a *datacenter* — and is given many upstream servers to choose from.
  - After this milestone: If load balancer fails to connect to the first upstream server it chooses, tries a different one. Only sends an error to the client if *all* upstream servers are dead. Keeps track of dead servers.

Milestone 1: Failover with Passive Health Checks



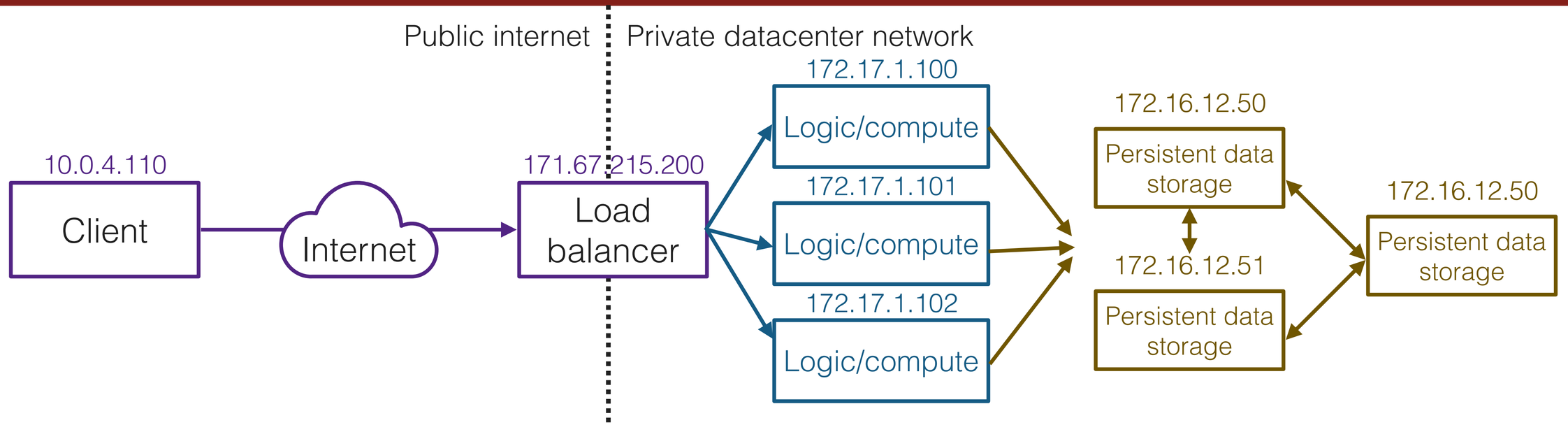
- Load balancer tries to connect to server A — fails
- Load balancer has other servers to try!

Milestone 1: Failover with Passive Health Checks



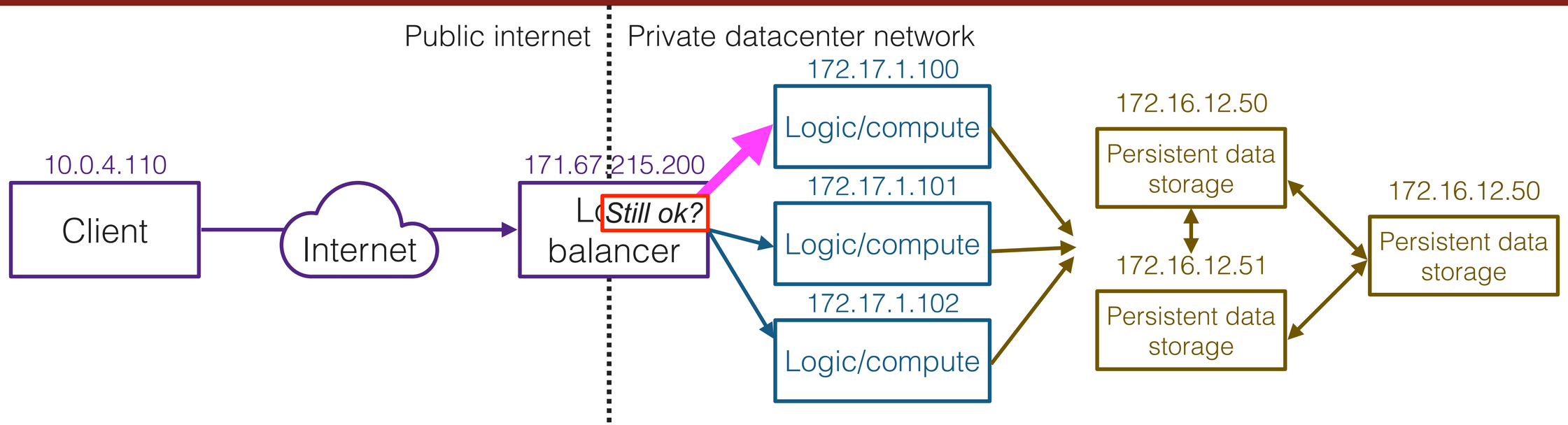
- Huge improvement to **availability**
- Note: for future requests, the LB should remember that this server is dead — don’t try to connect to it.

Milestone 2: Failover with active HTTP health checks



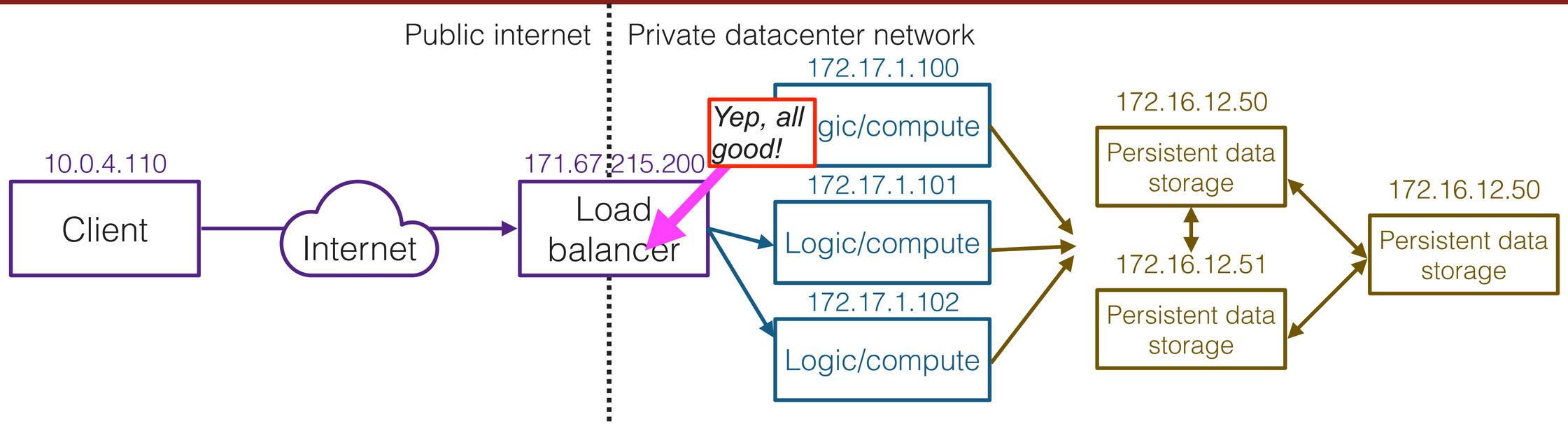
- Assume that all internal servers implement an *active health check* HTTP endpoint/path, given to your load balancer as part of configuration.
- E.g.: when they receive an HTTP request from your IP address for the “web page” “active-health-check”, they know to respond with an HTTP 200 (OK) message to indicate that they’re up and running.

Milestone 2: Failover with active HTTP health checks



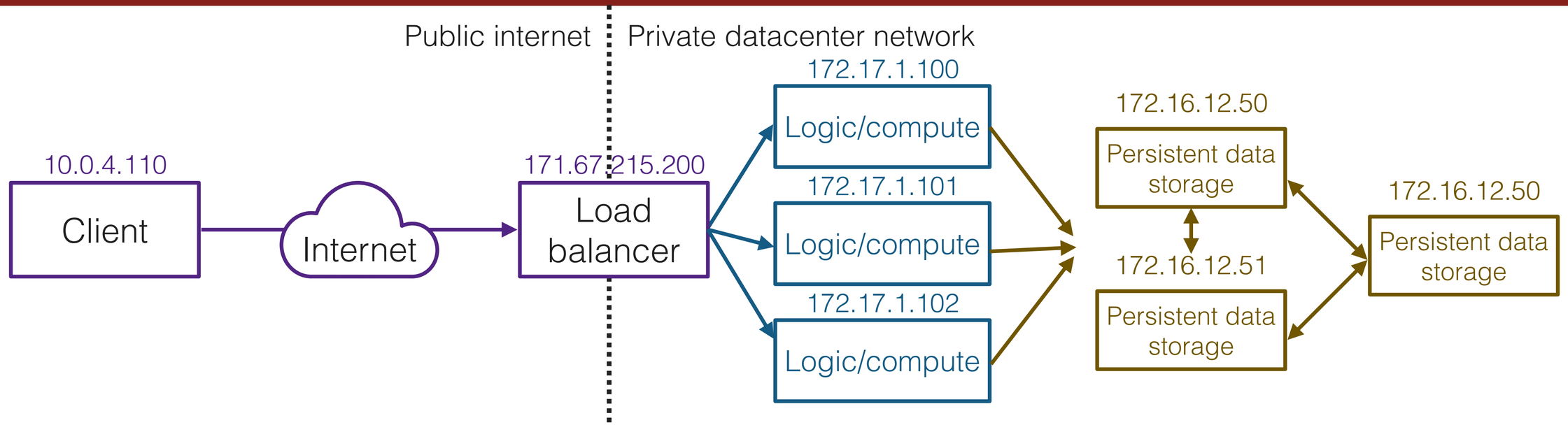
- Now: every *active_health_check* seconds, send an active “health check” request to each server.
- Based on the response (OK, not OK, or no response) — conclude that the server is alive or unresponsive. Record this info.

Milestone 2: Failover with active HTTP health checks



- Note: this should not impact your server's proxying performance. You'll need one or more threads to make this happen.

Milestone 3: Rate Limiting



- Protect your datacenter from being intentionally or unintentionally overwhelmed!
- Limit the rate at which clients can send requests

Milestone 3: Rate Limiting

- Many ways to do this — we recommend the *fixed window* algorithm
- Rate limiter tracks counters for each IP within a fixed time window (e.g., 12:00PM-12:01PM). At the end of the window, reset the counter.
- If a client exceeds `max_requests_per_window`, send a TOO_MANY_REQUESTS error response to the client.

Milestone 4: Add Multithreading

- Make it FAST
- Open-ended
- Baseline requirements: some amount of multithreading; meaningful speed and concurrency improvement over the sequential version, and ability to justify your design.
- Extension: do some real performance tuning!

Optional extensions

- Milestone 5: watch the event-driven programming material, and re-implement the load balancer using asynchronous I/O
- Watch the channels material and re-implement milestone 2 using *channels* instead of threads.
- Additional features:
  - Connection pooling
  - More sophisticated rate limiting
  - Other load balancing algorithms (rather than *random*)
  - Caching
  - Web application firewall
  - Others?

Bonus Rust syntax slides: Custom Errors

TL;DR: we use custom errors in project 2, and there's syntax in the starter code for how to use it. Think of this as, "there are different error value types, depending on what went wrong."

Recap: Results and Generics

- Reminder: a Result is an enum. It has two **variants**:
  - Ok(T) represents “success” and can contain a value of type *T* (generic).
  - Err(E) represents “error” and contains an error value of type *E* (generic)
- E.g., a **Result<int, String>** can either be an **Ok(int)** — Ok variant with integer inside — or an **Err(String)** — Err variant with a String inside.

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Recap: Results Contain Data

Example from the starter code: for setting up (“binding to”) a server socket.

Extract the value that was stored in the Ok, and store it in the variable “listener”

Match on the enum variant returned by `bind`: Ok(T) or Err(E)

```
let listener = match TcpListener::bind(&options.bind) {
        Ok(listener) => listener,
        Err(err) => {
            log::error!("Could not bind to {}: {}", options.bind, err);
            std::process::exit(1);
        }
    };
```

Extract the value that was stored in the Err, and store it in `err`; log it, then exit.

Custom Result types (custom error types)

- Emphasis: there is a *value* inside of the Err variant, and that value has a type.
- Often, libraries will have a **custom Result type** that requires a **custom error value type.**
- For example, the std::io library (I/O operations like read or write):
  - These operations often return a Result.
  - If the Result is **Err**, the **Err** will ***always*** contain an error value of type std::io::Error, a struct with some information about the error.
- This could get challenging! What if you're writing a function that calls lots of functions from other libraries, each of which has a different Err case? What Result type should your function return?

Custom errors

- **Error_chain** defines a *custom error type* for us.
- If a Result in our program is Err, the Err will encapsulate a value of this error type.
- This is like an enum that can be any of the variants listed here.
- **Result**s in our program are only generic over the data encapsulated in the Ok variant. Error type (returned in an Err) is fixed.

```
error_chain! {
    // Declare custom errors for use in this project
    errors {
        NoUpstreamServers
        BadResponse
    }
    // Automatically convert other errors, such as
    // std::io::Error, into our custom Error type
    foreign_links {
        IoError(std::io::Error);
        ResponseError(response::Error);
    }
}
```

- error_chain helps **translates error types from other libraries** into our error type!
- Will happen automatically if we specify under `foreign_links`.
- E.g., any `std::io::Error` in project 2 will automatically be converted into a `ResponseError`!

Custom errors: TL;DR

- Usage:

```
// return Err(ErrorKind::NoUpstreamServers.into())
```

  - What does this do?
    - Creates an ErrorKind (enum) of the variant NoUpstreamServers (which we've defined in our `error_chain`)
    - Converts it into our custom error value type *(into() is a type conversion)*
    - Creates a new Err (Result object), storing this ^ inside of it
- The ErrorKind variants are meant to be descriptive about what happened. Use them appropriately, and add more if you need to!

Vec::Retain

- We store the list of upstream servers (in the starter code) as a vector
  - You can change this, but you don't have to!
- If you keep using vectors, you might, at some point, need to remove an element with a particular value from a vector.
  - *E.g., you know that a server with a particular IP address is down — remove its IP address from a vector of "active" servers*
- We recommend doing this with retain:
  - Takes in a closure (function) that returns true or false; keeps only the elements for which the function returns `True`
  - (I.e., removes all elements not matching predicate)

Examples

```
let mut vec = vec![1, 2, 3, 4];
vec.retain(|&x| x % 2 == 0);
assert_eq!(vec, [2, 4]);
```

TCPStream and TCPListener

- TCP is a *transport layer* protocol that HTTP is built on top of (take CS144!)
- TCPStream:
  - This is a “socket stream” in CS110 terms
  - It’s a stream that wraps a socket
  - A stream is just an object that you can write data to and read data from
  - Here: write data to write to the socket; read data to read from the socket
- TCPListener:
  - Sets up a “server socket” that listens for incoming connections