Systems Design

Carlos Cuevas
Adjunct Lecturer - Brooklyn College

Typical Software Engineer Interview

Typically you'll be given several distinct rounds of interviews:

- Data Structures & Algorithms
  - Leetcode style questions
- Behavioral
  - "Tell me about a time you disagreed with someone…"
- Systems Design
  - Primarily asked of Senior Engineers but it's become increasingly common to ask even entry level engineers

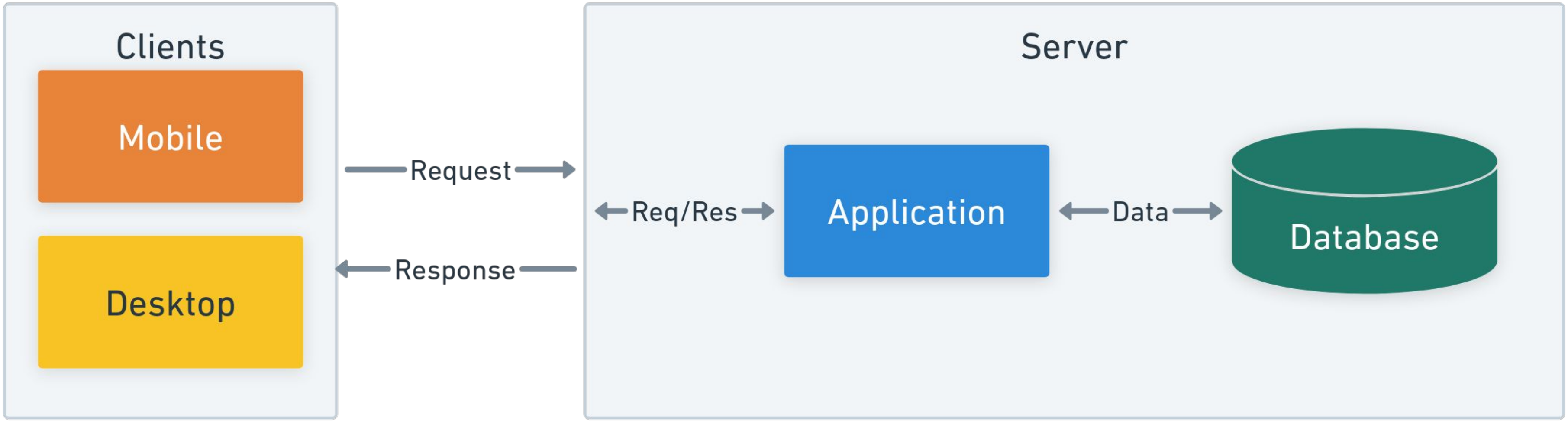
Systems Design Interview

- You'll be given a purposefully broad prompt.
- Design a system that accomplishes x. For example:
  - Chat System
  - URL Shortener
  - Youtube
  - Many Others
- It's on you to narrow scope as much as possible, but we'll get to that later.

Building Blocks

- Many systems have overlap
- Let's learn some common components of many large scale systems

Client - Server Model



How Does Software Talk to Other Software?

- What does it mean, exactly, for a client to make a request to the server?
- What does it mean, exactly, for a server to respond to a request?

Weather Application

- What might the request and response look like for a weather application hosted on the web?

- Request
  - Where do we send this request?
    - `http://www.weather-info.com/weather`

  - How?
    - HTTP GET method

  - How do we tell the app the location for which we want the weather?
    - `http://www.weather-info.com/weather`**`?zip=11210`**

- Response
  - Formatted data
    - JSON
      - `{ "temp": 76, "humidity": 40, "precip": false }`

Application Programming Interface (API)

- Set of rules and protocols that allows one software application to interact with another
- i.e. How one piece of software can communicate with another
- What's an API you've all had experience using?

Weather API

API Documentation

Endpoint	/weather
HTTP Method	GET
Parameters	integer zip
JSON Response	integer temp string humidity string precip

Request Code

```java
public static void main(String[] args) {
    String url = "/weather/?zip=11210";
    Response response = Request.Get(url).execute();
    String output = response
        .returnContent()
        .asString();
    System.out.println(output);
}
```

Response

```json
{
    "temp": 76,
    "humidity": 40,
    "precip": false
}
```

Weather API

Response Code

```
public static void respond(Request request) {
  String zip = getQueryParam("zip", request);

  String weatherData = getWeatherData(zip);

  Response response =
      Response.ok(weatherData, MediaType.APPLICATION_JSON).build();

  response.returnContent().writeTo(System.out);
}
```

What exactly is a Server?

- The term is overloaded

- Hardware
  - A physical* computer
  - on which **server software** is running

- Software
  - A process that is listening for requests and responding to those requests (i.e. serving)

* Could be:

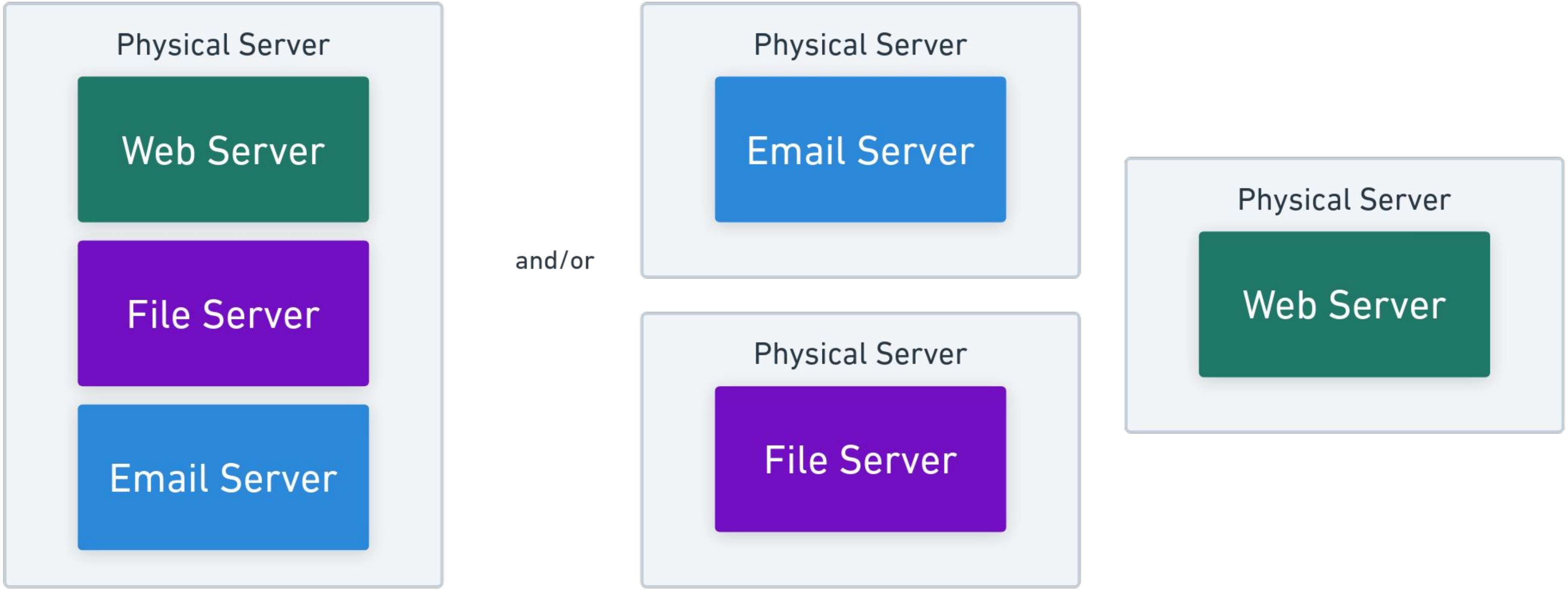
❖ Physical Computer
❖ Virtual Machine / Container
❖ Cloud Service

Server Software

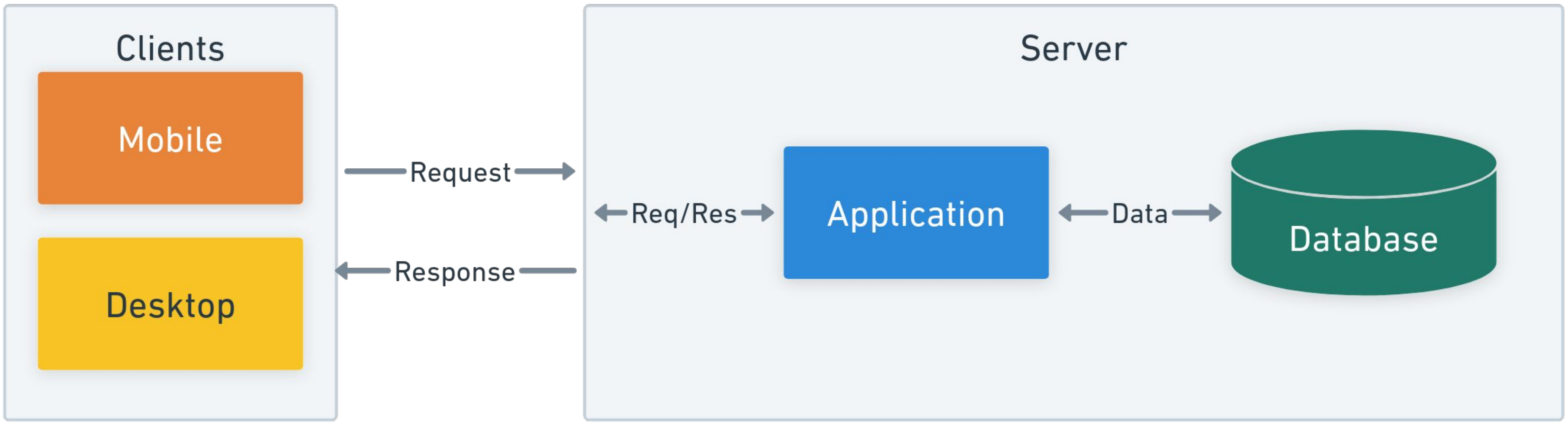
- Web / HTTP
  - Apache, Nginx
- File
  - FTP, Samba
- Email
  - Microsoft Exchange
- Database
  - MySQL, Oracle
- Many, many more

Hardware & Software

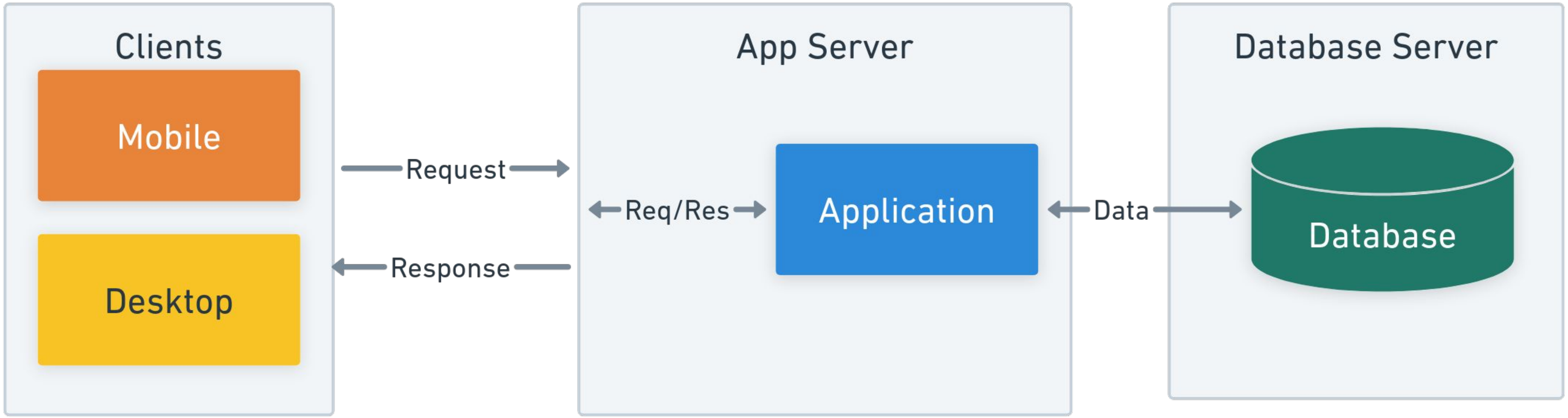
- One Physical Server can run multiple server applications
- Or, you can run multiple physical servers each running its own application.



Single Server

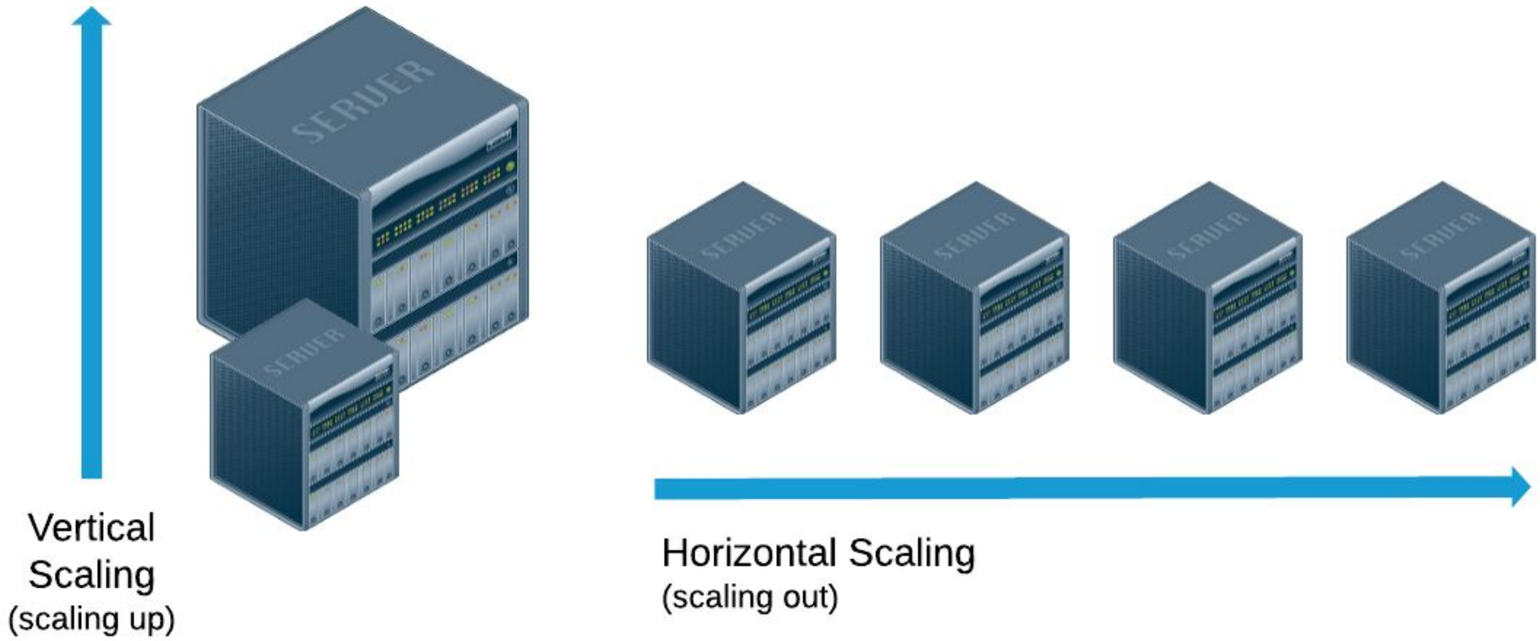


Multi-Server Setup



Horizontal vs Vertical Scaling

- Vertical: a more powerful computer
- Horizontal: more computers



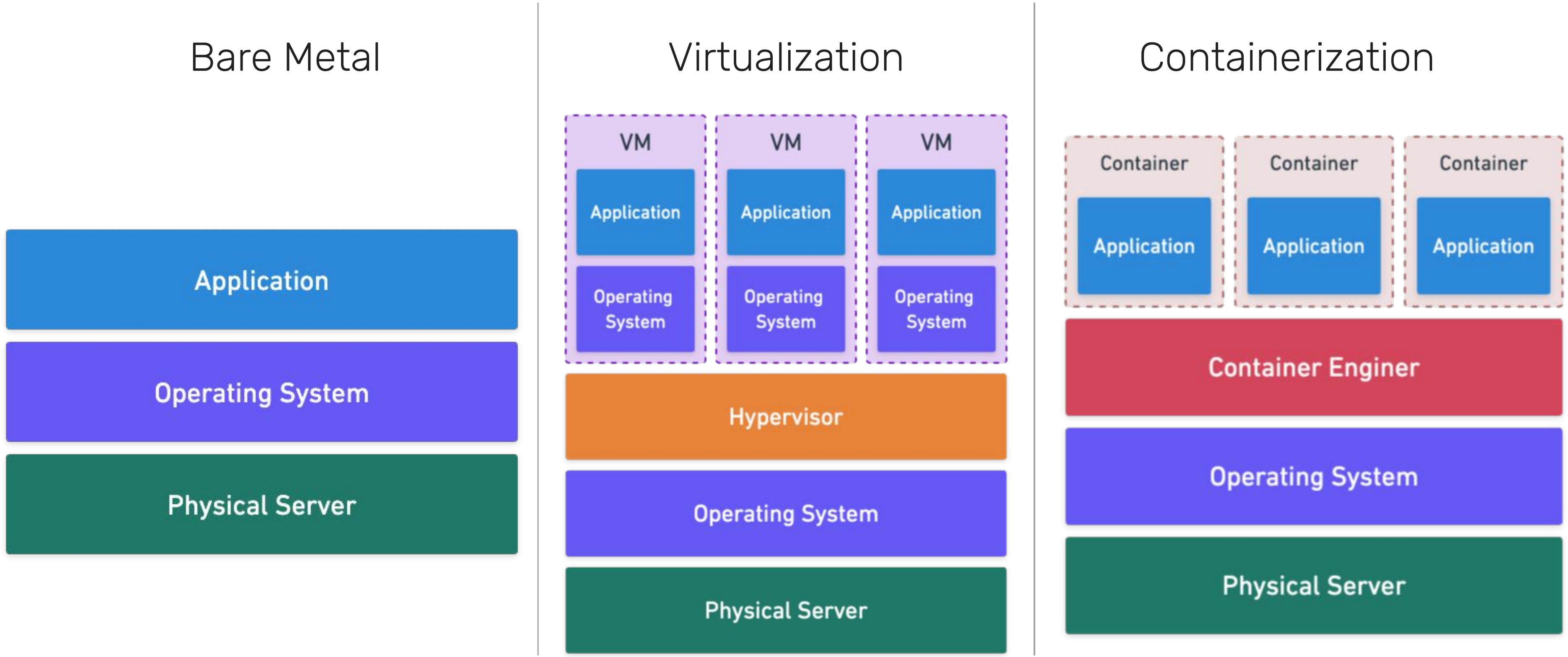
Horizontal vs Vertical Scaling

- There's a limit on how much you can vertically scale
  - After you've maxed out the most powerful CPU in existence, then what?
- Horizontal Scaling is, essentially, infinite
  - Just add more instances and distribute the load

Parallelization

- You have 10 washing machine size bags of laundry to do
- With 1 washing machine, you can do 1 at a time
- With 10 washing machines, you can do 10 at a time (in parallel)
- <u>Horizontal Scaling</u>: add lots of washing machines

Virtualization & Containerization



Which requires more knowledge of what's under the hood?

Abstraction

- Hiding details

- Why bother?
  - Increases ease of use / reduces complexity (for the user, at least)
  - Increases portability / looser coupling

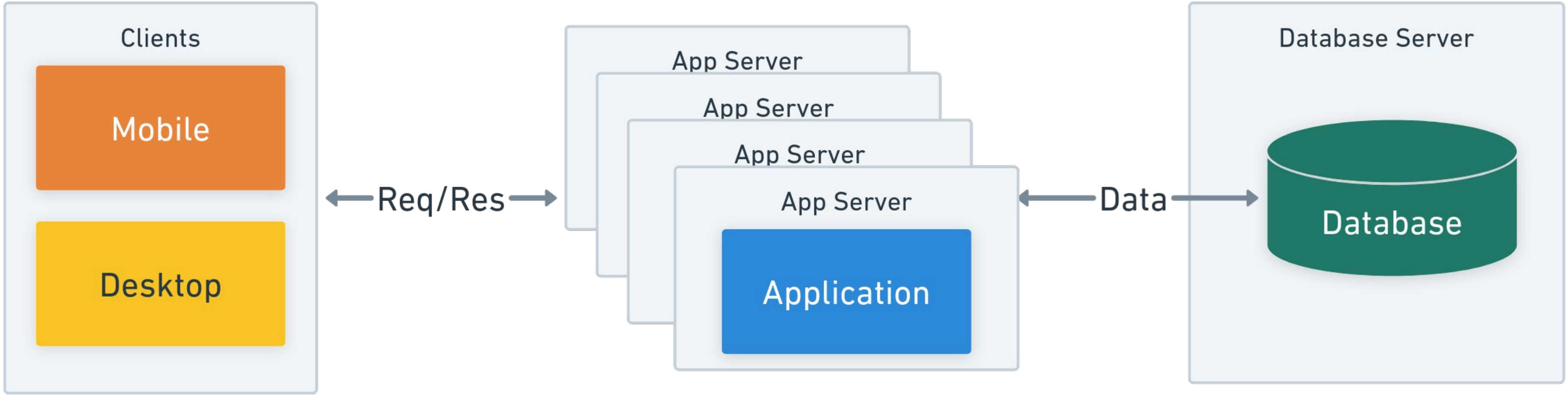
- Trade offs
  - Limited control
  - Increased overhead for the one implementing the abstraction layer

Containerized Applications

- Containers can be easily spun up and down in milliseconds, enabling you to quickly react to fluctuating demands

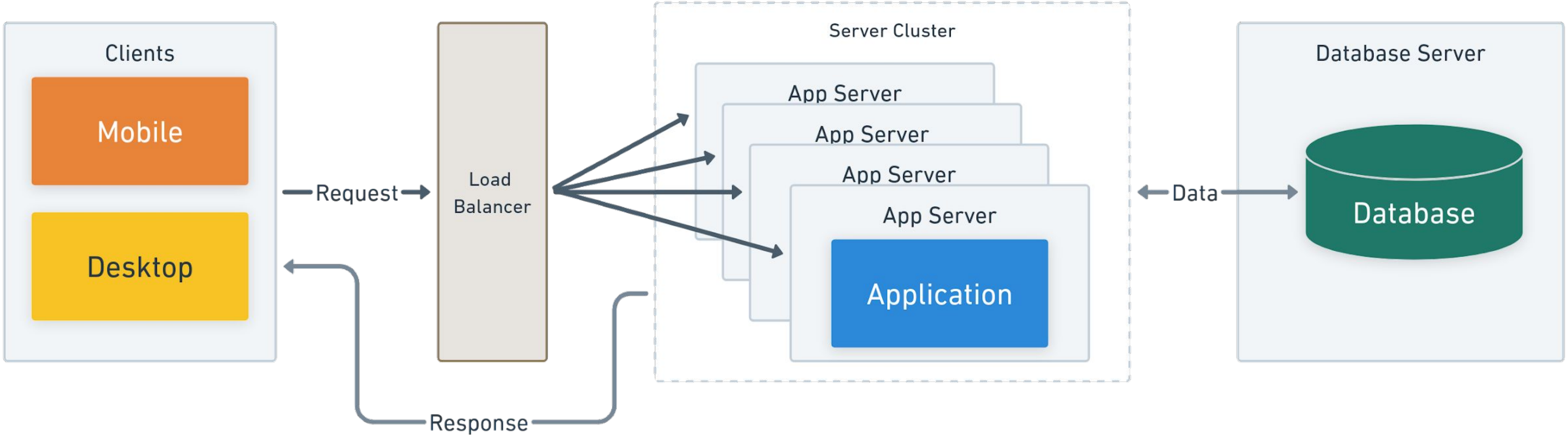
- Containers are self-contained units that bundle an application with all its dependencies. This allows them to run consistently across different environments, regardless of the underlying infrastructure.

Horizontal Scaling



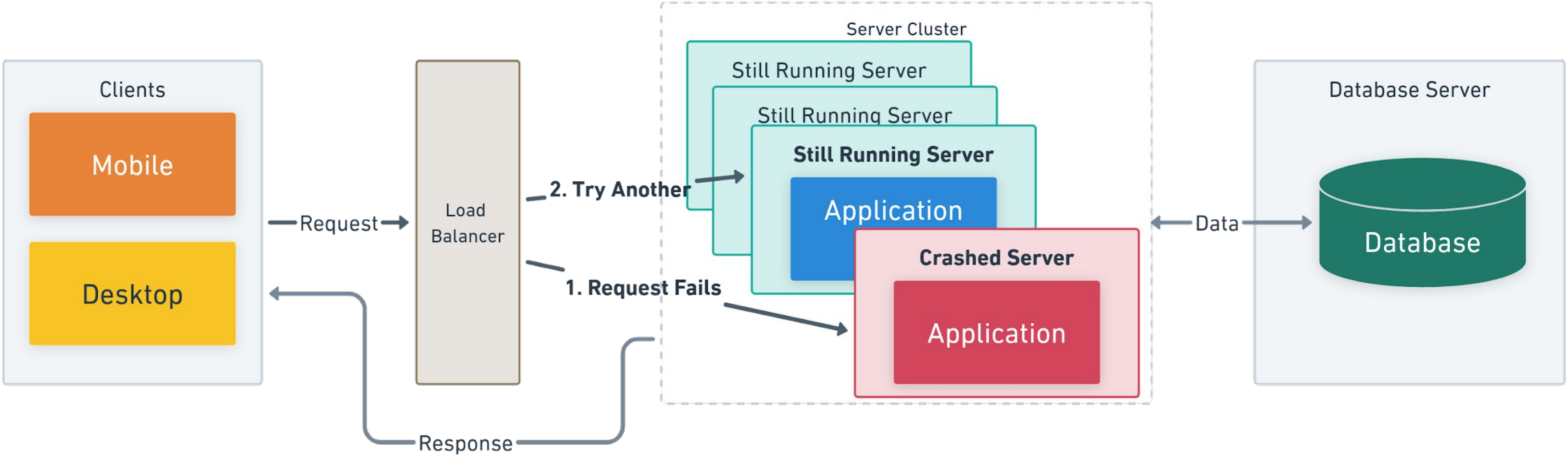
I've added more servers, but now what? How do we make sure the same server isn't used for every request?

Load Balancing



Load Balancers ensure an even distribution of traffic.

Horizontal Scaling + Load Balancing = Availability

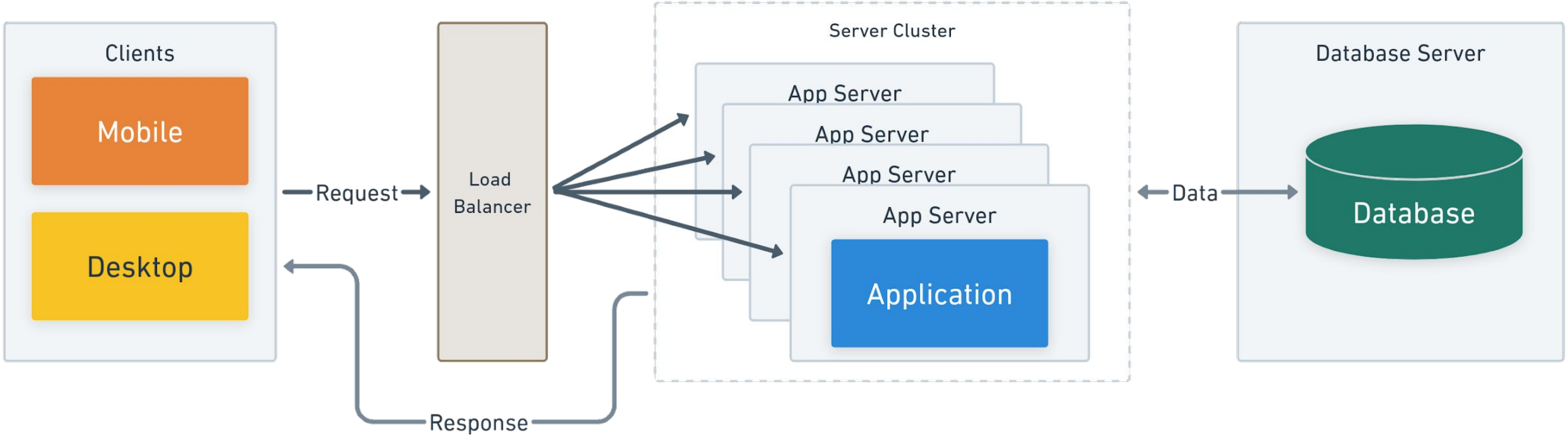


REST API

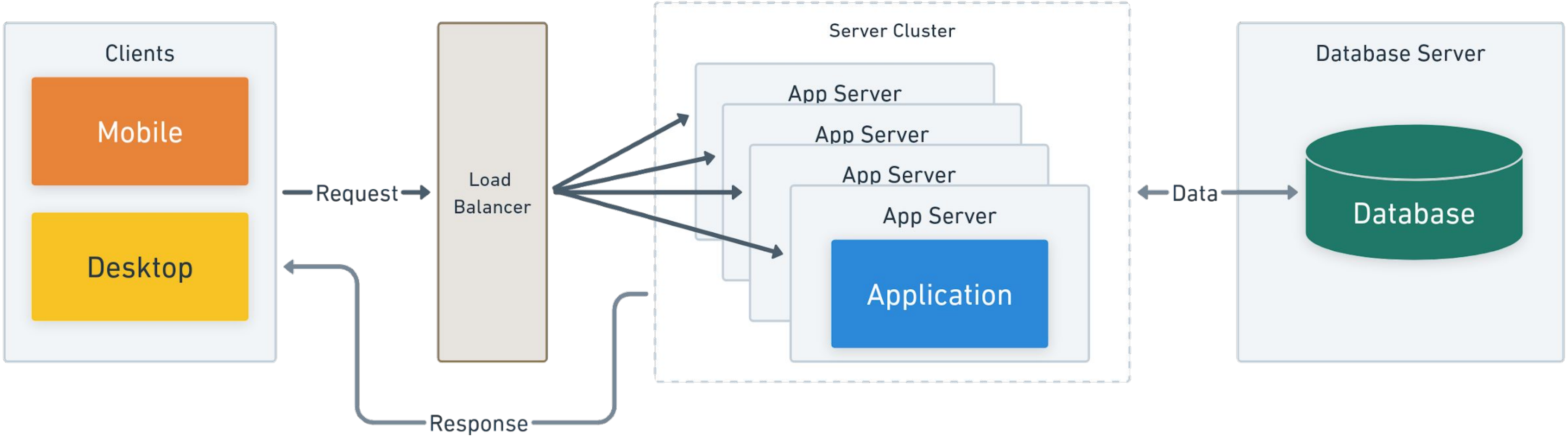
- A style of API design
- Clients use HTTP methods such as GET, PUT, DELETE, etc. to access server data.
- Stateless
    - Each request is processed purely based on the information provided within the request
    - The server doesn't "remember" anything about a client's previous requests

Why does Statelessness matter?

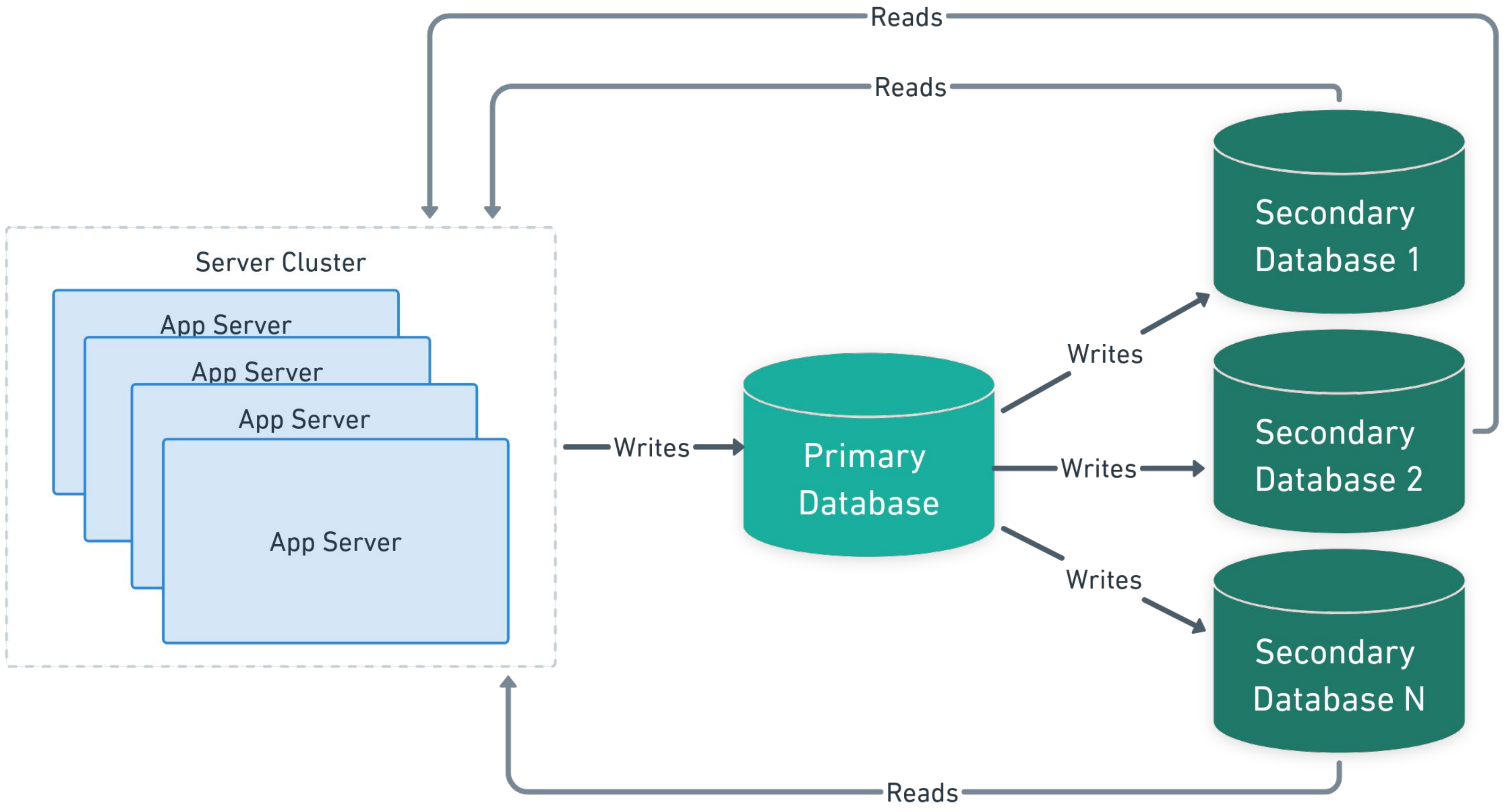
- Since each request can be treated independently, a request can be routed to any one of the servers



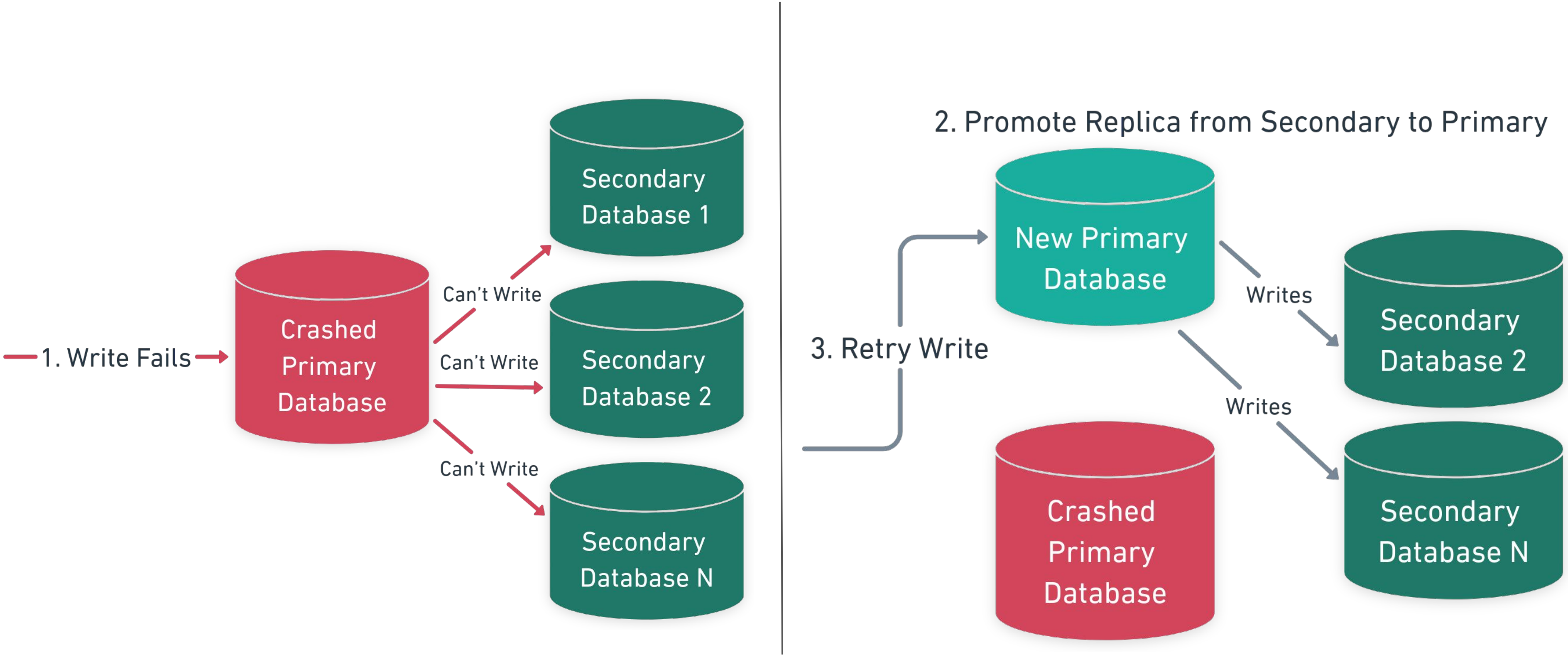
What else can we scale up?



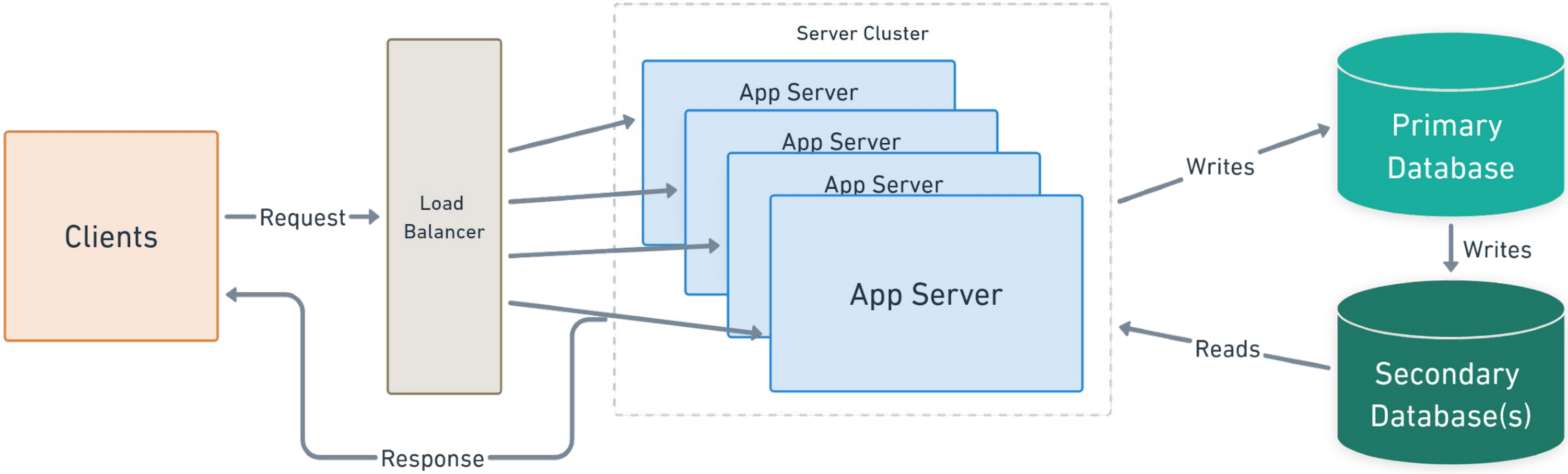
Database Replication



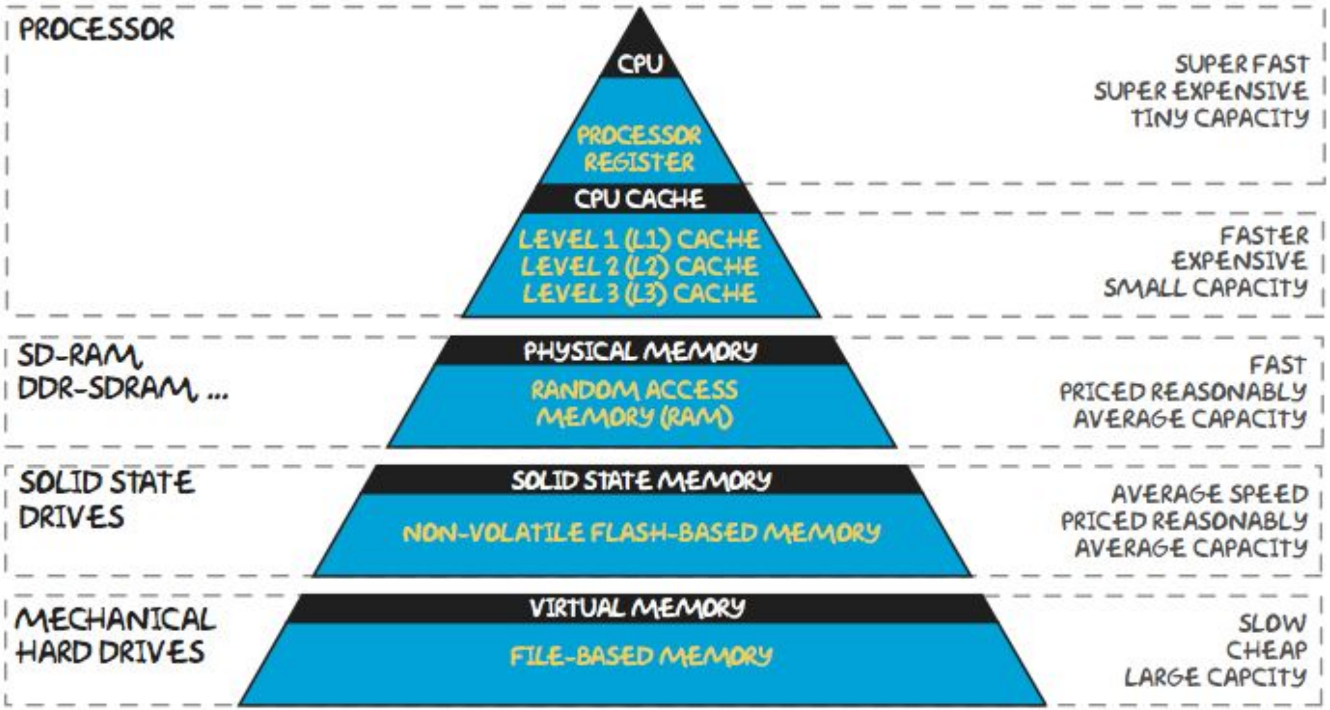
Database Replication improves Availability



Anything more we can do?



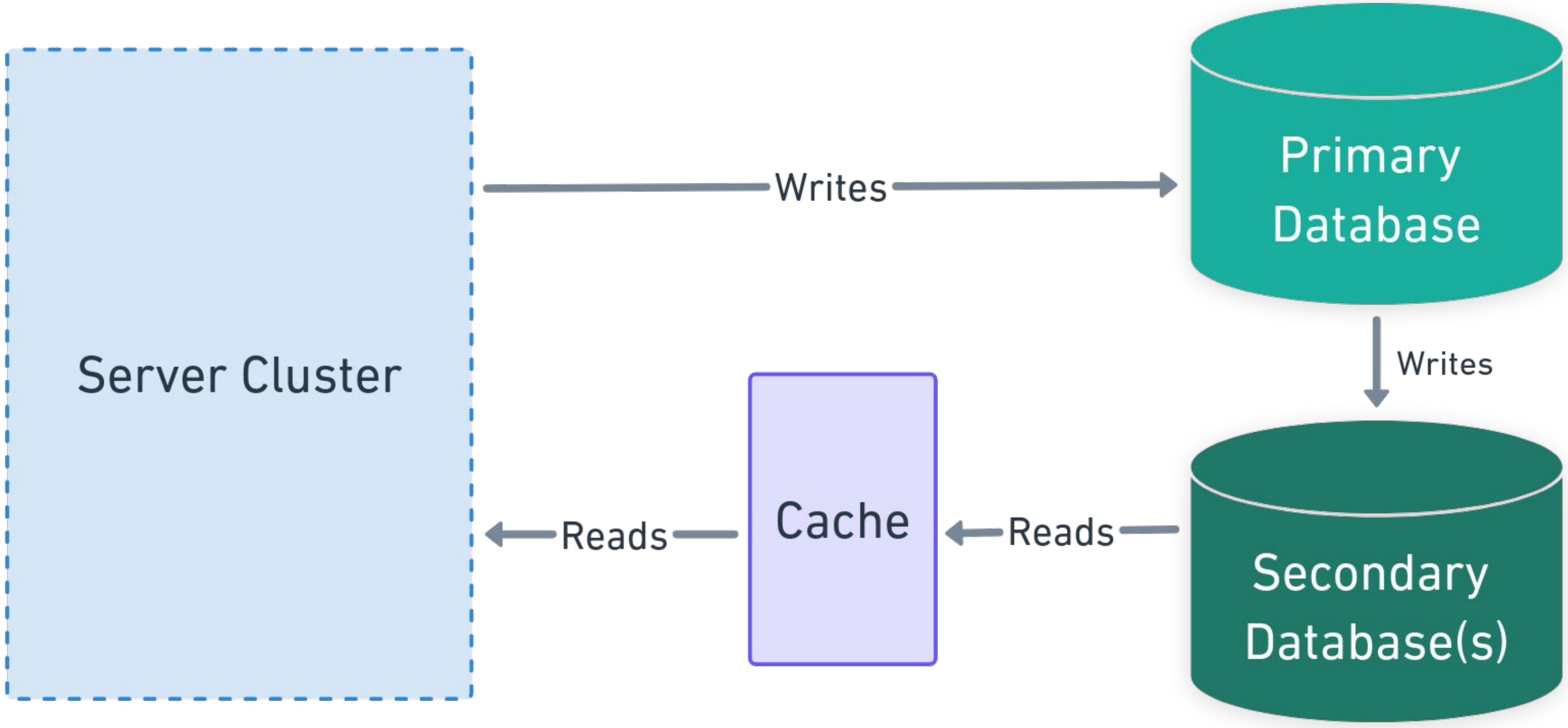
Databases are fast, but we can do better



Caching

- Even with replication, calls to the database are expensive
- Let's read from something faster, when we can

Cache

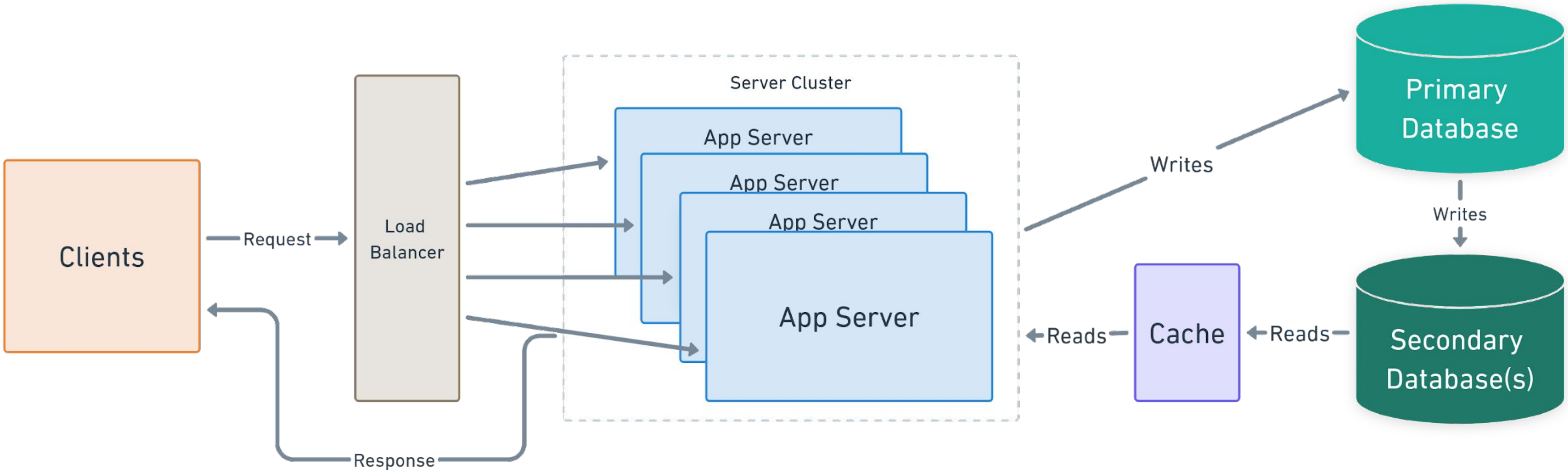


Server Cluster
Writes
Primary Database
Writes
Cache
Reads
Reads
Secondary Database(s)

Why are caches faster than the database?

- Caches are designed to be fast to read from and write to
  - Caches tend to be simple key/value stores.
  - Relational Databases enforce constraints and support sophisticated relationships between sets of data.
- Data is stored in memory (instead of on the disk)
  - Memory is faster than the disk.
- They, too, can be horizontally scaled.

Anything more we can do?

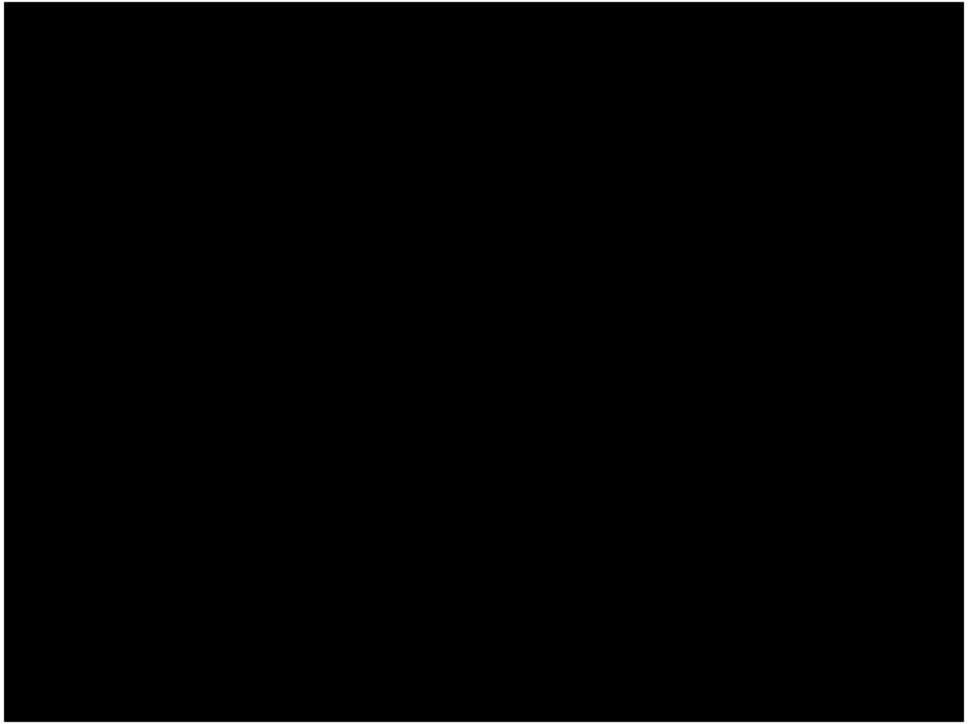


Some Insights

- The information being carried over a network is a physical thing like anything else.

- Imagine if we were passing mail through a pneumatic tube:
    - The further the distance between the sender and receiver, the longer it takes for the mail to arrive.
    - The same is true of electrons or photons over a cable.

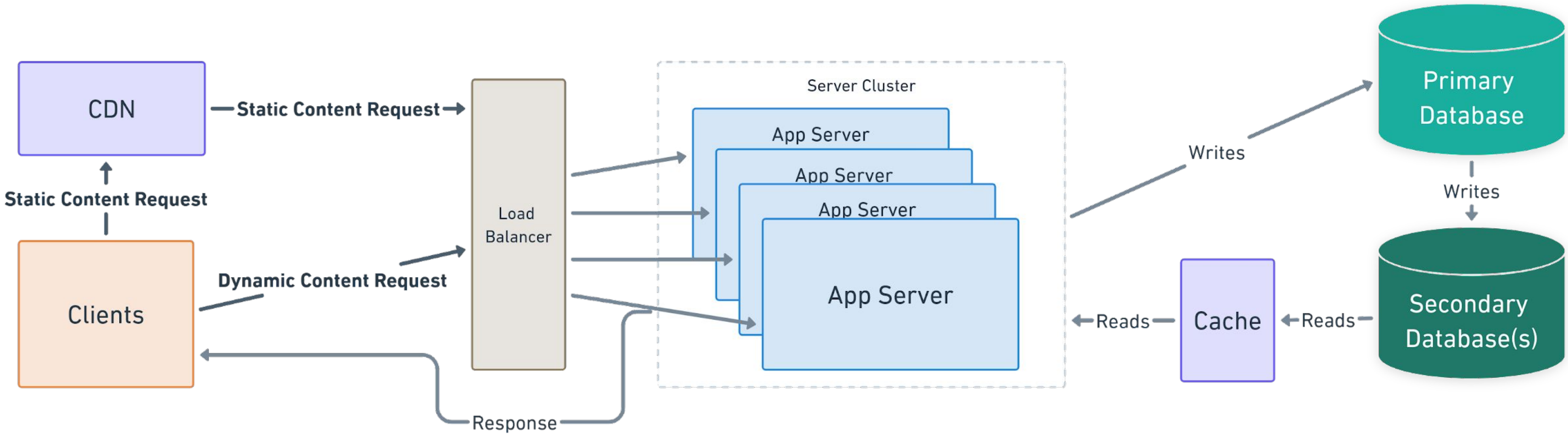
- Given that, would storing the data closer to our clients speed up responses?



Content Delivery Network (CDN)

- Networks of servers located in different regions around the world
- Direct users to the CDN instance closest to them
- Static content can be cached there
  - Media
    - Images
    - Video
  - Scripts (JS)
  - CSS
- Dynamic content too, in certain cases
  - If it changes infrequently
- Reduces latency
- Reduces number of requests made to your servers

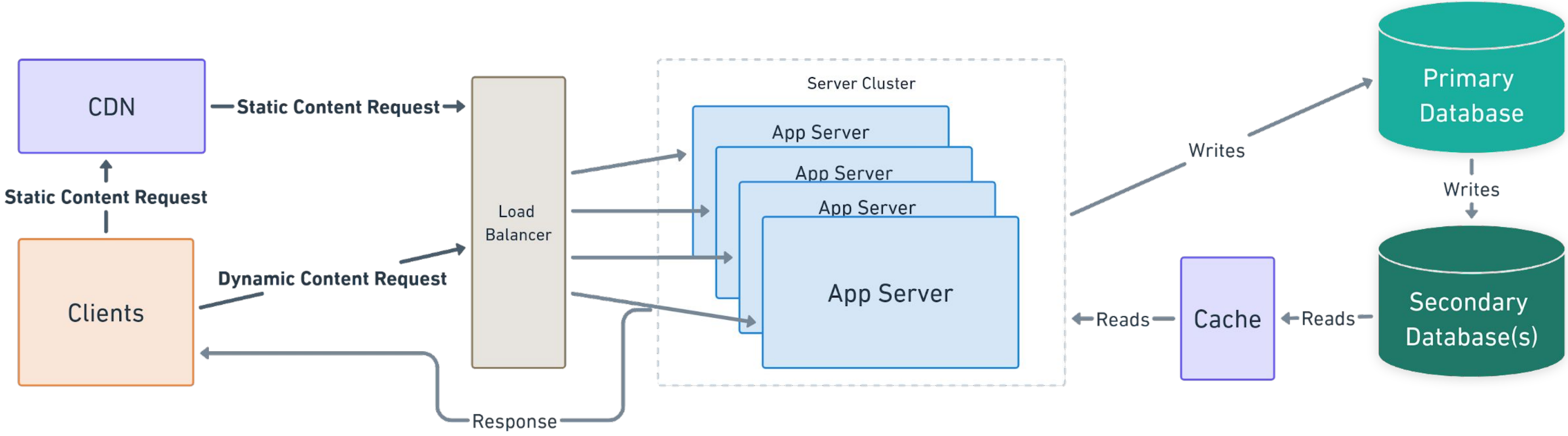
CDN



CDN
Static Content Request
Static Content Request
Clients
Dynamic Content Request
Load Balancer
Response
Server Cluster
App Server
App Server
App Server
App Server
Writes
Primary Database
Writes
Reads
Cache
Reads
Secondary Database(s)

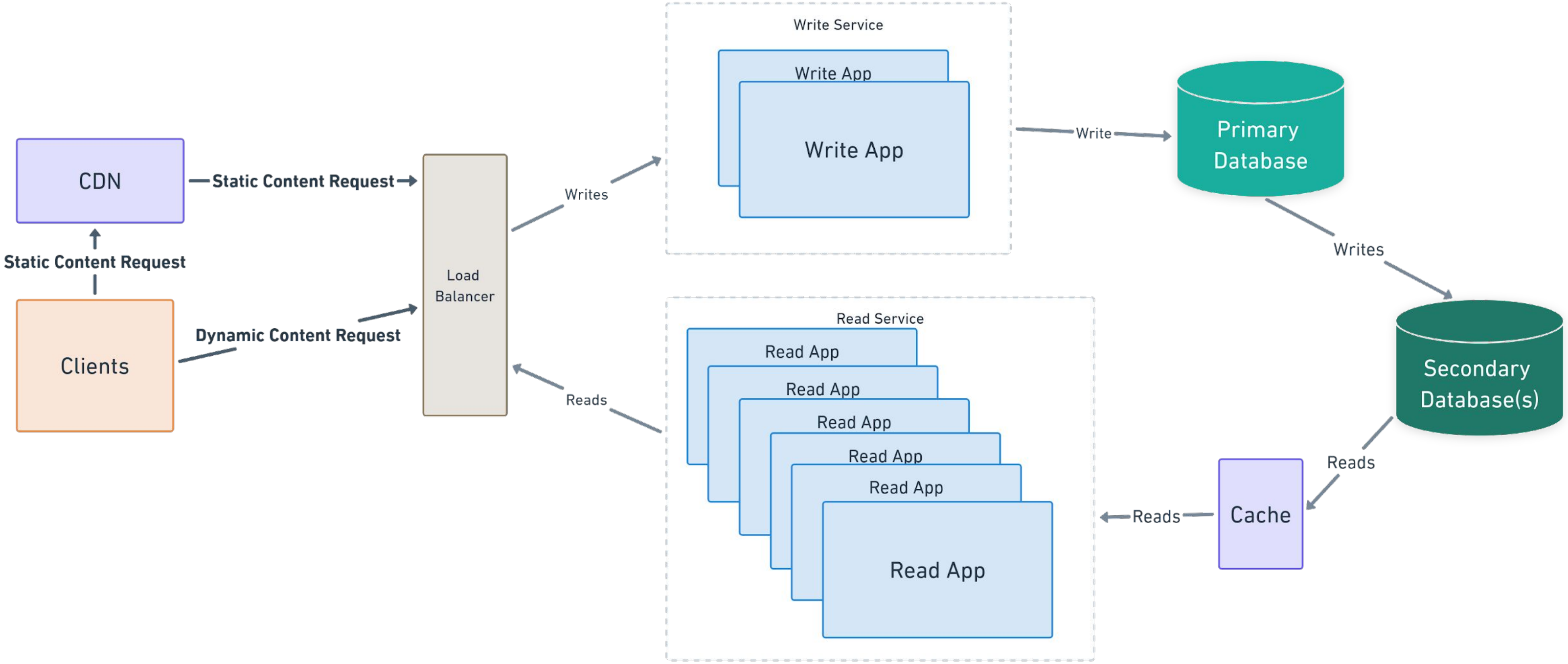
Anything more we can do?

- What if our application does more than one thing, but a lot more of one out of all the things it does? (e.g. way more reads than writes or vice versa)



Services

- We can break up our application into smaller functions/domains called services and scale those independently as well.



Systems Design Interview

- It's about the journey, not the destination.
    - There isn't one right answer.
    - Communication is as (if not more) important than your particular design.
    - Particularly, demonstrating an understanding of your decisions, the tradeoffs, etc.
- Above all, do not jump straight into designing. Ask **a lot** of questions first.

System Design Interview

https://www.amazon.com/System-Design-Interview-insiders-Second/dp/B08CMF2CQF

System Design Interview Framework - Alex Xu

1. Understand the problem & establish design scope
2. Propose high-level design & get buy-in
3. Design deep-dive
4. Wrap-up

Step 1: Understand the Problem & Establish Design Scope

1. Define the **Functional Requirements**
    - **What** the should the system do?
        - Narrow down the scope as much as possible
        - e.g. If the question is "Design Twitter", specify which exact parts
            - Timeline
            - Followers
            - DMs
            - Search
            - Media
            - Authentication
    - Explicitly list use cases
        - Logged In Users
            - Can post a tweet, can send a message, etc.
        - Logged Out Users
            - Can't post a tweet, can't send a message, etc.

Step 1: Understand the Problem & Establish Design Scope

2. Define **Non-Functional Requirements**
    - **How** the system should do it.
        - **Performance**: Describes how well the system performs under certain conditions, including response time, throughput, and scalability.
        - **Reliability**: Ensures the system operates correctly and reliably over time, including measures like availability, fault tolerance, and recovery.
        - **Scalability**: Addresses the system's ability to handle increased load or growth in terms of users, data, or transactions.
        - **Availability**: Specifies the percentage of time the system should be operational and accessible.
        - **Security**: Outlines the security measures and controls to protect the system from unauthorized access, data breaches, and other security threats.

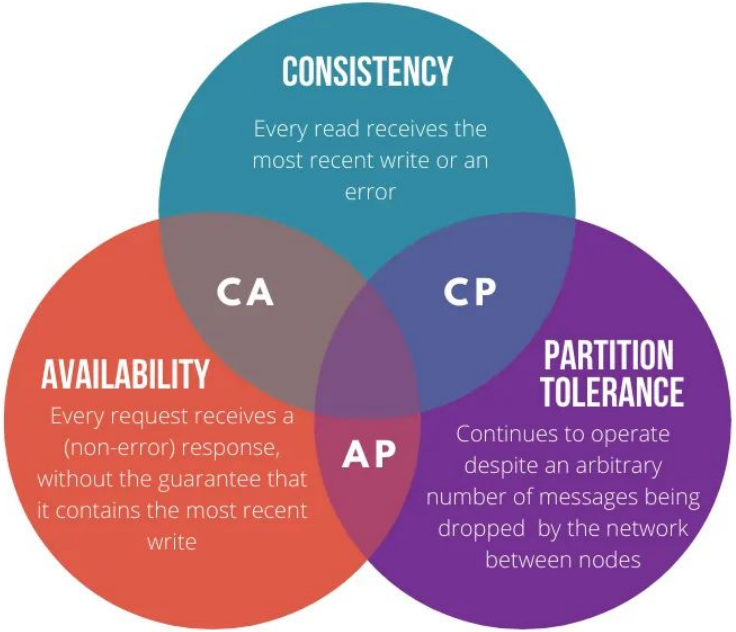
Functional vs Non-Functional Requirements

Functional Requirements	Non-Functional Requirements	
Users can post tweets	Availability	Highly Available
Users can delete tweets	Latency	<= 200 ms
Users can search tweets	Consistency	Eventual Consistency
Users can view another's timeline	Scalability	Highly scalable

CAP Theorem

In a **Distributed** system, you can only guarantee two of the following:

- Consistency
  - Reads return the exact same data for all users
- Availability
  - Every request receives a response
- Partition Tolerance
  - Continue working even if two nodes can't communicate



CAP Theorem: Why only 2 out of 3?

For Distributed Systems, Partition Tolerance is considered a necessity. So if a partial network outage occurs during a read/write, your system must continue operating. Do you:

- Cancel the operation?
  - Decreases **availability** but ensures **consistency**.

- Proceed with the operation?
  - Provides **availability** but risks **inconsistency**.

CAP Theorem

During interviews, be explicit about which choice you're making and why.

- Choose **consistency** over availability when
    - Data integrity is a high priority
        - Financial transactions
    - Read-heavy systems
        - If writes are rare, there will be few disruptions during a partial network outage
- Choose **availability** over consistency when
    - Data integrity isn't essential
        - Two users seeing a different number of likes on a tweet
    - User experience would be greatly harmed by unavailability
    - Or not harmed by temporary inconsistency (a.k.a. Eventual Consistency)
    - Write-heavy Systems
        - If writes are frequent, there will be many disruptions during a partial outage

Step 1: Understand the Problem & Establish Design Scope

3. Back-of-the-Envelope Estimations
    - Estimates you create to get a good feel for which designs will meet your requirements
        - Use a combination of thought experiments and common performance numbers
        - Pick whole numbers. Accuracy isn't important, you're only worried about the order of magnitude.

- Load
    - Requests Per Second
    - Data Volume
    - User Traffic

- Storage
    - Amount of Storage Required

- Resources:
    - Number of:
        - Servers
        - CPUs
        - Memory

- Network Bandwidth

- Latency

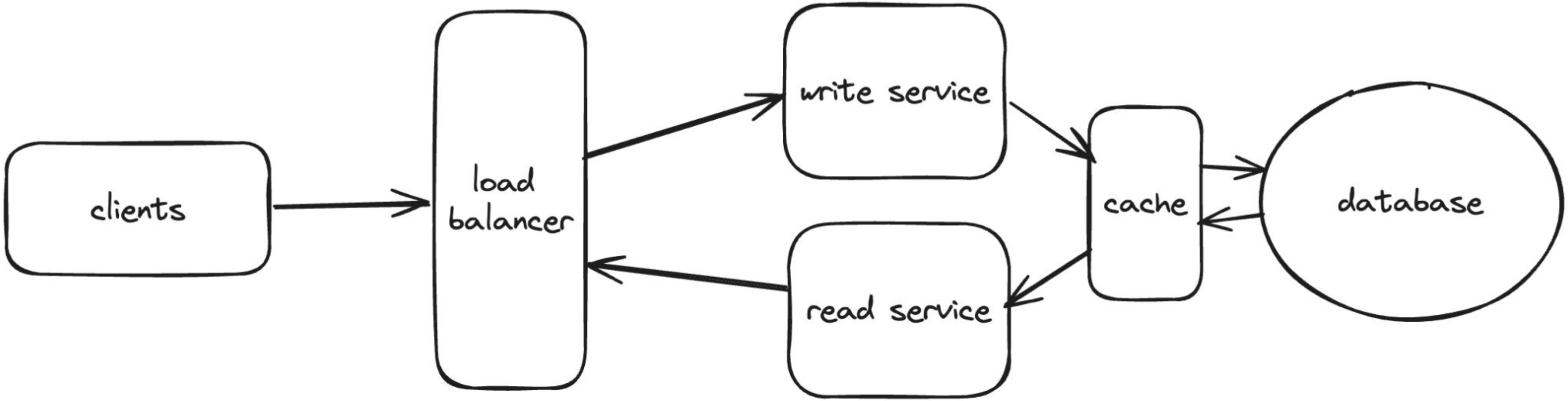
Step 1: Back-of-the-Envelope Estimations for Twitter

- Assumptions:
  - 300 million monthly active users.
  - 50% of users use Twitter daily.
  - Users post 2 tweets per day, on average.
  - 10% of tweets contain media.
  - Data is stored for 5 years.

- Estimations:
  - Query per second (QPS) estimate:
    - Daily active users (DAU) = 300 million * 50% = 150 million
    - Tweets QPS = 150 million * 2 tweets / 24 hour / 3600 seconds = ~3500
    - Peek QPS = 2 * QPS = ~7000
  - Media Storage estimate:
    - Average tweet size:
      - tweet_id = 64 bytes
      - text = 140 bytes
      - media = 1 MB
    - Media storage: 150 million * 2 * 10% * 1 MB = 30 TB per day
    - 5-year media storage: 30 TB * 365 * 5 = ~55 PB

Step 2: Propose High-Level Design & Get Buy-In

1. Sketch a diagram of your system's core functionalities
    - Use the requirements you gathered in step one as a checklist



Step 2: Propose High-Level Design & Get Buy-In

2. Define the API
   - What endpoints will you have to create?

Endpoint	`/tweet`
HTTP Method	`GET`
Parameters	`integer tweet_id`
JSON Response	`integer user_id` `string tweet_content`

Step 2: Propose High-Level Design & Get Buy-In

3. Define the Data Model
    - What tables will you need to create in your database?
    - What are the relationships between those tables?

tweets	
tweet_id	**int**
user_id	int
content	string
created_at	datetime

users	
user_id	**int**
username	string
email	string
created_at	datetime

likes	
like_id	**int**
tweet_id	int
user_id	int
created_at	datetime

Step 2: Propose High-Level Design & Get Buy-In

4. Get Buy-In
    - Check in with your interviewer, confirm you're on the right track
    - Make space for them to interject
    - Only move forward after you have buy-in from your interviewer

Step 3: Design Deep-Dive

- Work with the interviewer to identify and prioritize components in the architecture to focus in on.
- Make the system:
  - Faster
    - CDNs, Caches
  - Robust
    - Database Replication
    - Sharding
  - Secure
    - OAuth
    - ACLs
    - Encryption

Also makes it faster

Step 3: Design Deep-Dive

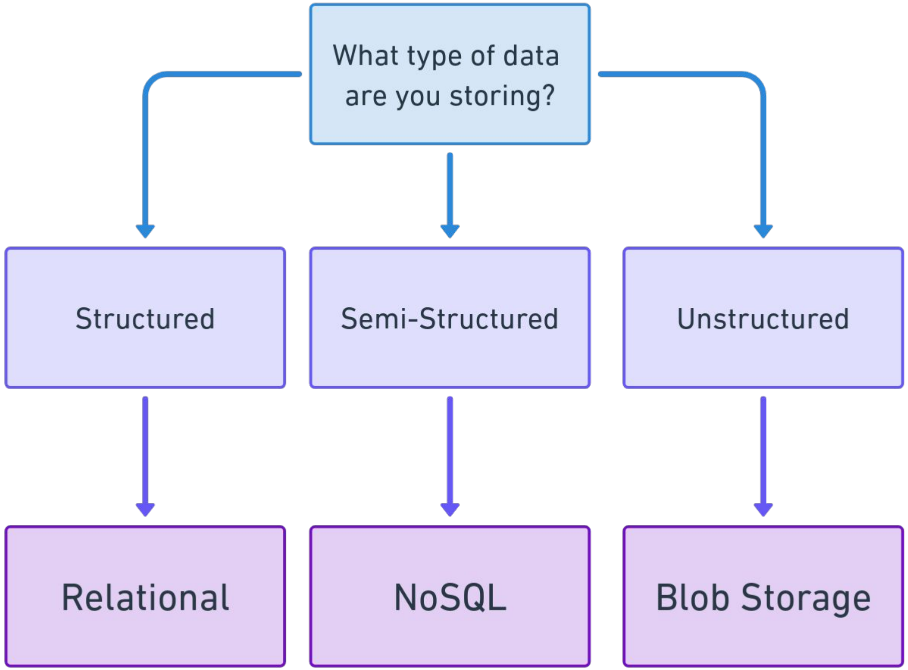
- Technology Choices
  - e.g. What sort of database?
    - Relational vs NoSQL
  - e.g. What sort of message queue/event streaming system?
    - Kafka, RabbitMQ, etc.
  - e.g. Long polling vs Websockets?

- The choices matter, but the ability to discuss the trade offs between each choice matters more.
  - If you're going to choose NoSQL over Relational, be prepared to explain why. What might change that decision?

Choice of Database System

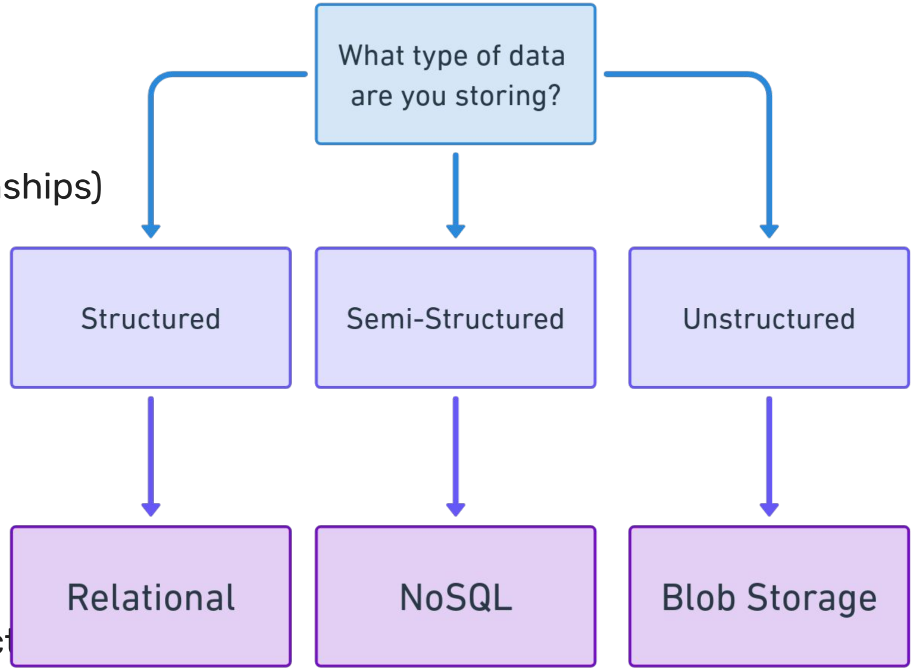
- Choose an appropriate database technology based on the type of data you're dealing with.

- Relational is often a safe bet



Structured, Semi-Structured, and Unstructured Data

- Structured Data
  - Highly organized
  - Consistent format
  - Multiple sets of data interlinked (relationships)
    - Customer Information
    - Financial Transactions
- Semi-Structured Data
  - Some/Simple organization
  - Variable pieces of data
    - JSON / XML
    - Log files
- Unstructured
  - Requires additional processing to extract
    - Images
    - Videos



Step Four: Wrap-Up

- Give the interviewer a recap of your design.
- Discuss possible improvements, if you had more time.
- Error Cases
  - e.g. server failure
- What metrics should you track?
  - e.g. HTTP Response Codes
- How will you monitor the system?
- Make space for the interviewer to pick something to drill into.

The Best Tip I Can Give

- Unless you're Allen Iverson, **YOU NEED PRACTICE.**
- Talk to your classmates, ask if they want to practice together
- Pramp

These slides can be found at:

https://carloscuevas.github.io/systems_design.pdf