Systems Design

Carlos Cuevas
Brooklyn College

Typical Software Engineer Interview

- Data Structures & Algorithms
- Practical Coding Interview
- Behavioral
- Systems Design

Systems Design Interview

- Design a system that accomplishes *x*
  - Chat System
  - URL Shortener
  - Youtube
  - Many Others

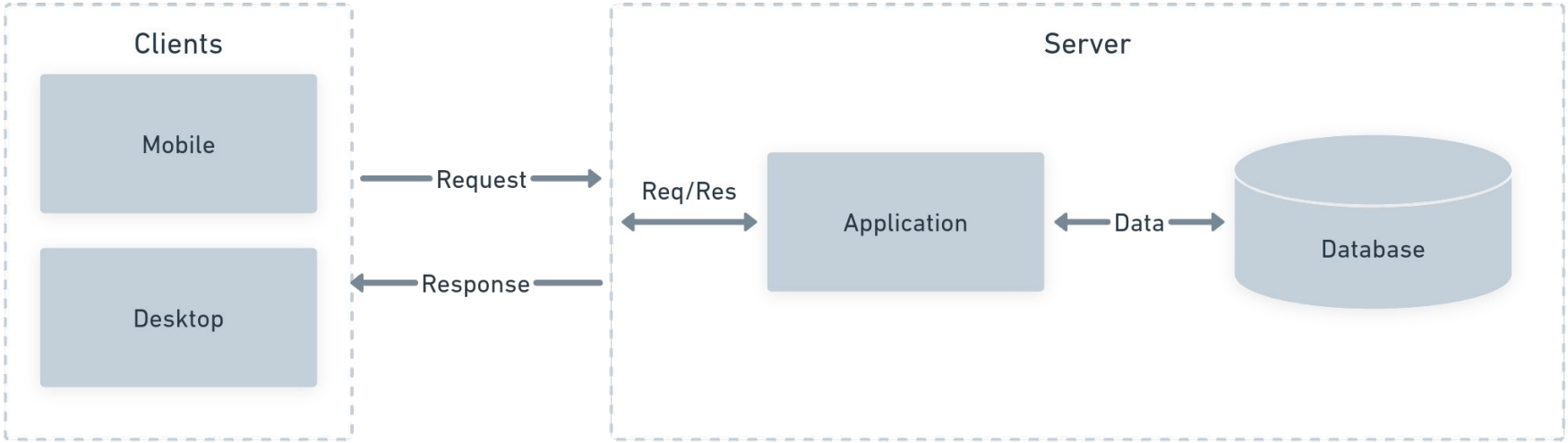
Systems Design Interview

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

Building Blocks

- Many systems have overlap
- Learn the common components of many large scale systems

Client - Server Model



What exactly is a Server?

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

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

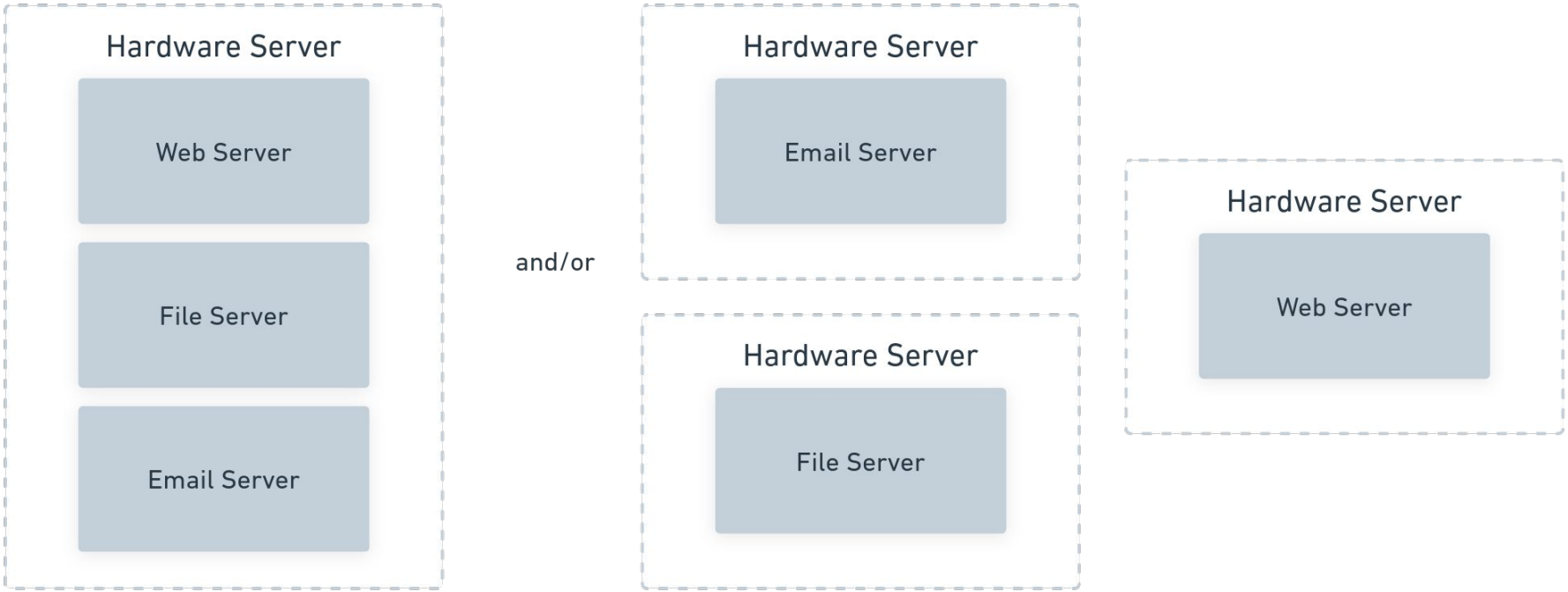
* Could be:

- Physical Computer
- Virtual Machine / Container
- Cloud Service

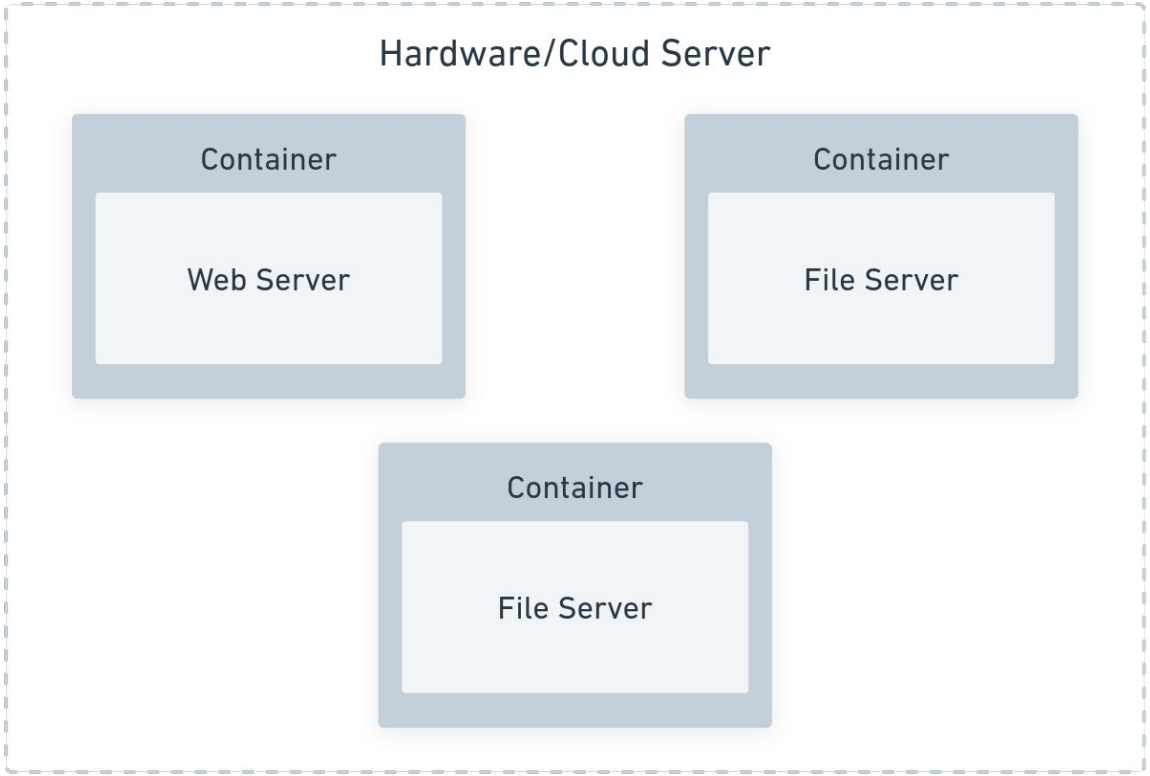
Types of (Software) Servers

- Web
  - Apache
- File
  - FTP
- Email
  - SMTP
- Database
  - MySQL
- Many many more

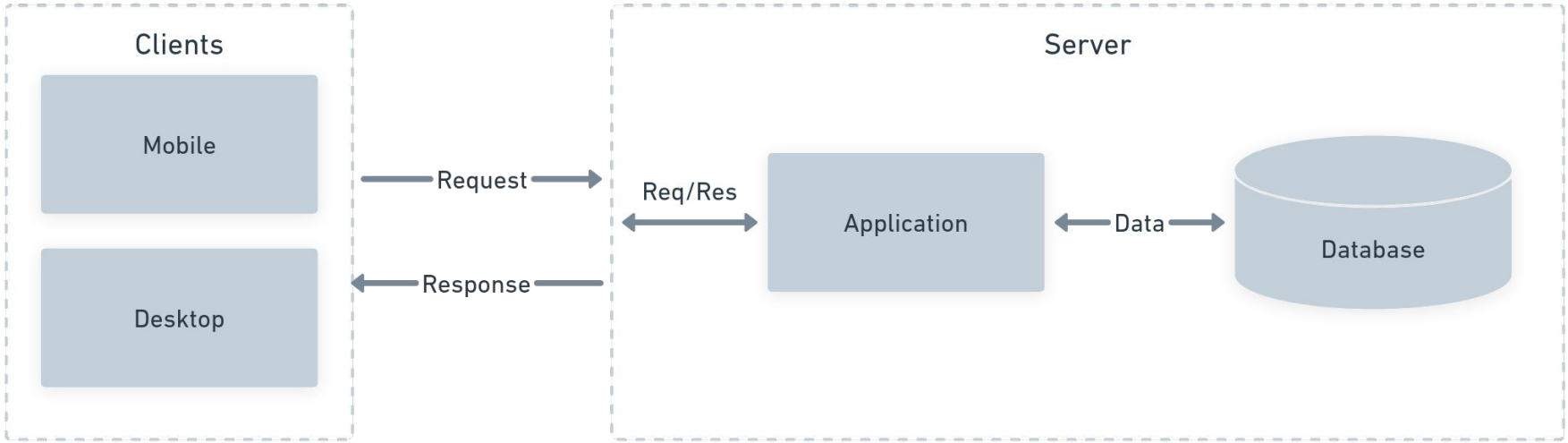
Hardware & Software



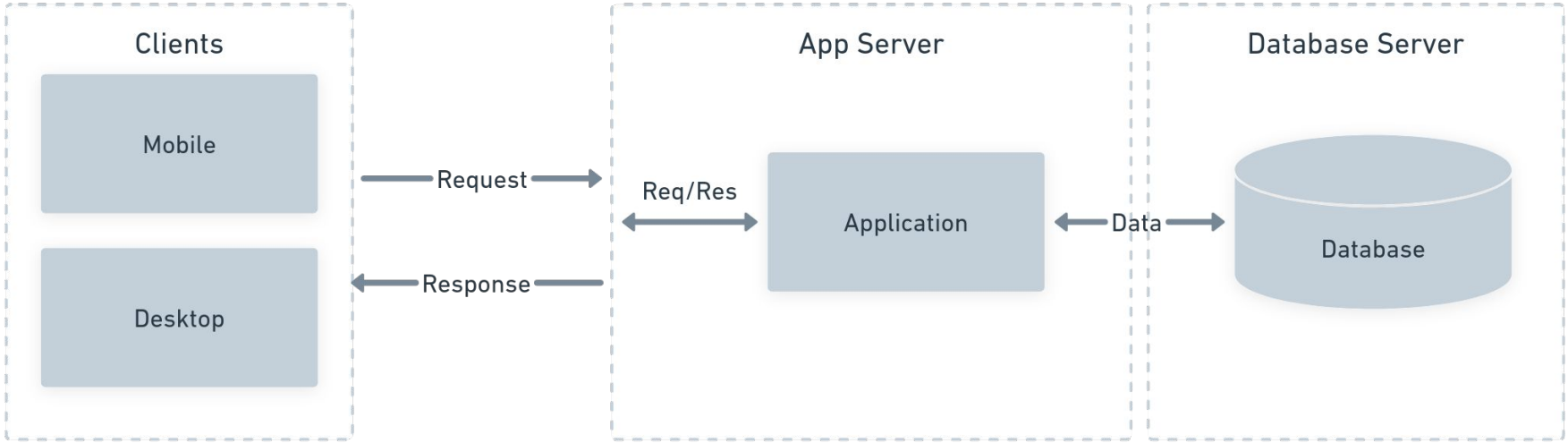
Containerization



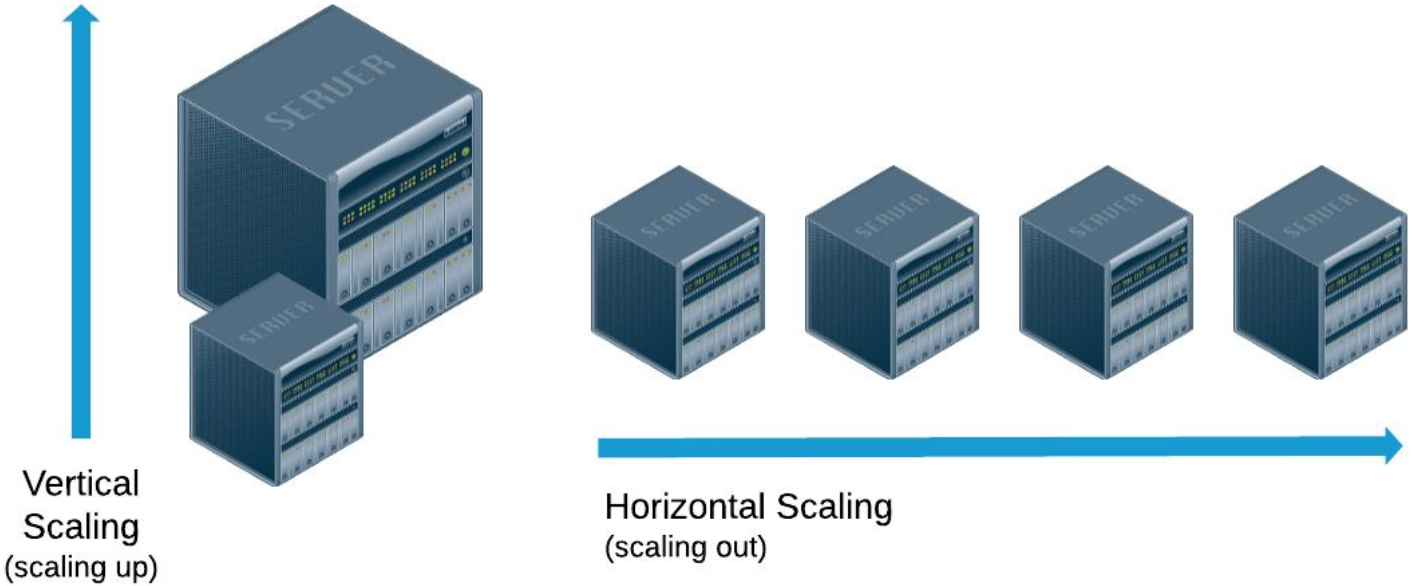
Single Server



Multi-Server Setup



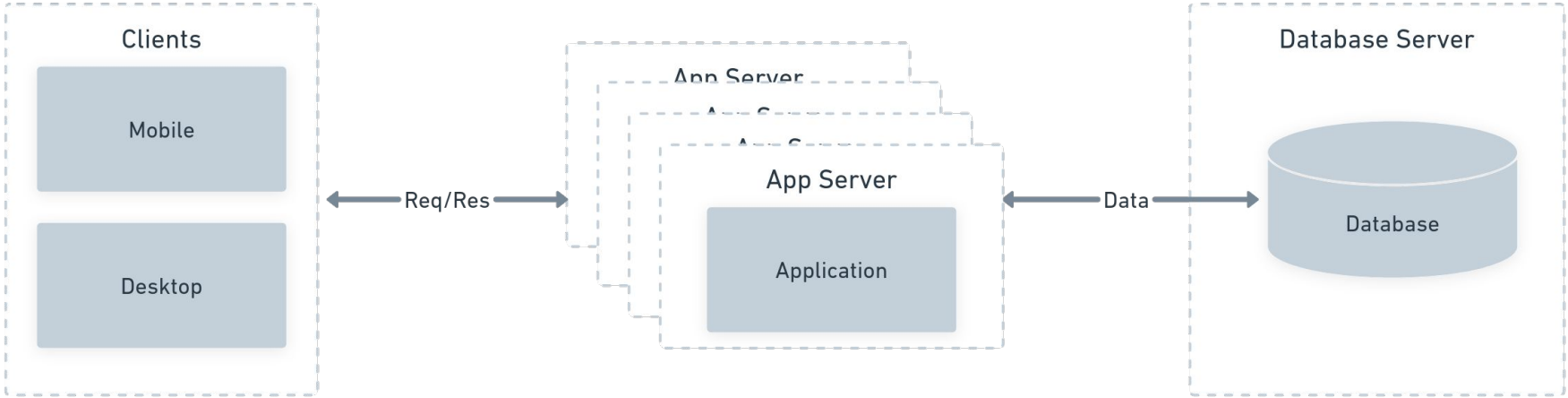
Horizontal vs Vertical Scaling



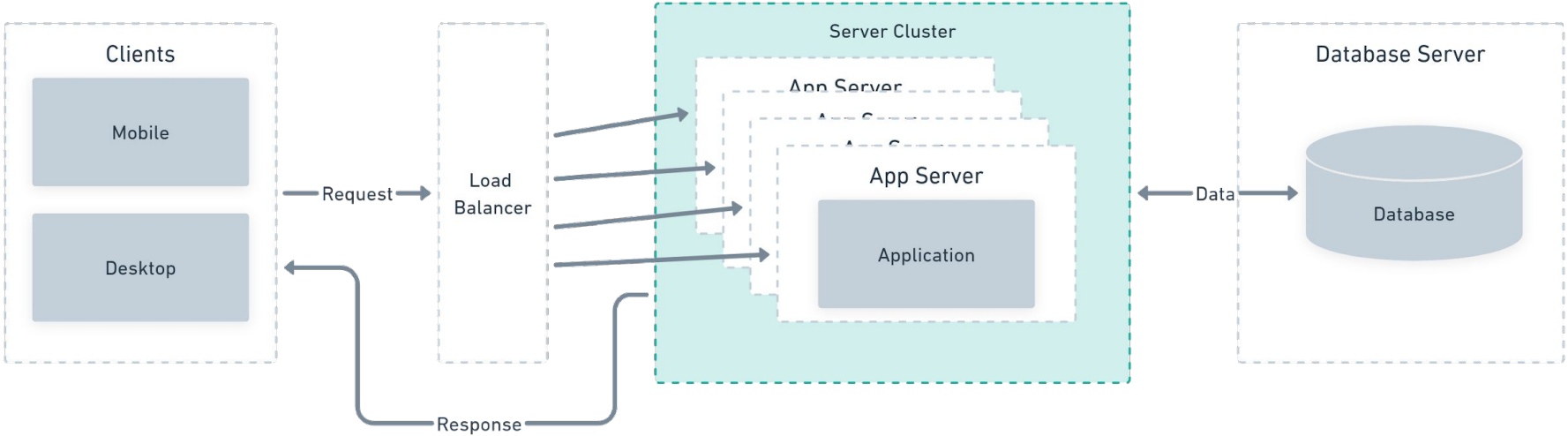
Horizontal vs Vertical Scaling

- 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

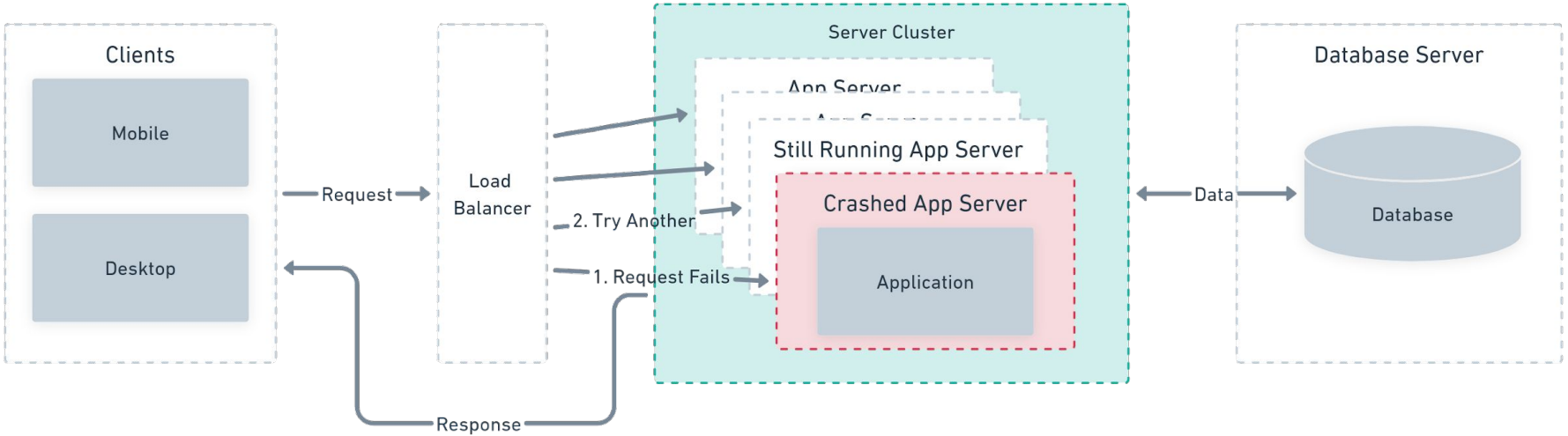
Horizontal Scaling



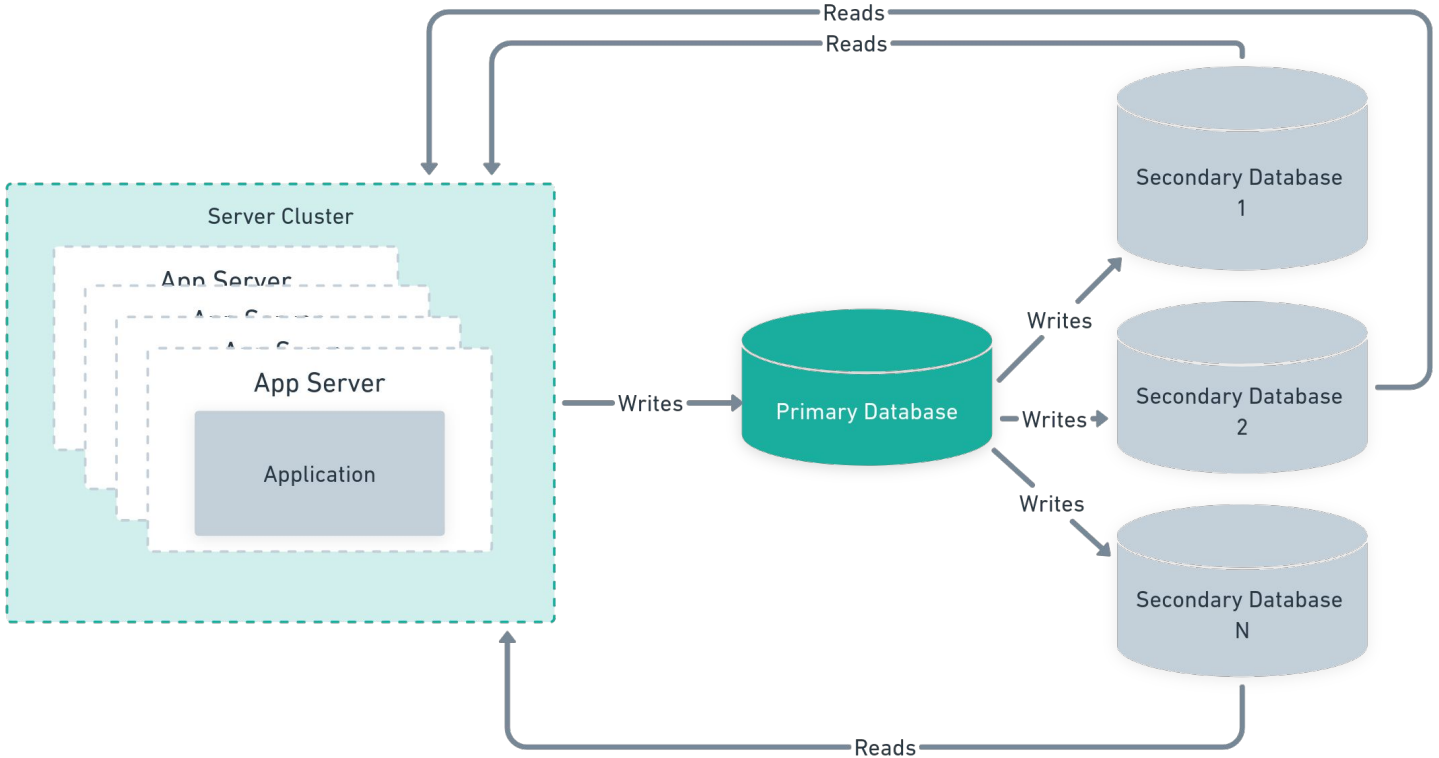
Load Balancing



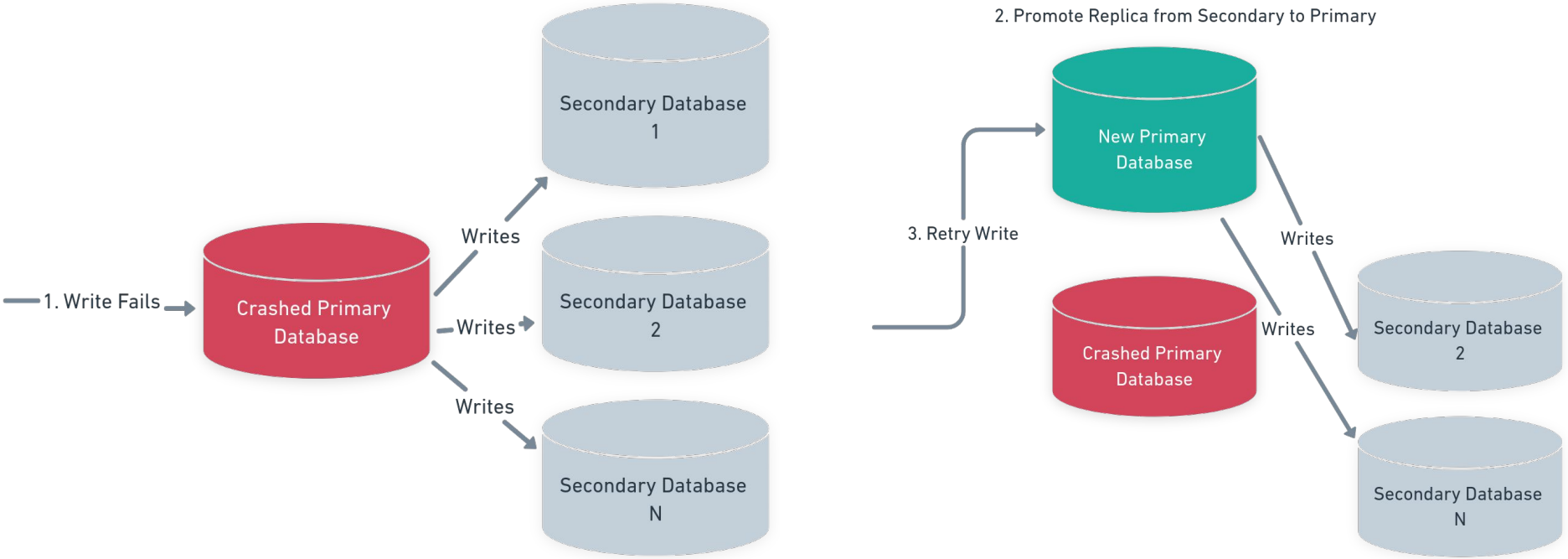
Horizontal Scaling + Load Balancing = Availability



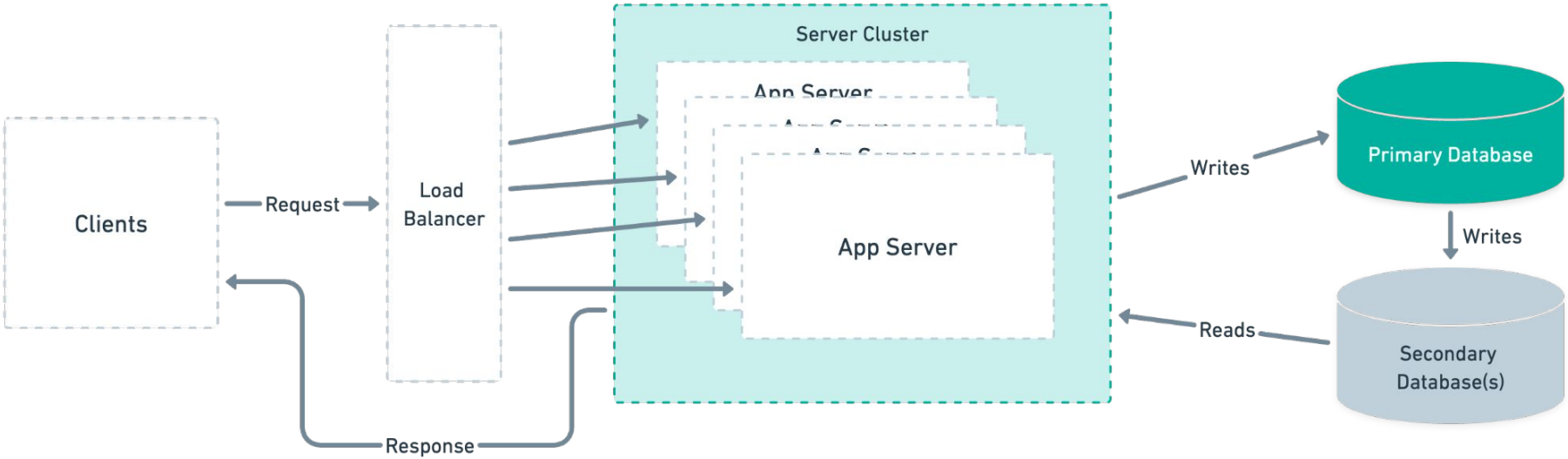
Database Replication



Database Replication improves Availability



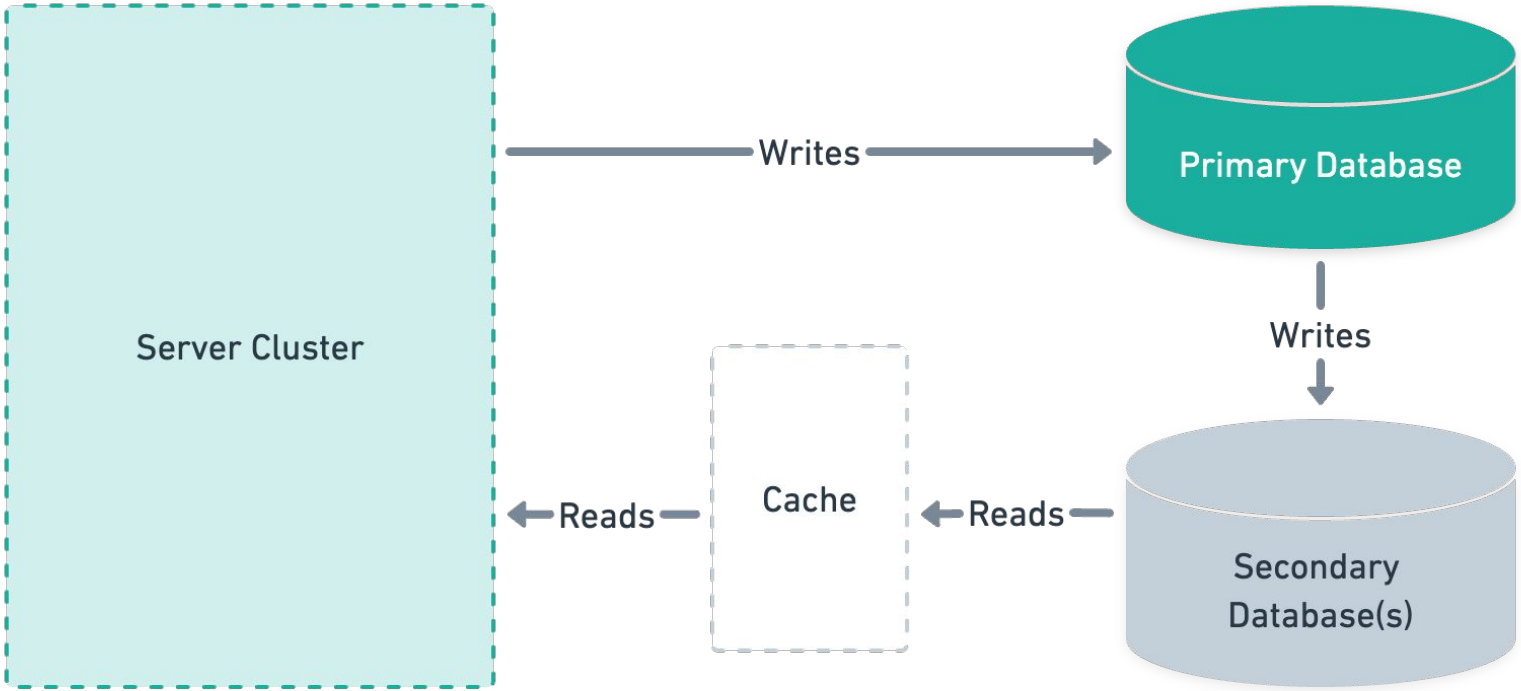
Anything more we can do?



Caching

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

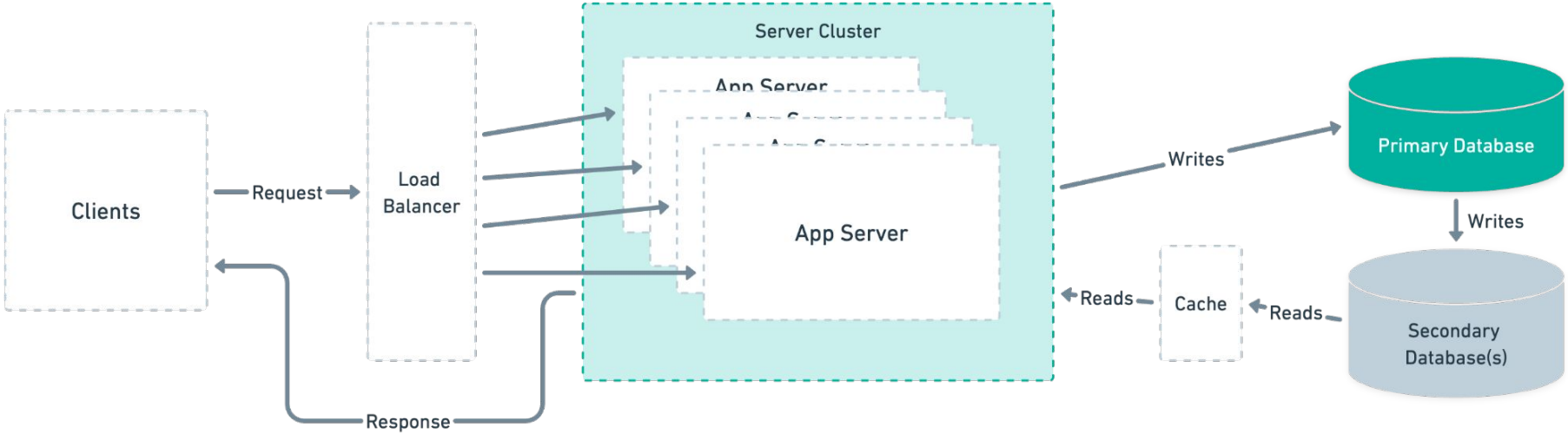
Cache


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

Why are caches faster than the database?

- Caches are designed to be fast to read from and write to
  - Relational Databases, in particular, require more overhead
- 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.

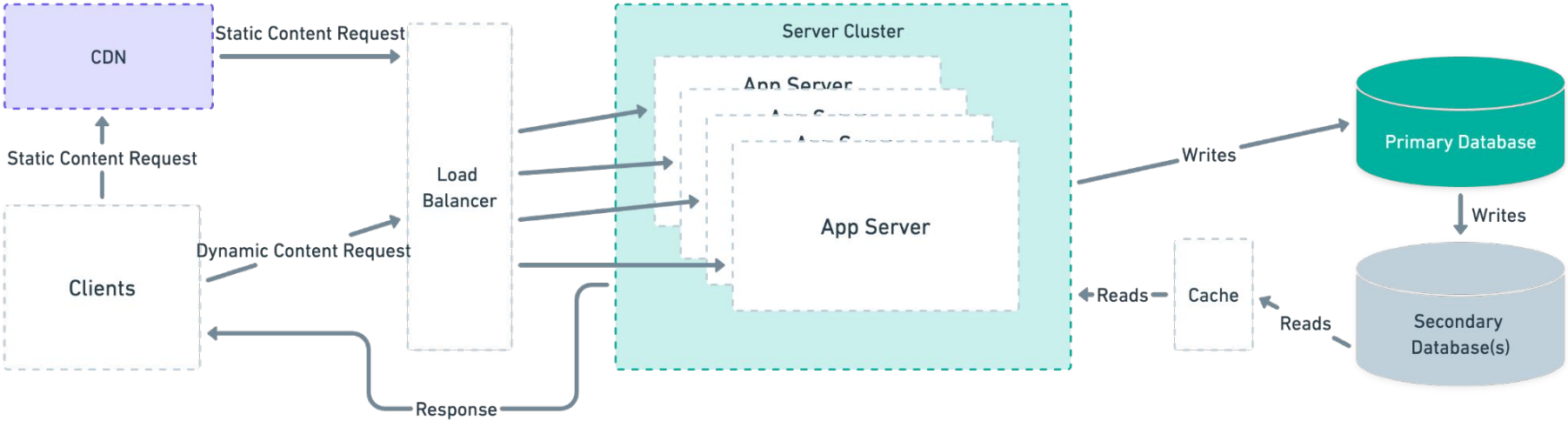
- Imagine if we're passing mail through a pneumatic tube:
  - The further the distance between the sender and receiver, the longer it takes for the mail to arrive.
  - The same is true of electrons or photons over a cable.

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

Content Delivery Network (CDN)

- Network of servers located in different regions of 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
- 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
Writes
Primary Database
Writes
Reads
Cache
Reads
Secondary Database(s)

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
  - Structured 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
- 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`

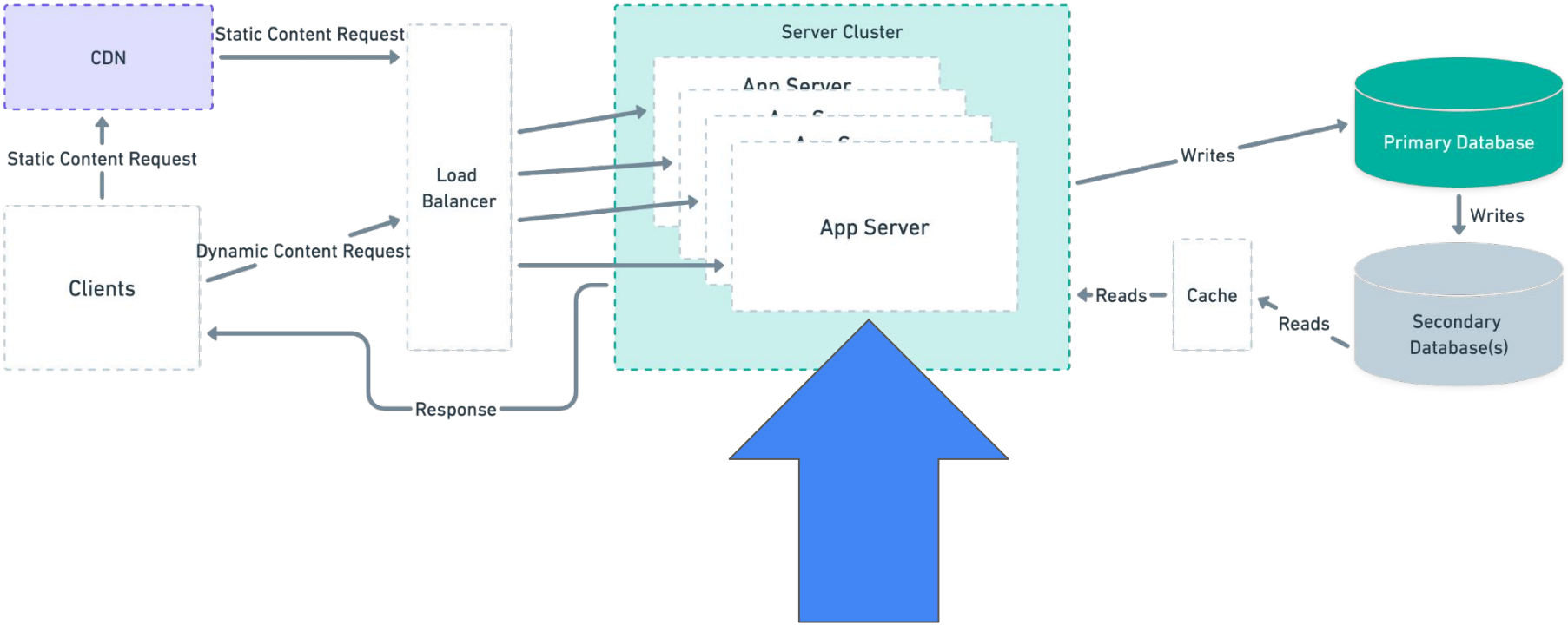
Request Code

```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

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

Where is your API implemented?



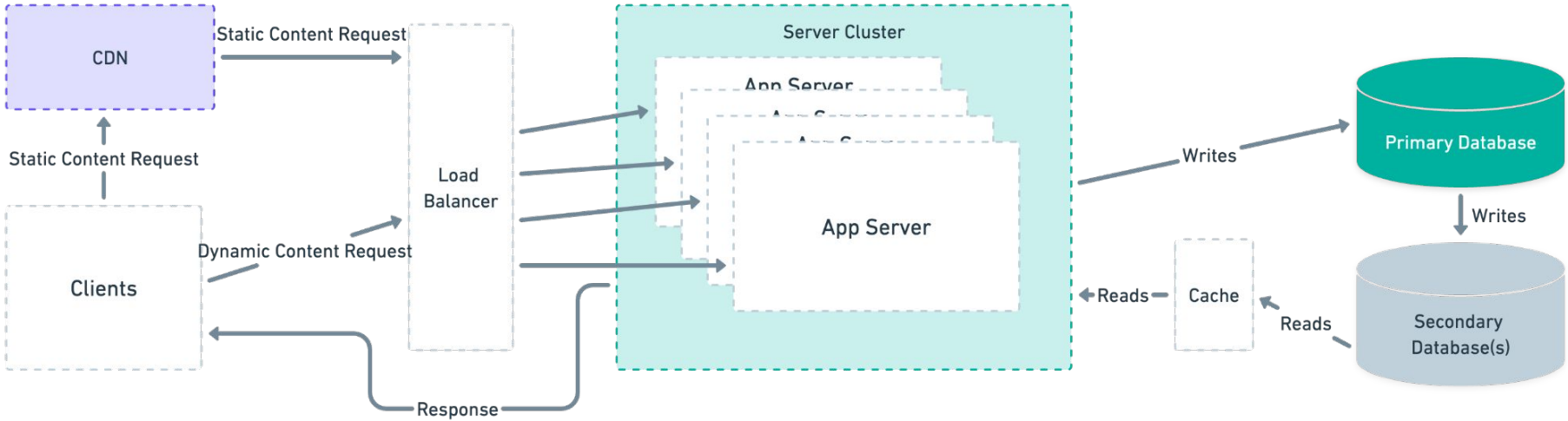
REST API

- Clients use HTTP methods such as GET, PUT, DELETE, etc. to access server data.

- Stateless

Why does Statelessness matter?

- Each request can be treated independently
- A request can be routed to any one of the servers



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

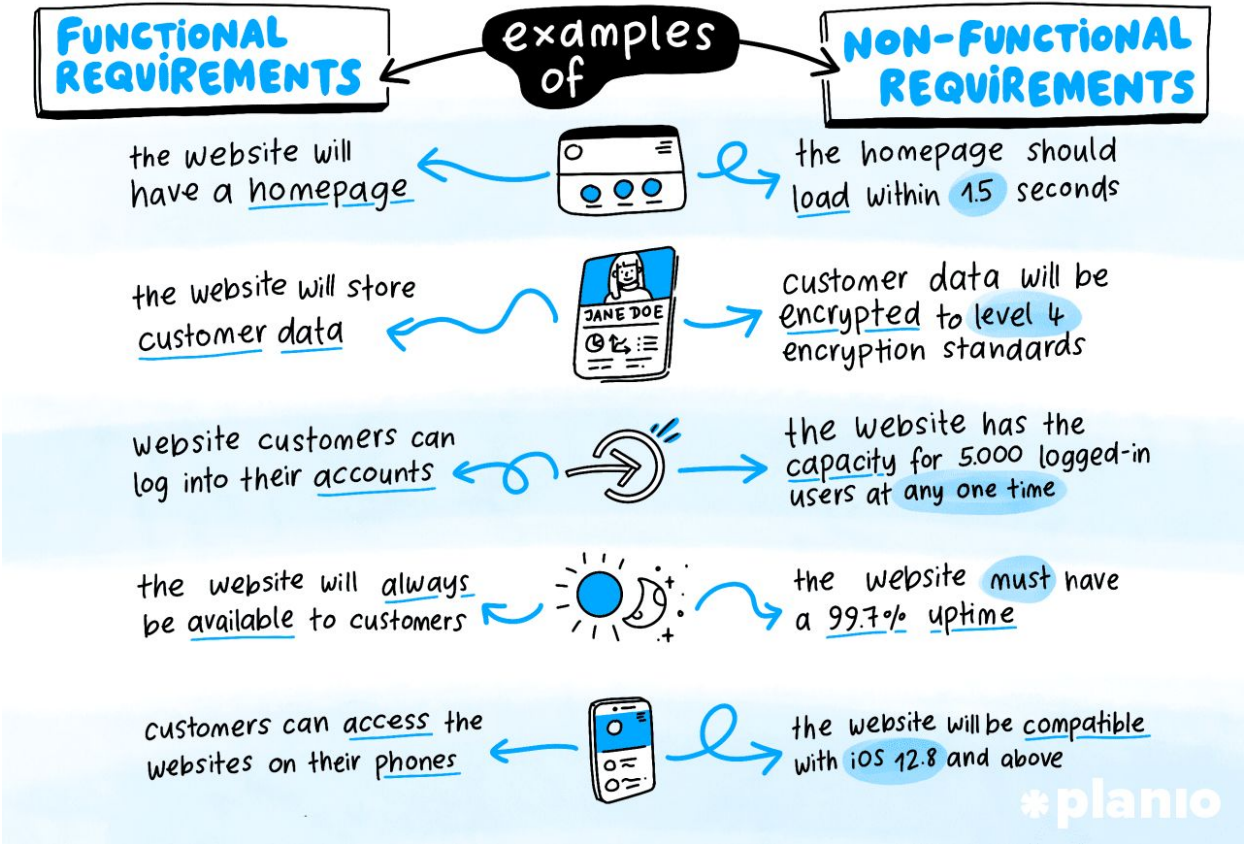
Step 1: Understand the Problem & Establish Design Scope

1. Functional requirements
    - What the System Should 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. Non-functional requirements
    - **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.

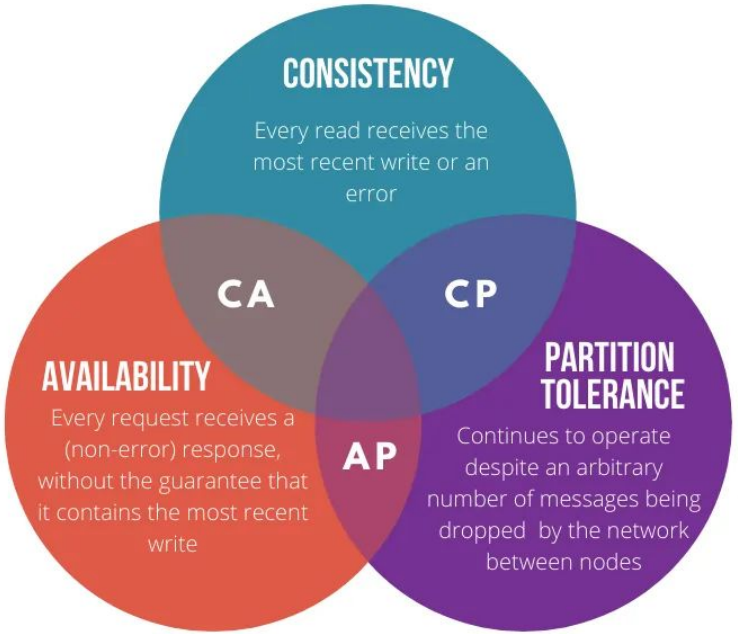
Functional vs Non-Functional Requirements



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 e.g. financial transactions
  - Read-heavy Systems
    - If writes are rare, there will be few disruptions anyway

- Choose **availability** over consistency when
  - Data integrity isn't essential e.g. 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
    - Writes are common, so disruptions would be common if consistency were chosen over availabilty

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

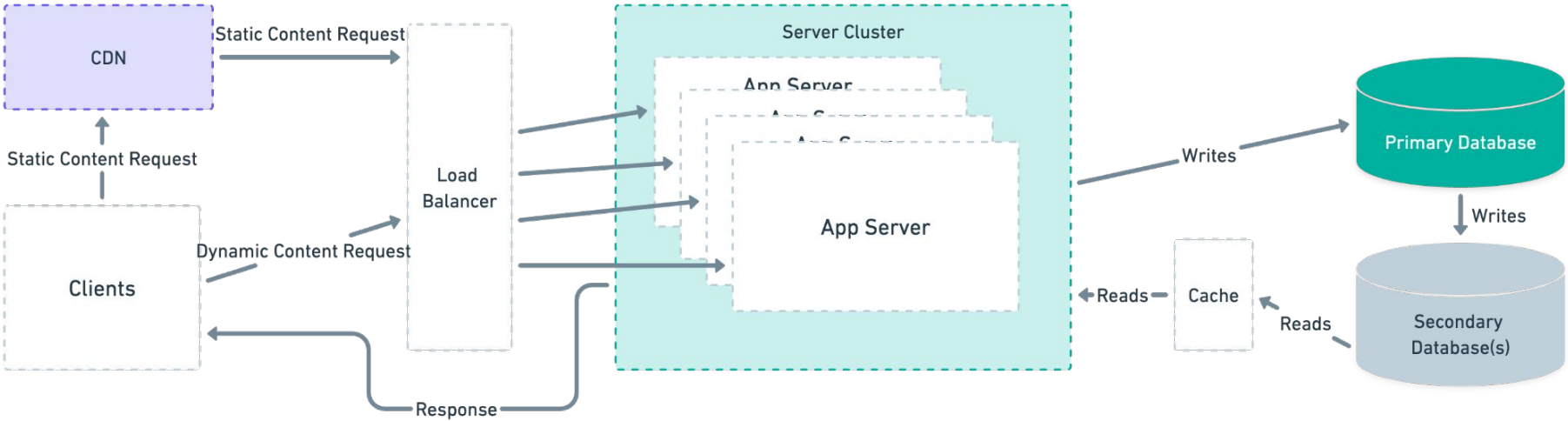
Step 1: Back-of-the-Envelope Estimations for Twitter

- 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 simple 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

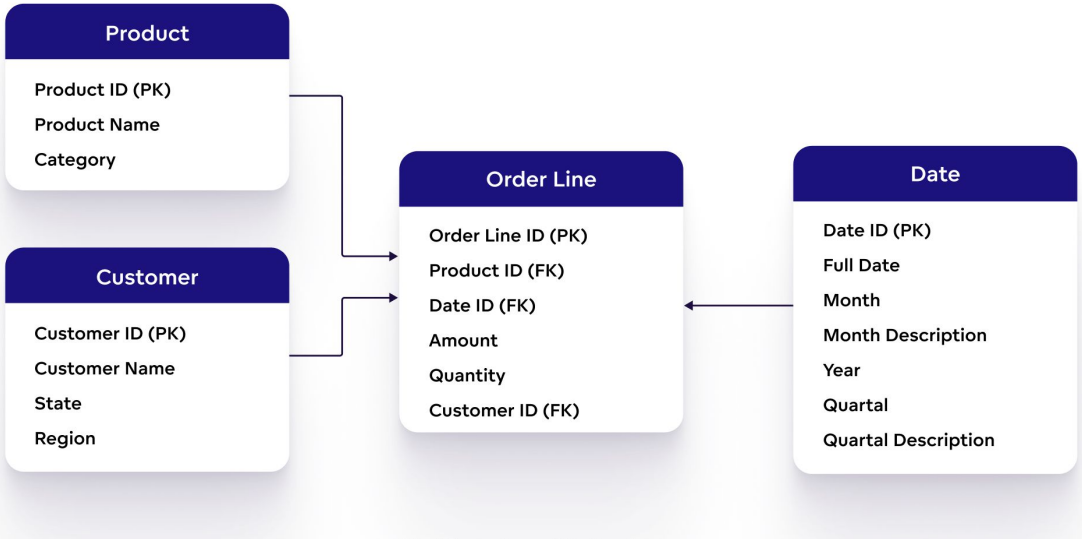
2. Define the API
    - What endpoints will you have to create?

Endpoint	`/weather`
HTTP Method	`GET`
Parameters	`integer zip`

JSON Response	`integer temp`
	`string humidity`
	`string precip`

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?



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

Step 3: Design Deep-Dive

- Make the system:
  - Faster
    - CDNs, Caches
  - Robust
    - Database Replication
    - Sharding
  - Secure
    - OAuth
    - ACLs
    - Encryption

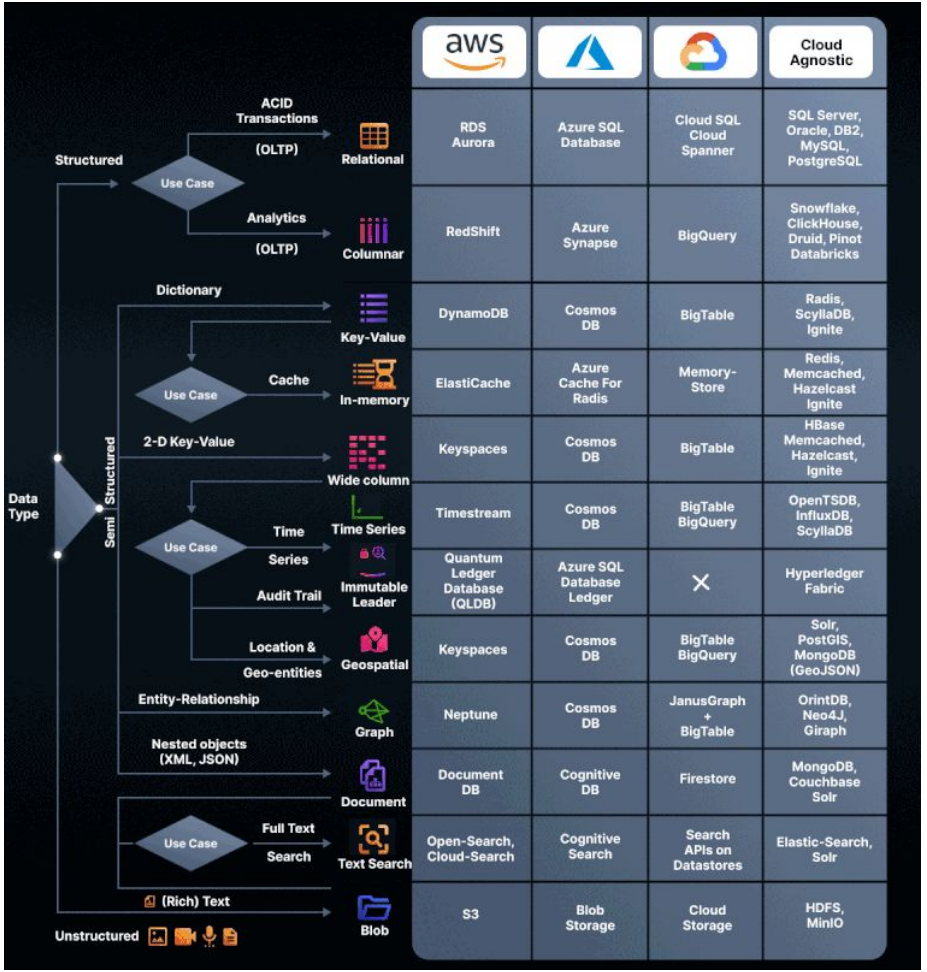
Also makes it faster

Step 3: Design Deep-Dive

- Technology Choices
  - What sort of database?
    - Relational vs NoSQL

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
- Blob storage for media
- NoSQL for performance / non-relational data



Step Four: Wrap-Up

- Give the interviewer a recap of your design
- Error Cases
- Edge Cases
- Metrics, Monitoring, Alerting
- Make space for the interviewer to pick something to drill into