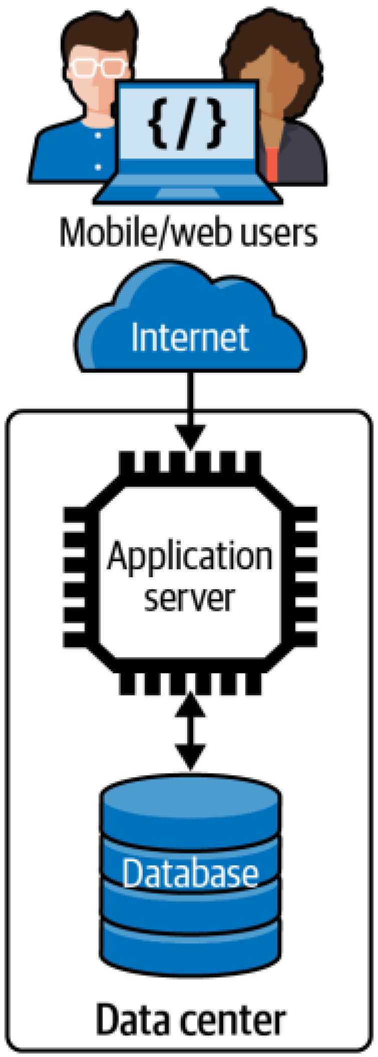
# Distributed Systems Architectures

## Introduction

- Consider that you are developing a distributed system, that accepts requests from users through web and mobile interfaces and has some *intelligent* features such as providing recommendations or notifications.

# Scaling Multi-tier Distributed Applications

- A typical software architecture for 'starter' systems would be as follows:
- Users submit requests to the application from their mobile app or Web browser.
- The **magic of Internet networking** delivers these requests to the application service
- Communications uses a standard application-level network protocol, typically HTTP.



## Scaling Multi-tier Distribu

Could be JEE,
Spring for Java,
Flask for Python, or others

- The application service code exploits a server execution environment that enables **multiple requests from multiple users** to be processed simultaneously.
- This approach leads to what is generally known as a monolithic architecture.
- If the request load stays relatively low, this application would process requests with consistency low latency.
- If the request load grows, latencies will increase as the CPU/memory becomes insufficient for concurrent requests.

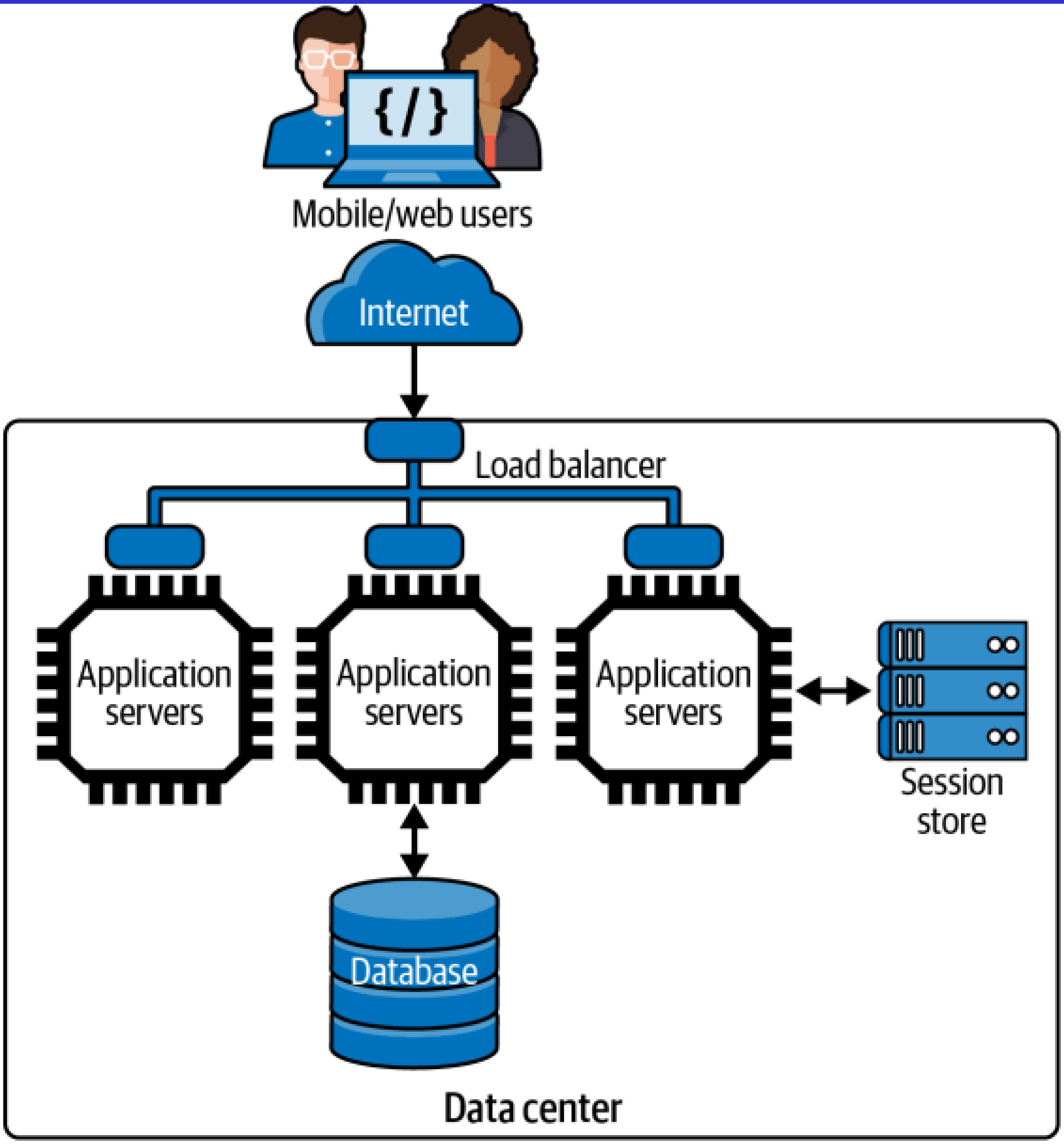
## Scaling Up Multi-tier Distributed Applications

- Scaling up the application service hardware…
- Example: "You might upgrade your server from a modest t3.xlarge instance with 4 (virtual) CPUs and 16GBs of memory to a **t3.2xlarge instance which doubles the number of CPUs and memory available** for the application "
- Pros: Simple
- Cons:
  - Failures?
  - Cost?
  - Capacity?

## Scaling Out Multi-tier Distributed Applications

- Scaling out by replicating a service and running multiple copies on multiple server nodes.
- This simple strategy increases an application’s capacity and hence scalability

# Scaling out Multi-tier Distributed Applications



## Scaling out Multi-tier Distributed Applications

- To successfully scale out an application, you need two fundamental elements:
  - Load balancer
  - Stateless services
- Load balancers receive requests and choose a service replica to process the request.
- The load balancer also relays the responses from the service back to the client.
- Load balancer requirements?

## Scaling out Multi-tier Distributed Applications

### Stateless services

- Load balancers must be able to send consecutive requests from the same client to different service instances for processing.
- Hence, the API implementations should retain no knowledge or state associated with an individual client’s session.
- When a user accesses an application, a user session is created by the service and a unique session is managed internally to identify the sequence of user interactions and track session state.
- Example of a session state?

## Scaling out Multi-tier Distributed Applications

- Scale out is attractive as, in theory, you can keep adding new (virtual) hardware and services to handle increased request loads and keep request
- Resilience to failures?
- Limitations?

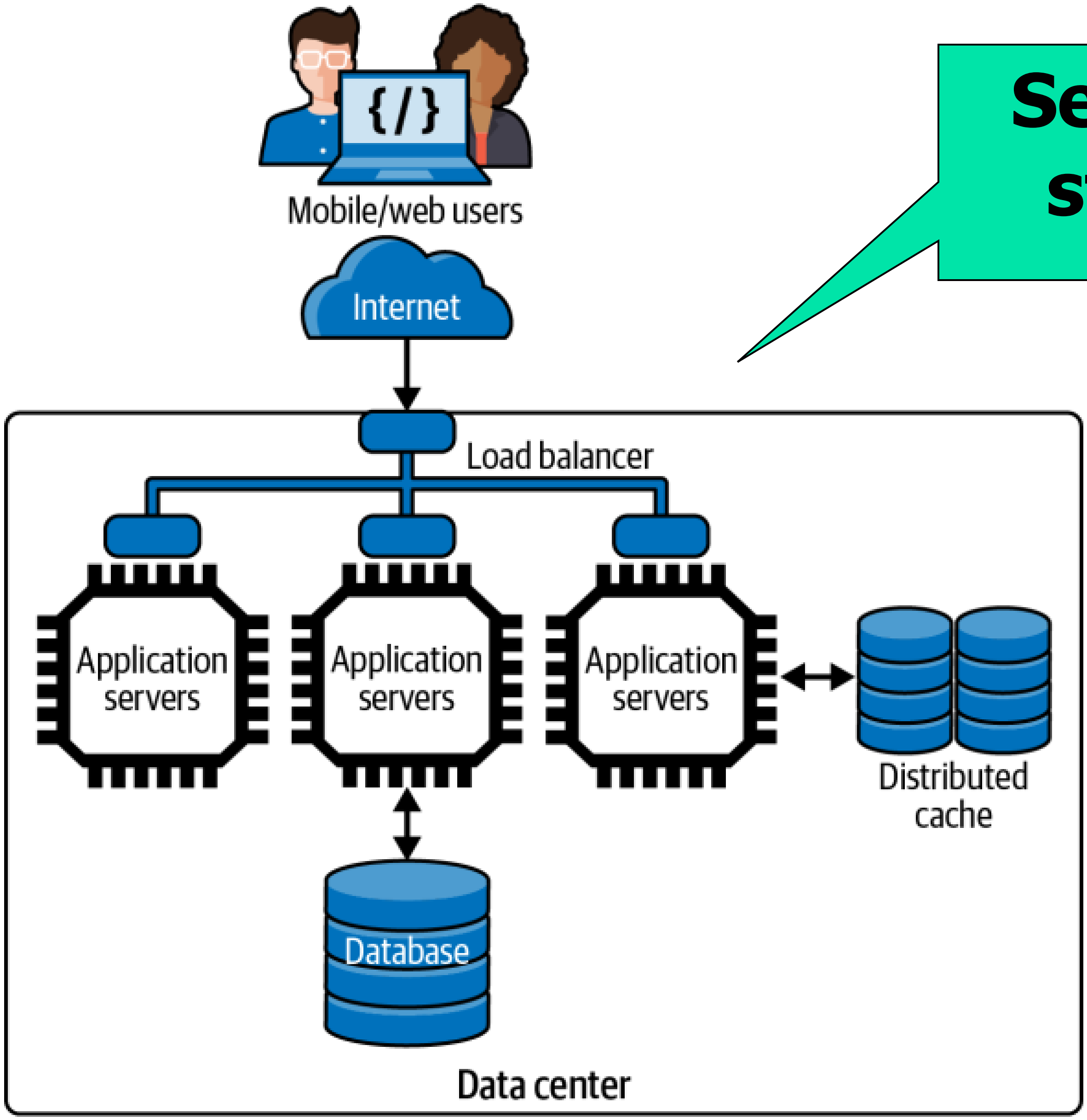
## Scaling the Database

- Scaling up by increasing the number of CPUs, memory and disks in a database server is one option.
- For example, Google Cloud Platform can provision a SQL database on a db-n1-highmem-96 node, which has 96 vCPUs, 624GB of memory, 30TBs of disk and can support 4000 connections. This will cost somewhere between $6K and $16K per year.
- Yet….Other factors need consideration…
- Alternatively, infrequent access to the database is an effective option.

## Scaling the Database **with Caching**

- Can be achieved by employing *distributed caching* in the scaled out service tier.
  - Which data can be stored?
  - Example?

# Scaling the Database **with Caching**



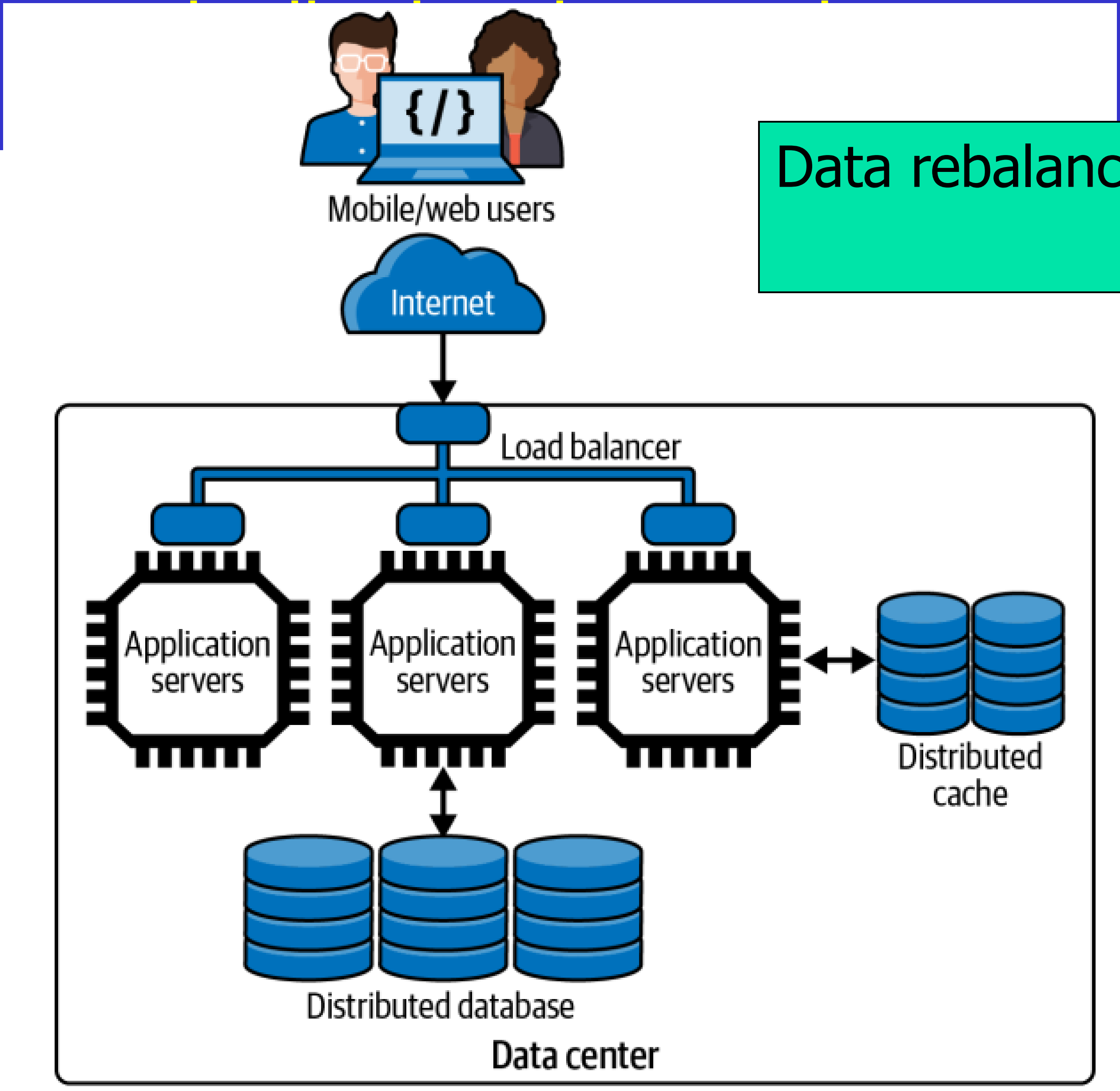
**Session store?**

## Scaling the Database **with Caching**

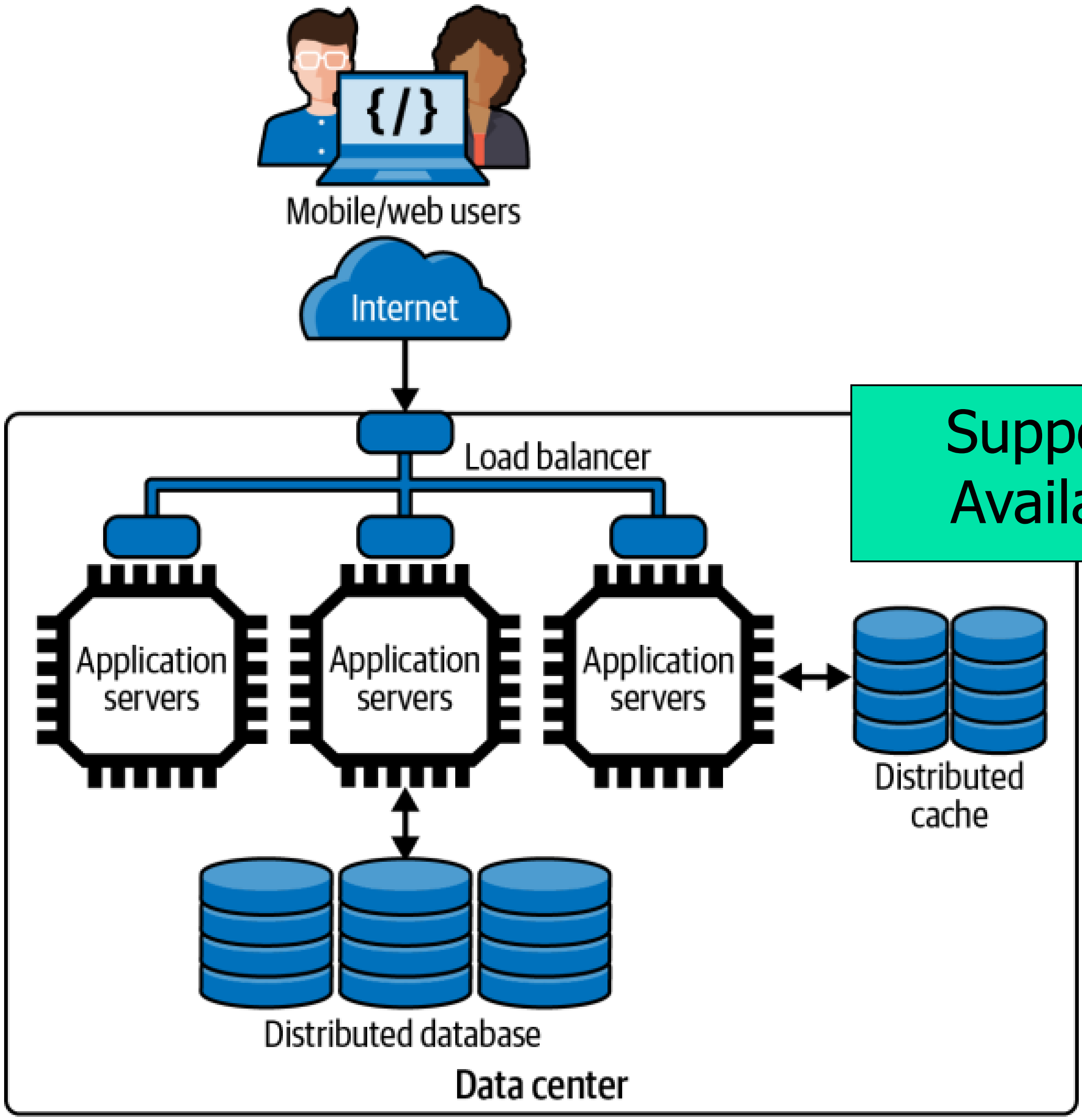
- Possible modifications to the application logic layer
  - The application logic layer needs to be modified to check for cached data within a distributed cache (such as Redis or memcached store)
  - If the cached data does not include the info, you would need to query the database and load the results into the cache as well as return it to the caller.
  - Invalidating cached results…

## Distributing the Database

- Still, many systems need to rapidly access terabyte and larger data stores that make a single database effectively prohibitive.
- In these systems, a distributed database is needed.
- A distributed database stores the data across multiple disks that are queried by multiple database engine replicas.
- These multiple engines logically appear to the application as a single database



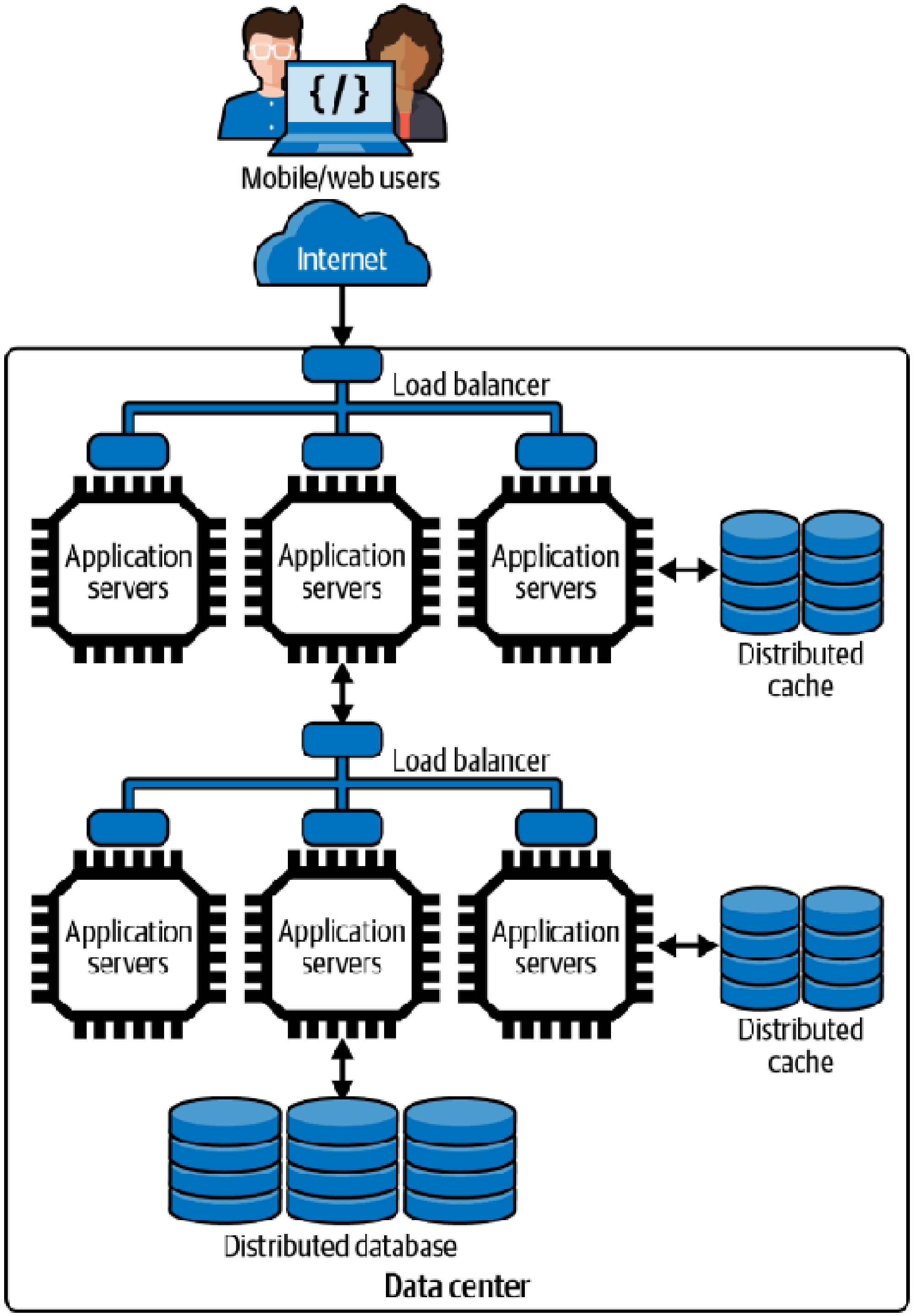
Data rebalancing?

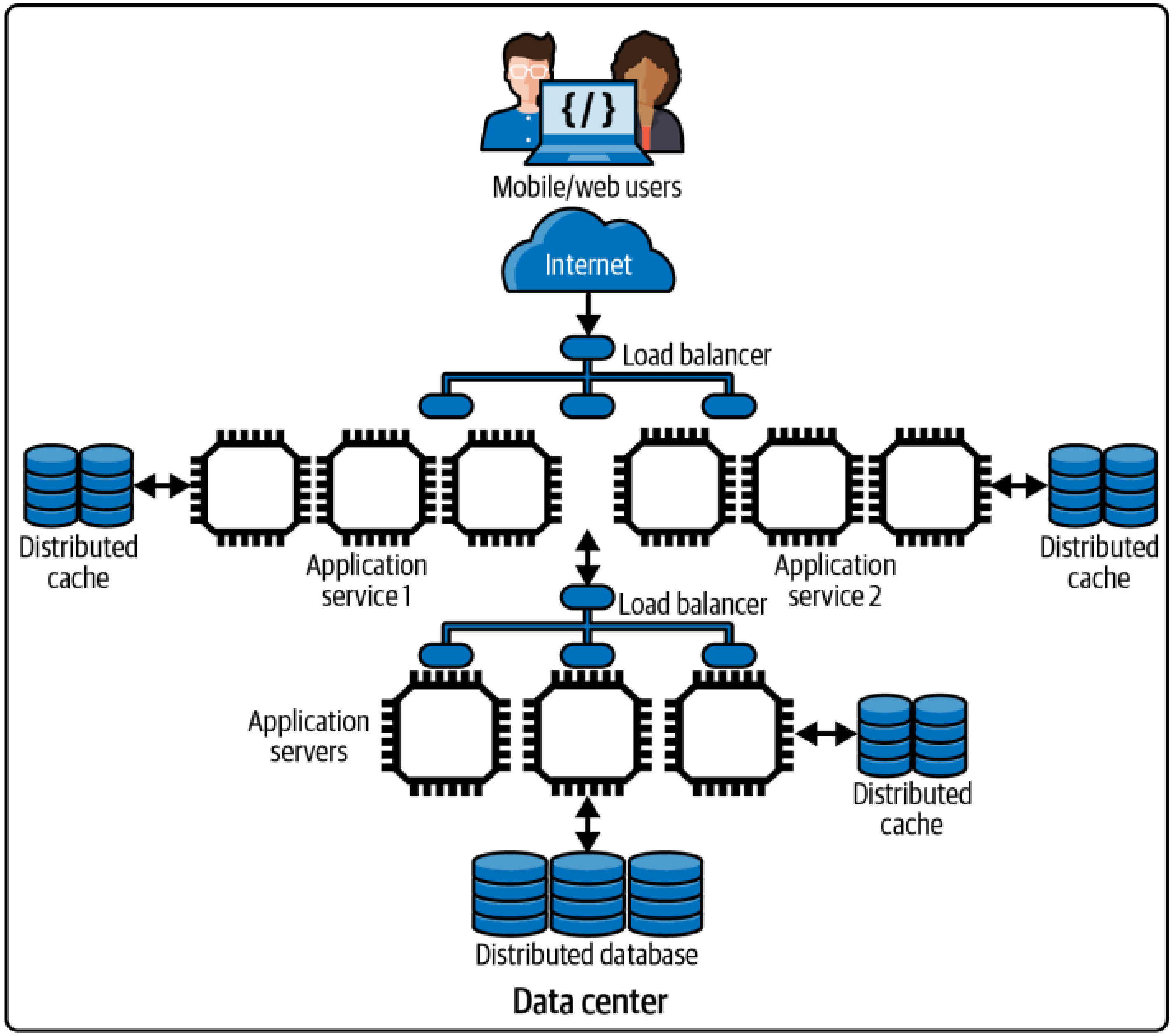


Support for Availability?

## Multiple Processing Tiers

- Any scalable system has many services that interact to process a request.
  - E.g. accessing a Web page on the Amazon.com can require in excess of 100 different services before a response is returned to the user .
- With stateless, cached, load-balanced services, we can extend these core design principles and build a multi-tiered application.
- In fulfilling a request, a service calls one or more downstream services


Mobile/web users
Internet
Load balancer
Application servers
Application servers
Application servers
Distributed cache
Load balancer
Application servers
Application servers
Application servers
Distributed cache
Distributed database
Data center


Mobile/web users
{/}
Internet
Load balancer
Distributed cache
Application service 1
Application service 2
Distributed cache
Load balancer
Application servers
Distributed cache
Distributed database
Data center

## Multiple Processing Tiers

- In addition, by breaking the application into multiple independent services, you can scale each based on the service demand.
- **If you see an increasing volume of requests from mobile users and decreasing volumes from Web users**, it’s possible to provision different numbers of instances for each service to satisfy demand.
- This is a major advantage of refactoring monolithic applications into multiple independent services, which can be separately built, tested, deployed and scaled.

## Increasing Responsiveness

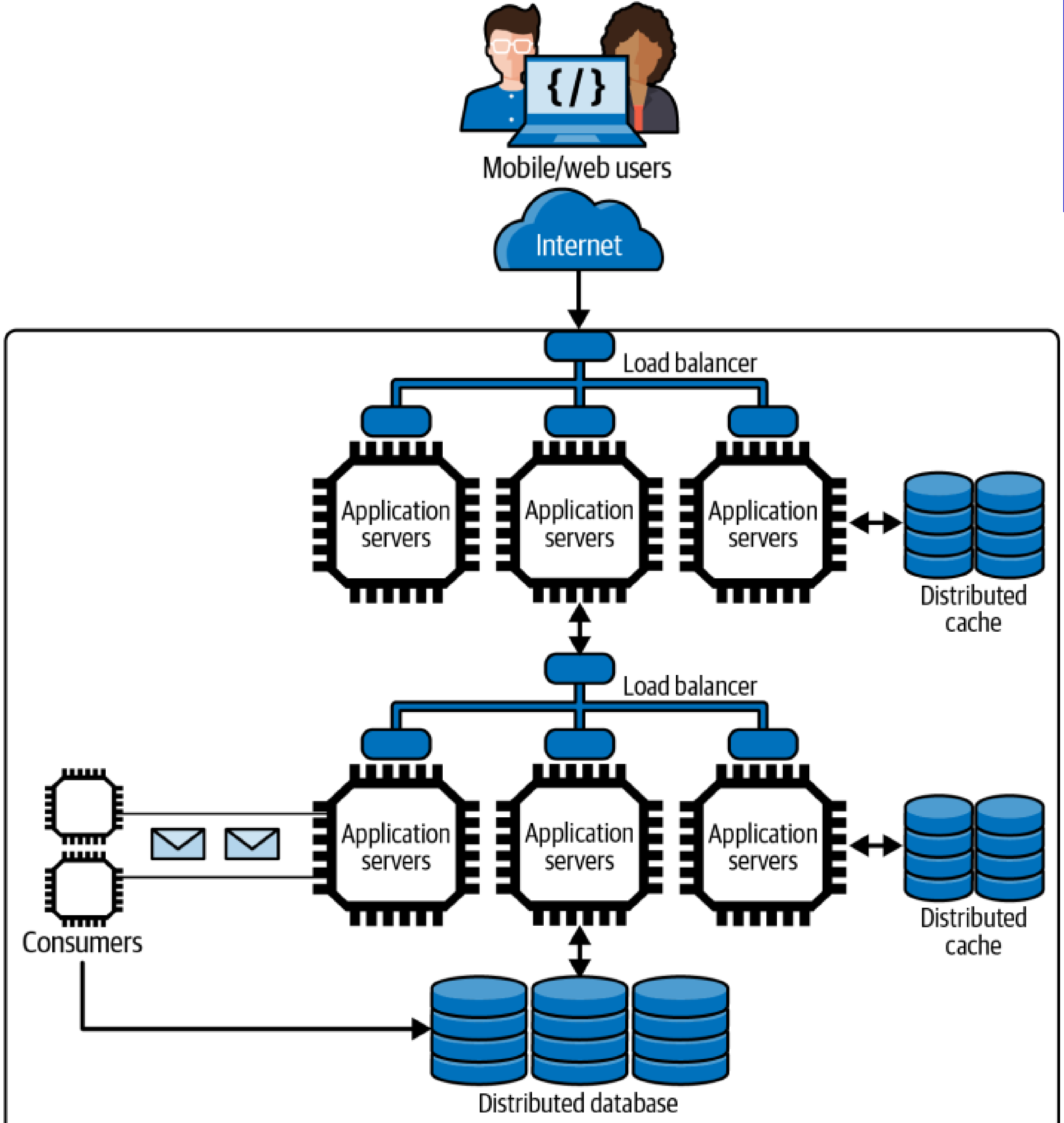
- You can decrease response times by using caching and precalculated responses, but many requests will still result in database accesses.
- For example, if a user updates their delivery address immediately prior to placing an order, the new delivery address must be persisted so that the user can confirm the address before they hit the "purchase" button.
- The response time in this case includes the time for the database write, which is confirmed by the response the user receives.

# Increasing Responsiveness

- Some update requests, however, can be successfully responded to without fully persisting the data in a database.
- The data about the event is sent to the service, which acknowledges receipt and concurrently stores the data in a remote queue for subsequent writing to the database.
- Distributed queueing platforms can be used to reliably send data from one service to another, typically but not always in a first-in, first-out (FIFO) manner.

## Increasing Responsiveness

- Writing a message to a queue is typically much faster than writing to a database,
  and this enables the request to be successfully acknowledged much more quickly.
- Another service is deployed to read messages from the queue and write the data to
  the database.
- When a user checks his data—maybe three hours or three days later—the data has been persisted successfully in the database.


Mobile/web users
Internet
Load balancer
Application servers
Application servers
Application servers
Distributed cache
Load balancer
Application servers
Application servers
Application servers
Distributed cache
Consumers
Distributed database

## Systems and Hardware Scalability

- Even the most carefully crafted software architecture and code will be limited in terms of scalability if the services and data stores run on inadequate hardware.

## Systems and Hardware Scalability

- There are some cases where upgrading the number of CPU cores and available memory is not going to buy you more scalability.
  - If code is single threaded, running it on a node with more cores is not going to improve performance. It'll just use one core at any time. The rest are simply not used.
  - If multithreaded code contains many serialized sections, only one threaded core can proceed at a time to ensure the results are correct. This phenomenon is described by Amdahl's law.

## Systems and Hardware

Two data points from Amdahl’s law are:

- If only 5% of a code executes serially, the rest in parallel, adding more than 2,048 cores has essentially no effect.
- If 50% of a code executes serially, the rest in parallel, adding more than 8 cores has essentially no effect.

## Systems and Hardware – An experiment

- The following figure shows how the throughput of a benchmark system improves as the database is deployed on more powerful (and expensive) hardware.
- The benchmark employs a Java service that accepts requests from a load generating client, queries a database, and returns the results to the client.
- The client, service, and database run on different hardware resources deployed in the same regions in the AWS cloud.

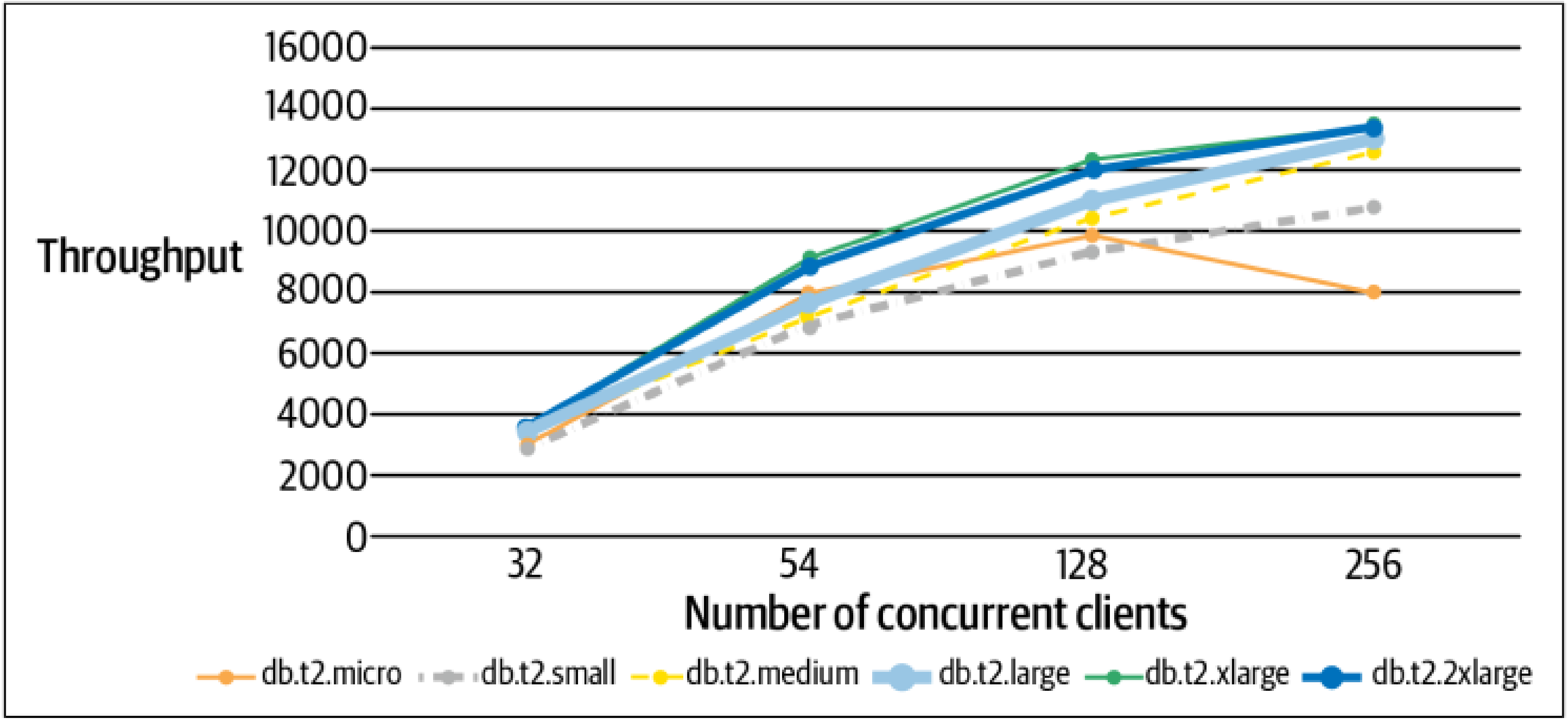


*Figure 2-8. An example of scaling up a database server*

- From this chart, it’s possible to make some straightforward observations:
  - In general, the more powerful the hardware selected for the database, the higher the throughput. That is good.
  - The difference between the db.t2.xlarge and db.t2.2xlarge instances in terms of throughput is minimal. This could be because the service tier is becoming a bottleneck, or the database model and queries are not exploiting the additional resources of the db.t2.2xlarge RDS instance.

- From this chart, it’s possible to make some straightforward observations:
  - The two least powerful instances perform pretty well until the request load is increased to 256 concurrent clients. The dip in throughput for these two instances indicates they are overloaded and things will only get worse if the request load increases.
- **Adding more hardware always increases costs, but may not always give the performance improvement you expect.**

## Google Classroom Info

- https://classroom.google.com/c/NTkzODQ3NDk3NTIw?cjc=j7xtpkz
- It is **YOUR** responsibility as a student to join this Google classroom to receive the course announcements.
- All course submissions will be done through this Google classroom

## Required Readings

- Chapter 2: “Distributed Systems Architectures: An Introduction ”, from the textbook: “Foundations of Scalable Systems”, Ian Gorton, O’reilly Media Inc., 2022.