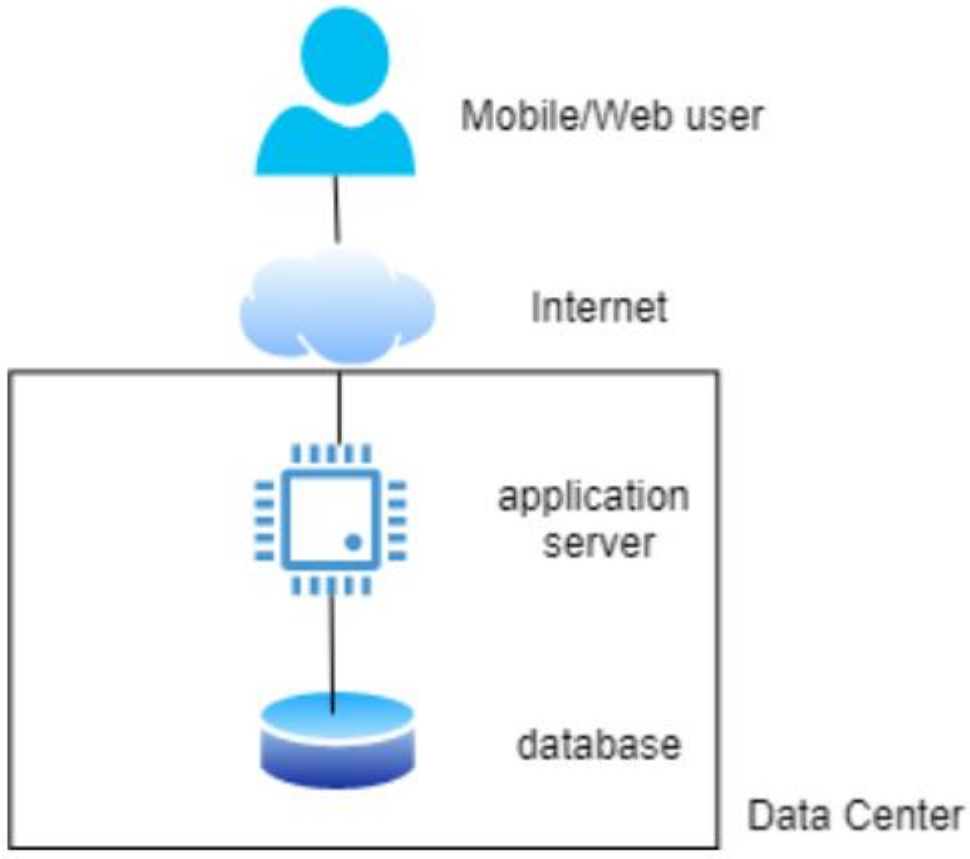
Scaling Distributed Systems

Scaling Multi-tier Distributed Applications

- A typical software architecture for ‘starter’ systems would be as follows:



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.

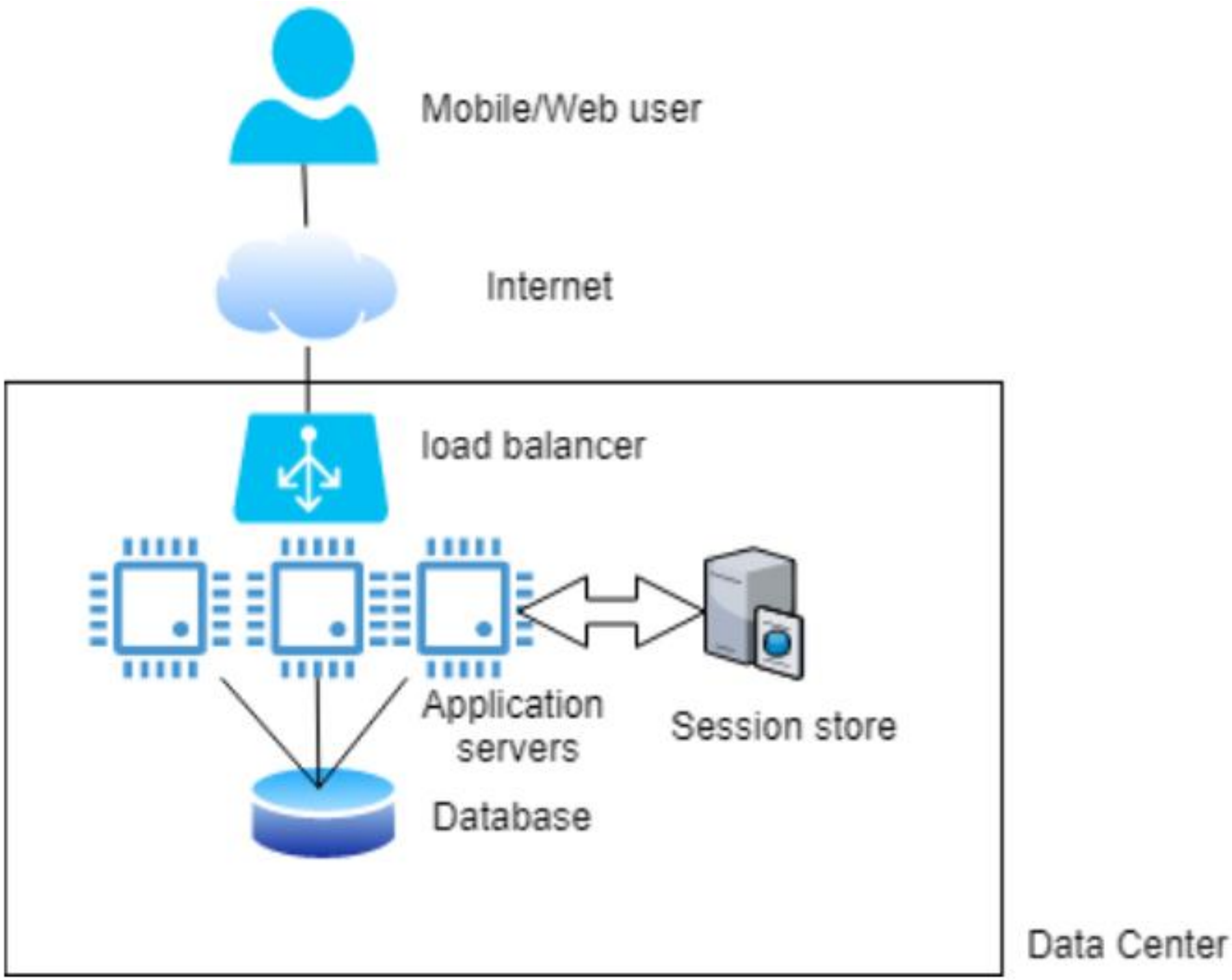
Scaling Up Multi-tier Distributed Applications

- 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.

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.

Scaling out Multi-tier Distributed Applications

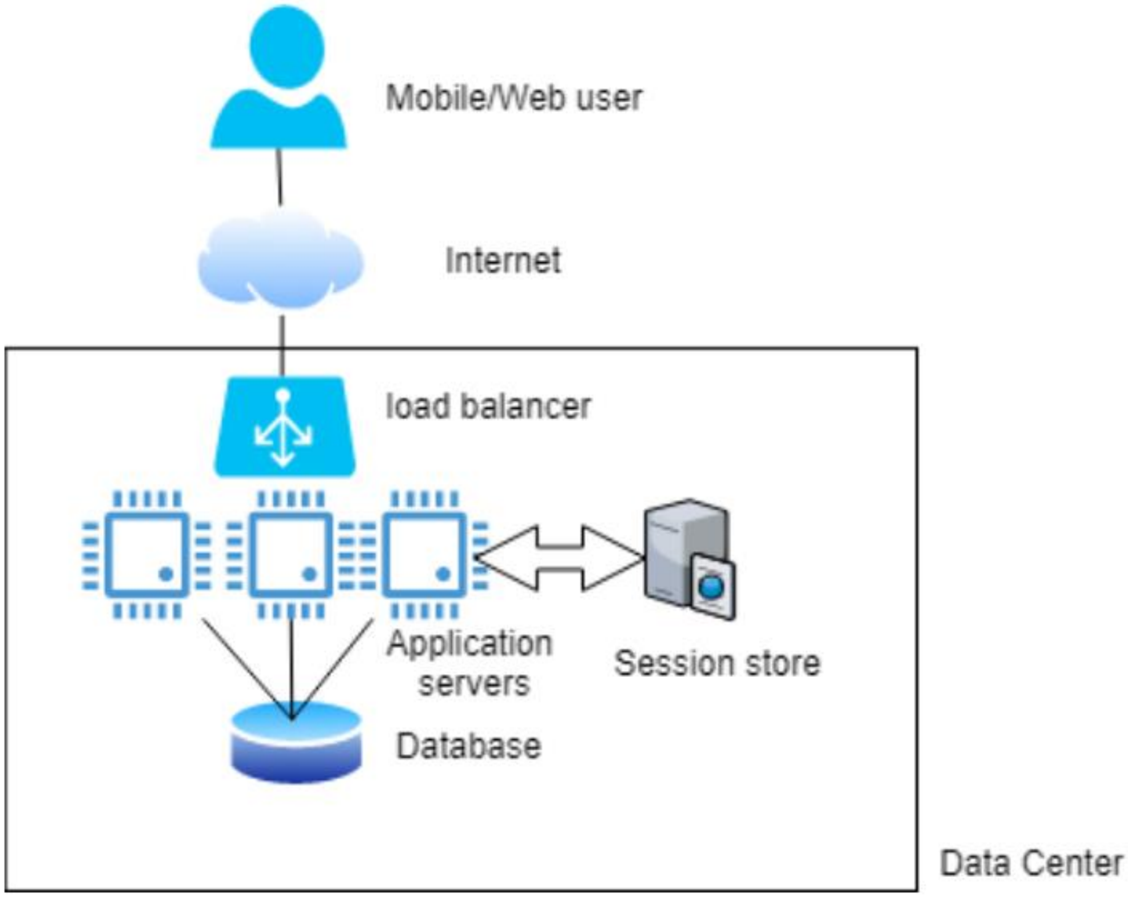


Figure 2-2. Scale out Architecture

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?

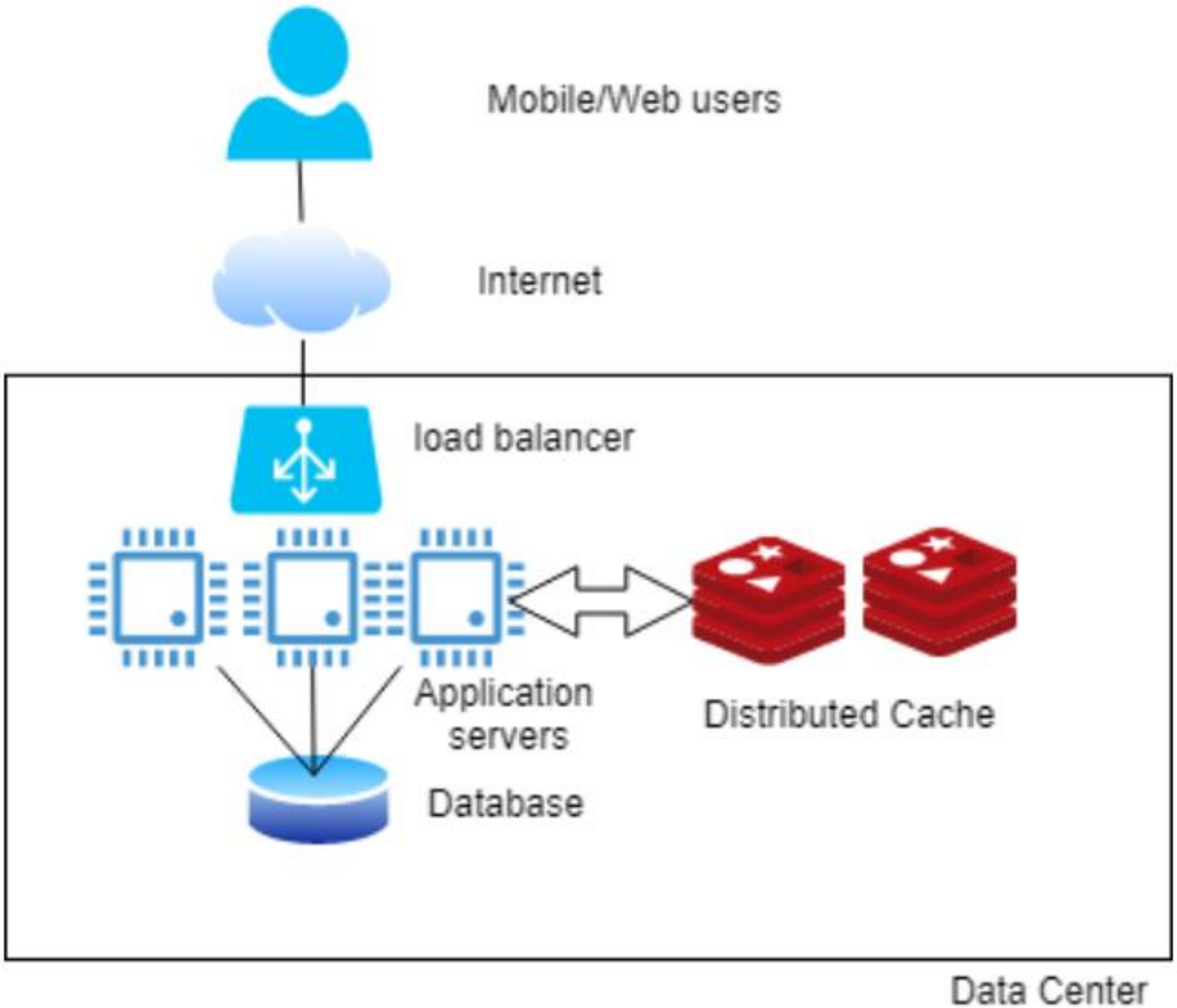
Scaling the Database

- 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**



Scaling the Database **with Caching**

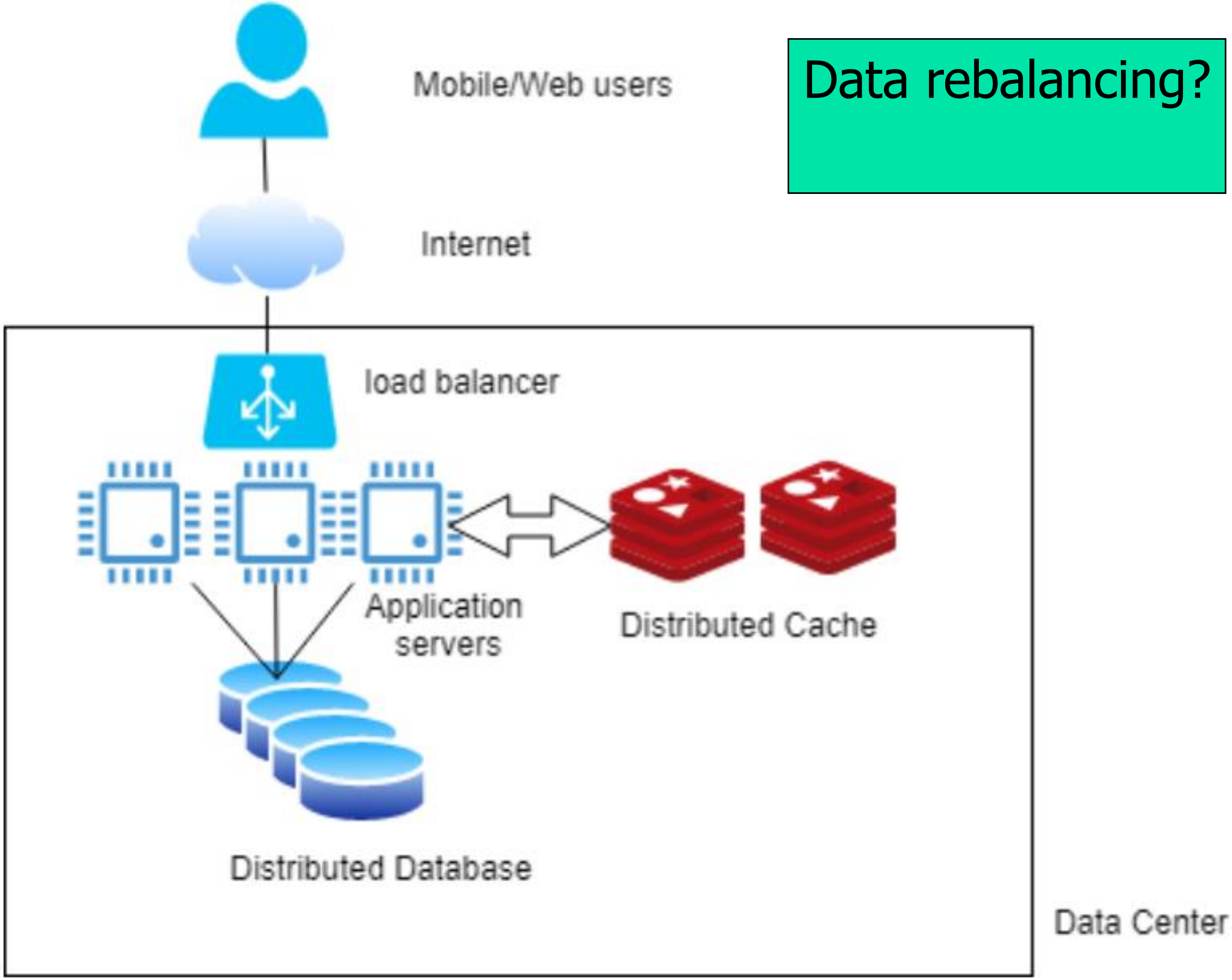
- Possible modifications to the application logic layer
  - The application logic layer needs to be modified to check for cached data.
  - If the cached data does not include the info, you would need to query the database and load the results into the cache as well as return it to the caller.
  - Invalidating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

Distributing the Database

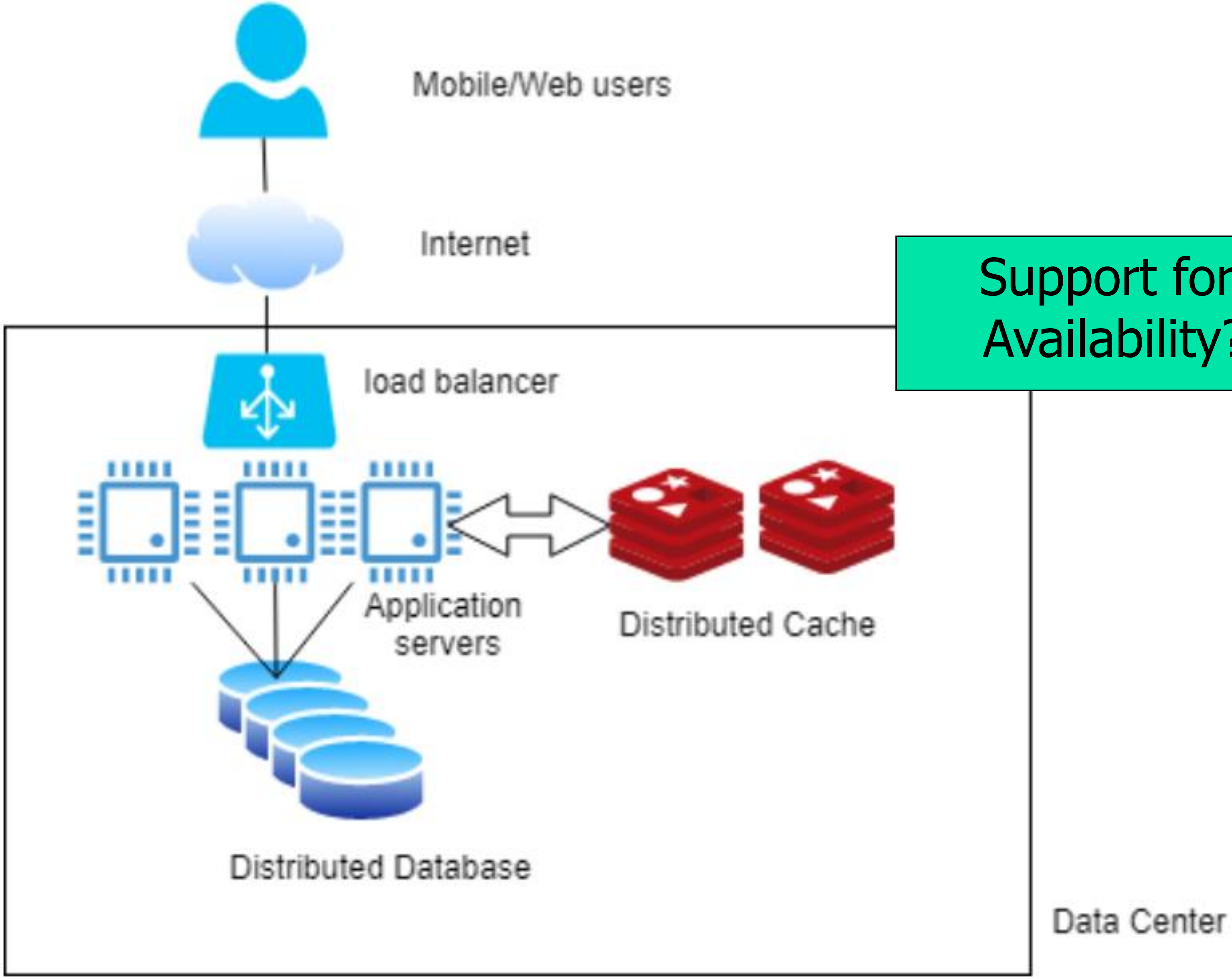
- Still, many systems need to rapidly access terabyte and larger data stores that make a single database effectively prohibitive.
- In these systems, a distributed database is needed.
- A distributed database stores the data across multiple disks that are queried by multiple database engine replicas.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

Distributing the Database

Data rebalancing?



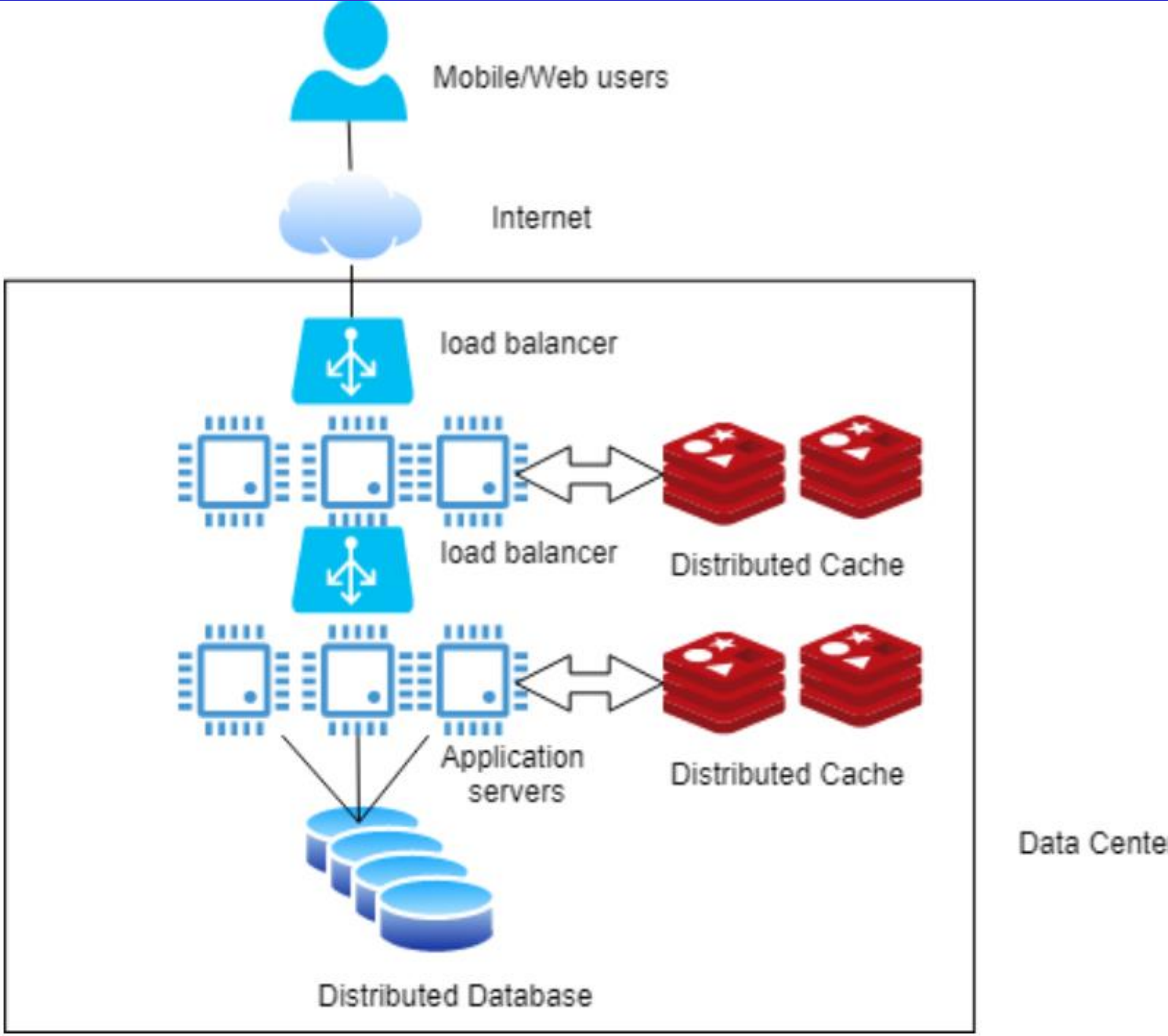
Distributing the Database

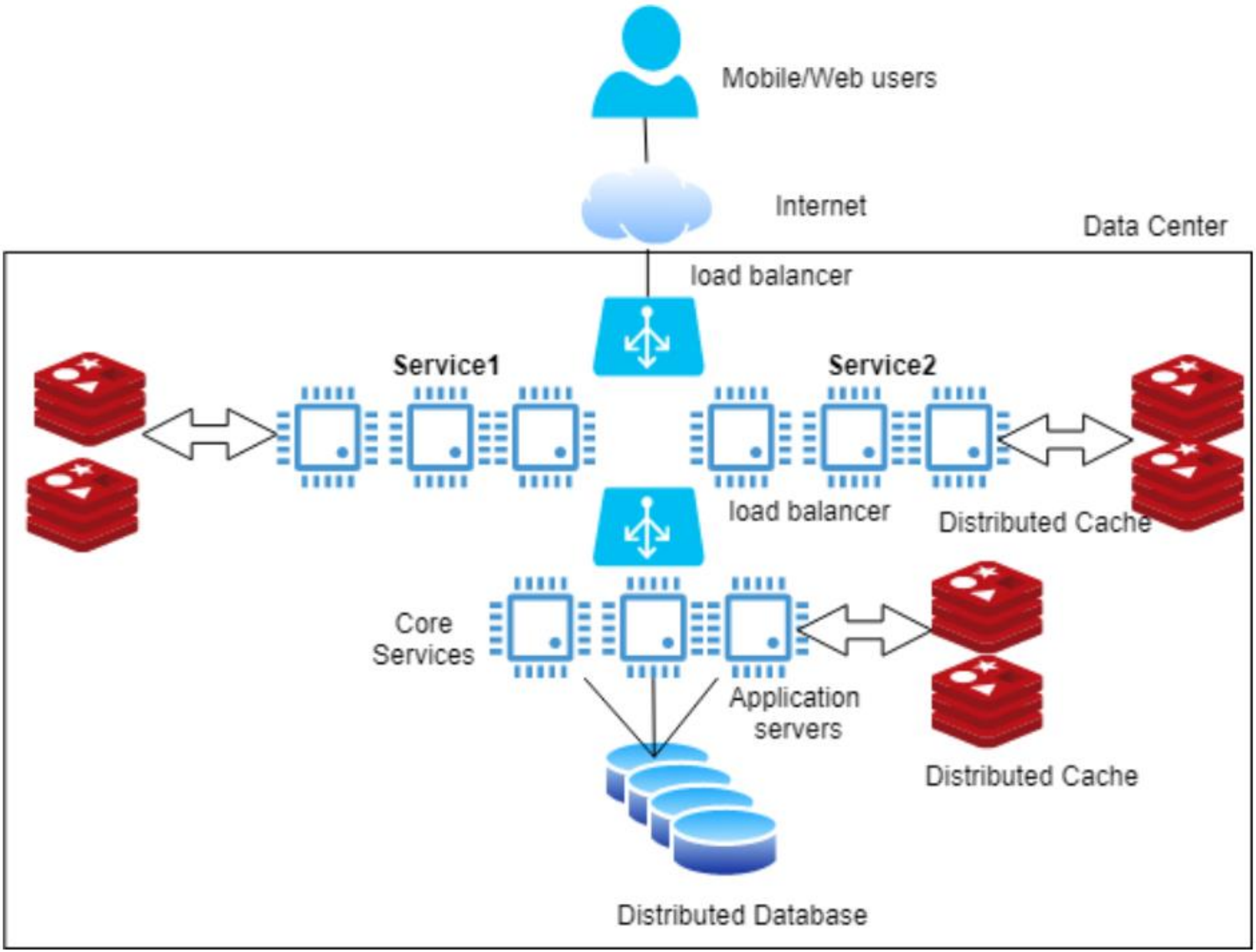


Support for Availability?

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
load balancer
Distributed Cache
Application servers
Distributed Cache
Data Center
Distributed Database


Mobile/Web users
Internet
Data Center
load balancer
Service1
Service2
load balancer
Distributed Cache
Core
Services
Application
servers
Distributed Cache
Distributed Database

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.

Required Readings

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