CHAPTER 2

Distributed Systems Architectures: A Whirlwind Tour

In this chapter we'll introduce some of the fundamental approaches to scaling a software system. The type of systems this book is oriented towards are the internet-facing systems we all utilize every day. I'll let you name your favorite. These systems accept requests from users through Web and mobile interfaces, store and retrieve data based on user requests or events (e.g. a GPS-based system), and have some *intelligent* features such as providing recommendations or providing notifications based on previous user interactions.

We'll start with a simple system design and show how it can be scaled. In the process, several concepts will be introduced that we'll cover in much more detail later in this book. Hence this chapter just gives a broad overview of these concepts and how they aid in scalability – truly a whirlwind tour!

Basic System Architecture

Virtually all massive scale systems start off small and grow due to their success. It's common, and sensible, to start with a development framework such as Ruby on Rails or Django or equivalent, which promotes rapid development to get a system quickly up and running. A typical, very simple software architecture for 'starter' systems which closely resembles what you get with rapid development frameworks is shown in Figure 1. This comprises a client tier, application service tier, and a database tier. If you use Rails or equivalent, you also get a framework which hardwires a Model-View-Controller (MVC) pattern for Web application processing and an Object-Relational Mapper (ORM) that generates SQL queries.

With this architecture, users submit requests to the application from their mobile app or Web browser. The magic of Internet networking (see Chapter 4) delivers these requests to the application service which is running on a machine hosted in some corporate or commercial cloud data center. Communications uses a standard network protocol, typically HTTP.

The application service runs code that supports an application programming interface (API) that clients use to format data and send HTTP requests to. Upon receipt of a request, the service executes the code associated with the requested API. In the process, it may read from or write to a database, depending on the semantics of the API. When the request is complete, the service sends the results to the client to display in their app or browser.

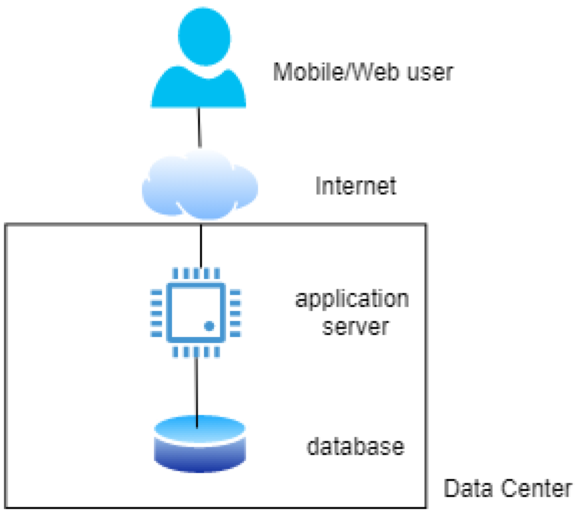

Figure 1 Basic Multi-Tier Distributed Systems Architecture

Many systems conceptually look exactly like this. The application service code exploits a server execution environment that enables multiple requests from multiple users to be processed simultaneously. There's a myriad of these application server technologies – JEE and Spring for Java, Flask for Python[1]– that are widely used in this scenario. This approach leads to what is generally known as a monolithic architecture[2]. Monoliths grow in complexity as the application becomes more feature rich. This eventually makes it hard to modify and test rapidly, and the execution footprint can become extremely heavyweight as all the API implementations run in the same application service.

Still, if request loads stay relatively low, this application architecture can suffice. The service has the capacity to process requests with consistently low latency. But if request loads keep growing, this means latencies will grow as the service has insufficient CPU/memory capacity for the concurrent request volume and hence requests will take longer to process. In these circumstances, our single server is overloaded and has become a bottleneck.

In this case, the first strategy for scaling is usually to 'scale up' the application service hardware. For example, if your application is running on AWS, 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[3].

Scale up is simple. It gets many real-world applications a long way to supporting larger workloads. It obviously just costs more money for hardware, but that's scaling for you.

It's inevitable however for many applications the load will grow to a level which will swamp a single server node, no matter how many CPUs and how much memory you have. That's when you need a new strategy – namely scaling out, or horizontal scaling, that we touched on in Chapter 1.

[1] https://en.wikipedia.org/wiki/Flask_(web_framework)

[2] Mark Richards and Neal Ford, Fundamentals of Software Architecture: An Engineering Approach 1st Edition, O'Reilly Media, 2020

[3] https://aws.amazon.com/ec2/instance-types/

Scale Out

Scaling out relies on the ability to replicate a service in the architecture and run multiple copies on multiple server nodes. Requests from clients are distributed across the replicas so that in theory, if we have N replicas, each server node processes {#requests/N}. This simple strategy increases an application's capacity and hence scalability.

To successfully scale out an application, we need two fundamental elements in our design. As illustrated in Figure 2, these are:

1) **Load balancer:** All user requests are sent to a load balancer, which chooses a service replica to process the request. Various strategies exist for choosing a target service, all with the core aim of keeping each resource equally busy. The load balancer also relays the responses from the service back to the client. Most load balancers belong to a class of Internet components known as reverse proxies[4], which control access to server resources for client requests. As an intermediary, reverse proxies add an extra network hop for a request, and hence need to be extremely low latency to minimize the overheads they introduce. There are many off-the-shelf load balancing solutions as well as cloud-provider specific ones, and we'll cover the general characteristics of these in much more detail in Chapter XXX.
2) **Stateless services:** For load balancing to be effective and share requests evenly, the load balancer must be free to send consecutive requests from the same client to different service instances for processing. This means the API implementations in the services must 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 identified is managed internally to identify the sequence of user interactions and track session state. A classic example of session state is a shopping cart. To use a load balancer effectively, the data representing the current contents of a user's cart must be stored somewhere – typically a data store – such that any service replica can access this state when it receives a request as part of a user session. In Figure 2 this is labeled as a *Session Store*.

[4] https://en.wikipedia.org/wiki/Reverse_proxy

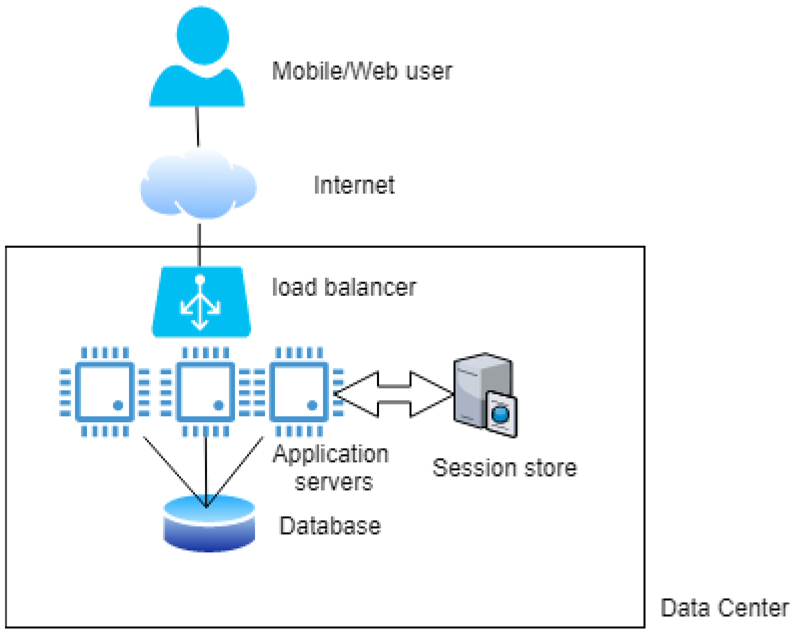

Figure 2 Scale out Architecture

Scale out is attractive as, in theory, you can keep adding new (virtual) hardware and services to handle increased request loads and keep request latencies consistent and low. As soon as you see latencies rising, you deploy another server instance. This requires no code changes and hence is relatively cheap – you just pay for the hardware you deploy.

Scale out has another highly attractive feature. If one of the services fails, the requests it is processing will be lost. But as the failed service manages no session state, these requests can be simply reissued by the client and sent to another service instance for processing. This means the application is resilient to failures in the service software and hardware, thus enhancing the application's *availability*. Availability is a key feature of distributed systems, and one we will discuss in depth in Chapter XXX.

Unfortunately, as with any engineering solution, simple scaling out has limits. As you add new service instances, the request processing capacity grows, potentially infinitely. At some stage however, reality will bite and the capability of your single database to provide low latency query responses will diminish. Slow queries will mean longer response times for clients. If requests keep arriving faster than they are being processed, some system component will fail due to resource exhaustion and clients will see exceptions and request timeouts. Essentially your database has become a bottleneck that you must engineer away in order to scale your application further.

Scaling the Database with Caching

Scaling up by increasing the number of CPUs, memory and disks in a database server can go a long way to scaling a system. For example, at the time of writing 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, which sounds a good deal to me! Scaling up is a very common database scalability strategy.

Large databases need constant care and attention from highly skilled database administrators to keep them tuned and running fast. There's a lot of wizardry in this job – e.g. query tuning, disk partitioning, indexing, on-node caching – and hence database administrators are valuable people that you want to be very nice to. They can make you application services highly responsive indeed.

In conjunction with scale up, a highly effective approach is querying the database as infrequently as possible in your services. This can be achieved by employing distributed caching in the service tier. Caching stores recently retrieved and commonly accessed database results in memory so they can be quickly retrieved without placing a burden on the database. For data that is frequently read and changes rarely, your processing logic can be modified to first check a distributed cache, such as a Redis[5] or memcached[6] store. These cache technologies are essentially distributed Key-Value stores with very simple APIs. This scheme is illustrated in Figure 3. Note that the *Session Store* from Figure 2 has disappeared. This is because we can use a general-purpose distributed cache to store session identifiers along with application data.

Accessing the cache requires a remote call from your service, but if the data you need is in the cache, on a fast network this is far less expensive than querying the database instance. Introducing a caching layer also requires your processing logic to be modified to check for cached data. If what you want is not in the cache, your code must still query the database and load the results into the cache as well as return it to the caller. You also need to decide when to remove or invalidate cached results – this depends on your application's tolerance to serving stale results to clients and the volume of data you cache.

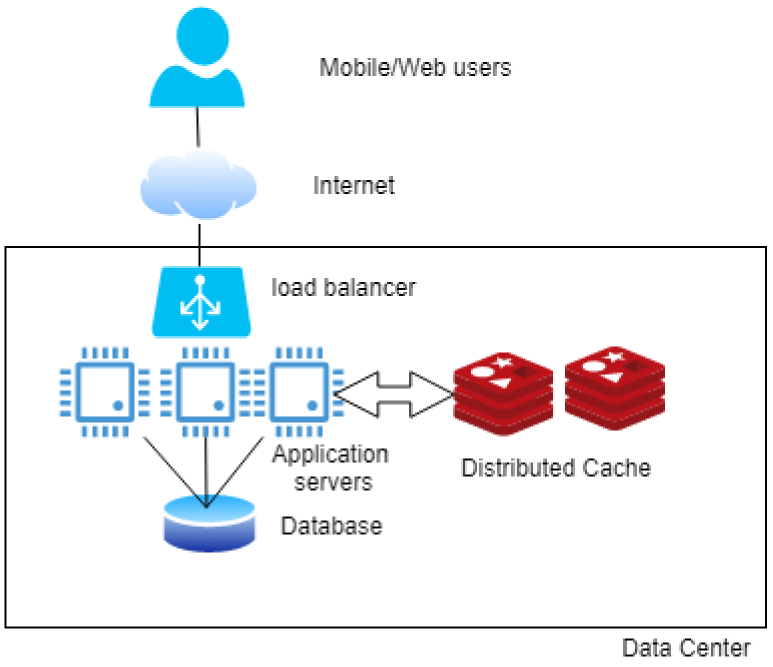

Figure 3 Introducing Distributed Caching

A well-designed caching scheme can be absolutely invaluable in scaling a system. Caching works great for data that rarely changes and is accessed frequently, such as inventory, event and

[5] https://redis.io/

[6] https://memcached.org/

contact data. If you can handle a large percentage, like 80% or more, of read requests from your cache, then you effectively buy extra capacity at your databases as they are not involved in handling requests.

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.

Distributing the Database

There are more distributed database technologies around in 2020 than you probably want to imagine. It's a complex area, and one we'll cover extensively later in Chapter XXX. In very general terms, there are two major categories:

1) Distributed SQL stores from major vendors such as Oracle and IBM. These enable organizations to scale out their SQL database relatively seamlessly by storing 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, hence minimizing code changes.
2) Distributed so-called NoSQL stores from a whole array of vendors. These products use a variety of data models and query languages to distribute data across multiple nodes running the database engine, each with their own locally attached storage. Again, the location of the data is transparent to the application, and typically controlled by the design of the data model using hashing functions on database keys. Leading products in this category are Cassandra, MongoDB and Neo4j.

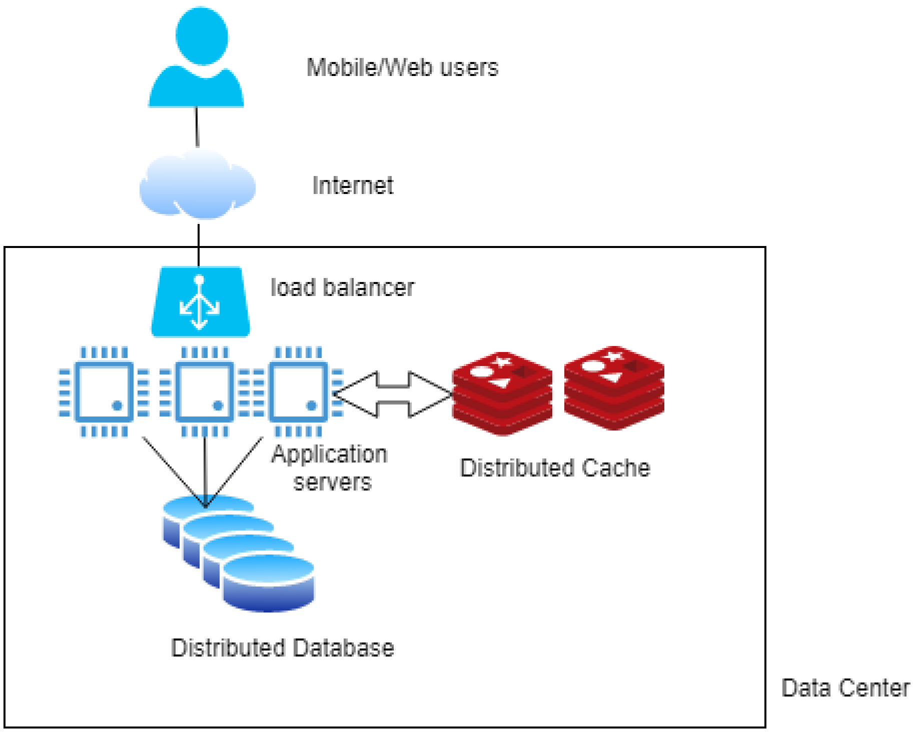

Figure 4 Scaling the Data Tier using a Distributed Database

Figure 4 shows how our architecture incorporates a distributed database. As the data volumes grow, a distributed database has features to enable the number of storage nodes to be increased. As nodes are added (or removed), the data managed across all nodes is *rebalanced* to attempt to ensure the processing and storage capacity of each node is equally utilized.

Distributed databases also promote availability. They support replicating each data storage node so if one fails or cannot be accessed due to network problems, another copy of the data is available. The models utilized for replication and the trade-offs these require (spoiler – consistency) are covered in later chapters.

If you are utilizing a major cloud provider, there are also two deployment choices for your data tier. You can deploy your own virtual resources and build, configure, and administer your own distributed database servers. Alternatively, you can utilize cloud-hosted databases. The latter simplifies the administrative effort associated with managing, monitoring and scaling the database, as many of these tasks essentially become the responsibility of the cloud provider you choose. As usual, the no free lunch principle applies.

Multiple Processing Tiers

Any realistic system that we need to scale will have many different services that interact to process a request. For example, accessing a Web page on the Amazon.com web site can require in excess of 100 different services being called before a response is returned to the user[7].

The beauty of the stateless, load balanced, cached architecture we are elaborating in this chapter is that we can extend the core design principles and build a multi-tiered application. In fulfilling a request, a service can call one or more dependent services, which in turn are replicated and load-balanced. A simple example is shown in Figure 5. There are many nuances in how the services interact, and how applications ensure rapid responses from dependent services. Again, we'll cover these in detail in later chapters.

[7] https://www.allthingsdistributed.com/2019/08/modern-applications-at-aws.html

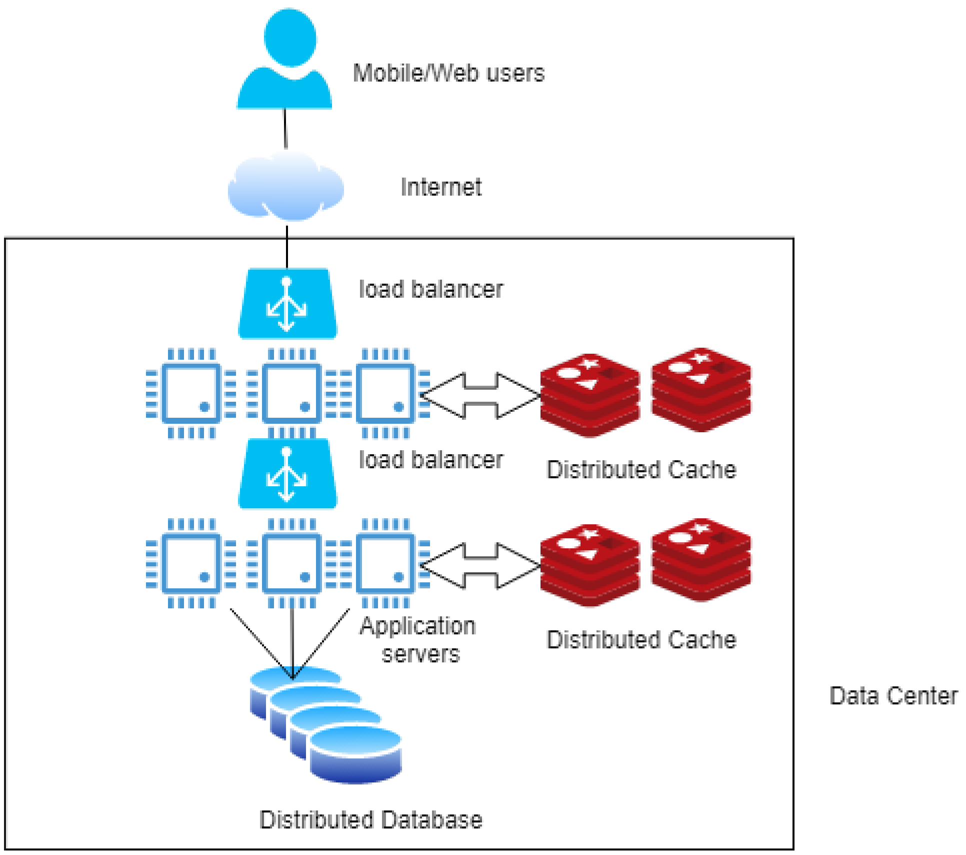

Figure 5 Scaling Processing Capacity with Multiple Tiers

This design also promotes having different, load balanced services at each tier in the architecture. For example, Figure 6 illustrates two replicated Internet-facing services that both utilized a core service that provides database access. Each service is load balanced and employs caching to provide high performance and reliability. This design is often used to provide a service for Web clients and a service for mobile clients, each of which can be scaled independently based on the load they experience. Its commonly called the Backend For Frontend (BFF) pattern[8].

[8] https://samnewman.io/patterns/architectural/bff/

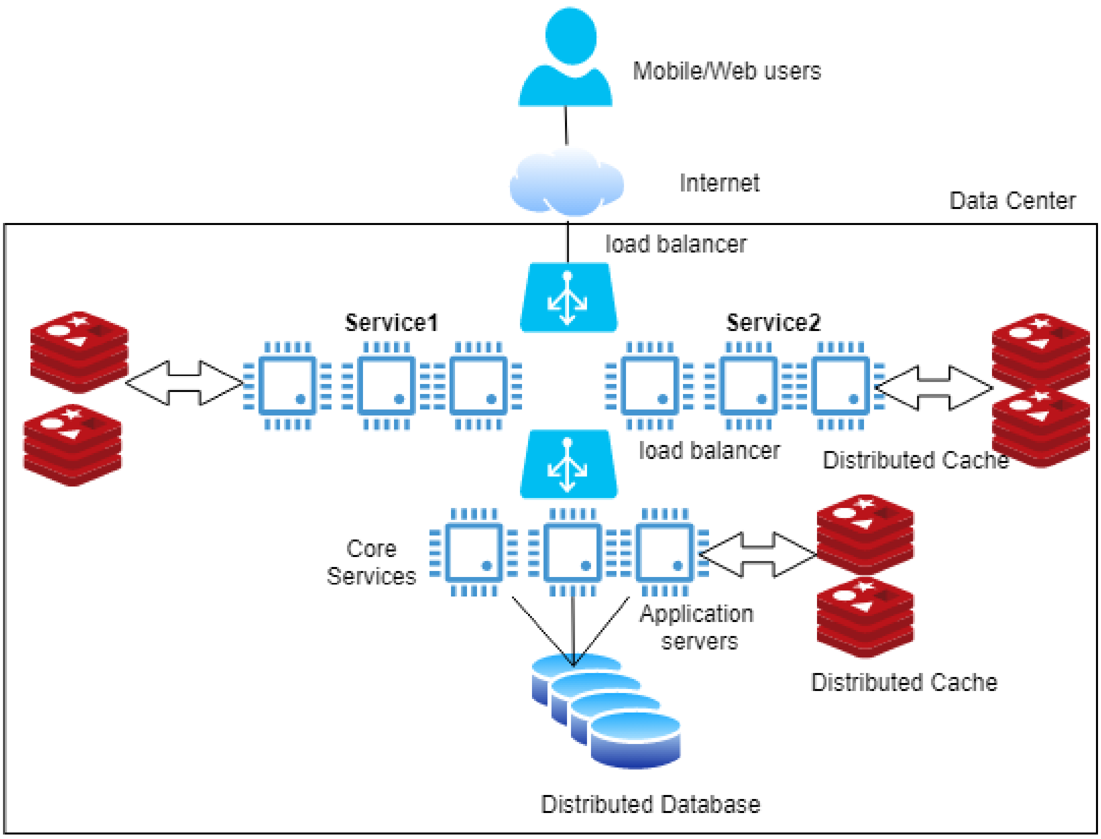

Figure 6 Scalable Architecture with Multiple Services

In addition, by breaking the application into multiple independent services, we can scale each based on the service demand. If for example we see an increasing volume of requests from mobile users and decreasing volumes from Web users, we can 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

Most client application requests expect a response. A user might want to see all auction items for a given product category or see the real estate that is available for sale in a given location. In these examples, the client sends a request and waits until a response is received. This time interval between sending the request and receiving the result is the latency of the request. We can decrease latencies by using caching and precalculated responses, but many requests will still result in a database access.

A similar scenario exists for requests that update data in an application. 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 latency in this case includes the time for the database write, which is confirmed by the response the user receives.

Some update requests however can be successfully responded to without fully persisting the data in a database. For example, the skiers and snowboarders amongst you will be familiar with lift ticket scanning systems that check you have a valid pass to ride the lifts that day. They also record which lifts you take, the time you get on, and so on. Nerdy skier/snowboarders can then use the resort's mobile app to see how many lifts they ride in a day.

As a person waits to get on a lift, a scanner device validates the pass using an RFID chip reader. The information about the rider, lift, and time are then sent over the Internet to a data capture service operated by the ski resort. The lift rider doesn't have to wait for this to occur, as the latency could slow down their loading process. There's also no expectation from the lift rider that they can instantly use their app to ensure this data has been captured. They just get on the lift, talk smack with their friends, and plan their next run.

Service implementations can exploit this type of scenario to improve responsiveness. 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. Writing a message to a queue is 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 the user checks their lift rides – maybe 3 hours or 3 days later – the data has been persisted successfully. The basic architecture to implement this approach is illustrated in Figure 7.

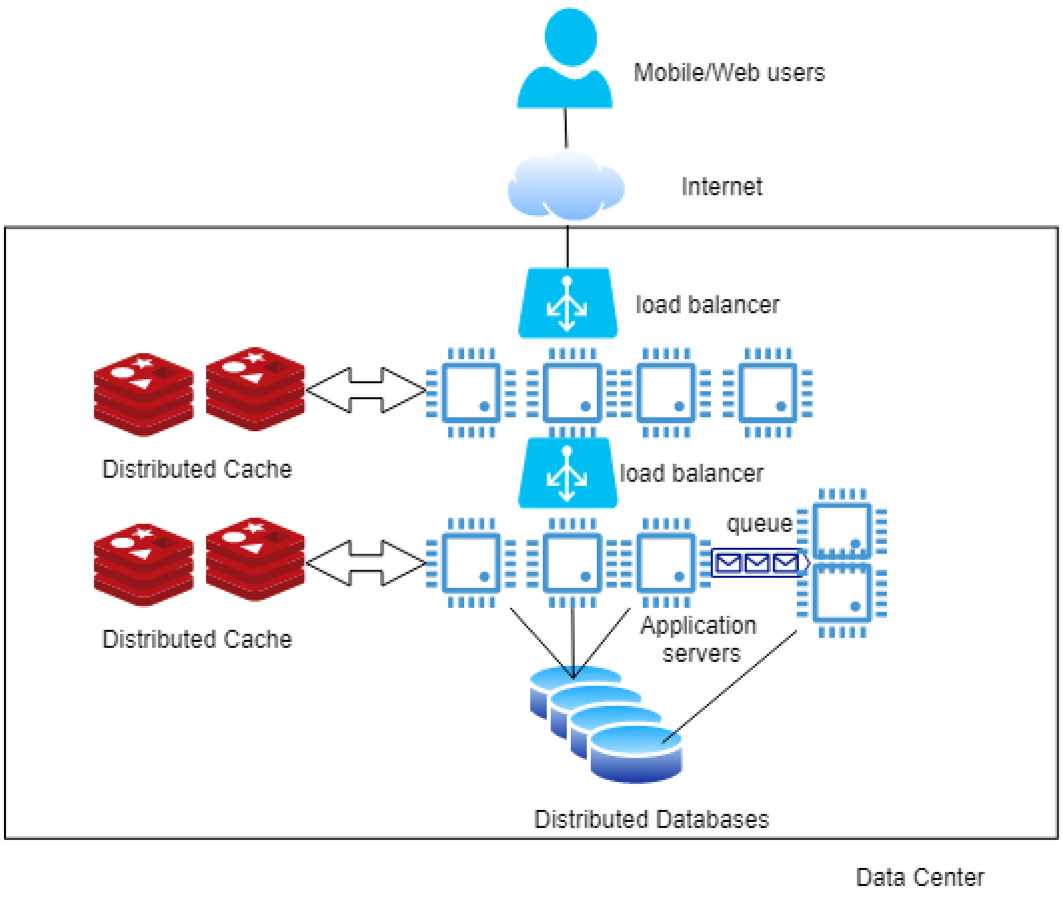

Figure 7 Increasing Responsiveness with Queueing

Whenever the results of a write operation are not immediately needed, an application can use this approach to improve responsiveness and hence scalability. Many queueing technologies exist that applications can utilize, and we'll discuss how these operate in later chapters. These queueing platforms all provide asynchronous communications. A *producer* service writes to the queue, which acts as temporary storage, while another *consumer* service removes messages from the queue and makes the necessary updates to, in our example, a database that stores skier lift ride details.

The key is that the data *eventually* gets persisted. Eventually typically means a few seconds at most but use cases that employ this design should be resilient to longer delays without impacting the user experience.

Summary and Further Reading

This chapter has provided a whirlwind tour of the major approaches we can utilize to scale out a system as a collection of communicating services and distributed databases. Much detail has been brushed over, and as you no doubt know, in software systems the devil is in the detail. Subsequent chapters will therefore progressively start to explore these details, starting with concurrent and distributed systems fundamentals. Next there are sections on the application service and data management tiers as portrayed in the basic distributed systems architecture blueprint described in this chapter.

Another area this chapter has skirted around is the subject of software architecture. We've used the term *services* for distributed components in an architecture that implement application business logic and database access. These services are independently deployed processes that communicate using remote communications mechanisms such as HTTP. In architectural terms, these services are most closely mirrored by those in the Service Oriented Architecture (SOA) pattern, an established architectural approach for building distributed systems. A more modern evolution of this approach revolves around microservices. These tend to be more cohesive, encapsulated services that promote continuous development and deployment.

We'll touch on these differences in a later chapter. If you'd like a much more in depth discussion of these, and software architecture issues in general, then Mark Richards' and Neal Ford's book is an excellent place to start.

Mark Richards and Neal Ford, Fundamentals of Software Architecture: An Engineering Approach 1st Edition, O'Reilly Media, 2020

Finally, there's a class of *big data* software architectures that address some of the issues that come to the fore with very large data collections. One of the most prominent is data reprocessing. This occurs when data that has already been stored and analyzed needs to be re-analyzed due to code changes. This reprocessing may occurs due to software fixes, or the introduction of new algorithms that can derive more insights from the original raw data. There's a good discussion of the Lambda and Kappa architectures, both of which are prominent in this space, in this article.

Jay Krepps, Questioning the Lambda Architecture, https://www.oreilly.com/radar/questioning-the-lambda-architecture/