Chapter 3

Infrastructure as a Service

Infrastructure as a Service (IaaS) is the most fundamental service model as it provides users access to the virtualized hardware, including processors, storage, and networking, and allows them to build an entire software stack that they desire, including operating systems, libraries and runtimes, and applications.

In this chapter, we will learn the key services that an IaaS providet offers and how to use these services to develop a cloud application. We will often use the IaaS services from Amazon Web Services (AWS) as examples. We choose AWS because it provides representative IaaS services that can be found in other cloud providers; it is also the first IaaS provider and today the largest cloud provider in the world.

There are three key types of IaaS services that we typically need to develop a cloud app: compute (AWS EC2), which provides computing resources for executing the applications, storage (AWS EBS, S3), which provides storage resources for storing the data that applications needs to access, and messaging (AWS SQS) for the distributed applications and their distributed components to communicate across the network.

3.1 Motivating Example

Before we discuss the IaaS services in detail, let us look at an interesting success story of IaaS. Animoto.com is a SaaS provider that provides video making as a service. Its customers can upload materials that they want to use, such as photos and video clips, and use the provided templates to customize and generate their videos. On April 18th 2008, Animoto.com launched a viral ad campaign on Facebook. As illustrated in Figure 3.1, this led to a huge spike in the demand of its service, resulting in its number of users growing from 5,000 to 750,000 in just three days. Fortunately, Animoto.com was hosted on AWS, and it was able to handle this sudden surge of demand by quickly growing its number of EC2 instances from 40 to 4000. Even though it is a really outdated story, it still serves as a good example of the power of cloud computing. Without the

cloud, a company like Animoto.com would not be able to deal with the sudden change of demand quickly and cost-effectively.

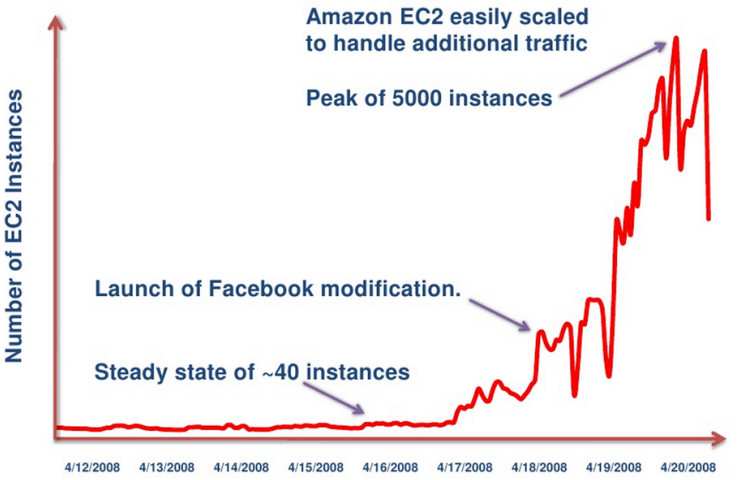

Figure 3.1: The success story of Animoto.com on EC2

In the following, we will learn how to use the basic IaaS resources. Although the discussions here are based on AWS, we can also find similar IaaS resources from the other providers.

3.2 Compute

Compute is the most basic service that a cloud needs to provide to allow its customers to run their applications and perform computations. With IaaS, compute resources are provided in the form of virtual machines (VMs). VMs serve two important purposes for an IaaS cloud. First, VMs are resource containers, each providing the necessary hardware resources, including CPUs, memory, and storage and network devices, for a customer to run her applications. At the same time, the amount of resources that an application can consume is also defined by the VM where it runs. For the sake of manageability, IaaS providers commonly offer a limited category of different types of VMs, called instance types in EC2. The various VM types differ in the amount of CPU, memory, and I/O resources. For example, t2.micro, the instance type that the AWS Free Tier provides, has 1 vCPU and 1 GB of memory; t2.large has 2 vCPUs and 8 GB of memory. Some VM types also differ in capabilities. For example, m5d.large has a local SSD to provide faster storage to the VM; p2.xlarge has a GPU for accelerating the VM's workload. To achieve vertical scaling, a user can change the type of VMs that her application runs on; to achieve horizontal scaling, she can change the number of instances of the same VM type that her

application uses.

The second important purpose of VMs in an IaaS cloud is to provide the execution environments, including OSes and libraries/runtimes, that the customers' applications need to run. A user can install and customize the entire software stack in a VM, which is then encapsulated by the VM image. With this image, the user can start VM instances of any type of her choice on any IaaS resource in the cloud. As discussed in Chapter 1, users can specify the location of their instances at some higher level of abstraction, e.g., Availability Zones (AZs) in AWS, but they have no control or even knowledge of the location at lower levels of abstraction such as individual servers that are used to host the VMs.

VM instances are instantiated from VM images. The relationship between instances and images is analogous to the relationship between processes and programs. A program contains the code and data, and is often stored as a file on a file system. When the program is brought into memory and executed on CPU, it becomes a process which is an instance of the program in execution. Many processes can be instantiated from the same program, and each process can be given a different amount of resources to execute. A VM image contains an entire file system that stores the entire software stack, including the OS, system programs, libraries, and application programs needed for starting a machine. When these programs from the image are loaded into memory and executed on CPU of a host, they become an instance of the VM in execution. Many VM instances can be launched from the same image, and each VM instance can be launched using a different instance type and therefore executed with a different amount of resources specified by the instance type.

Specifically in AWS, VM images for launching EC2 VM instances are managed as Amazon Machine Images (AMIs). AWS provides a catalog of different AMIs with preinstalled OSes such as Linux, Windows, and MacOS and even preinstalled software frameworks such as PyTorch for machine learning and Hadoop for data analytics. Users can also customize these template images or upload their own images as AMIs for launching EC2 instances that have their desired software environment.

During the lifecycle of an instance, it goes through various states as shown in the state diagram Figure 3.3. An instance is launched from an AMI and starts to run. A running instance can be rebooted, stopped, or terminated. The difference between the latter two cases is that a stopped instance's state still exists in the system and can be started again later, whereas a terminated instance is removed from the system. Both cases do not incur EC2 charges since the instance is not running on EC2 any more, but a stopped instance still incurs charges for storing the instance's state, which will be explained further in the following IaaS storage section.

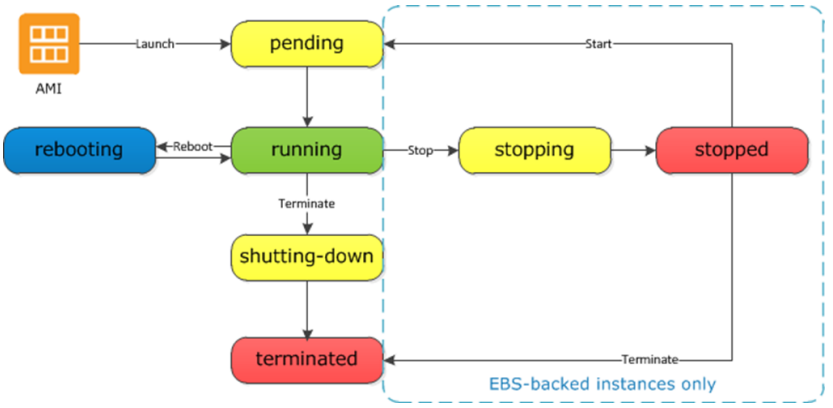

Figure 3.2: The lifecycle of EC2 instances

3.3 Storage

There are generally three types of storage resources in an IaaS cloud to support VMs, including local volume storage, remote volume storage, and general-purpose storage. Each of them has different characteristics and can be used in different ways to support IaaS VMs and applications. We need to understand these storage resources and use them properly.

An IaaS user manages both the VMs and the applications running on the VMs, and the user needs storage for two types of data: VM data and application data. The former is what the VMs needs to run, and the latter is the applications need to run, including input and output.

Let us first understand the concept of disk volumes which is the basic abstraction for VM storage. Just like a physical machine can have multiple disks, a VM has multiple virtual disks. Each disk is typically installed with a file system to allow the applications to store and organize data using folders and files. Such disks are often called volumes. Among all the volumes that are attached to a machine, one of them has to provide the OS, which is called the root volume, so the machine can load the OS and run it. The other volumes can store applications. The VM images discussed in the previous section are copies of such VM volumes that can be saved and used later to launch VM instances. For example, when a user creates an EC2 instance from an AMI image, the chosen image is copied to a volume and the volume is attached to the instance so the instance can start and run. In the rest of this section, we will discuss the three different types of storage resources for an IaaS application in detail.

Volume Store

On a personal computer, VM volumes are stored on local disks, often as files on a local file system. In a cloud, storage resources are consolidated into storage

servers and shared by all the VM hosts via storage virtualization. In this setup, a VM instance running on a VM host accesses its VM volumes *remotely* stored on the storage servers over the network. These shared volume storage resources are offered as a cloud service to users to store their volumes and provide them to their VM instances. In AWS, this service is the Elastic Block Store (EBS) service. It is called block store because data is stored and accessed as fixed-size blocks.

A volume store provides VM volumes accessible by VM instances anywhere in the same zone, because the storage resources are shared by all the VM hosts. Volumes can be attached to any instance in the same zone. An instance can simultaneously have multiple volumes attached to it. But the volumes stored in the volume store of one zone are not available to the instances in a different zone. The volume store also provides high reliability by replicating the volumes across the storage servers in the same zone. Volumes can persist independently from the instances—a volume can be created without being attached to any instance, and a volume can be saved after the instance that it was previously attached to is terminated.

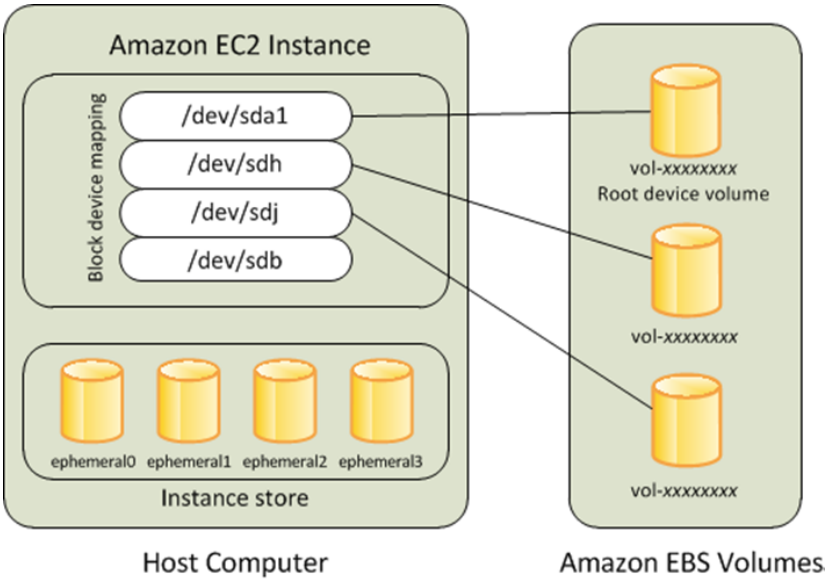

Figure 3.3: EBS and Instance Store provide volume storage to EC2 VM instances

Local Volume Store

The VM hosts in an IaaS cloud can also provide *local* volume storage to the instances that they are hosting. Volumes stored on such a local volume store

typical provides better performance to the instances than the volumes accessed over the network from a *remote* volume store discussed above. But local volume store cannot be offered as a standalone service as the *remote* volume store because it is tied to the specific host and it is available to only the instances that are running on the same host. Therefore, local volume store is typically provided as a feature to high-performance instance types. In AWS, this local volume store is called Instance Store.

When such an instance starts, it will be given local volumes to provide high-performance storage. The lifecycle of local volumes is tied to the instance that they are attached to—they are deleted when the instance is stopped or terminated. This is because no VM host can be dedicated to any VM; every time an instance is launched, it is run on a host that is likely to be different from the host where a previous instance of the same VM ran, and therefore any local volumes on that previous host will not be local any more to the new host and have no reason to persist any more when the previous instance was stopped or terminated.

General-purpose Object Store

A general-purpose object store can store any kind of data, including both structured and unstructured data, data of any format, and data of any size. In AWS, this service is the Simple Storage Service (S3). Such a general-purpose storage service is often offered as an object store because of the generalizability and scalability of object storage. Object storage stores data as self-contained objects, where each object stores both data and metadata and has a unique identifier for addressing the object. The self-contained nature allows objects to store any kind of data with any type of attributes. Object storage also eliminates the need of any complex structure such as a directory tree in a file system for naming the objects. Instead, a general-purpose cloud object store allows users to organize data into buckets, where each bucket serves as a container of arbitrary objects and each bucket has a globally unique address. The object store also provides a simple interface, mainly just "get" and "put" operations, for accessing the objects, which is much simpler than file systems. These designs all help make the object store scalable.

Compared to the other storage services that an IaaS cloud offers, this general-purpose object store has several advantages. First, it is highly reliable. Volume store is also reliable, but it replicates data only within an AZ, and thus cannot tolerate failures occurred at the AZ level. In comparison, the object store replicates data across the AZs in the same region, and can thus tolerate failures occurred at a larger scale. Second, the object store is typically cheaper per unit of data storage, and is more suited for long-term storage of large datasets. Finally, because the object store data is replicated across a cloud region, it is available to all the AZs in the region; in comparison, local volume store is available to the only host that it resides, and remote volume store is available to only the AZ that it resides.

This general-purpose object store provides important storage services to IaaS VMs. Specifically, it is commonly used to store the snapshots of VM volumes. A volume snapshot is a point-in-time copy of the data on the volume. Just like a photo snapshot captures the state of a scenery at the time when the snapshot was taken, a volume snapshot captures the state of a volume at the time when the snapshot was taken. Volume snapshots have several important uses. First, they provide backups of volumes, and can be used to recover the volumes in case of failures. Second, a volume can have a series of snapshots taken at different points of time, which allow the volume to be rolled back to any previous state captured by the snapshots. For example, if an update made to the OS on a volume becomes undesirable and there is no clean way to undo the update, the user can simply roll back the volume to the snapshot taken right before the update and continue to use the volume from there. Finally, volume snapshots provide an efficient way to clone volumes and use them to launch multiple instances each with a dedicated volume clone.

However, different from the local and remote volume stores, VM instances cannot be directly launched from the volume snapshots. The volume snapshots have to be loaded into a volume store first before they can be used to launch VM instances. Performance is the main reason for this restriction. On one hand, the object store is much slower and farther away from the VM hosts than the volume stores; on the other hand, the object store's interface does not allow the objects to be directly partially modified, which is inefficient for instance executions. Figure 3.4 illustrates how both the volume stores (Instance Store, EBS) and the general-purpose object store (S3) are used to support cloud VM storage.

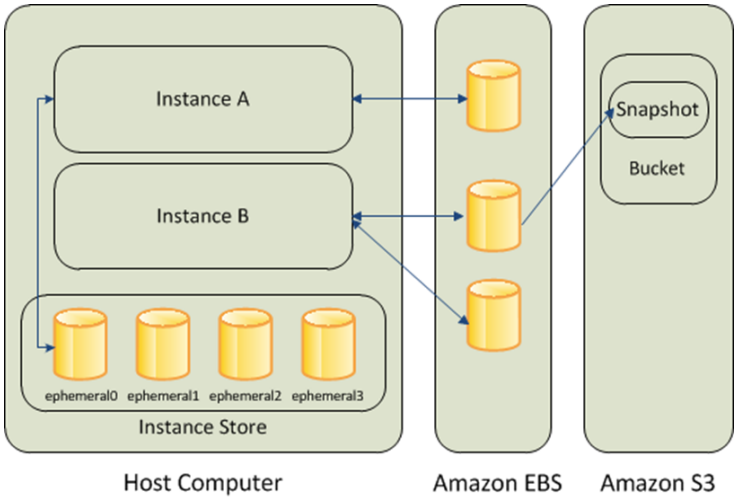

Figure 3.4: Volume stores and object store provide storage to cloud VMs

3.4 Communication

A cloud application is natively a distributed application, meaning that the application has components distributed across network, and these components communicate through the network. There are two important reasons for this distributed computing nature of cloud applications. First, to follow the service-oriented architecture (SOA), the integral components of an application need to be loosely coupled and communicate with one another over network. Second, to achieve elasticity, the application needs to scale by utilizing multiple instances to process in parallel, which also requires communication across the network. To support this network communication for an IaaS application, there are two basic questions that need to be answered: first, how to address the instances where the application's distributed components run on, and second, how to facilitate the data exchange among these components over network.

3.4.1 Addressing

An IaaS VM instance automatically gets a private IP address and host name from the IaaS cloud's internal private network. These private addresses are useful for a cloud application's internal components to communicate with one another.

An instance can also request a public IP address and host name for communicating with its external components such as the clients of the cloud application. However, because public IP addresses are limited resources, users typically need to pay for the use of public addresses. In AWS, an instance can get an ephemeral public IP while it is running, but this IP is recycled once the instance is stopped or terminated. If the user wants a permanent public IP, she needs to request one from the AWS Elastic IP service, and then can assign it to any of her instances. Moreover, to encourage users to make good use of these scarce resources, cloud providers typically charge more for public IPs that are assigned to a user but not used by any of the user's instances.

Therefore, an IaaS developer needs to carefully manage the addresses for her instances, and pay attention to the difference between public vs. private addresses and static vs. ephemeral addresses. For example, for a multi-tier cloud application, the web tier needs a public address to receive requests from its clients across the Internet, whereas the application tier and the data tier need only private addresses for communicating with the other tiers.

3.4.2 Messaging

There are different ways for processes in a distributed system to communicate with one another, from the basic socket programming which requires the client and server to explicitly create a connection between each other using socket-based endpoints, to the more advanced remote procedure calls (RPCs) which allows the client to invoke the server's service in the same way how it invokes a

local procedure call. However, both these methods require the client and server to be tightly coupled, i.e., the client needs to directly communicate with the server, which has several serious limitations for cloud applications. For example, it assumes that the server is available and ready when the client makes the request; otherwise, the client has to be blocked until the request is processed. The analogy to this in the human communication is phone calls, where a phone call cannot go through unless both parties are simultaneously available. To enable a loosely coupled system, cloud applications require a different method for communication, message queue based communication, which is another fundamental service provided by IaaS clouds. Example of this message queue service in AWS is the Simple Queuing Service (SQS).

Message-queue-based communication typically works as follows. The application first creates a message queue for delivering messages from the sender to the receiver. The sender deposits messages in the queue, and the receiver retrieves the messages from the queue. There can be multiple senders and multiple receivers sharing the same queue. For example, a multi-tier application can have multiple instances for each tier such as a web tier cluster and an application tier cluster; when the web tier communicates with the application tier via a message queue, all the web tier instances are senders and all the application tier instances are receivers.

A cloud message queue service is elastic, which means that a user can create as many queues as needed, each queue can store as many messages as needed, and there can be as many senders and receivers as needed working on the same queue at the same time, while the queue always provides the same level of performance. Such elasticity is achieved by employing a distributed system to provide the necessary processing and storage resources for storing and delivering the messages.

At the same time, a cloud message queue service is also highly reliable which means a sender's messages will be delivered to the receiver. Specifically, it typically provides a guarantee to deliver a message *at least once*, which means that a message received by the queue will always be delivered, but duplicates of the same message may be delivered. The message queue service achieves reliably delivery by replicating the message on its distributed system. The at-least-once guarantee is weaker than the guarantee of delivering each message exactly once, but the latter typically requires the message queue system to achieve stronger consistency at the cost of reduced scalability. Therefore, the at-least-once guarantee provides a good tradeoff between performance and reliability. On the other hand, an application using the message queue service can typically handle the redundant delivery of the same message; for example, it can assign a unique ID to each message and use this ID to eliminate the received redundant copies of the same message.

Compared to socket-based and RPC-based communication, message queues allow the senders and receivers to be loosely coupled—they do not communicate with each other directly; instead, the message queue service brokers the communication between the senders and receivers. Following the analogy given

at the start of this section, if direct communication is like making a phone call, message-queue-based communication is like using a shared mailbox for a sender to send mails to a receiver, except that this "mailbox" in the cloud is far more scalable and reliable than the real-world mailboxes.

This loosely coupled communication through cloud message queues has several important advantages compared to the tightly coupled alternatives. First, it allows the sender and receiver to operate asynchronously. Processes in a distributed system each operates at its own pace, and it is in general difficult to synchronize their progress. Instead of waiting for the other party to be ready, now the sender and receiver interacts with only the message queue, which is always ready, and can immediately return to continue their executions. Second, message queues provide isolation between the senders and receivers. Tightly-coupled senders and receivers affect each other when failures happen. When one party fails, the other party has to be blocked and waiting for the failed party to be recovered. Using message queues, senders and receivers can operate asynchronously, they can also each deal with failures independently. The messages sent before a sender fails are reliably stored in the queues, and the messages that are in a queue are always ready to be consumed after a failed receiver recovers. Third, a cloud message queue also serves as a salable and reliable buffer to allow the senders and receivers to produce and consume data, respectively, at different speed. For example, a web tier that processes HTTP requests and creates jobs for the application tier is often faster than an application tier that performs the computationally intensive jobs (such as machine learning and data analytics). At the same time, the burstiness inherent in cloud workload also exacerbate this mismatch in speed. If the web tier directly sends the jobs to the application tier, either the web tier has to be throttled to allow the application tier to catch up, or the application tier has to buffer the jobs on its own. In comparison, using a message queue for communication, the queue provides a transparent buffer to bridge the speed gap between the sender and receiver and absorb the burstiness in the workload. Finally, message queues can also be utilized by a cloud application to implement autoscaling and achieve elasticity, which is discussed next.

3.5 Autoscaling

A cloud application must be elastic. First, the application must use elastic "ingredients", i.e., resources that can dynamically scale out and in according to the demand of the application. In this chapter, we have already learned all the elastic resources that IaaS provides, including VM instances, volume store, data store, and message queues, for an application to use for its development. Second, the application itself must be able to dynamically scale out and in itself according to the demand of the requests that it services. Just like a cook needs to know how to make use of the good ingredients to make a good dish, a cloud developer needs to know how to make use of the elastic resources that the

cloud provides to develop an elastic application. Recall the Aniomto.com story we discussed in the last lecture. Animoto.com was able to handle the sudden surge of demand (from 5,000 users to 750,000 users), not only because it was using EC2, but also because it was able to automatically scale itself from 40 to 4,000 instances. Hence, an elastic cloud application must support autoscaling, meaning that it must be able to acquire and relinquish resources on demand, and it must make efficient use of the resources without leaving them idle and wasted.

In general, IaaS provides limited support of autoscaling. With AWS, a user can use the CloudWatch service to monitor certain metrics of the user's application, such as the CPU utilization of the instances used by the application. The user can set up a trigger for scaling when the utilization exceeds a certain threshold (e.g., 90% CPU utilization), and the scaling action can be creating a certain number of new instances. The application can then use the Elastic Load Balancing (ELB) service to automatically distribute the load across all the instances. The opposite process happens when the application needs to scale in as the demand drops. However, there are limited triggers and actions that are supported by AWS. In general, an IaaS provider lacks visibility into a user's VMs and has limited information for making autoscaling decisions on behalf of the user's application.

To have more control and flexibility of when to scaling and how to scaling, a cloud application can implement autoscaling on its own using a message queue service such as SQS. The basic idea is to use message queues to decouple the various components in an application. So when one component needs to invoke the service of another component it does so by depositing the request in a queue that is shared between these two components, where the other component retrieves the requests from the queue and perform the requested service. The length of every queue—the number of pending requests therefore indicates the load of the component that needs to process these requests. When one component becomes fully subscribed, i.e., it does not have any more instances to handle the pending requests in its queue, the application can increase the number of instances for that component according to the number of pending requests. Conversely, when the demand drops, the application can also scale in the number of instances by stopping or terminating the idle instances accordingly. Of course if not every request is the same and not every job require the same amount of work, more information about the queued jobs are needed, in addition to the queue length, in order to make better autoscaling decisions.

Figure 3.5 illustrates an example of the use of two SQS queues to autoscale the application tier ("processing servers") of a two-tier application. Note that in this example, the web tier does not require autoscaling, because it is determined that the application tier is the bottleneck (likely because the requests processed by the application tier is more intensive than the requests of the web tier). As the load of the system changes, bottlenecks may shift from one component to another. For example, as the user demand continues to grow, the single-instance-based web tier will eventually become insufficient to handle the

incoming HTTP workload and become the new bottleneck; at this point, even though the application tier can autoscale, it will not grow further because it is capped by how many requests the web tier can process. Therefore, autoscaling needs to be determined given the range of load that the system expects to be servicing.

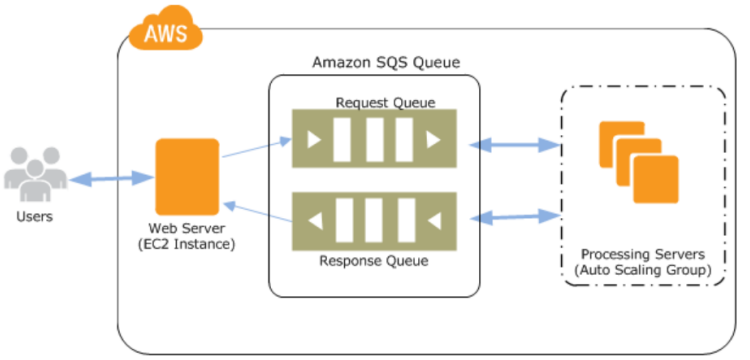

Figure 3.5: SQS-based autoscaling