17 Scaling up and down: Autoscaling and CloudWatch

This chapter covers

- Creating an Auto Scaling group with a launch template
- Using autoscaling to change the number of virtual machines
- Scaling a synchronous decoupled app behind a load balancer (ALB)
- Scaling an asynchronous decoupled app using a queue (SQS)

Suppose you're organizing a party to celebrate your birthday. How much food and drink do you need to buy? Calculating the right numbers for your shopping list is difficult due to the following factors:

- How many people will attend? You received several confirmations, but some guests will cancel at short notice or show up without letting you know in advance. Therefore, the number of guests is vague.
- How much will your guests eat and drink? Will it be a hot day, with everybody drinking a lot? Will your guests be hungry? You need to guess the demand for food and drink based on experiences from previous parties as well as weather, time of day, and other variables.

Solving the equation is a challenge because there are many unknowns. Being a good host, you'll order more food and drink than needed so no guest will be hungry or thirsty for long. It may cost you more money than necessary, and you may end up wasting some of it, but this possible waste is the risk you must take to ensure you have enough for unexpected guests and circumstances.

Before the cloud, the same was true for our industry when planning the capacity of our IT infrastructure. Planning to meet future demands for your IT infrastructure was nearly impossible. To prevent a supply gap, you needed to add extra capacity on top of the planned demand to prevent running short of resources. When procuring hardware for a data center, we always had to buy hardware based on the demands of the future. We faced the following uncertainties when making these decisions:

- How many users need to be served by the infrastructure?
- How much storage would the users need?
- How much computing power would be required to handle their requests?

To avoid supply gaps, we had to order more or faster hardware than needed, causing unnecessary expenses.

On AWS, you can use services on demand. Planning capacity is less and less important. You can scale from one EC2 instance to thousands of EC2 instances. Storage can grow from gigabytes to petabytes. You can scale on demand, thus replacing capacity planning. AWS calls the ability to scale on demand *elasticity*.

Public cloud providers like AWS can offer the needed capacity with a short waiting time. AWS serves more than a million customers, and at that scale, it isn't a problem to provide you with 100 additional virtual machines within minutes if you suddenly need them. This allows you to address another problem: recurring traffic patterns, as shown in figure 17.1. Think about the load on your infrastructure during the day versus at night, on a weekday versus the weekend, or before Christmas versus the rest of year. Wouldn't it be nice if you could add capacity when traffic grows and remove capacity when traffic shrinks? That's what this chapter is all about.

Figure 17.1 Typical traffic patterns for a web shop

Scaling the number of virtual machines is possible with Auto Scaling groups (ASG) and *scaling policies* on AWS. Autoscaling is part of the EC2 service and helps you scale the number of EC2 instances you need to fulfill the current load of your system. We introduced Auto Scaling groups in chapter 13 to ensure that a single virtual machine was running even if an outage of an entire data center occurred.

In this chapter, you'll learn how to manage a fleet of EC2 instances and adapt the size of the fleet depending on the current use of the infrastructure. To do so, you will use the concepts that you learned about in chapters 14 and 15 and enhance your setup with automatic scaling as follows:

- Using Auto Scaling groups to launch multiple virtual machines of the same kind as you did in chapters 13 and 14
- Changing the number of virtual machines based on CPU load with the help of CloudWatch alarms, which is a new concept we are introducing in this chapter
- Changing the number of virtual machines based on a schedule to adapt to recurring traffic patterns—something you will learn about in this chapter
- Using a load balancer as an entry point to the dynamic EC2 instance pool as you did in chapter 14
- Using a queue to decouple the jobs from the dynamic EC2 instance pool, similar to what you learned in chapter 14

Examples are 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.
The following prerequisites are required to scale your application horizontally, which means increasing and decreasing the number of virtual machines based on the current workload:

- The EC2 instances you want to scale need to be *stateless*. You can achieve stateless servers by storing data with the help of services like RDS (SQL database), DynamoDB (NoSQL database), EFS (network filesystem), or S3 (object store) instead of storing data on disks (instance store or EBS) that are available only to a single EC2 instance.
- An entry point to the dynamic EC2 instance pool is needed to distribute the workload across multiple EC2 instances. EC2 instances can be decoupled synchronously with a load balancer or asynchronously with a queue.

We introduced the concept of the stateless server in part 3 of this book and explained how to use decoupling in chapter 13. In this chapter, you'll return to the concept of the stateless server and also work through an example of synchronous and asynchronous decoupling.

17.1 Managing a dynamic EC2 instance pool

Imagine that you need to provide a scalable infrastructure to run a web application, such as a blogging platform. You need to launch uniform virtual machines when the number of requests grows and terminate virtual machines when the number of requests shrinks. To adapt to the current workload in an automated way, you need to be able to launch and terminate VMs automatically. Therefore, the configuration and deployment of the web application needs to be done during bootstrapping, without human interaction.

In this section, you will create an Auto Scaling group. Next, you will learn how to change the number of EC2 instances launched by the Auto Scaling group based on scheduled actions. Afterward, you will learn how to scale based on a utilization metric provided by CloudWatch. Auto Scaling groups allows you to manage such a dynamic EC2 instance pool in the following ways:

- Dynamically adjust the number of virtual machines that are running
- Launch, configure, and deploy uniform virtual machines

The Auto Scaling group grows and shrinks within the bounds you define. Defining a minimum of two virtual machines allows you to make sure at least two virtual machines are running in different availability zones to plan for failure. Conversely, defining a maximum number of virtual machines ensures you are not spending more money than you intended for your infrastructure. As figure 17.2 shows, autoscaling consists of three parts:

- A launch template that defines the size, image, and configuration of virtual machines
- An Auto Scaling group that specifies how many virtual machines need to be running based on the launch template
- Scaling plans that adjust the desired number of EC2 instances in the Auto Scaling group based on a plan or dynamically

Figure 17.2 Autoscaling consists of an Auto Scaling group and a launch template, launching and terminating uniform virtual machines.

If you want multiple EC2 instances to handle a workload, it’s important to start identical virtual machines to build a homogeneous foundation. Use a launch template to define and configure new virtual machines. Table 17.1 shows the most important parameters for a launch template.

Table 17.1 Launch template parameters

Name	Description	Possible values
`ImageId`	Image from which to start a virtual machine	ID of an Amazon Machine Image (AMI)
`InstanceType`	Size for new virtual machines	Instance type (such as `t2.micro`)
`UserData`	User data for the virtual machine used to execute a script during bootstrapping	BASE64-encoded string
`NetworkInterfaces`	Configures the network interfaces of the virtual machine. Most importantly, this parameter allows you to attach a public IP address to the instance.	List of network interface configurations
`IamInstanceProfile`	Attaches an IAM instance profile linked to an IAM role	Name or Amazon Resource Name (ARN, an ID) of an IAM instance profile

After you create a launch template, you can create an Auto Scaling group that references it. The Auto Scaling group defines the maximum, minimum, and desired number of virtual machines. *Desired* means this number of EC2 instances should be running. If the current number of EC2 instances is below the desired number, the Auto Scaling group will add EC2 instances. If the current number of EC2 instances is above the desired number, EC2 instances will be terminated. The desired capacity can be changed automatically based on load or a schedule, or manually. *Minimum* and *maximum* are the lower and upper limits for the number of virtual machines within the Auto Scaling group.

The Auto Scaling group also monitors whether EC2 instances are healthy and replaces broken instances. Table 17.2 shows the most important parameters for an Auto Scaling group.

Table 17.2 Auto Scaling group parameters

Name	Description	Possible values
`DesiredCapacity`	Desired number of healthy virtual machines	Integer
`MaxSize`	Maximum number of virtual machines; the upper scaling limit	Integer
`MinSize`	Minimum number of virtual machines; the lower scaling limit	Integer
`HealthCheckType`	How the Auto Scaling group checks the health of virtual machines	`EC2` (health of the in-stance) or `ELB` (health check of instance performed by a load balancer)
`HealthCheckGracePeriod`	Period for which the health check is paused after the launch of a new instance to wait until the instance is fully bootstrapped	Number of seconds
`LaunchTemplate`	ID (`LaunchTemplateId`) and version of launch template used as a blueprint when spin-ning up virtual machines	ID and version of launch template
`TargetGroupARNs`	The target groups of a load balancer, where autoscaling registers	List of tar-get group ARNs

	new instances automatically	
`VPCZoneIdentifier`	List of subnets in which to launch EC2 instances	List of subnet identifiers of a VPC

If you specify multiple subnets with the help of `VPCZoneIdentifier` for the Auto Scaling group, EC2 instances will be evenly distributed among these subnets and, thus, among availability zones.

Don’t forget to define a health check grace period

If you are using the ELB’s health check for your Auto Scaling group, make sure you specify a `HealthCheckGracePeriod` as well. Specify a health check grace period based on the time it takes from launching an EC2 instance until your application is running and passes the ELB’s health check. For a simple web application, a health check period of five minutes is suitable.

The next listing shows how to set up such a dynamic EC2 instance pool with the help of a CloudFormation template.

Listing 17.1 Auto Scaling group and launch template for a web app

```
# [...]
LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'
  Properties:
    LaunchTemplateData:
      IamInstanceProfile:
        Name: !Ref InstanceProfile
      ImageId: 'ami-028f2b5ee08012131'        ①
      InstanceType: 't2.micro'                ②
      NetworkInterfaces:
      - AssociatePublicIpAddress: true        ③
        DeviceIndex: 0
        Groups:                               ④
        - !Ref WebServerSecurityGroup
      UserData:                               ⑤
        'Fn::Base64': !Sub |
          #!/bin/bash -x
          yum -y install httpd
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    TargetGroupARNs:                          ⑥
    - !Ref LoadBalancerTargetGroup
    LaunchTemplate:                           ⑦
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
    MinSize: 2                                    ⑧
    MaxSize: 4                                    ⑨
    HealthCheckGracePeriod: 300                   ⑩
    HealthCheckType: ELB                          ⑪
    VPCZoneIdentifier:                            ⑫
    - !Ref SubnetA
    - !Ref SubnetB
# [...]
```

① Image (AMI) from which to launch new virtual machines

② Instance type for new EC2 instances

③ Associates a public IP address with new virtual machines

④ Attaches these security groups when launching new virtual machines

⑤ The script executed during the bootstrap of virtual machines

⑥ Registers new virtual machines on the target group of the load balancer

⑦ References the launch template

⑧ Minimum number of EC2 instances

⑨ Maximum number of EC2 instances

⑩ Waits 300 seconds before terminating a new virtual machine because of an unsuccessful health check

⑪ Uses the health check from the ELB to check the health of the EC2 instances

⑫ Starts the virtual machines in these two subnets of the VPC

In summary, Auto Scaling groups are a useful tool if you need to start multiple virtual machines of the same kind across multiple availability zones. Additionally, an Auto Scaling group replaces failed EC2 instances automatically.

17.2 Using metrics or schedules to trigger scaling

So far in this chapter, you’ve learned how to use an Auto Scaling group and a launch template to manage virtual machines. With that in mind, you can change the desired capacity of the Auto Scaling group manually so new instances will be started or old instances will be terminated to reach the new desired capacity.

To provide a scalable infrastructure for a blogging platform, you need to increase and decrease the number of virtual machines in the pool automatically by adjusting the desired capacity of the Auto Scaling group with scaling policies. Many people surf the web during their lunch break, so you might need to add virtual machines every day between 11 a.m. and 1 p.m. You also need to adapt to unpredictable load patterns—for example, if articles hosted on your blogging platform are shared frequently through social networks. Figure 17.3 illustrates two ways to change the number of virtual machines, as described in the following list.

- *Defining a schedule*—The timing would increase or decrease the number of virtual machines according to recurring load patterns (such as decreasing the number of virtual machines at night).
- *Using a CloudWatch alarm*—The alarm will trigger a scaling policy to increase or decrease the number of virtual machines based on a metric (such as CPU usage or number of requests on the load balancer).

Figure 17.3 Triggering autoscaling based on CloudWatch alarms or schedules

Scaling based on a schedule is less complex than scaling based on a CloudWatch metric, because it's difficult to find a metric on which to scale reliably. On the other hand, scaling based on a schedule is less precise, because you have to overprovision your infrastructure to be able to handle unpredicted spikes in load.

17.2.1 Scaling based on a schedule

When operating a blogging platform, you might notice the following load patterns:

- *One-time actions*—Requests to your registration page increase heavily after you run a TV advertisement in the evening.
- *Recurring actions*—Many people seem to read articles during their lunch break, between 11 a.m. and 1 p.m.

Luckily, scheduled actions adjust your capacity with one-time or recurring actions. You can use different types of actions to react to both load pattern types.

The following listing shows a one-time scheduled action increasing the number of web servers at 12:00 UTC on January 1, 2018. As usual, you'll find the code in the book's code repository on GitHub: **https://github.com/AWSinAction/code3**. The CloudFormation template for the WordPress example is located in /chapter17/ wordpress-schedule.yaml.

Listing 17.2 Scheduling a one-time scaling action

```
OneTimeScheduledActionUp:
  Type: 'AWS::AutoScaling::ScheduledAction'          ①
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup      ②
    DesiredCapacity: 4                               ③
    StartTime: '2025-01-01T12:00:00Z'                ④
```

① Defining a scheduled action

② Name of the Auto Scaling group

③ Sets the desired capacity to 4

④ Changes the setting at 12:00 UTC on January 1, 2025

You can also schedule recurring scaling actions using cron syntax. The code example shown next illustrates how to use two scheduled actions to increase the desired capacity during business hours (08:00 to 20:00 UTC) every day.

Listing 17.3 Scheduling a recurring scaling action that runs at 20:00 UTC every day

```
RecurringScheduledActionUp:
  Type: 'AWS::AutoScaling::ScheduledAction'            ①
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup
    DesiredCapacity: 4                                 ②
    Recurrence: '0 8 * * *'                            ③
 RecurringScheduledActionDown:
  Type: 'AWS::AutoScaling::ScheduledAction'
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup
    DesiredCapacity: 2                                 ④
    Recurrence: '0 20 * * *'                           ⑤
```

① Defining a scheduled action

② Sets the desired capacity to 4

③ Increases the capacity at 08:00 UTC every day

④ Sets the desired capacity to 2

⑤ Decreases the capacity at 20:00 UTC every day

`Recurrence` is defined in Unix cron syntax format as shown here:

```
* * * * *
| | | | |
| | | | +- day of week (0 - 6) (0 Sunday)
| | | +--- month (1 - 12)
| | +----- day of month (1 - 31)
| +------- hour (0 - 23)
+--------- min (0 - 59)
```

We recommend using scheduled scaling actions whenever your infrastructure's capacity requirements are predictable—for example, an internal system used during work hours only, or a marketing action planned for a certain time.

17.2.2 Scaling based on CloudWatch metrics

Predicting the future is hard. Traffic will increase or decrease beyond known patterns from time to time. For example, if an article published on your blogging platform is heavily shared through social media, you need to be able to react to unplanned load changes and scale the number of EC2 instances.

You can adapt the number of EC2 instances to handle the current workload using CloudWatch alarms and scaling policies. CloudWatch helps monitor virtual machines and other services on AWS. Typically, services publish usage metrics to CloudWatch, helping you to evaluate the available capacity. The types of scaling policies follow:

1. *Step scaling*—Allows more advanced scaling because multiple scaling adjustments are supported, depending on how much the threshold you set has been exceeded.
2. *Target tracking*—Frees you from defining scaling steps and thresholds. You need only to define a target (such as CPU utilization of 70%), and the number of EC2 instances is adjusted accordingly.
3. *Predictive scaling*—Uses machine learning to predict load. It works best for cyclical traffic and recurring workload patterns (see "Predictive Scaling for Amazon EC2 Auto Scaling" at **http://mng.bz/RvYO** to learn more).
4. *Simple scaling*—A legacy option that was replaced with step scaling.

All of the scaling policies use metrics and alarms to scale the number of EC2 instances based on the current workload. As shown in figure 17.4, the virtual machines publish metrics to CloudWatch constantly. A CloudWatch alarm monitors one of these metrics and triggers a scaling action if the defined threshold is reached. The scaling policy then increases or decreases the desired capacity of the Auto Scaling group.

Figure 17.4 Triggering autoscaling based on a CloudWatch metric and alarm

An EC2 instance publishes several metrics to CloudWatch by default: CPU, network, and disk utilization are the most important. Unfortunately, no metric currently exists for a virtual machine’s memory usage. You can use these metrics to scale the number of VMs if a bottleneck is reached. For example, you can add EC2 instances if the CPU is working to capacity. The following parameters describe a CloudWatch metric:

- `Namespace`—Defines the source of the metric (such as AWS/EC2)
- `Dimensions`—Defines the scope of the metric (such as all virtual machines belonging to an Auto Scaling group)
- `MetricName`—Unique name of the metric (such as `CPUUtilization`)

CloudWatch alarms are based on CloudWatch metrics. Table 17.3 explains the alarm parameters in detail.

Table 17.3 Parameters for a CloudWatch alarm that triggers scaling based

on CPU usage of all virtual machines belonging to an Auto Scaling group

Context	Name	Description	Possible values
Condition	`Statistic`	Statistical function applied to a metric	`Average`, `Sum`, `Minimum`, `Maximum`, `SampleCount`
Condition	`Period`	Defines a time-based slice of values from a metric	Seconds (multiple of 60)
Condition	`EvaluationPeriods`	Number of periods to evaluate when checking for an alarm	Integer
Condition	`Threshold`	Threshold for an alarm	Number
Condition	`ComparisonOperator`	Operator to compare the threshold against the result from a statistical function	`GreaterThanOrEqualToThreshold`, `GreaterThanThreshold`, `LessThanThreshold`, `LessThanOrEqualToThreshold`
Metric	`Namespace`	Source of the metric	`AWS/EC2` for metrics from the EC2 service
Metric	`Dimensions`	Scope of the metric	Depends on the metric; references the Auto Scaling group for an aggregated metric over all associated EC2 instances

Metric	`MetricName`	Name of the metric	For example, `CPUUtilization`
Action	`AlarmActions`	Actions to trigger if the threshold is reached	Reference to the scaling policy

You can define alarms on many different metrics. You'll find an overview of all namespaces, dimensions, and metrics that AWS offers at **http://mng.bz/8E0X**. For example, you could scale based on the load balancer's metric counting the number of requests per target, or the networking throughput of your EC2 instances. You can also publish custom metrics—for example, metrics directly from your application like thread pool usage, processing times, or user sessions.

Scaling based on CPU load with VMs that offer burstable performance

Some virtual machines, such as instance families `t2` and `t3`, offer burstable performance. These virtual machines offer a baseline CPU performance and can burst performance for a short time based on credits. If all credits are spent, the instance operates at the baseline. For a `t2.micro` instance, baseline performance is 10% of the performance of the underlying physical CPU.

Using virtual machines with burstable performance can help you react to load spikes. You save credits in times of low load and spend credits to burst performance in times of high load. But scaling the number of virtual machines with burstable performance based on CPU load is tricky because your scaling strategy must take into account whether your instances have enough credits to burst performance. Consider searching for another metric to scale (such as number of sessions) or using an instance type without burstable performance.

You've now learned how to use autoscaling to adapt the number of virtual machines to the workload. It's time to bring this into action.

17.3 Decoupling your dynamic EC2 instance pool

If you need to scale the number of virtual machines running your blogging platform based on demand, Auto Scaling groups can help you provide the right number of uniform virtual machines, and scaling schedules or CloudWatch alarms can increase or decrease the desired number of EC2 instances automatically. But how can users reach the EC2 instances in the pool to browse the articles you're hosting? Where should the HTTP request be routed?

Chapter 14 introduced the concept of decoupling: synchronous decoupling with ELB and asynchronous decoupling with SQS. If you want to use autoscaling to grow and shrink the number of virtual machines, you need to decouple your EC2 instances from the clients, because the interface that's reachable from outside the system needs to stay the same no matter how many EC2 instances are working behind the scenes.

Figure 17.5 shows how to build a scalable system based on synchronous or asynchronous decoupling. A load balancer acts as the entry point for synchronous decoupling, by distributing requests among a fleet of virtual machines. A message queue is used as the entry point for asynchronous requests, and messages from producers are stored in the queue. The virtual machines then poll the queue and process the messages asynchronously.

Figure 17.5 Decoupling allows you to scale the number of virtual machines dynamically.

Decoupled and scalable applications require stateless servers. A stateless server stores any shared data remotely in a database or storage system. The following two examples implement the concept of a stateless server:

- *WordPress blog*—Decoupled with ELB, scaled with autoscaling and CloudWatch based on CPU utilization, and data outsourced to a MySQL database (RDS) and a network filesystem (EFS)
- *URL2PNG taking screenshots of URLs*—Decoupled with a queue (SQS), scaled with autoscaling and CloudWatch based on queue length, and data outsourced to a NoSQL database (DynamoDB) and an object store (S3)

17.3.1 Scaling a dynamic EC2 instance pool synchronously decoupled by a load balancer

Answering HTTP(S) requests is a synchronous task. If a user wants to use your web application, the web server has to answer the corresponding requests immediately. When using a dynamic EC2 instance pool to run a web application, it's common to use a load balancer to decouple the EC2 instances from user requests. The load balancer forwards HTTP(S) requests to multiple EC2 instances, acting as a single entry point to the dynamic EC2 instance pool.

Suppose your company has a corporate blog for publishing announcements and interacting with the community, and you're responsible for hosting the blog. The marketing department complains about slow page speed and even timeouts in the evening, when traffic reaches its daily peak. You want to use the elasticity of AWS by scaling the number of EC2 instances based on the current workload.

Your company uses the popular blogging platform WordPress for its corporate blog. Chapters 2 and 10 introduced a WordPress setup based on EC2 instances and RDS (MySQL database). In this chapter, we'd like to complete the example by adding the ability to scale.

Figure 17.6 shows the final, extended WordPress example. The following services are used for this highly available scaling architecture:

- EC2 instances running Apache to serve WordPress, a PHP application
- RDS offering a MySQL database that's highly available through Multi-AZ deployment
- EFS storing PHP, HTML, and CSS files as well as user uploads such as images and videos
- ELB to synchronously decouple the web servers from visitors
- Autoscaling and CloudWatch to scale the number of EC2 instances based on the current CPU load of all running virtual machines

Figure 17.6 Autoscaling web servers running WordPress, storing data on RDS and EFS, decoupled with a load balancer scaling based on load

As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code3. The CloudFormation template for the WordPress example is located in /chapter17/wordpress.yaml.

Execute the following command to create a CloudFormation stack that spins up the scalable WordPress setup. Replace `$Password` with your own password consisting of eight to 30 letters and digits.

```
$ aws cloudformation create-stack --stack-name wordpress \
➥ --template-url https:/ /s3.amazonaws.com/\
➥ awsinaction-code3/chapter17/wordpress.yaml --parameters \
➥ "ParameterKey=WordpressAdminPassword,ParameterValue=$Password" \
➥ --capabilities CAPABILITY_IAM
```

It will take up to 15 minutes for the stack to be created. This is a perfect time to grab some coffee or tea. Log in to the AWS Management Console, and navigate to the AWS CloudFormation service to monitor the process of the CloudFormation stack named `wordpress`. You have time to look through the most important parts of the CloudFormation template, shown in the following three listings.

Create a blueprint to launch EC2 instances, also known as a launch template, as illustrated next.

Listing 17.4 Creating a scalable, HA WordPress setup, part 1

```
LaunchTemplate:
    Type: 'AWS::EC2::LaunchTemplate'        ①
    Metadata:  # [...]                      ②
    Properties:
      LaunchTemplateData:
        IamInstanceProfile:                 ③
          Name: !Ref InstanceProfile
        ImageId: !FindInMap [RegionMap,
➥ !Ref 'AWS::Region', AMI]               ④
        Monitoring:                         ⑤
          Enabled: false
        InstanceType: 't2.micro'            ⑥
        NetworkInterfaces:                  ⑦
        - AssociatePublicIpAddress: true    ⑧
          DeviceIndex: 0
          Groups:
          - !Ref WebServerSecurityGroup     ⑨
        UserData: # [...]                   ⑩
```

① Creates a launch template for autoscaling

② Contains the configuration for the EC2 instance applied during bootstrapping

③ Configures the IAM instance profile for the virtual machines, allowing the machines to authenticate and authorize for AWS services

④ Selects the image from a map with AMIs organized by region

⑤ Disables detailed monitoring of the EC2 instances to reduce costs—enable this for production workloads.

⑥ Configures the instance type

⑦ Defines the network interface for the virtual machine

⑧ Enables a public IP address for the virtual machine

⑨ A list of security groups that should be attached to the EC2 instance

⑩ The user data contains a script to install and configure WordPress automatically.

Second, the Auto Scaling group shown in the next listing launches EC2 instances based on the launch template.

Listing 17.5 Creating a scalable, HA WordPress setup, part 2

```
AutoScalingGroup:
    Type: 'AWS::AutoScaling::AutoScalingGroup'                   ①
    DependsOn:                                                   ②
    - EFSMountTargetA
    - EFSMountTargetB
    Properties:
      TargetGroupARNs:                                                        ③
      - !Ref LoadBalancerTargetGroup
      LaunchTemplate:                                                         ④
        LaunchTemplateId: !Ref LaunchTemplate
        Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
      MinSize: 2                                                              ⑤
      MaxSize: 4                                                              ⑥
      HealthCheckGracePeriod: 300                                             ⑦
      HealthCheckType: ELB                                                    ⑧
      VPCZoneIdentifier:                                                      ⑨
      - !Ref SubnetA
      - !Ref SubnetB
      Tags:                                                                   ⑩
      - PropagateAtLaunch: true
        Value: wordpress
        Key: Name
```

① Creates an Auto Scaling group

② Because the EC2 instances require access to the EFS filesystem, waits until CloudFormation creates the mount targets

③ Registers and unregisters virtual machines on the target group of the load balancer

④ References the latest version of the launch template

⑤ Launches at least two machines and ensures that at least two virtual machines are running, one for each of the two AZs for high availability

⑥ Launches no more than four machines

⑦ Waits five minutes to allow the EC2 instance and web server to start before evaluating the health check

⑧ Uses the ELB health check to monitor the health of the virtual machines

⑨ Launches VMs into two different subnets in two different AZs for high availability

⑩ Adds a tag, including a name for all VMs launched by the ASG

You will learn how to create CloudWatch alarms for scaling in the next example. For now, we are using a target-tracking scaling policy that creates CloudWatch alarms automatically in the background. A target-tracking scaling policy works like the thermostat in your home: you define the target, and the thermostat constantly adjusts the heating power to reach the target. Predefined metric specifications to use with target tracking follow:

- `ASGAverageCPUUtilization`—Scale based on the average CPU usage among all instances within an Auto Scaling group
- `ALBRequestCountPerTarget`—Scale based on the number of requests forwarded from the Application Load Balancer (ALB) to a target
- `ASGAverageNetworkIn` and `ASGAverageNetworkOut`—Scale based on the average number of bytes received or sent

In some cases, scaling based on CPU usage, request count per target, or network throughput does not work. For example, you might have another bottleneck you need to scale on, such as disk I/O. Any CloudWatch metric can be used for target tracking as well. Only one requirement exists: adding or removing instances must affect the metric proportionally. For example, request latency is not a valid metric for target tracking, because adjusting the number of instances does not affect the request latency directly.

The following listing shows a target-tracking policy based on the average CPU utilization of all EC2 instances of the Auto Scaling group.

Listing 17.6 Creating a scalable, HA WordPress setup, part 3

```
ScalingPolicy:
  Type: 'AWS::AutoScaling::ScalingPolicy'                          ①
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup                    ②
    PolicyType: TargetTrackingScaling                              ③
    TargetTrackingConfiguration:                                   ④
      PredefinedMetricSpecification:                               ⑤
        PredefinedMetricType: ASGAverageCPUUtilization             ⑥
      TargetValue: 70                                              ⑦
    EstimatedInstanceWarmup: 60                                    ⑧
```

① Creates a scaling policy

② Adjusts the desired capacity of the Auto Scaling group

③ Creates a scaling policy tracking a specified target

④ Configures the target tracking

⑤ Uses a predefined scaling metric

⑥ Average CPU usage across all EC2 instances of the ASG

⑦ Defines the target at 70% CPU usage

⑧ Excludes newly launched EC2 instances from CPU metric for 60 seconds to avoid scaling on load caused due to the bootstrapping of the VM and your application

Follow these steps after the CloudFormation stack reaches the state `CREATE_COMPLETE` to create a new blog post containing an image:

1. Select the CloudFormation stack `wordpress`, and switch to the Outputs tab.
2. Open the link shown for key `URL` with a web browser.
3. Search for the Log In link in the navigation bar, and click it.
4. Log in with username `admin` and the password you specified when creating the stack with the CLI.
5. Click Posts in the menu on the left.
6. Click Add New.
7. Type in a title and text, and upload an image to your post.
8. Click Publish.
9. Go back to the blog by clicking on the View Post link.

Now you're ready to scale. We've prepared a load test that will send 500,000 requests to the WordPress setup within a few minutes. Don't worry about costs: the usage is covered by the Free Tier. After three minutes, new virtual machines will be launched to handle the load. The load test takes 10 minutes. Another 15 minutes later, the additional VMs will disappear. Watching this is fun; you shouldn't miss it.

NOTE If you plan to do a big load test, consider the AWS Acceptable Use Policy at **https://aws.amazon.com/aup** and ask for permission before you begin (see also **http://mng.bz/2r8m**).

Simple HTTP load test

We're using a tool called Apache Bench to perform a load test of the WordPress setup. The tool is part of the `httpd-tools` package available from the Amazon Linux package repositories.

Apache Bench is a basic benchmarking tool. You can send a specified number of HTTP requests by using a specified number of threads. We're using the following command for the load test to send 500,000 requests to the load balancer using 15 threads. The load test is limited to 600 seconds, and we're using a connection timeout of 120 seconds. Replace `$UrlLoadBalancer` with the URL of the load balancer:

```
$ ab -n 500000 -c 15 -t 300 -s 120 -r $UrlLoadBalancer
```

Update the CloudFormation stack with the following command to start the load test:

```
$ aws cloudformation update-stack --stack-name wordpress \
➥ --template-url https:/ /s3.amazonaws.com/\
➥ awsinaction-code3/chapter17/wordpress-loadtest.yaml --parameters \
➥ ParameterKey=WordpressAdminPassword,UsePreviousValue=true  \
➥ --capabilities CAPABILITY_IAM
```

Watch for the following things to happen, using the AWS Management Console:

1. Open the CloudWatch service, and click Alarms on the left.
2. When the load test starts, the alarm called `TargetTracking-wordpress-AutoScalingGroup--AlarmHigh-` will reach the `ALARM` state after about 10 minutes.
3. Open the EC2 service, and list all EC2 instances. Watch for two additional instances to launch. At the end, you'll see five instances total (four web servers and the EC2 instance running the load test).
4. Go back to the CloudWatch service, and wait until the alarm named `TargetTracking-wordpress-AutoScalingGroup--AlarmLow-` reaches the `ALARM` state.
5. Open the EC2 service, and list all EC2 instances. Watch for the two additional instances to disappear. At the end, you'll see three instances total (two web servers and the EC2 instance running the load test).

The entire process will take about 30 minutes.

You've now watched autoscaling in action: your WordPress setup can now adapt to the current workload. The problem with pages loading slowly or even timeouts in the evening is solved.

Cleaning up
Execute the following commands to delete all resources corresponding to the WordPress setup:

```
$ aws cloudformation delete-stack --stack-name wordpress
```

17.3.2 Scaling a dynamic EC2 instances pool asynchronously decoupled by a queue

Imagine that you're developing a social bookmark service where users can save and share their links. Offering a preview that shows the website being linked to is an important feature. But the conversion from URL to PNG is causing high load during the evening, when most users add new bookmarks to your service. Because of that, customers are dissatisfied with your application's slow response times.

You will learn how to dynamically scale a fleet of EC2 instances to asynchronously generate screenshots of URLs in the following example. Doing so allows you to guarantee low response times at any time because the load-intensive workload is isolated into background jobs.

Decoupling a dynamic EC2 instance pool asynchronously offers an advantage if you want to scale based on workload: because requests don't need to be answered immediately, you can put requests into a queue and scale the number of EC2 instances based on the length of the queue. This gives you an accurate metric to scale, and no requests will be lost during a load peak because they're stored in a queue.

To handle the peak load in the evening, you want to use autoscaling. To do so, you need to decouple the creation of a new bookmark and the process of generating a preview of the website. Chapter 14 introduced an application called URL2PNG that transforms a URL into a PNG image. Figure 17.7 shows the architecture, which consists of an SQS queue for asynchronous decoupling as well as S3 for storing generated images. Creating a bookmark will trigger the following process:

1. A message is sent to an SQS queue containing the URL and the unique ID of the new bookmark.
2. EC2 instances running a Node.js application poll the SQS queue.
3. The Node.js application loads the URL and creates a screenshot.
4. The screenshot is uploaded to an S3 bucket, and the object key is set to the unique ID.
5. Users can download the screenshot directly from S3 using the unique ID.

Figure 17.7 Autoscaling virtual machines that convert URLs into images, decoupled by an SQS queue

A CloudWatch alarm is used to monitor the length of the SQS queue. If the length of the queue reaches five, an additional virtual machine is started to handle the workload. When the queue length goes below five, another CloudWatch alarm decreases the desired capacity of the Auto Scaling group.

The code is in the book's code repository on GitHub at https://github.com/AWSinAction/code3. The CloudFormation template for the URL2PNG example is located at chapter17/url2png.yaml.

Execute the following command to create a CloudFormation stack that spins up the URL2PNG application:

```
$ aws cloudformation create-stack --stack-name url2png \
➥ --template-url https:/ /s3.amazonaws.com/\
➥ awsinaction-code3/chapter17/url2png.yaml \
➥ --capabilities CAPABILITY_IAM
```

It will take up to five minutes for the stack to be created. Log in to the AWS Management Console, and navigate to the AWS CloudFormation service to monitor the process of the CloudFormation stack named `url2p-ng`.

We're using the length of the SQS queue to scale the number of EC2 instances. Because the number of messages in the queue does not correlate with the number of EC2 instances processing messages from the queue, it is not possible to use a target-tracking policy. Therefore, you will use a step-scaling policy in this scenario, as illustrated here.

Listing 17.7 Monitoring the length of the SQS queue

```
# [...]
HighQueueAlarm:
  Type: 'AWS::CloudWatch::Alarm'
  Properties:
    EvaluationPeriods: 1                                    ①
    Statistic: Sum                                          ②
    Threshold: 5                                            ③
    AlarmDescription: 'Alarm if queue length is higher than 5.'
    Period: 300                                             ④
    AlarmActions:                                           ⑤
    - !Ref ScalingUpPolicy
    Namespace: 'AWS/SQS'                                    ⑥
    Dimensions:                                              ⑦
    - Name: QueueName
      Value: !Sub '${SQSQueue.QueueName}'
    ComparisonOperator: GreaterThanThreshold                ⑧
    MetricName: ApproximateNumberOfMessagesVisible          ⑨
# [...]
```

① Number of time periods to evaluate when checking for an alarm

② Sums up all values in a period

③ Alarms if the threshold of five is reached

④ Uses a period of 300 seconds because SQS metrics are published every five minutes

⑤ Increases the number of desired instances by one through the scaling policy

⑥ The metric is published by the SQS service.

⑦ The queue, referenced by name, is used as the dimension of the metric.

⑧ Alarms if the sum of the values within the period is greater than the threshold of five

⑨ The metric contains an approximate number of messages pending in the queue.

The CloudWatch alarm triggers a scaling policy. The scaling policy shown in the following listing defines how to scale. To keep things simple, we are using a step-scaling policy with only a single step. Add additional steps if you want to react to a threshold breach in a more fine-grained way.

Listing 17.8 A step-scaling policy adding one more instance to an Auto Scaling group

```
# [...]
ScalingUpPolicy:
  Type: 'AWS::AutoScaling::ScalingPolicy'                    ①
  Properties:
    AdjustmentType: 'ChangeInCapacity'                       ②
    AutoScalingGroupName: !Ref AutoScalingGroup              ③
    PolicyType: 'StepScaling'                                ④
    MetricAggregationType: 'Average'                         ⑤
    EstimatedInstanceWarmup: 60                              ⑥
    StepAdjustments:                                         ⑦
    - MetricIntervalLowerBound: 0                            ⑧
      ScalingAdjustment: 1                                   ⑨
# [...]
```

① Creates a scaling policy

② The scaling policy increases the capacity by an absolute number.

③ Attaches the scaling policy to the Auto Scaling group

④ Creates a scaling policy of type step scaling

⑤ The aggregation type used when evaluating the steps, based on the metric defined within the CloudWatch alarm that triggers the scaling policy

⑥ The metrics of a newly launched instance are ignored for 60 seconds while it boots up.

⑦ Defines the scaling steps. We use a single step in this example.

⑧ The scaling step is valid from the alarms threshold to infinity.

⑨ Increases the desired capacity of the Auto Scaling group by 1

To scale down the number of instances when the queue is empty, a CloudWatch alarm and scaling policy with the opposite values needs to be defined.

You're now ready to scale. We've prepared a load test that will quickly generate 250 messages for the URL2PNG application. A virtual machine will be launched to process jobs from the SQS queue. After a few minutes, when the load test is finished, the additional virtual machine will disappear. Update the CloudFormation stack with the following command to start the load test:

```
$ aws cloudformation update-stack --stack-name url2png \
➥ --template-url https:/ /s3.amazonaws.com/\
➥ awsinaction-code3/chapter17/url2png-loadtest.yaml \
➥ --capabilities CAPABILITY_IAM
```

Watch for the following things to happen, with the help of the AWS Management Console:

1. Open the CloudWatch service, and click Alarms at left.
2. When the load test starts, the alarm called `url2png-`**`HighQueueAlarm-*`** will reach the `ALARM` state after a few minutes.
3. Open the EC2 service, and list all EC2 instances. Watch for an additional instance to launch. At the end, you'll see three instances total (two workers and the EC2 instance running the load test).
4. Go back to the CloudWatch service, and wait until the alarm named `url2png `**`-LowQueueAlarm-*`** reaches the `ALARM` state.
5. Open the EC2 service, and list all EC2 instances. Watch for the additional instance to disappear. At the end, you'll see two instances total (one worker and the EC2 instance running the load test).

The entire process will take about 20 minutes.

You've just watched autoscaling in action. The URL2PNG application can now adapt to the current workload, and the problem with slowly generated screenshots has been solved.

Cleaning up

Execute the following commands to delete all resources corresponding to the URL2PNG exa

```
$ URL2PNG_BUCKET=aws cloudformation describe-stacks --stack-name url2p
➥ --query "Stacks[0].Outputs[?OutputKey=='BucketName'].OutputValue" \
➥ --output text

$ aws s3 rm s3://${URL2PNG_BUCKET} --recursive

$ aws cloudformation delete-stack --stack-name url2png
```

Whenever distributing an application among multiple EC2 instances, you should use an Auto Scaling group. Doing so allows you to spin up identical instances with ease. You get the most out of the possibilities of the cloud when scaling the number of instances based on a schedule or a metric depending on the load pattern.

Summary

- You can use autoscaling to launch multiple identical virtual machines by using a launch template and an Auto Scaling group.
- EC2, SQS, and other services publish metrics to CloudWatch (CPU usage, queue length, and so on).
- CloudWatch alarms can change the desired capacity of an Auto Scaling group. This allows you to increase the number of virtual machines based on CPU usage or other metrics.
- Using stateless machines is a best practice, when scaling the number of machines according to the current workload automatically.
- To distribute load among multiple virtual machines, synchronous decoupling with the help of a load balancer or asynchronous decoupling with a message queue is necessary.