Listing 13.3 Launching a Jenkins virtual machine with autoscaling in two AZs

```
# [...]
LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'
  Properties:
    LaunchTemplateData:
      IamInstanceProfile:
        Name: !Ref IamInstanceProfile
      ImageId: !FindInMap [RegionMap,
➥ !Ref 'AWS::Region', AMI]
      Monitoring:
        Enabled: false
      InstanceType: 't2.micro'
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeviceIndex: 0
        Groups:
        - !Ref SecurityGroup
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1 --stack ${AWS::StackName}
          ➥ --resource AutoScalingGroup --region ${AWS::Region}' ERR
```

The blueprint used by the Auto Scaling group when launching an EC2 instance

Selects the AMI (in this case, Amazon Linux 2)

Attaches an IAM role to the EC2 instance to grant access for the Session Manager

By default, EC2 sends metrics to CloudWatch every five minutes. You can enable detailed instance monitoring to get metrics every minute for an additional cost.

The instance type for the virtual machine

Configures the network interface (ENI) of the EC2 instance

Associates a public IP address when launching the instance

Attaches a security group allowing ingress on port 8080 to the instance

The EC2 instance will execute the script loaded from user data at the end of the boot process. The script installs and configures Jenkins.

```
          # Installing Jenkins
          amazon-linux-extras enable epel=7.11 && yum -y clean metadata
          yum install -y epel-release && yum -y clean metadata
          yum install -y java-11-amazon-corretto-headless daemonize
          wget -q -T 60 http://ftp-chi.osuosl.org/pub/jenkins/
          ➥ redhat-stable/jenkins-2.319.1-1.1.noarch.rpm
          rpm --install jenkins-2.319.1-1.1.noarch.rpm

          # Configuring Jenkins
          # [...]

          # Starting Jenkins
          systemctl enable jenkins.service
          systemctl start jenkins.service
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
          ➥ --resource AutoScalingGroup --region ${AWS::Region}
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    LaunchTemplate:
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
```

References the launch template

Auto Scaling group responsible for launching the virtual machine

```
    Tags:
    - Key: Name
      Value: 'jenkins-multiaz'
      PropagateAtLaunch: true
    MinSize: 1
    MaxSize: 1
    VPCZoneIdentifier:
    - !Ref SubnetA
    - !Ref SubnetB
    HealthCheckGracePeriod: 600
    HealthCheckType: EC2
  # [...]
```

Adds those tags to the Auto Scaling group itself as well as to all EC2 instances launched by the Auto Scaling group

Maximum number of EC2 instances

Minimum number of EC2 instances

Launches the virtual machines in subnet A (created in availability zone A) and subnet B (created in availability zone B)

Waits 10 minutes before considering the health check of a newly launched instance

Uses the internal health check of the EC2 service to discover problems with the virtual machine

The CloudFormation stack might be already up and running. Execute the following command to grab the public IP address of the virtual machine. If no IP address appears, the virtual machine isn't started yet. Wait another minute, and try again:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
➥ Values=jenkins-multiaz" "Name=instance-state-code,Values=16" \
➥ --query "Reservations[0].Instances[0].\
➥ [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-0cff527cda42afbcc",
  "34.235.131.229",
  "172.31.38.173",
  "subnet-28933375"
]
```

Instance ID of the virtual machine

Public IP address of the virtual machine

Subnet ID of the virtual machine

Private IP address of the virtual machine

Open http://$PublicIP:8080 in your browser, and replace `$PublicIP` with the public IP address from the output of the previous `describe-instances` command. The web interface for the Jenkins server appears.

Execute the following command to terminate the virtual machine and test the recovery process with autoscaling. Replace `$InstanceId` with the instance ID from the output of the previous `describe` command:

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

After a few minutes, the Auto Scaling group detects that the virtual machine was terminated and starts a new virtual machine. Rerun the `describe-instances` command until the output contains a new running virtual machine, as shown here:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
➥ Values=jenkins-multiaz" "Name=instance-state-code,Values=16" \
➥ --query "Reservations[0].Instances[0].\
➥ [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
```

```
[
  "i-0293522fad287bdd4",
  "52.3.222.162",
  "172.31.37.78",
  "subnet-45b8c921"
]
```

The instance ID, the public IP address, the private IP address, and probably even the subnet ID have changed for the new instance. Open http://$PublicIP:8080 in your browser, and replace `$PublicIP` with the public IP address from the output of the previous `describe-instances` command. The web interface from the Jenkins server appears.

You've now built a highly available architecture consisting of an EC2 instance with the help of autoscaling. Two problems with the current setup follow:

- The Jenkins server stores data on disk. When a new virtual machine is started to recover from a failure, this data is lost because a new disk is created.
- The public and private IP addresses of the Jenkins server change after a new virtual machine is started for recovery. The Jenkins server is no longer available under the same endpoint.

You'll learn how to solve these problems in the next part of the chapter.

Cleaning up

It's time to clean up to avoid unwanted costs. Execute the following command to delete all resources corresponding to the Jenkins setup:

```
$ aws cloudformation delete-stack --stack-name jenkins-multiaz
$ aws cloudformation wait stack-delete-complete \
➥  --stack-name jenkins-multiaz
```

Waits until the stack is deleted

13.2.3 Pitfall: Recovering network-attached storage

The EBS service offers network-attached storage for virtual machines. Remember that EC2 instances are linked to a subnet, and the subnet is linked to an availability zone. EBS volumes are also located only in a single availability zone. If your virtual machine is started in another availability zone because of an outage, the EBS volume cannot be accessed from the other availability zones. Let's say your Jenkins data is stored on an EBS volume in availability zone `us-east-1a`. As long as you have an EC2 instance running in the same availability zone, you can attach the EBS volume. If this availability zone becomes unavailable and you start a new EC2 instance in availability zone `us-east-1b`, however, you can't access that EBS volume in `us-east-1a`, which means that you can't recover Jenkins because you don't have access to the data. See figure 13.5.

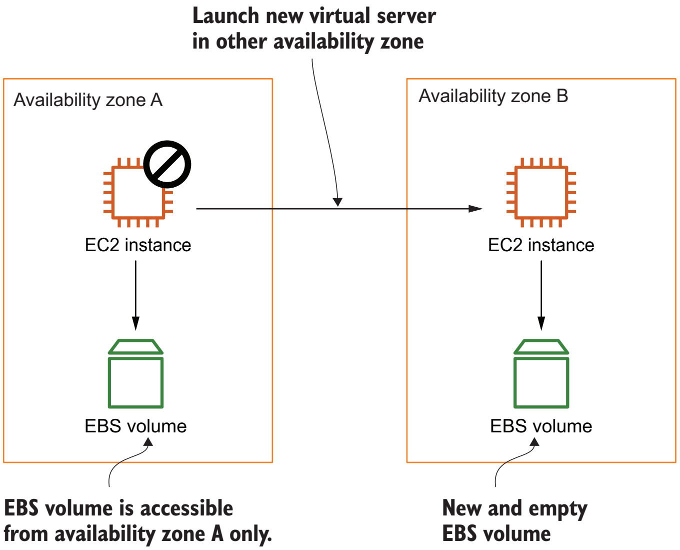


Figure 13.5 An EBS volume is available only in a single availability zone.

Don't mix availability and durability guarantees

An EBS volume is guaranteed to be available for 99.999% of the time. So if an availability zone outage occurs, the volume is no longer available. This does not imply that you lose any data. As soon as the availability zone is back online, you can access the EBS volume again with all its data.

An EBS volume guarantees that you won't lose any data in 99.9% of the time. This guarantee is called the durability of the EBS volume. If you have 1,000 volumes in use, you can expect that you will lose one of the volumes and its data a year.

You have multiple solutions for this problem:

- Outsource the state of your virtual machine to a managed service that uses multiple availability zones by default: RDS, DynamoDB (NoSQL database), EFS (NFSv4.1 share), or S3 (object store).
- Create snapshots of your EBS volumes regularly, and use these snapshots if an EC2 instance needs to recover in another availability zone. EBS snapshots are stored on S3 and, thus, are available in multiple availability zones. If the EBS volume is the root volume of the ECS instance, create AMIs to back up the EBS volume instead of a snapshot.
- Use a distributed third-party storage solution to store your data in multiple availability zones: GlusterFS, DRBD, MongoDB, and so on.

The Jenkins server stores data directly on disk. To outsource the state of the virtual machine, you can't use RDS, DynamoDB, or S3; you need a file-level storage solution instead. As you've learned, an EBS volume is available only in a single availability zone, so this isn't the best fit for the problem. But do you remember EFS from chapter 9? EFS provides network file storage (over NFSv4.1) and replicates your data automatically between availability zones in a region.

To embed EFS into the Jenkins setup, as shown in listing 13.4, you have to make the following three modifications to the Multi-AZ template from the previous section:

1. Create an EFS filesystem.
2. Create EFS mount targets in each availability zone.
3. Adjust the user data to mount the EFS filesystem. Jenkins stores all its data under /var/lib/jenkins.

Listing 13.4 Storing Jenkins state on EFS

```
# [...]
FileSystem:
  Type: 'AWS::EFS::FileSystem'
  Properties: {}
MountTargetSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'EFS Mount target'
    SecurityGroupIngress:
    - FromPort: 2049
      IpProtocol: tcp
      SourceSecurityGroupId: !Ref SecurityGroup
      ToPort: 2049
    VpcId: !Ref VPC
MountTargetA:
  Type: 'AWS::EFS::MountTarget'
  Properties:
    FileSystemId: !Ref FileSystem
    SecurityGroups:
    - !Ref MountTargetSecurityGroup
    SubnetId: !Ref SubnetA
MountTargetB:
  Type: 'AWS::EFS::MountTarget'
  Properties:
    FileSystemId: !Ref FileSystem
    SecurityGroups:
    - !Ref MountTargetSecurityGroup
    SubnetId: !Ref SubnetB
# [...]
LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'
  Properties:
    LaunchTemplateData:
      # [...]
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1 --stack ${AWS::StackName}
          ➥ --resource AutoScalingGroup --region ${AWS::Region}' ERR
```

Creates an Elastic File System (EFS), which provides a NFS (network filesystem)

Creates a security group used to grant network traffic from the EC2 instance to EFS

Allows incoming traffic on port 2049 used by NFS

The mount target provides a network interface for the filesystem.

The mount target is attached to a subnet.

Therefore, you need a mount target per subnet.

The blueprint used by the Auto Scaling group to launch virtual machines

```
        # Installing Jenkins
        # [...]

        # Mounting EFS volume
        mkdir -p /var/lib/jenkins
        echo "${FileSystem}:/ /var/lib/jenkins efs tls,_netdev 0 0"
        ➥ >> /etc/fstab
        while ! (echo > /dev/tcp/${FileSystem}.efs.${AWS::Region}.
        ➥ amazonaws.com/2049) >/dev/null 2>&1; do sleep 5; done
        mount -a -t efs
        chown -R jenkins:jenkins /var/lib/jenkins

        # Configuring Jenkins
        # [...]

        # Starting Jenkins
        systemctl enable jenkins.service
        systemctl start jenkins.service
        /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
        ➥ --resource AutoScalingGroup --region ${AWS::Region}
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    LaunchTemplate:
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
    Tags:
    - Key: Name
      Value: 'jenkins-multiaz-efs'
      PropagateAtLaunch: true
    MinSize: 1
    MaxSize: 1
    VPCZoneIdentifier:
    - !Ref SubnetA
    - !Ref SubnetB
    HealthCheckGracePeriod: 600
    HealthCheckType: EC2
  # [...]
```

Creates a folder used by Jenkins to store data if it does not exist yet

Adds an entry to the configuration file for volumes

Waits until the EFS filesystem becomes available

Mounts the EFS filesystem

Changes the ownership of the mounted directory to make sure Jenkins is able to write and read files

Creates the Auto Scaling group

References the launch template defined above

You can find the CloudFormation template for this example on GitHub and on S3. Download a snapshot of the repository at https://github.com/AWSinAction/code3/archive/main.zip. The file we're talking about is located at chapter13/multiaz-efs.yaml. On S3, the same file is located at https://s3.amazonaws.com/awsinaction-code3/chapter13/multiaz-efs.yaml.

Execute the following command to create the new Jenkins setup that stores state on EFS. Replace $Password with a password consisting of 8–40 characters:

```
$ aws cloudformation create-stack --stack-name jenkins-multiaz-efs \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code3/chapter13/multiaz-efs.yaml \
➥ --parameters "ParameterKey=JenkinsAdminPassword,
➥ ParameterValue=$Password" \
➥ --capabilities CAPABILITY_IAM
```

The creation of the CloudFormation stack will take a few minutes. Run the following command to get the public IP address of the virtual machine. If no IP address appears, the virtual machine isn't started yet. In this case, wait another minute, and try again:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
➥ Values=jenkins-multiaz-efs" "Name=instance-state-code,Values=16" \
➥ --query "Reservations[0].Instances[0].\
➥ [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-0efcd2f01a3e3af1d",
  "34.236.255.218",
  "172.31.37.225",
  "subnet-0997e66d"
]
```

Instance ID of the virtual machine

Public IP address of the virtual machine

Private IP address of the virtual machine

Subnet ID of the virtual machine

Next, create a new Jenkins job by following these steps:

1. Open http://$PublicIP:8080/newJob in your browser, and replace `$PublicIP` with the public IP address from the output of the previous `describe` command.
2. Log in with user `admin` and the password you chose when starting the CloudFormation template.
3. Select the Install Suggested Plugins option.
4. Keep the default for Jenkins URL and click Save and Finish.
5. Click Start Using Jenkins.
6. Click New Item to create a new project.
7. Type in AWS in Action as the name for the new project.
8. Select Freestyle Project as the job type, and click OK.

You've made some changes to the state of Jenkins stored on EFS. Now terminate the EC2 instance with the following command, and you will see that Jenkins recovers from the failure without data loss. Replace `$InstanceId` with the instance ID from the output of the previous `describe` command:

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

After a few minutes, the Auto Scaling group detects that the virtual machine was terminated and starts a new virtual machine. Rerun the `describe-instances` command shown next until the output contains a new running virtual machine:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
➥ Values=jenkins-multiaz-efs" "Name=instance-state-code,Values=16" \
➥ --query "Reservations[0].Instances[0].\
➥ [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
```

```
[
  "i-07ce0865adf50cccf",
  "34.200.225.247",
  "172.31.37.199",
  "subnet-0997e66d"
]
```

The instance ID, the public IP address, the private IP address, and probably even the subnet ID have changed for the new instance. Open http://$PublicIP:8080 in your browser, and replace `$PublicIP` with the public IP address from the output of the previous `describe-instances` command. The web interface from the Jenkins server appears, and it still contains the AWS in Action job you created recently.

You've now built a highly available architecture consisting of an EC2 instance with the help of autoscaling. State is now stored on EFS and is no longer lost when an EC2 instance is replaced. There is one problem left: the public and private IP addresses of the Jenkins server change after a new virtual machine is started for recovery. The Jenkins server is no longer available under the same endpoint.

Cleaning up

It's time to clean up to avoid unwanted costs. Execute the following command to delete all resources corresponding to the Jenkins setup:

```
$ aws cloudformation delete-stack --stack-name jenkins-multiaz-efs
$ aws cloudformation wait stack-delete-complete \
      ➥ --stack-name jenkins-multiaz-efs
```

Waits until the stack is deleted

You'll learn how to solve the last problem next.

13.2.4 Pitfall: Network interface recovery

Recovering a virtual machine using a CloudWatch alarm in the same availability zone, as described at the beginning of this chapter, is easy because the private IP address and the public IP address stay the same automatically. You can use these IP addresses as an endpoint to access the EC2 instance, even after a failover.

When it comes to creating a virtual network in the cloud (VPC), you need to be aware of the following dependencies, as figure 13.6 illustrates:

- A VPC is always bound to a region.
- A subnet within a VPC is linked to an availability zone.
- A virtual machine is launched into a single subnet.

You can't keep the private IP address when using autoscaling to recover from a EC2 instance or availability zone outage. If a virtual machine has to be started in another

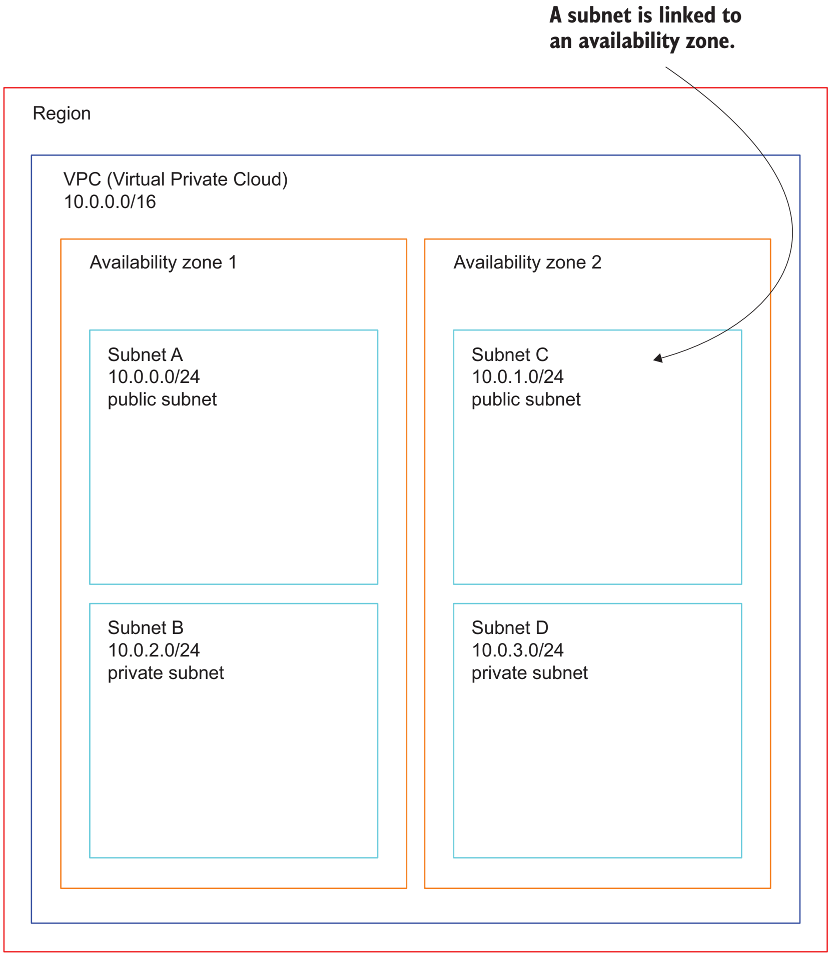
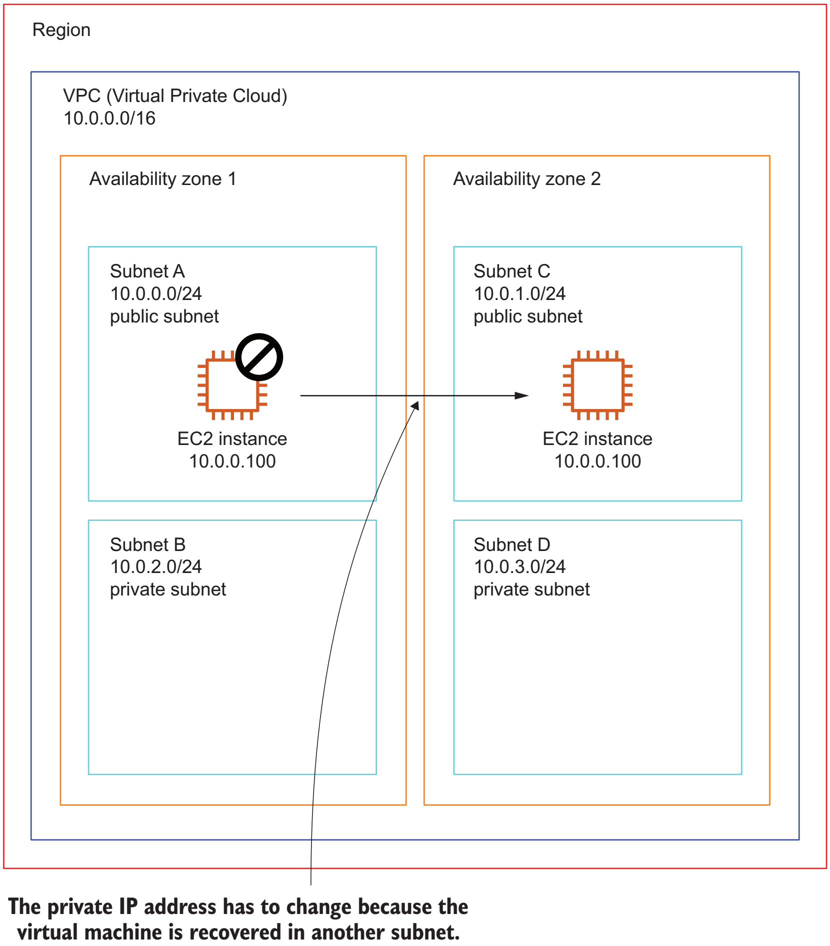


Figure 13.6 A VPC is bound to a region, and a subnet is linked to an availability zone.

availability zone, it must be started in another subnet. Therefore, it's not possible to use the same private IP address for the new virtual machine, as figure 13.7 shows.

By default, you also can't use an Elastic IP as a public IP address for a virtual machine launched by autoscaling. The requirement for a static endpoint to receive requests is common, though. For the use case of a Jenkins server, developers want to bookmark an IP address or a hostname to reach the web interface. Different possibilities exist for providing a static endpoint when using autoscaling to build high availability for a single virtual machine, as described here:

- Allocate an Elastic IP, and associate this public IP address during the bootstrap of the virtual machine.
- Create or update a DNS entry linking to the current public or private IP address of the virtual machine.



The private IP address has to change because the virtual machine is recovered in another subnet.

Figure 13.7 The virtual machine starts in another subnet in case of a failover and changes the private IP address.

- Use an Elastic Load Balancer (ELB) as a static endpoint that forwards requests to the current virtual machine.

To use the second solution, you need to link a domain with the Route 53 (DNS) service; we've chosen to skip this solution because you need a registered domain to implement it. The ELB solution is covered in chapter 14, so we'll skip it in this chapter as well. We'll focus on the first solution: allocating an Elastic IP and associating this public IP address during the virtual machine's bootstrap.

Execute the following command to create the Jenkins setup based on autoscaling again, using an Elastic IP address as static endpoint:

```
$ aws cloudformation create-stack --stack-name jenkins-multiaz-efs-eip \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code3/chapter13/multiaz-efs-eip.yaml \
➥ --parameters "ParameterKey=JenkinsAdminPassword,
➥ ParameterValue=$Password" \
➥ --capabilities CAPABILITY_IAM
```

You can find the CloudFormation template for this example on GitHub and on S3. Download a snapshot of the repository at https://github.com/AWSinAction/code3/archive/main.zip. The file we're talking about is located at chapter13/multiaz-efs-eip.yaml. On S3, the same file is located at https://s3.amazonaws.com/awsinaction-code3/chapter13/multiaz-efs-eip.yaml.

The command creates a stack based on the template shown in listing 13.5. The differences from the original template spinning up a Jenkins server with autoscaling follow:

- Allocating an Elastic IP
- Adding the association of an Elastic IP to the script in the user data
- Creating an IAM role and policy to allow the EC2 instance to associate an Elastic IP

Listing 13.5 Using an EIP as a static endpoint for a virtual machine

```
# [...]
ElasticIP:
  Type: 'AWS::EC2::EIP'
  Properties:
    Domain: vpc
  DependsOn: VPCGatewayAttachment
IamRole:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Version: '2012-10-17'
      Statement:
      - Effect: Allow
        Principal:
          Service: 'ec2.amazonaws.com'
        Action: 'sts:AssumeRole'
    Policies:
    - PolicyName: ec2
      PolicyDocument:
        Version: '2012-10-17'
        Statement:
        - Action: 'ec2:AssociateAddress'
          Resource: '*'
          Effect: Allow
    - PolicyName: ssm
      PolicyDocument:
        Version: '2012-10-17'
        Statement:
        - Effect: Allow
          Action:
          - 'ssmmessages:*'
          - 'ssm:UpdateInstanceInformation'
```

Creates a static public IP address

Creates an IAM role granting access to AWS services to the EC2 instance

The IAM role can be used only by EC2 instances.

The IAM policy allows access to the EC2 API action called AssociateAddress, which is used to associate an Elastic IP with an EC2 instance.

The other IAM policy enables access to the Session Manager, enabling you to open a terminal connection with the EC2 instance.

```
          - 'ec2messages:*'
          Resource: '*'
IamInstanceProfile:
  Type: 'AWS::IAM::InstanceProfile'
  Properties:
    Roles:
    - !Ref IamRole
LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'
  Properties:
    LaunchTemplateData:
      IamInstanceProfile:
        Name: !Ref IamInstanceProfile
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI]
      Monitoring:
        Enabled: false
      InstanceType: 't2.micro'
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeviceIndex: 0
        Groups:
        - !Ref SecurityGroup
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1 --stack ${AWS::StackName}
          ➥ --resource AutoScalingGroup --region ${AWS::Region}' ERR

          # Attaching EIP
          INSTANCE_ID="$(curl
          ➥ -s http://169.254.169.254/latest/meta-data/instance-id)"
          aws --region ${AWS::Region} ec2 associate-address
          ➥ --instance-id $INSTANCE_ID
          ➥ --allocation-id ${ElasticIP.AllocationId}
          ➥ --allow-reassociation
          sleep 30

          # Installing Jenkins [...]
          # Mounting EFS volume [...]
          # Configuring Jenkins [...]

          # Starting Jenkins
          systemctl enable jenkins.service
          systemctl start jenkins.service
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
          ➥ --resource AutoScalingGroup --region ${AWS::Region}
```

An IAM instance profile is needed to be able to attach an IAM role to an EC2 instance.

The launch template defines the blueprint for launching the EC2 instance.

Attaches the IAM instance profile defined when starting the virtual machine

Gets the ID of the running instance from the metadata service (see http://mng.bz/deAX for details)

The EC2 instance associates the Elastic IP address with itself by using the AWS CLI.

If the query returns output shown in the following listing, containing a URL, a user, and a password, the stack has been created and the Jenkins server is ready to use. Open the URL in your browser, and log in to the Jenkins server with user admin and the password you've chosen. If the output is null, try again in a few minutes:

```
$ aws cloudformation describe-stacks --stack-name jenkins-multiaz-efs-eip \
➥ --query "Stacks[0].Outputs"
```

You can now test whether the recovery of the virtual machine works as expected. To do so, you'll need to know the instance ID of the running virtual machine. Run the following command to get this information:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
➥ Values=jenkins-multiaz-efs-eip" "Name=instance-state-code,Values=16" \
➥ --query "Reservations[0].Instances[0].InstanceId" --output text
```

Execute the following command to terminate the virtual machine and test the recovery process triggered by autoscaling. Replace `$InstanceId` with the instance from the output of the previous command:

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

Wait a few minutes for your virtual machine to recover. Because you're using an Elastic IP assigned to the new virtual machine on bootstrap, you can open the same URL in your browser, as you did before the termination of the old instance.

Cleaning up

It's time to clean up to avoid unwanted costs. Execute the following command to delete all resources corresponding to the Jenkins setup:

```
$ aws cloudformation delete-stack --stack-name jenkins-multiaz-efs-eip
$ aws cloudformation wait stack-delete-complete \
➥ --stack-name jenkins-multiaz-efs-eip
```

Waits until the stack is deleted

Now the public IP address of your virtual machine running Jenkins won't change, even if the running virtual machine needs to be replaced by another virtual machine in another availability zone.

Last but not least, we want to come back to the concept of an availability zone and dive into some of the details.

13.2.5 Insights into availability zones

A region consists of multiple availability zones. Each availability zone consists of at least one isolated data center. The identifier for an availability zone consists of the identifier for the region (such as us-east-1) and a character (a, b, c, ...). So us-east-1a is the identifier for an availability zone in region us-east-1. To distribute resources across the different availability zones, the AZ identifier is mapped to one or multiple data centers randomly when creating an AWS account. This means us-east-1a might point to a different availability zone in your AWS account than it does in our AWS account.

We recommend that you take some time to explore the worldwide infrastructure provided by AWS. You can use the following commands to discover all regions available for your AWS account:

```
$ aws ec2 describe-regions
{
  "Regions": [
    {
      "Endpoint": "ec2.eu-north-1.amazonaws.com",
      "RegionName": "eu-north-1",
      "OptInStatus": "opt-in-not-required"
    },
    {
      "Endpoint": "ec2.ap-south-1.amazonaws.com",
      "RegionName": "ap-south-1",
      "OptInStatus": "opt-in-not-required"
    },
    [...]
    {
      "Endpoint": "ec2.us-west-2.amazonaws.com",
      "RegionName": "us-west-2",
      "OptInStatus": "opt-in-not-required"
    }
  ]
}
```

Lists all regions available for your AWS account

The endpoint URL, used to access the EC2 service in the region

The name of the region

Newer regions require an opt-in.

Next, to list all availability zones for a region, execute the following command and replace `$Region` with `RegionName` of a region from the previous output:

```
$ aws ec2 describe-availability-zones --region $Region
{
  "AvailabilityZones": [
    {
      "State": "available",
      "OptInStatus": "opt-in-not-required",
      "Messages": [],
      "RegionName": "us-east-1",
      "ZoneName": "us-east-1a",
      "ZoneId": "use1-az1",
      "GroupName": "us-east-1",
      "NetworkBorderGroup": "us-east-1",
      "ZoneType": "availability-zone"
    },
    {
      "State": "available",
      "OptInStatus": "opt-in-not-required",
      "Messages": [],
      "RegionName": "us-east-1",
      "ZoneName": "us-east-1b",
      "ZoneId": "use1-az2",
      "GroupName": "us-east-1",
      "NetworkBorderGroup": "us-east-1",
      "ZoneType": "availability-zone"
    },
    [...]
    {
      "State": "available",
      "OptInStatus": "opt-in-not-required",
```

Lists the availability zones of a region

The region name

The name of the availability zone might point to different data centers in different AWS accounts.

The ID of the availability zone points to the same data centers in all AWS accounts.

```
            "Messages": [],
            "RegionName": "us-east-1",
            "ZoneName": "us-east-1f",
            "ZoneId": "use1-az5",
            "GroupName": "us-east-1",
            "NetworkBorderGroup": "us-east-1",
            "ZoneType": "availability-zone"
        }
    ]
}
```

At the end of the chapter, you will learn how to analyze resilience requirements and derive an AWS architecture from the results.

13.3 Architecting for high availability

Before you begin implementing highly available or even fault-tolerant architectures on AWS, you should start by analyzing your disaster-recovery requirements. Disaster recovery is easier and cheaper in the cloud than in a traditional data center, but building for high availability increases the complexity and, therefore, the initial costs as well as the operating costs of your system. The recovery time objective (RTO) and recovery point objective (RPO) are standards for defining the importance of disaster recovery from a business point of view.

Recovery time objective (RTO) is the time it takes for a system to recover from a failure; it's the length of time until the system reaches a working state again, defined as the system service level, after an outage. In the example with a Jenkins server, the RTO would be the time until a new virtual machine is started and Jenkins is installed and running after a virtual machine or an entire availability zone goes down.

Recovery point objective (RPO) is the acceptable data-loss time caused by a failure. The amount of data loss is measured in time. If an outage happens at 10:00 a.m. and the system recovers with a data snapshot from 09:00 a.m., the time span of the data loss is one hour. In the example of a Jenkins server using autoscaling, the RPO would be zero, because data is stored on EFS and is not lost during an AZ outage. Figure 13.8 illustrates the definitions of RTO and RPO.

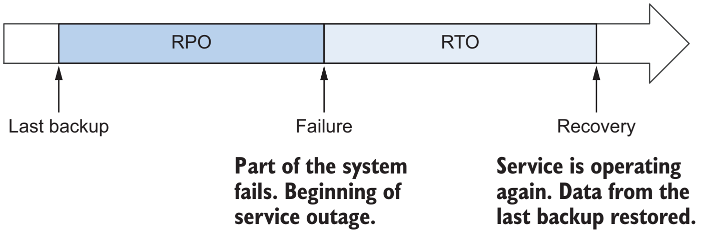

RPO
RTO
Last backup
Failure
Recovery
Part of the system fails. Beginning of service outage.
Service is operating again. Data from the last backup restored.

Figure 13.8 Definitions of RTO and RPO

13.3.1 RTO and RPO comparison for a single EC2 instance

You've learned about two possible solutions for making a single EC2 instance highly available. When choosing the solution, you have to know the application's business requirements. Can you tolerate the risk of being unavailable if an availability zone goes down? If so, EC2 instance recovery is the simplest solution, where you don't lose any data. If your application needs to survive an unlikely availability zone outage, your safest bet is autoscaling with data stored on EFS, but this method also has performance effects compared to storing data on EBS volumes. As you can see, there is no one-size-fits-all solution. You have to pick the solution that fits your business problem best. Table 13.2 compares the solutions.

Table 13.2 Comparison of high availability for a single EC2 instance

	RTO	RPO	Availability
EC2 instance, data stored on EBS root volume: recovery triggered by a CloudWatch alarm	About 10 minutes	No data loss	Recovers from a failure of a virtual machine but not from an outage of an entire availability zone
EC2 instance, data stored on EBS root volume: recovery triggered by autoscaling	About 10 minutes	All data lost	Recovers from a failure of a virtual machine and from an outage of an entire availability zone
EC2 instance, data stored on EBS root volume with regular snapshots: recovery triggered by autoscaling	About 10 minutes	Realistic time span for snapshots: between 30 minutes and 24 hours	Recovers from a failure of a virtual machine and from an outage of an entire availability zone
EC2 instance, data stored on EFS filesystem: recovery triggered by autoscaling	About 10 minutes	No data loss	Recovers from a failure of a virtual machine and from an outage of an entire availability zone

If you want to be able to recover from an outage of an availability zone and need to decrease the RPO, you should try to achieve a stateless server. Using storage services like RDS, EFS, S3, and DynamoDB can help you to do so. See part 3 if you need help with using these services.

13.3.2 AWS services come with different high availability guarantees

It is important to note that some AWS services are highly available or even fault-tolerant by default. Other services provide building blocks to achieve a highly available architecture. As described next, you can use multiple availability zones or even multiple regions to build a highly available architecture, as figure 13.9 shows:

- Route 53 (DNS) and CloudFront (CDN) operate globally over multiple regions and are highly available by default.
- S3 (object store), EFS (network filesystem) and DynamoDB (NoSQL database) use multiple availability zones within a region so they can withstand a data center outage.

- The Relational Database Service (RDS) offers the ability to deploy a primary-standby setup, called Multi-AZ deployment, so you can fail over into another availability zone with a short downtime, if necessary.
- A virtual machine runs in a single availability zone. AWS offers a tool to build an architecture based on EC2 instances that can fail over into another availability zone: autoscaling.

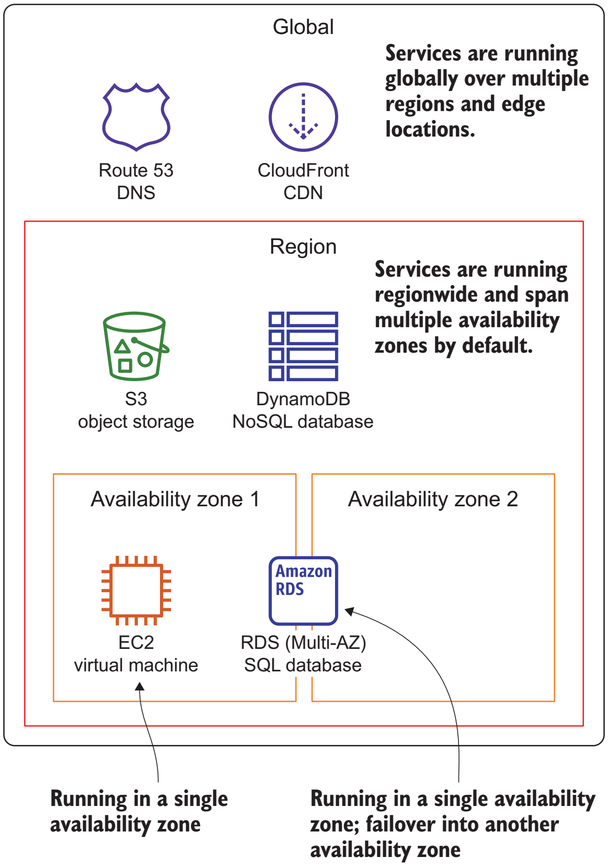

Global
Services are running globally over multiple regions and edge locations.
Route 53
DNS
CloudFront
CDN
Region
Services are running regionwide and span multiple availability zones by default.
S3
object storage
DynamoDB
NoSQL database
Availability zone 1
Availability zone 2
Amazon RDS
EC2
virtual machine
RDS (Multi-AZ)
SQL database
Running in a single availability zone
Running in a single availability zone; failover into another availability zone

Figure 13.9 AWS services can operate in a single availability zone, over multiple availability zones within a region, or even globally.

When planning for failure, it is also important to consider the service-level objective (SLO) and service-level agreement (SLA) committed to by AWS. Most services define an SLA, which helps you as a customer when estimating the availability of an architecture. You can read them here: http://mng.bz/rn7Z.

When designing a system for AWS, you need to look into the SLA and resilience specifications of each building block. To do so, check the AWS documentation, which includes a section on resilience for most services.

Summary

- A virtual machine fails if the underlying hardware or virtualization layer fails.
- You can recover a failed virtual machine with the help of a CloudWatch alarm: by default, data stored on EBS, as well as the private and public IP addresses, stays the same.
- An AWS region consists of multiple isolated groups of data centers called availability zones.
- Recovering from a data center outage is possible when using multiple availability zones.
- Use autoscaling to replace a failed virtual machine, even in the event of a data center outage. The pitfalls are that you can no longer blindly rely on EBS volumes and, by default, IP addresses will change.
- Recovering data in another availability zone is tricky when stored on EBS volumes instead of managed storage services like RDS, EFS, S3, and DynamoDB.
- Some AWS services use multiple availability zones by default, but virtual machines run in a single availability zone.

14 Decoupling your infrastructure: Elastic Load Balancing and Simple Queue Service

This chapter covers

- The reasons for decoupling a system
- Synchronous decoupling with load balancers to distribute requests
- Hiding your backend from users and message producers
- Asynchronous decoupling with message queues to buffer message peaks

Imagine that you want some advice from us about using AWS, and therefore, we plan to meet in a café. To make this meeting successful, we must:

- Be available at the same time
- Be at the same place
- Find each other at the café

The problem with making our meeting happing is that it's *tightly coupled* to a location. We live in Germany; you probably don't. We can solve that problem by decoupling our meeting from the location. So, we change plans and schedule a Google Hangout session. Now we must:

- Be available at the same time
- Find each other in Google Hangouts

Google Hangouts (and other video/voice chat services) does *synchronous decoupling*. It removes the need to be at the same place, while still requiring us to meet at the same time.

We can even decouple from time by using email. Now we must:

- Find each other via email

Email does *asynchronous decoupling*. You can send an email when the recipient is asleep, and they can respond later when they're awake.

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.

NOTE To fully understand this chapter, you'll need to have read and understood the concept of autoscaling covered in chapter 13.

In summary, to meet up, we have to be at the same place (the café), at the same time (3 p.m.) and find each other (I have black hair and I'm wearing a white shirt). Our meeting is tightly coupled to a location and a place. We can decouple a meeting in the following two ways:

- *Synchronous decoupling*—We can now be at different places, but we still have to find a common time (3 p.m.) and find each other (exchange Skype IDs, for instance).
- *Asynchronous decoupling*—We can be at different places and now also don't have to find a common time. We only have to find each other (exchange email addresses).

A meeting isn't the only thing that can be decoupled. In software systems, you can find a lot of tightly coupled components, such as the following:

- A public IP address is like the location of our meeting: to make a request to a web server, you must know its public IP address, and the virtual machine must be connected to that address. If you want to change the public IP address, both parties are involved in making the appropriate changes. The public IP address is tightly coupled with the web server.
- If you want to make a request to a web server, the web server must be online at the same time. Otherwise, your request will be rejected. A web server can be offline for many reasons: someone might be installing updates, a hardware failure, and so on. The client is tightly coupled with the web server.

AWS offers solutions for synchronous and asynchronous decoupling. Typically, synchronous decoupling is used when the client expects an immediate response. For example, a user expects an response to the request to load the HTML of a website with very little latency. The Elastic Load Balancing (ELB) service provides different types of load balancers that sit between your web servers and the client to decouple your requests synchronously. The client sends a request to the ELB, and the ELB forwards the request to a virtual machine or similar target. Therefore, the client does not need to know about the target; it knows only about the load balancer.

Asynchronous decoupling is different and commonly used in scenarios where the client does not expect an immediate response. For example, a web application could scale and optimize an image uploaded by the user in the background and use the raw image until that process finished in the background. For asynchronous decoupling, AWS offers the *Simple Queue Service* (SQS), which provides a message queue. The producer sends a message to the queue, and a receiver fetches the message from the queue and processes the request.

You'll learn about both the ELB and the SQS services in this chapter. Let's start with ELB.

14.1 Synchronous decoupling with load balancers

Exposing a single EC2 instance running a web server to the outside world introduces a dependency: your users now depend on the public IP address of the EC2 instance. As soon as you distribute the public IP address to your users, you can't change it anymore. You're faced with the following problems:

- Changing the public IP address is no longer possible because many clients rely on it.
- If you add an additional EC2 instance (and IP address) to handle the increasing load, it's ignored by all current clients: they're still sending all requests to the public IP address of the first server.

You can solve these problems with a DNS name that points to your server. But DNS isn't fully under your control. DNS resolvers cache responses. DNS servers cache entries, and sometimes they don't respect your time-to-live (TTL) settings. For example, you might ask DNS servers to only cache the name-to–IP address mapping for one minute, but some DNS servers might use a minimum cache of one day. A better solution is to use a load balancer.

A load balancer can help decouple a system where the requester awaits an immediate response. Instead of exposing your EC2 instances (running web servers) to the outside world, you expose only the load balancer to the outside world. The load balancer then forwards requests to the EC2 instances behind it. Figure 14.1 shows how this works.

The requester (such as a web browser) sends an HTTP request to the load balancer. The load balancer then selects one of the EC2 instances and copies the original

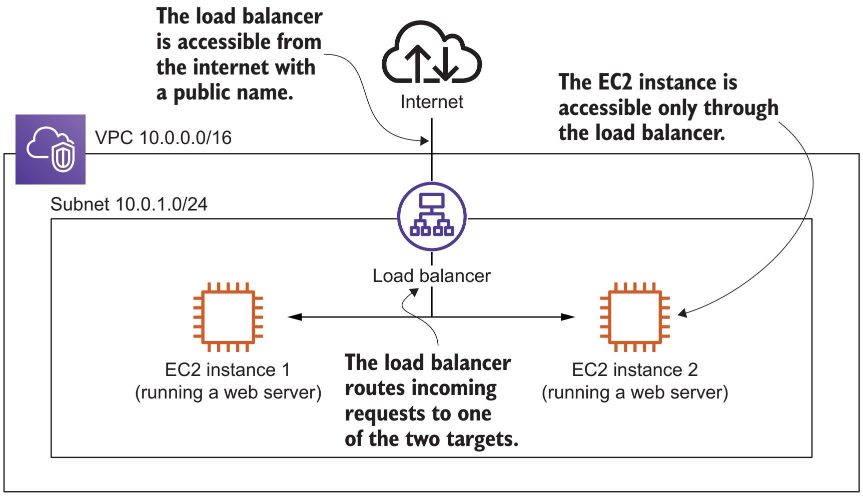

The load balancer is accessible from the internet with a public name.
Internet
The EC2 instance is accessible only through the load balancer.
VPC 10.0.0.0/16
Subnet 10.0.1.0/24
Load balancer
EC2 instance 1 (running a web server)
The load balancer routes incoming requests to one of the two targets.
EC2 instance 2 (running a web server)

Figure 14.1 A load balancer synchronously decouples your EC2 instances.

HTTP request to send to the EC2 instance that it selected. The EC2 instance then processes the request and sends a response. The load balancer receives the response and sends the same response to the original requester.

AWS offers different types of load balancers through the Elastic Load Balancing (ELB) service. All load balancer types are fault tolerant and scalable. They differ in supported protocols and features as follows:

- *Application Load Balancer (ALB)*—HTTP, HTTPS
- *Network Load Balancer (NLB)*—TCP, TCP TLS
- *Classic Load Balancer (CLB)*—HTTP, HTTPS, TCP, TCP TLS

Consider the CLB deprecated. As a rule of thumb, use the ALB whenever the HTTP/HTTPS protocol is all you need, and the NLB for all other scenarios.

NOTE The ELB service doesn't have an independent management console. It's integrated into the EC2 Management Console.

Load balancers can be used with more than web servers—you can use load balancers in front of any systems that deal with request/response-style communication, as long as the protocol is based on TCP.

14.1.1 Setting up a load balancer with virtual machines

AWS shines when it comes to integrating services. In chapter 13, you learned about Auto Scaling groups. You'll now put an ALB in front of an Auto Scaling group to decouple traffic to web servers by removing the dependency between your users and the EC2 instance's public IP address. The Auto Scaling group will make sure you always have two web servers running. As you learned in chapter 13, that's the way to protect against downtime caused by hardware failure. Servers that are started in the Auto Scaling group can automatically register with the ALB.

Figure 14.2 shows what the setup will look like. The interesting part is that the EC2 instances are no longer accessible directly from the public internet, so your users don't know about them. They don't know if there are two or 20 EC2 instances running behind the load balancer. Only the load balancer is accessible, and it forwards requests to the backend servers behind it. The network traffic to load balancers and backend EC2 instances is controlled by security groups, which you learned about in chapter 5.

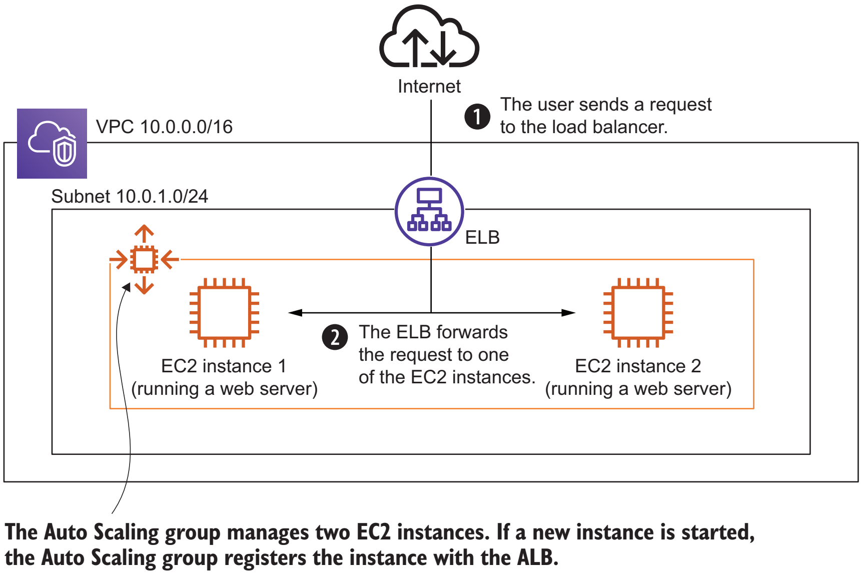

Internet
1 The user sends a request to the load balancer.
VPC 10.0.0.0/16
Subnet 10.0.1.0/24
ELB
2 The ELB forwards the request to one of the EC2 instances.
EC2 instance 1 (running a web server)
EC2 instance 2 (running a web server)
The Auto Scaling group manages two EC2 instances. If a new instance is started, the Auto Scaling group registers the instance with the ALB.

Figure 14.2 The load balancer evaluates rules so it can forward incoming rules to a specific target group.

If the Auto Scaling group adds or removes EC2 instances, it will also register new EC2 instances at the load balancer's target group and deregister EC2 instances that have been removed.

An ALB consists of the following three required parts and one optional part:

- *Load balancer*—Defines some core configurations, like the subnets the load balancer runs in, whether the load balancer gets public IP addresses, whether it uses IPv4 or both IPv4 and IPv6, and additional attributes.
- *Listener*—The listener defines the port and protocol that you can use to make requests to the load balancer. If you like, the listener can also terminate TLS for you. A listener links to a target group that is used as the default if no other listener rules match the request.

- *Target group*—A target group defines your group of backends. The target group is responsible for checking the backends by sending periodic health checks. Usually backends are EC2 instances, but they could also be a container running on Elastic Container Service (ECS) as well as Elastic Kubernetes Service (EKS), a Lambda function, or a machine in your data center connected with your VPC.
- *Listener rule*—Optional. You can define a listener rule. The rule can choose a different target group based on the HTTP path or host. Otherwise, requests are forwarded to the default target group defined in the listener.

Figure 14.3 shows the ALB parts.

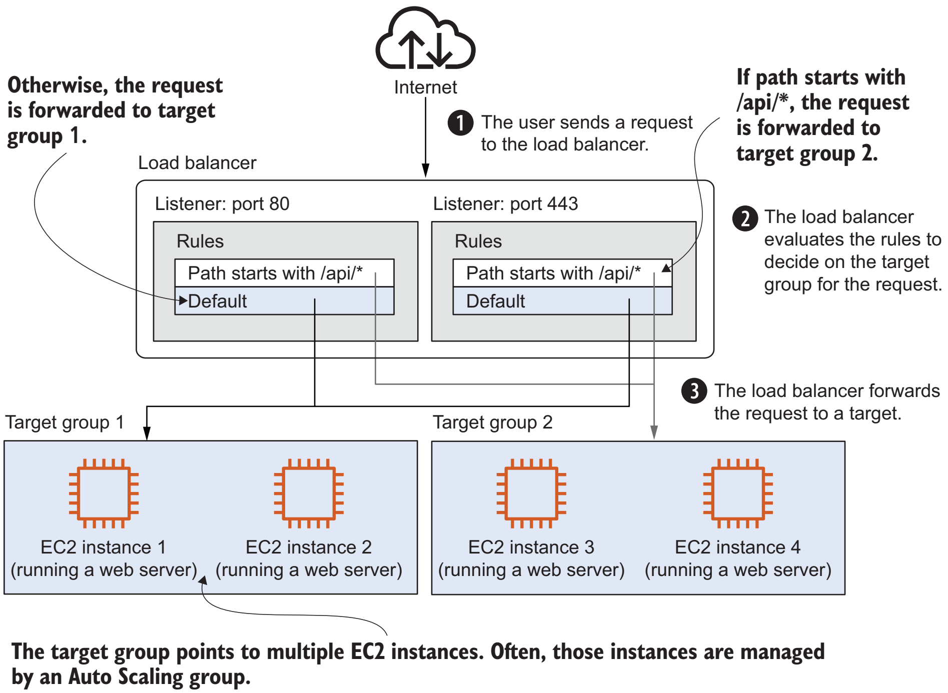

Internet
1 The user sends a request to the load balancer.
Otherwise, the request is forwarded to target group 1.
If path starts with /api/*, the request is forwarded to target group 2.
Load balancer
Listener: port 80
Rules
Path starts with /api/*
Default
Listener: port 443
Rules
Path starts with /api/*
Default
2 The load balancer evaluates the rules to decide on the target group for the request.
3 The load balancer forwards the request to a target.
Target group 1
Target group 2
EC2 instance 1 (running a web server)
EC2 instance 2 (running a web server)
EC2 instance 3 (running a web server)
EC2 instance 4 (running a web server)
The target group points to multiple EC2 instances. Often, those instances are managed by an Auto Scaling group.

Figure 14.3 Creating an ALB, listener, and target group. Also, the Auto Scaling group registers instances at the target group automatically.

The following three listings implement the example shown in figure 14.3. The first listing shows a CloudFormation template snippet to create an ALB and its firewall rules, the security group.

Listing 14.1 Creating a load balancer and connecting it to an Auto Scaling group A

```
# [...]
LoadBalancerSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'alb-sg'
    VpcId: !Ref VPC
    SecurityGroupIngress:
    - CidrIp: '0.0.0.0/0'
      FromPort: 80
      IpProtocol: tcp
      ToPort: 80
LoadBalancer:
  Type: 'AWS::ElasticLoadBalancingV2::LoadBalancer'
  Properties:
    Scheme: 'internet-facing'
    SecurityGroups:
    - !Ref LoadBalancerSecurityGroup
    Subnets:
    - !Ref SubnetA
    - !Ref SubnetB
    Type: application
  DependsOn: 'VPCGatewayAttachment'
```

Only traffic on port 80 from the internet will reach the load balancer.

The ALB is publicly accessible (use internal instead of internet-facing to define a load balancer reachable only from a private network).

Attaches the ALB to the subnets

Assigns the security group to the load balancer

The second listing configures the load balancer to listen on port 80 for incoming HTTP requests. It also creates a target group. The default action of the listener forwards all incoming requests to the target group.

Listing 14.2 Creating a load balancer and connecting it to an Auto Scaling group B

```
Listener:
  Type: 'AWS::ElasticLoadBalancingV2::Listener'
  Properties:
    DefaultActions:
    - TargetGroupArn: !Ref TargetGroup
      Type: forward
    LoadBalancerArn: !Ref LoadBalancer
    Port: 80
    Protocol: HTTP
TargetGroup:
  Type: 'AWS::ElasticLoadBalancingV2::TargetGroup'
  Properties:
    HealthCheckIntervalSeconds: 10
    HealthCheckPath: '/index.html'
    HealthCheckProtocol: HTTP
    HealthCheckTimeoutSeconds: 5
    HealthyThresholdCount: 3
    UnhealthyThresholdCount: 2
    Matcher:
      HttpCode: '200-299'
    Port: 80
    Protocol: HTTP
    VpcId: !Ref VPC
```

The load balancer forwards all requests to the default target group.

The load balancer listens on port 80 for HTTP requests.

Every 10 seconds …

… HTTP requests are made to /index.html.

If HTTP status code is 2XX, the backend is considered healthy.

The web server on the EC2 instances listens on port 80.

Shown in the third listing is the missing part: the targets. In our example, we are using an Auto Scaling group to launch EC2 instances. The Auto Scaling group registers the virtual machine at the target group.

Listing 14.3 Creating a load balancer and connecting it to an Auto Scaling group C

```
LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'
  # [...]
  Properties:
    LaunchTemplateData:
      IamInstanceProfile:
        Name: !Ref InstanceProfile
      ImageId: !FindInMap [RegionMap, !Ref 'AWS::Region', AMI]
      Monitoring:
        Enabled: false
      InstanceType: 't2.micro'
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeviceIndex: 0
        Groups:
        - !Ref WebServerSecurityGroup
      UserData: # [...]
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    LaunchTemplate:
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
    MinSize: !Ref NumberOfVirtualMachines
    MaxSize: !Ref NumberOfVirtualMachines
    DesiredCapacity: !Ref NumberOfVirtualMachines
    TargetGroupARNs:
    - !Ref TargetGroup
    VPCZoneIdentifier:
    - !Ref SubnetA
    - !Ref SubnetB
  CreationPolicy:
    ResourceSignal:
      Timeout: 'PT10M'
  DependsOn: 'VPCGatewayAttachment'
```

Keeps two EC2 instances running (MinSize ⇐ DesiredCapacity ⇐ MaxSize)

The Auto Scaling group registers new EC2 instances with the default target group.

The connection between the ALB and the Auto Scaling group is made in the Auto Scaling group description by specifying `TargetGroupARNs`.

The full CloudFormation template is located at http://mng.bz/VyKO. Create a stack based on that template by clicking on the Quick-Create link at http://mng.bz/GRgO, and then visit the output of your stack with your browser. Every time you reload the page, you should see one of the private IP addresses of a backend web server.

To get some detail about the load balancer in the graphical user interface, navigate to the EC2 Management Console. The subnavigation menu on the left has a Load Balancing section where you can find a link to your load balancers. Select the one and only load balancer. You will see details at the bottom of the page. The details contain a Monitoring tab, where you can find charts about latency, number of requests, and much more. Keep in mind that those charts are one minute behind, so you may have to wait until you see the requests you made to the load balancer.

Cleaning up

Delete the CloudFormation stack you created.

14.2 Asynchronous decoupling with message queues

Synchronous decoupling with ELB is easy; you don't need to change your code to do it. But for asynchronous decoupling, you have to adapt your code to work with a message queue.

A message queue has a head and a tail. You can add new messages to the tail while reading messages from the head. This allows you to decouple the production and consumption of messages. Now, why would you want to decouple the producers/requesters from consumers/receivers? You can achieve the following key benefits:

- *The queue acts as a buffer.* Producers and consumers don't have to run at the same speed. For example, you can add a batch of 1,000 messages in one minute while your consumers always process 10 messages per second. Sooner or later, the consumers will catch up, and the queue will be empty again.
- *The queue hides your backend.* Similar to the load balancer, message producers have no knowledge of the consumers. You can even stop all consumers and still produce messages. This is handy while doing maintenance on your consumers.

When decoupled, the producers and consumers don't know each other; they both only know about the message queue. Figure 14.4 illustrates this principle.

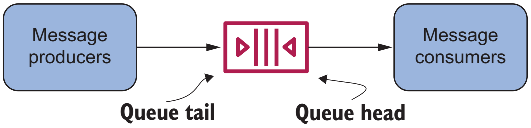


Figure 14.4 Producers send messages to a message queue while consumers read messages.

As you decoupled the sender from the receiver, the sender could even put new messages into the queue while no one is consuming messages, with the message queue acting as a buffer. To prevent message queues from growing infinitely large, messages are saved for only a certain amount of time. If you consume a message from a message queue, you must acknowledge the successful processing of the message to permanently delete it from the queue.

How do you implement asynchronous decoupling on AWS? That's where the Simple Queue Service (SQS) comes into play. SQS offers simple but highly scalable—throughput and storage—message queues that guarantee the delivery of messages at least once with the following characteristics:

- Under rare circumstances, a single message will be available for consumption twice. This may sound strange if you compare it to other message queues, but you'll see how to deal with this problem later in the chapter.
- SQS doesn't guarantee the order of messages, so you may read messages in a different order than they were produced. Learn more about the message order at the end of this section.

This limitation of SQS is also beneficial for the following reasons:

- You can put as many messages into SQS as you like.
- The message queue scales with the number of messages you produce and consume.
- SQS is highly available by default.
- You pay per message.

The pricing model is simple: $0.24 to $0.40 USD per million requests. Also, the first million requests per month are free. It is important to know that producing a message counts as a request, and consuming is another request. If your payload is larger than 64 KB, every 64 KB chunk counts as one request.

We have observed that many applications default to a synchronous process. That's probably because we are used to the request-response model and sometimes forget to think outside the box. However, replacing a synchronous with an asynchronous process enables many advantages in the cloud. Most importantly, scaling becomes much easier when you have a queue that can buffer requests for a while. Therefore, you will learn how to transition to an asynchronous process with the help of SQS next.

14.2.1 Turning a synchronous process into an asynchronous one

A typical synchronous process looks like this: a user makes a request to your web server, something happens on the web server, and a result is returned to the user. To make things more concrete, we'll talk about the process of creating a preview image of a URL in the following example, illustrated in figure 14.5:

1. The user submits a URL.
2. The web server downloads the content at the URL, takes a screenshot, and renders it as a PNG image.
3. The web server returns the PNG to the user.

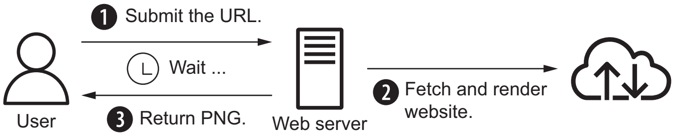


Figure 14.5 A synchronous process to create a screenshot of a website.

With one small trick, this process can be made asynchronous and benefit from the elasticity of a message queue, for example, during peak traffic, as shown in figure 14.6:

1. The user submits a URL.
2. The web server puts a message into a queue that contains a random ID and the URL.
3. The web server returns a link to the user where the PNG image will be found in the future. The link contains the random ID (such as http://$Bucket.s3.amazonaws.com/$RandomId.png).
4. In the background, a worker consumes the message from the queue.
5. The worker downloads the content and converts the content into a PNG.
6. Next, the worker uploads the image to S3.
7. At some point, the user tries to download the PNG at the known location. If the file is not found, the user should reload the page in a few seconds.

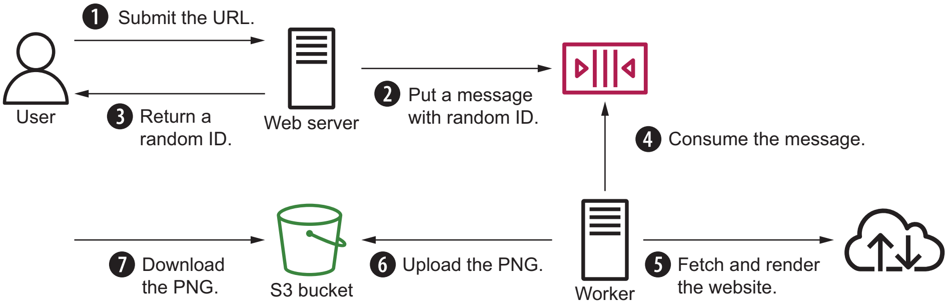


Figure 14.6 The same process, but asynchronous

If you want to make a process asynchronous, you must manage the way the process initiator tracks the process status. One way of doing that is to return an ID to the initiator that can be used to look up the process. During the process, this ID is passed from step to step.

14.2.2 Architecture of the URL2PNG application

You'll now create a basic but decoupled piece of software named URL2PNG that renders a PNG from a given web URL. You'll use Node.js to do the programming part, and you'll use SQS as the message queue implementation. Figure 14.7 shows how the URL2PNG application works.

On the message producer side, a small Node.js script generates a unique ID, sends a message to the queue with the URL and ID as the payload, and returns the ID to the user. The user now starts checking whether a file is available on the S3 bucket using the returned ID as the filename.

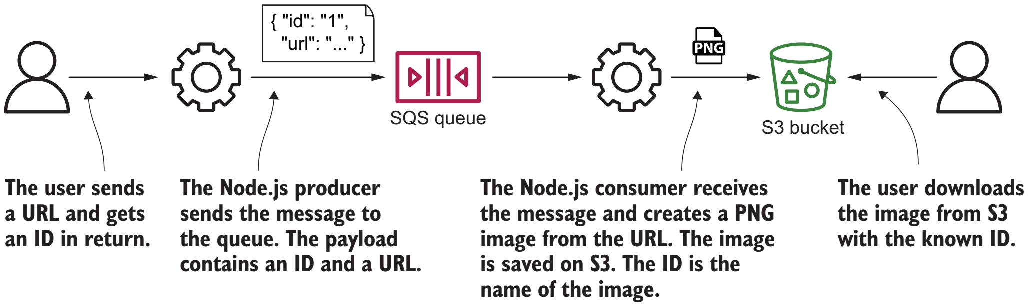

{ "id": "1",
"url": "..." }
SQS queue
PNG
S3 bucket
The user sends a URL and gets an ID in return.
The Node.js producer sends the message to the queue. The payload contains an ID and a URL.
The Node.js consumer receives the message and creates a PNG image from the URL. The image is saved on S3. The ID is the name of the image.
The user downloads the image from S3 with the known ID.

Figure 14.7 Node.js producer sends a message to the queue. The payload contains an ID and URL.

Simultaneously, on the message consumer side, a small Node.js script reads a message from the queue, generates the screenshot of the URL from the payload, and uploads the resulting image to an S3 bucket using the unique ID from the payload as the filename.

To complete the example, you need to create an S3 bucket with web hosting enabled. Execute the following command, replacing `$yourname` with your name or nickname to prevent name clashes with other readers (remember that S3 bucket names have to be globally unique across all AWS accounts):

```
$ aws s3 mb s3://url2png-$yourname
```

Now it's time to create the message queue.

14.2.3 Setting up a message queue

Creating an SQS queue is easy: you only need to specify the name of the queue as follows:

```
$ aws sqs create-queue --queue-name url2png
{
  "QueueUrl": "https://queue.amazonaws.com/878533158213/url2png"
}
```

The returned `QueueUrl` is needed later in the example, so take note of it.

14.2.4 Producing messages programmatically

You now have an SQS queue to send messages to. To produce a message, you need to specify the queue and a payload. You'll use Node.js in combination with the AWS SDK to make requests to AWS.

Installing and getting started with Node.js

Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit https://nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 14.

After Node.js is installed, you can verify that everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v14.*`. Now you're ready to run JavaScript examples like URL2PNG.

Do you want to get started with Node.js? We recommend *Node.js in Action* (second edition) by Alex Young et al. (Manning, 2017), or the video course *Node.js in Motion* by PJ Evans (Manning, 2018).

Here's how the message is produced with the help of the AWS SDK for Node.js; it will be consumed later by the URL2PNG worker. The Node.js script can then be used like this (don't try to run this command now—you need to install and configure URL2PNG first):

```
$ node index.js "http://aws.amazon.com"
PNG will be available soon at
http://url2png-$yourname.s3.amazonaws.com/XYZ.png
```

As usual, you'll find the code in the book's code repository on GitHub https://github.com/AWSinAction/code3. The URL2PNG example is located at /chapter14/url2png/. The next listing shows the implementation of index.js.

Listing 14.4 index.js: Sending a message to the queue

```
const AWS = require('aws-sdk');
var { v4: uuidv4 }  = require('uuid');
const config = require('./config.json');
const sqs = new AWS.SQS({});
```

Creates an SQS client

```
if (process.argv.length !== 3) {
  console.log('URL missing');
  process.exit(1);
}
```

Checks whether a URL was provided

```
const id = uuidv4();
```

Creates a random ID

```
const body = {
  id: id,
  url: process.argv[2]
};
```

The payload contains the random ID and the URL.

```
sqs.sendMessage({
  MessageBody: JSON.stringify(body),
```

Invokes the sendMessage operation on SQS

Converts the payload into a JSON string

```
  QueueUrl: config.QueueUrl
}, (err) => {
  if (err) {
    console.log('error', err);
  } else {
    console.log('PNG will be soon available at http://' + config.Bucket
    ➥ + '.s3.amazonaws.com/' + id + '.png');
  }
});
```

Queue to which the message is sent (was returned when creating the queue)

Before you can run the script, you need to install the Node.js modules. Run `npm install` in your terminal to install the dependencies. You'll find a config.json file that needs to be modified. Make sure to change `QueueUrl` to the queue you created at the beginning of this example, and change `Bucket` to `url2png-$yourname`.

Now you can run the script with `node index.js "http://aws.amazon.com"`. The program should respond with something like `PNG will be available soon at http://url2png-$yourname.s3.amazonaws.com/XYZ.png`. To verify that the message is ready for consumption, you can ask the queue how many messages are inside as follows. Replace `$QueueUrl` with your queue's URL:

```
$ aws sqs get-queue-attributes \
➥ --queue-url "$QueueUrl" \
➥ --attribute-names ApproximateNumberOfMessages
{
  "Attributes": {
    "ApproximateNumberOfMessages": "1"
  }
}
```

SQS returns only an approximation of the number of messages. This is due to the distributed nature of SQS. If you don't see your message in the approximation, run the command again and eventually you will see your message. Next, it's time to create the worker that consumes the message and does all the work of generating a PNG.

14.2.5 Consuming messages programmatically

Processing a message with SQS takes the next three steps:

1. Receive a message.
2. Process the message.
3. Acknowledge that the message was successfully processed.

You'll now implement each of these steps to change a URL into a PNG.

To receive a message from an SQS queue, you must specify the following:

- `QueueUrl`—The unique queue identifier.
- `MaxNumberOfMessages`—The maximum number of messages you want to receive (from one to 10). To get higher throughput, you can get messages in a batch. We usually set this to 10 for best performance and lowest overhead.

- `VisibilityTimeout`—The number of seconds you want to remove this message from the queue to process it. Within that time, you must delete the message, or it will be delivered back to the queue. We usually set this to the average processing time multiplied by four.
- `WaitTimeSeconds`—The maximum number of seconds you want to wait to receive messages if they're not immediately available. Receiving messages from SQS is done by polling the queue. AWS allows long polling, for a maximum of 20 seconds. When using long polling, you will not get an immediate response from the AWS API if no messages are available. If a new message arrives within 10 seconds, the HTTP response will be sent to you. After 20 seconds, you also get an empty response.

The following listing shows how this is done with the SDK.

Listing 14.5 worker.js: Receiving a message from the queue

```
const fs = require('fs');
const AWS = require('aws-sdk');
const puppeteer = require('puppeteer');
const config = require('./config.json');
const sqs = new AWS.SQS();
const s3 = new AWS.S3();

async function receive() {
  const result = await sqs.receiveMessage({
    QueueUrl: config.QueueUrl,
    MaxNumberOfMessages: 1,
    VisibilityTimeout: 120,
    WaitTimeSeconds: 10
  }).promise();

  if (result.Messages) {
    return result.Messages[0]
  } else {
    return null;
  }
};
```

Invokes the receiveMessage operation on SQS

Consumes no more than one message at once

Takes the message from the queue for 120 seconds

Long poll for 10 seconds to wait for new messages

Checks whether a message is available

Gets the one and only message

The receive step has now been implemented. The next step is to process the message. Thanks to the Node.js module `puppeteer`, it's easy to create a screenshot of a website, as demonstrated here.

Listing 14.6 worker.js: Processing a message (take screenshot and upload to S3)

```
async function process(message) {
  const body = JSON.parse(message.Body);
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
```

The message body is a JSON string. You convert it back into a JavaScript object.

Launches a headless browser

```
  await page.goto(body.url);
  page.setViewport({ width: 1024, height: 768})
  const screenshot = await page.screenshot();

  await s3.upload({ Uploads screenshot to S3
    Bucket: config.Bucket,
    Key: `${body.id}.png`,
    Body: screenshot,
    ContentType: 'image/png',
    ACL: 'public-read',
  }).promise();

  await browser.close();
};
```

Takes a screenshot

The S3 bucket to which to upload the image

The key, consisting of the random ID generated by the client and included in the SQS message

Allows anyone to read the image from S3 (public access)

Sets the content type to make sure browsers are showing the image correctly

The only step that's missing is to acknowledge that the message was successfully consumed, as shown in the next listing. This is done by deleting the message from the queue after successfully completing the task. If you receive a message from SQS, you get a `ReceiptHandle`, which is a unique ID that you need to specify when you delete a message from a queue.

Listing 14.7 worker.js: Acknowledging a message (deletes the message from the queue)

```
async function acknowledge(message) {
  await sqs.deleteMessage({
    QueueUrl: config.QueueUrl,
    ReceiptHandle: message.ReceiptHandle
  }).promise();
};
```

Invokes the deleteMessage operation on SQS

ReceiptHandle is unique for each receipt of a message.

You have all the parts; now it's time to connect them, as shown next.

Listing 14.8 worker.js: Connecting the parts

```
async function run() {
  while(true) {
    const message = await receive();
    if (message) {
      console.log('Processing message', message);
      await process(message);
      await acknowledge(message);
    }
    await new Promise(r => setTimeout(r, 1000));
  }
};

run();
```

An endless loop polling and processing messages

Receives a message

Processes the message

Acknowledges the message by deleting it from the queue

Sleeps for one second to decrease number of requests to SQS

Starts the loop

Now you can start the worker to process the message that is already in the queue. Run the script with `node worker.js`. You should see some output that says the worker is in the process step and then switches to `Done`. After a few seconds, the screenshot should be uploaded to S3. Your first asynchronous application is complete.

Remember the output you got when you invoked `node index.js "http://aws.amazon.com"` to send a message to the queue? It looked similar to this: http://url2png-$yourname.s3.amazonaws.com/XYZ.png. Now put that URL in your web browser, and you will find a screenshot of the AWS website (or whatever you used as an example).

You've created an application that is asynchronously decoupled. If the URL2PNG service becomes popular and millions of users start using it, the queue will become longer and longer because your worker can't produce that many PNGs from URLs. The cool thing is that you can add as many workers as you like to consume those messages. Instead of only one worker, you can start 10 or 100. The other advantage is that if a worker dies for some reason, the message that was in flight will become available for consumption after two minutes and will be picked up by another worker. That's fault tolerant! If you design your system to be asynchronously decoupled, it's easy to scale and create a good foundation to be fault tolerant. The next chapter will concentrate on this topic.

Cleaning up

Delete the message queue as follows:

```
$ aws sqs delete-queue --queue-url "$QueueUrl"
```

And don't forget to clean up and delete the S3 bucket used in the example. Issue the following command, replacing `$yourname` with your name:

```
$ aws s3 rb --force s3://url2png-$yourname
```

14.2.6 Limitations of messaging with SQS

Earlier in the chapter, we mentioned a few limitations of SQS. This section covers them in more detail. But before we start with the limitations, the benefits include these:

- You can put as many messages into SQS as you like. SQS scales the underlying infrastructure for you.
- SQS is highly available by default.
- You pay per message.

Those benefits come with some tradeoffs. Let's have a look at those limitations in more detail now.

SQS doesn't guarantee that a message is delivered only once

Two reasons a message might be delivered more than once follow:

- *Common reason*—If a received message isn't deleted within `VisibilityTimeout`, the message will be received again.

- *Rare reason*—A `DeleteMessage` operation doesn't delete all copies of a message because one of the servers in the SQS system isn't available at the time of deletion.

The problem of repeated delivery of a message can be solved by making the message processing idempotent. *Idempotent* means that no matter how often the message is processed, the result stays the same. In the URL2PNG example, this is true by design: if you process the message multiple times, the same image will be uploaded to S3 multiple times. If the image is already available on S3, it's replaced. Idempotence solves many problems in distributed systems that guarantee messages will be delivered at least once.

Not everything can be made idempotent. Sending an email is a good example: if you process a message multiple times and it sends an email each time, you'll annoy the addressee.

In many cases, processing at least once is a good tradeoff. Check your requirements before using SQS if this tradeoff fits your needs.

SQS doesn't guarantee the message order

Messages may be consumed in a different order than that in which you produced them. If you need a strict order, you should search for something else. If you need a stable message order, you'll have difficulty finding a solution that scales like SQS. Our advice is to change the design of your system so you no longer need the stable order, or put the messages in order on the client side.

SQS FIFO (first in, first out) queues

FIFO queues guarantee the order of messages and have a mechanism to detect duplicate messages. If you need a strict message order, they are worth a look. The disadvantages are higher pricing and a limitation of 3,000 operations per second. Check out the documentation at http://mng.bz/xM7Y for more information.

SQS doesn't replace a message broker

SQS isn't a message broker like ActiveMQ—SQS is only a message queue. Don't expect features like message routing or message priorities. Comparing SQS to ActiveMQ is like comparing DynamoDB to MySQL.

Amazon MQ

AWS announced an alternative to Amazon SQS in November 2017: Amazon MQ provides Apache ActiveMQ as a service. Therefore, you can use Amazon MQ as a message broker that speaks the JMS, NMS, AMQP, STOMP, MQTT, and WebSocket protocols.

Go to the Amazon MQ Developer Guide at http://mng.bz/AVP7 to learn more.

Summary

- Decoupling makes things easier because it reduces dependencies.
- Synchronous decoupling requires two sides to be available at the same time, but the sides don't have to know each other.
- With asynchronous decoupling, you can communicate without both sides being available.
- Most applications can be synchronously decoupled without touching the code, by using a load balancer offered by the ELB service.
- A load balancer can make periodic health checks to your application to determine whether the backend is ready to serve traffic.
- Asynchronous decoupling is possible only with asynchronous processes, but you can modify a synchronous process to be an asynchronous one most of the time.
- Asynchronous decoupling with SQS requires programming against SQS with one of the SDKs.

15 Automating deployment: CodeDeploy, CloudFormation, and Packer

About 20 years ago, we rented our first virtual machine. Our goal was to deploy WordPress, a content management system. To do so, we logged in to the machine using SSH, downloaded WordPress, installed the scripting language PHP and the web server Apache, edited the configuration files, and started the web server.

To this day, the following steps for deploying software—whether open source, proprietary, or homegrown—have remained the same:

1. Fetch source code or binaries
2. Install dependencies
3. Edit configuration files
4. Start services

These activities are also summarized under the term *configuration management*. The two main reasons for why automating deployments is a must-have in the cloud follow:

- To ensure high availability and scalability, you need to configure an Auto Scaling group to launch EC2 instances automatically. A new machine could spin up at any time, so deploying changes manually is not an option.
- Manual changes are error prone and expensive to reproduce. Automating increases reliability and reduces the costs per deployment.

From what we have learned from our consulting clients, organizations that implement automated deployments have a higher chance of success in the cloud.

In this chapter, you will learn how to automate the deployment of an application. We want to introduce three approaches with different pros and cons, so that you can pick the solution that best fits your situation. We use the following three options and will provide an overview that helps you make a decision at the end of the chapter:

- AWS CodeDeploy to deploy to running EC2 instances
- AWS CloudFormation, Auto Scaling groups, and user data to perform a rolling update
- Bundling an application into a customized AMI for immutable deployments with Packer by HashiCorp

Examples are 100% covered by the Free Tier

The examples in this chapter are completely covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything. Keep in mind that this only applies if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete each section within a few days; you'll clean up your account at the end of each section.

Chapter requirements

This chapter assumes that you have a basic understanding of the following components:

- Ensuring high availability by launching EC2 instances with an Auto Scaling group (chapter 13)
- Distributing requests with Elastic Load Balancing (chapter 14)

On top of that, the example included in this chapter makes intensive use of the following:

- Automating cloud infrastructure with CloudFormation (chapter 4)

Let's look at an example to see how this process might play out. Imagine you are the organizer of a local AWS meetup, and you want to provide a service allowing the community members to edit documents collaboratively. Therefore, you decided to deploy Etherpad, a web application, on EC2. The architecture, described next, is straightforward and illustrated in figure 15.1:

- Application Load Balancer forwards incoming requests.
- Auto Scaling launches and monitors exactly one virtual machine.
- The EC2 instance runs Etherpad.
- The RDS database instance stores the documents.

Unfortunately, Etherpad does not support clustering, which means it is not possible to run it on more than one machine in parallel.

You are sold on the idea of automating deployments and are looking for the right tool for the job. The first option might sound familiar to you. We are using a solution that applies changes with the help of agents installed on the virtual machines.

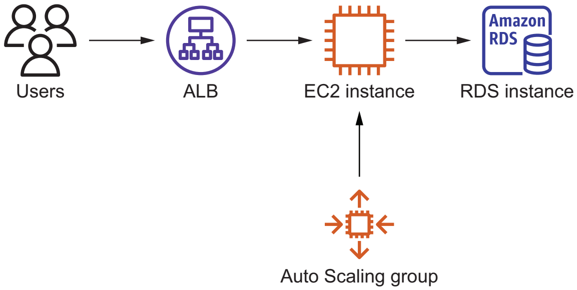


Figure 15.1 Etherpad infrastructure: ALB, EC2 instance, Auto Scaling group, and RDS database instance

15.1 In-place deployment with AWS CodeDeploy

You have two main reasons for performing in-place deployments. First, the speed of the deployment matters. Second, your application persists state on the virtual machine, so you try to avoid replacing running machines. So, for example, if you are running a database-like system that persists data on disk, the in-place deployment approach is a good fit.

The purpose of AWS CodeDeploy is to automate deployments on EC2, Fargate, and Lambda. Even on-premises machines are supported. The fully managed deployment service is free for EC2, Fargate, and Lambda and costs $0.02 per on-premises machine update. Figure 15.2 shows how CodeDeploy works for EC2 instances, as described here:

1. An engineer uploads a zip archive, including deployment instructions and the binaries or source code.
2. An engineer creates a deployment by choosing the revision and target instances.
3. An agent running on the EC2 instances pulls deployment tasks from CodeDeploy.
4. The agent downloads the zip archive from an S3 bucket.
5. The agent executes the instructions specified in the deployment artifact and copies the binaries or source code.
6. The agent sends a status update to CodeDeploy.

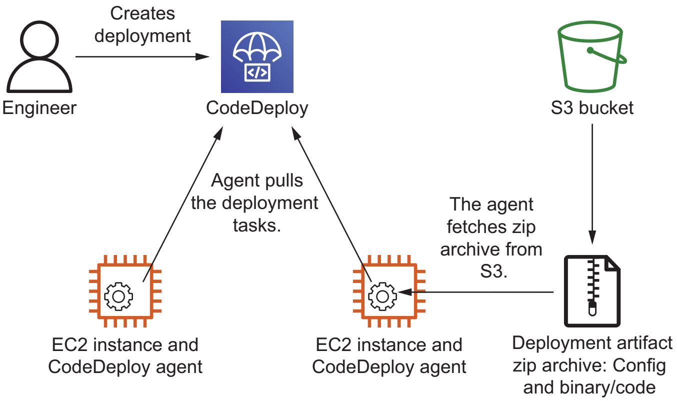

Creates deployment
Engineer
CodeDeploy
S3 bucket
Agent pulls the deployment tasks.
The agent fetches zip archive from S3.
EC2 instance and CodeDeploy agent
EC2 instance and CodeDeploy agent
Deployment artifact zip archive: Config and binary/code

Figure 15.2 CodeDeploy automates deployments to EC2 instances.

This approach is called *in-place deployment*. The EC2 instances keep running while the agent rolls out the change. We want to introduce the following important components of CodeDeploy before we proceed:

- *Application*—Specifies a name and the compute platform (EC2/on-premises, ECS, or Lambda)
- *Deployment group*—Points to the targets (an Auto Scaling group in our example)
- *Revision*—References the code to deploy on S3 or GitHub
- *Deployment*—Rolls out a specific revision to a target group

How do you deploy the web application Etherpad with CodeDeploy? Start with setting up the infrastructure consisting of an Application Load Balancer, an Auto Scaling group, and an RDS database instance. We prepared a CloudFormation template to do just this. 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 /chapter15/codedeploy.yaml.

Use the following command to create a CloudFormation stack based on the template. The command will exit after the stack has been created successfully:

```
aws cloudformation deploy --stack-name etherpad-codedeploy \
➥ --template-file chapter15/codedeploy.yaml --capabilities CAPABILITY_IAM
```

Besides creating an Application Load Balancer, an Auto Scaling group, and an RDS database instance, the CloudFormation template also creates the following resources required to deploy Etherpad with CodeDeploy:

- *S3 bucket*—Stores the deployment artifacts (zip files)
- *CodeDeploy application*—Manages deployments of Etherpad
- *CodeDeploy deployment group*—Points to the Auto Scaling group
- *IAM role*—Required by CodeDeploy
- *Parameter*—Stores the database endpoint in the Systems Manager Parameter Store

The next listing shows a deeper look into the resources specified in the CloudFormation template /chapter15/codedeploy.yaml.

Listing 15.1 The CodeDeploy application and deployment group

```
# [...]
ArtifactBucket:
  Type: 'AWS::S3::Bucket'
  Properties: {}
Application:
  Type: 'AWS::CodeDeploy::Application'
  Properties:
    ApplicationName: 'etherpad-codedeploy'
    ComputePlatform: 'Server'
DeploymentGroup:
```

The S3 bucket to store the deployment artifacts

The CodeDeploy application—a collection of deployment groups and revisions

The CodeDeploy deployment group specifies the targets for a deployment.

```
  Type: 'AWS::CodeDeploy::DeploymentGroup'
  Properties:
    ApplicationName: !Ref Application
    DeploymentGroupName: 'etherpad-codedeploy'
    AutoScalingGroups:
    - !Ref AutoScalingGroup
    DeploymentConfigName: 'CodeDeployDefault.AllAtOnce'
    LoadBalancerInfo:
      TargetGroupInfoList:
      - Name: !GetAtt LoadBalancerTargetGroup.TargetGroupName
    ServiceRoleArn: !GetAtt CodeDeployRole.Arn
CodeDeployRole:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Version: '2012-10-17'
      Statement:
      - Effect: Allow
        Principal:
          Service: 'codedeploy.amazonaws.com'
        Action: 'sts:AssumeRole'
    ManagedPolicyArns:
    - 'arn:aws:iam::aws:policy/service-role/AWSCodeDeployRole'
DatabaseHostParameter:
  Type: 'AWS::SSM::Parameter'
  Properties:
    Name: '/etherpad-codedeploy/database_host'
    Type: 'String'
    Value: !GetAtt 'Database.Endpoint.Address'
# [...]
```

The deployment group points to the Auto Scaling group.

Because only one EC2 instance is running, deploys to all instances at once

Considers the load balancer's target group when performing deployments

The IAM role to use for deployments

The IAM role grants access to autoscaling, load balancing, and other services that are relevant for a deployment.

Stores the database hostname in the Systems Manager's Parameter Store

By default, CodeDeploy comes with the deployment configurations defined in table 15.1.

Table 15.1 The predefined deployment configurations for CodeDeploy

Name	Description
`CodeDeployDefault.AllAtOnce`	Deploy to all targets at once.
`CodeDeployDefault.HalfAtATime`	Deploy to half of the targets at a time.
`CodeDeployDefault.OneAtATime`	Deploy to targets one by one.

On top of that, CodeDeploy allows you to define your own deployment configurations, such as deploying to 25% of the targets at a time.

After you have deployed and prepared the infrastructure for Etherpad and CodeDeploy, it is about time to create the first deployment. Before you do so, you need to gather some information. Use the following command to get the outputs from the CloudFormation stack named etherpad-codedeploy:

```
aws cloudformation describe-stacks --stack-name etherpad-codedeploy \
➥ --query "Stacks[0].Outputs"
```

In our case, the output looks like this:

```
[
  {
    "OutputKey": "ArtifactBucket",
    "OutputValue": "etherpad-codedeploy-artifactbucket-12vah1x44tpg7",
    "Description": "Name of the artifact bucket"
  },
  {
    "OutputKey": "URL",
    "OutputValue": "http://ether-LoadB-...us-east-1.elb.amazonaws.com",
    "Description": "The URL of the Etherpad application"
  }
]
```

Copies the name of the S3 bucket used to store the deployment artifacts

Opens the URL in your browser to access Etherpad

Note the `ArtifactBucket` output. You will need the name of the S3 bucket in the following step. Also, open the `URL` output in your browser. The ALB shows an error page, because you haven't deployed the application yet.

Creating a deployment artifact, also called a revision, is the next step. To do so, you need to create a zip file containing the source code, the scripts, and a configuration file. You'll find an example to deploy Etherpad in the book's code repository on GitHub: https://github.com/AWSinAction/code3 at /chapter15/etherpad-lite-1.8.17.zip. We encourage you to unzip the file to take a closer look at its contents.

The deployment instruction file needed for CodeDeploy is called an AppSpec file: appspec.yml. The following listing explains the AppSpec file that we prepared to deploy Etherpad.

Listing 15.2 The AppSpec file used to deploy Etherpad with the help of CodeDeploy

```
version: 0.0
os: linux
files:
- source: .
  destination: /etherpad
hooks:
  BeforeInstall:
  - location: hook_before_install.sh
    timeout: 60
  AfterInstall:
  - location: hook_after_install.sh
    timeout: 60
  ApplicationStart:
  - location: hook_application_start.sh
    timeout: 180
    runas: ec2-user
  ValidateService:
  - location: hook_validate_service.sh
    timeout: 300
    runas: ec2-user
```

Odd but true: the latest version of the App Spec file format is 0.0.

CodeDeploy supports Linux and Windows.

Copies all files from the archive to /etherpad

Hooks allow you to run scripts during the deployment process.

Triggered before CodeDeploy copies the source files

Triggered after CodeDeploy copies the source files

Triggered to start the application

Triggered to validate the service after starting the application

Now let's inspect the contents of the `hook_after_install.sh` script in the next listing.

Listing 15.3 Executing the script after the install step

```
#!/bin/bash -ex

TOKEN=`curl -X PUT "http://169.254.169.254/
➥ latest/api/token"
➥ -H "X-aws-ec2-metadata-token-ttl-seconds: 60"`
AZ=`curl -H "X-aws-ec2-metadata-token: $TOKEN" -v
➥ http://169.254.169.254/latest/meta-data/placement/availability-zone`
REGION=${AZ::-1}
DATABASE_HOST=$(aws ssm get-parameter --region ${REGION}
➥ --name "/etherpad-codedeploy/database_host"
➥ --query "Parameter.Value" --output text)

chown -R ec2-user:ec2-user /etherpad/
cd /etherpad/
rm -fR node_modules/
echo "
{
\"title\": \"Etherpad\",
\"dbType\": \"mysql\",
\"dbSettings\": {
\"host\": \"${DATABASE_HOST}\",
\"port\": \"3306\",
\"database\": \"etherpad\",
\"user\": \"etherpad\",
\"password\": \"etherpad\"
},
\"exposeVersion\": true
}
" > settings.json
```

Gets a token to access the EC2 metadata service

Fetches the availability zone of the EC2 instance from the metadata service

Removes the last character of the availability zone to get the region

Fetches the database host name from the Systems Manager Parameter Store

Makes sure all files belong to ec2-user instead of root

Cleans up the Node.js modules used by Etherpad

Generates the settings.json file for Etherpad containing the database host

Next, create a zip file containing the Etherpad source code as well as the appspec.yml file. As mentioned before, we have already prepared the zip file for you. All you have to do is upload it to S3 by executing the following command. Make sure to replace `$BucketName` with the artifact bucket name:

```
aws s3 cp chapter15/etherpad-lite-1.8.17.zip \
➥ s3://$BucketName/etherpad-lite-1.8.17.zip
```

Now you are ready to deploy Etherpad. Use the following command to create a deployment. Don't forget to replace `$BucketName` with the name of your artifact bucket:

```
aws deploy create-deployment --application-name etherpad-codedeploy \
➥ --deployment-group-name etherpad-codedeploy \
➥ --revision "revisionType=S3,
➥ s3Location={bucket=$BucketName,
➥ key=etherpad-lite-1.8.17.zip,bundleType=zip}"
```

Use the AWS Management Console or the following command to check whether the deployment succeeded. Replace `$DeploymentId` with the deployment ID printed to the console from the previous command:

```
$ aws deploy get-deployment --deployment-id $DeploymentId
```

Reload or open the URL of your Etherpad application and create a new pad. Click the settings icon to check the current version of Etherpad. The version is `c85ab49`, which is the latest Git commit ID for version 1.8.17.

Next, imagine you want to update to version 1.8.18 of Etherpad to roll out a fix for a security problem. The first step is to upload the revision. Don't forget to replace `$BucketName` with the name of your artifact bucket:

```
$ aws s3 cp chapter15/etherpad-lite-1.8.18.zip \
➥ s3://$BucketName/etherpad-lite-1.8.18.zip
```

Afterward, create another deployment to roll out version 1.8.18 with the following command:

```
$ aws deploy create-deployment --application-name etherpad-codedeploy \
➥ --deployment-group-name etherpad-codedeploy \
➥ --revision "revisionType=S3,
➥ s3Location={bucket=$BucketName,
➥ key=etherpad-lite-1.8.18.zip,bundleType=zip}"
```

Again, use the AWS Management Console or the following command to check progress, as shown next. Replace `$DeploymentId` with the deployment ID printed to the console from the previous command:

```
$ aws deploy get-deployment --deployment-id $DeploymentId
```

After the deployment is successful, reload the Etherpad web application. Check the version, which should be `4b96ff6` after the update. Congratulations—you have successfully deployed Etherpad with the help of CodeDeploy.

Cleaning up

Don't forget to clean up your AWS account before proceeding with the next step using the following code. Replace `$BucketName` with the name of the bucket used to store deployment artifacts:

```
$ aws s3 rm --recursive s3://${BucketName}
$ aws cloudformation delete-stack --stack-name etherpad-codedeploy
```

By the way, CodeDeploy will also make sure that the latest revision is deployed to any EC2 instances launched by the Auto Scaling group, such as if the health check failed and the Auto Scaling group replaced the failed EC2 instance.

Speed is an important advantage of in-place deployments with CodeDeploy. However, applying changes to long-running virtual machines, also called snowflake servers, is risky. A deployment might fail on a machine because of a change applied weeks before. It is difficult to reproduce the exact state of a machine.

That's why we prefer a different approach. To roll out a new revision, instead of modifying a running machine, spin up a new virtual machine. By doing so, each deployment starts from the same state, which increases reliability. You will learn how to implement a rolling update of EC2 instances with the help of CloudFormation in the following section.

Blue-green deployments with CodeDeploy

We have omitted that CodeDeploy supports blue-green deployments. With this method, a new machine is started instead of an in-place update. However, we prefer using CloudFormation for rolling updates because of its simplicity.

15.2 Rolling update with AWS CloudFormation and user data

CloudFormation is the Infrastructure as Code tool designed to manage AWS resources in an automated way. In addition, you can also use CloudFormation to orchestrate a rolling update of EC2 instances in an Auto Scaling group.

In contrast to an in-place update, a rolling update does not cause any downtime for the users. We use this approach whenever the virtual machines are disposable, meaning whenever our application does not persist any data on a local disk or in memory. For example, we use CloudFormation and user data to deploy WordPress, Jenkins, or a homegrown worker to crawl websites.

Back to our previous example: let's deploy Etherpad on EC2 with the help of CloudFormation. Figure 15.3 explains the process, which is laid out here:

1. The engineer initiates an update of the CloudFormation stack.
2. CloudFormation orchestrates a rolling update of the EC2 instance launched by the Auto Scaling group.
3. The Auto Scaling group launches a new EC2 instance based on the updated launch template, which includes a deployment script.
4. The EC2 instance fetches and executes a user data script.
5. The script fetches the source code from GitHub, creates a settings file, and starts the application.
6. The Auto Scaling group terminates the old EC2 instance.

To get started, deploy a CloudFormation stack based on the template we prepared to deploy Etherpad, as shown next. You'll find the CloudFormation template /chapter15/

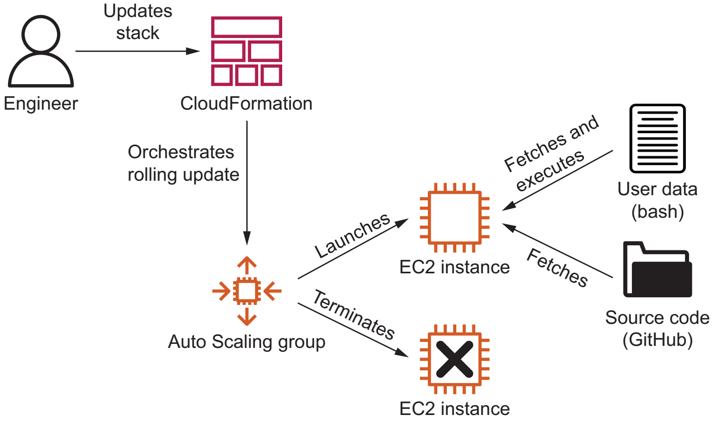


Figure 15.3 Rolling update orchestrated by CloudFormation with the help of a user data script

cloudformation.yaml in the book's code repository on GitHub https://github.com/AWSinAction/code3:

```
aws cloudformation deploy --stack-name etherpad-cloudformation \
➥ --template-file chapter15/cloudformation.yaml \
➥ --parameter-overrides EtherpadVersion=1.8.17 \
➥ --capabilities CAPABILITY_IAM
```

It will take about 10 minutes until the CloudFormation stack has been created and the command returns. Afterward, use the following command to get the URL of Etherpad:

```
aws cloudformation describe-stacks --stack-name etherpad-cloudformation \
➥ --query "Stacks[0].Outputs[0].OutputValue" --output text
```

Open the URL in your browser, and create a pad. Click the settings icon to check the current version of Etherpad as you did in the previous section. The version is `c85ab49`, which is the latest Git commit ID for version 1.8.17.

Great, but how did we deploy Etherpad to the EC2 instance? The following listing shows the CloudFormation template chapter15/cloudformation.yaml and answers this question.

Listing 15.4 Adding a bash script to the user data

```
# [...]
LaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'
  Properties:
    LaunchTemplateData:
      # [...]
      ImageId: !FindInMap [RegionMap,
➥ !Ref 'AWS::Region', AMI]
```

The Auto Scaling group uses the launch template as a blueprint when launching EC2 instances.

Picks the Amazon Linux 2 AMI dependent on the current region

```
      InstanceType: 't2.micro'
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1 --stack ${AWS::StackName}
➥ --resource AutoScalingGroup --region ${AWS::Region}' ERR

          # Install nodejs and git
          curl -fsSL https://rpm.nodesource.com/setup_14.x | bash -
          yum install -y nodejs git

          # Fetch, configure, and start Etherpad as non-root user
          su ec2-user -c '
          cd /home/ec2-user/
          git clone --depth 1 --branch ${EtherpadVersion}
➥ https://github.com/AWSinAction/etherpad-lite.git
          cd etherpad-lite/
          echo "
          {
            \"title\": \"Etherpad\",
            \"dbType\": \"mysql\",
            \"dbSettings\": {
              \"host\": \"${Database.Endpoint.Address}\",
              \"port\": \"3306\",
              \"database\": \"etherpad\",
              \"user\": \"etherpad\",
              \"password\": \"etherpad\"
            },
            \"exposeVersion\": true
          }
          " > settings.json
          ./src/bin/run.sh &'

          /opt/aws/bin/cfn-signal -e 0 --stack ${AWS::StackName}
➥ --resource AutoScalingGroup --region ${AWS::Region}
# [...]
```

Selects the instance type applicable for the Free Tier

Here is where the magic happens: the user data defined here is accessible by the EC2 instance during runtime.

The operating system will execute this bash script at the end of the boot process.

If any step fails, the script will abort and notify CloudFormation by calling cfn-signal.

Fetches Etherpad from the GitHub repository

Creates a settings file for Etherpad containing the database host name

Notifies CloudFormation about a successful deployment

Starts Etherpad

Let's repeat how this deployment works:

1. The launch template specifies a script to add to the user data when launching an EC2 instance.
2. The Auto Scaling group launches an EC2 instance based on the Amazon Linux 2 image.
3. At the end of the boot process, the EC2 instance executes the bash script fetched from user data.
4. The bash script fetches, configures, and starts Etherpad.

In summary, user data is a way to inject a script into the boot process of an EC2 instance—a simple but powerful concept. Note that by default the script is executed only during the first boot of the instance, so don't rely on the script to start services.

Debugging a user data script

In case you need to debug a user data script, use the Session Manager to connect to the EC2 instance. Have a look at the /var/log/cloud-init-output.log log file shown next, which contains the outputs of the user data script at the end:

```
$ less /var/log/cloud-init-output.log
```

So far, we have deployed Etherpad on EC2 by injecting a bash script into the boot process with the help of user data. But how do we perform a rolling update to update Etherpad from version 1.8.17 to 1.8.18? See the next listing for details.

Listing 15.5 Updating the Auto Scaling group or the referenced launch template

```
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    TargetGroupARNs:
    - !Ref LoadBalancerTargetGroup
    LaunchTemplate:
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
    MinSize: '1'
    MaxSize: '2'
    HealthCheckGracePeriod: 300
    HealthCheckType: ELB
    VPCZoneIdentifier:
    - !Ref SubnetA
    - !Ref SubnetB
    Tags:
    - PropagateAtLaunch: true
      Value: etherpad
      Key: Name
  CreationPolicy:
    ResourceSignal:
      Timeout: PT10M
  UpdatePolicy:
    AutoScalingRollingUpdate:
      PauseTime: PT10M
      WaitOnResourceSignals: true
      MinInstancesInService: 1
```

The resource defines the Auto Scaling group.

References the launch template we saw earlier

Etherpad does not support clustering; therefore, we are launching a single machine.

To enable zero-downtime deployments, we must launch a second machine during the deployment process.

The update policy specifies the behavior of CloudFormation in case of changes to the launch template.

That's where the magic happens: the configuration of the rolling update.

The Auto Scaling group awaits a success signal from launching the EC2 instance within 10 minutes (see cfn-signal in user data script).

The Auto Scaling group waits for a signal from the EC2 instance.

Makes sure the instance is up and running during the update to ensure zero downtime deployment

Let's make the update from version 1.8.17 to 1.8.18 happen. Execute the following command to update the CloudFormation stack. You might want to open the AWS Management Console and watch the running EC2 instances. CloudFormation will spin up a new EC2 instance. As soon as the new EC2 instance is ready, CloudFormation will terminate the old one:

```
$ aws cloudformation deploy --stack-name etherpad-cloudformation \
➥ --template-file cloudformation.yaml \
➥ --parameter-overrides EtherpadVersion=1.8.18 \
➥ --capabilities CAPABILITY_IAM
```

After updating the CloudFormation stack, reload the Etherpad web application. Check the version, which should be `4b96ff6`.

Congratulations! You have deployed a new version of Etherpad without any downtime for the users. Also, you learned how to use user data to bootstrap an EC2 instance in an automated way.

Cleaning up

Don't forget to clean up your AWS account before proceeding with the next step as follows:

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

One flaw with launching an EC2 instance from a base image like Amazon Linux 2 and using a user data script to deploy an application is reliability. Many things can and will go wrong when executing the deployment script. For example, GitHub could be down, so the EC2 instance cannot download Etherpad's source code. The same is true for other repositories, for example, the RPM repository used to install Node.js.

You will learn how to mitigate this risk in the next section by building AMIs, including everything that is needed to start Etherpad without any external dependencies. As an added benefit, this approach allows you to spin up an EC2 instance faster, because the boot process does not include deployment steps.

15.3 Deploying customized AMIs created by Packer

In this section, we present using customized Amazon Machine Images (AMIs), also called *immutable servers*. An immutable server starts from an image and is ready to go. To deploy a change, a new image is created, and the old server is replaced by a new one based on the new image.

Because a new image is needed for every change, it is advisable to automate the process of creating it. Usually, we stick to the tools AWS provides. However, we cannot recommend the EC2 Image Builder offered by AWS because it is complicated to use and doesn't seem to be designed to build images when you own the source code.

Instead, we recommend Packer, a tool provided by HashiCorp, which is very easy to use. Figure 15.4 illustrates how Packer works, as described here:

1. Launch an EC2 instance.
2. Connect to the EC2 instance via the Systems Manager.
3. Run the provisioner script.
4. Stop the EC2 instance.
5. Create an AMI.
6. Terminate the EC2 instance.

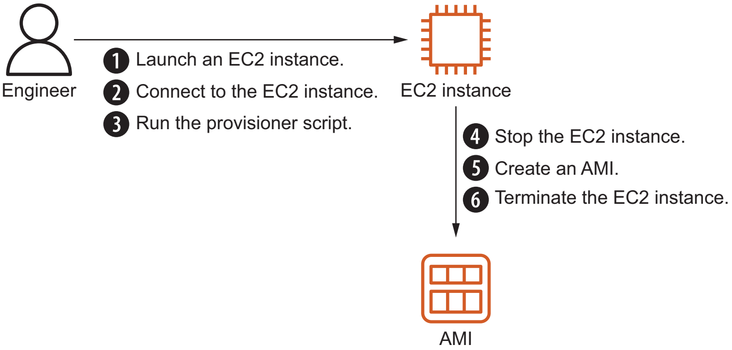

Engineer
1 Launch an EC2 instance.
2 Connect to the EC2 instance.
3 Run the provisioner script.
EC2 instance
4 Stop the EC2 instance.
5 Create an AMI.
6 Terminate the EC2 instance.
AMI

Figure 15.4 Packer automates the process of creating AMIs.

To use Packer, you need to define a template. You'll find the Packer template to build an AMI for Etherpad at chapter15/etherpad.pkr.hcl in the book's code repository on GitHub at https://github.com/AWSinAction/code3. The template starts with configuring Packer and the required plugins, as shown next.

Listing 15.6 The Packer template to build an Etherpad AMI, part 1

```
packer {
  required_plugins {
    amazon = {
      version = ">= 0.0.2"
      source  = "github.com/hashicorp/amazon"
    }
  }
}
```

Next, you need to define a source AMI as well as the details for the EC2 instance, which Packer will launch to build the AMI. By default, Packer uses SSH to connect to the EC2 instance. We are using the Systems Manager for increased security and usability instead.

IAM role ec2-ssm-core

Packer requires an IAM role named `ec2-ssm-core`. You created the role in the chapter 3 section, “Creating an IAM role.”

Listing 15.7 The Packer template to build an Etherpad AMI, part 2

```
source "amazon-ebs" "etherpad" {
  ami_name = "awsinaction-etherpad-{{timestamp}}"
  tags = {
    Name = "awsinaction-etherpad"
  }
  instance_type = "t2.micro"
  region        = "us-east-1"
  source_ami_filter {
    filters = {
      name = "amzn2-ami-hvm-2.0.*-x86_64-gp2"
      root-device-type    = "ebs"
      virtualization-type = "hvm"
    }
    most_recent = true
    owners      = ["137112412989"]
  }
  ssh_username         = "ec2-user"
  ssh_interface        = "session_manager"
  communicator         = "ssh"
  iam_instance_profile = "ec2-ssm-core"
  ami_groups = ["all"]
  ami_regions = ["us-east-1"]
}
```

Specifies the name for the new AMI adding {{timestamp}} to ensure uniqueness

The instance type for the temporary build instance

The region used to build the AMI

The source filter defines the base AMI from which to start.

Searches for Amazon Linux 2 images; the * represents all versions.

Picks the latest version of the Amazon Linux 2 images

Filters only AMIs owned by Amazon; the AWS account 137112412989 belongs to Amazon.

Tells Packer to use the Session Manager instead of plain SSH to connect with the temporary build instance

Attaches the IAM instance profile ec2-ssm-core, which is required for the Session Manager

Adds regions to distribute the AMI worldwide

The last part of the Packer template is shown in the next listing. In the build step, a shell provisioner is used to execute commands on the temporary build instance.

Listing 15.8 The Packer template to build an Etherpad AMI, part 3

```
build {
  name    = "awsinaction-etherpad"
  sources = [
    "source.amazon-ebs.etherpad"
  ]

  provisioner "shell" {
    inline = [
      "curl -fsSL https://rpm.nodesource.com/setup_14.x
➥ | sudo bash -",
```

References the source; see listing 15.7.

The shell provisioner executes a script on the temporary build instance.

Adds a YUM repository for Node.js

```
        "sudo yum install -y nodejs git",
        "sudo mkdir /opt/etherpad-lite",
        "sudo chown -R ec2-user:ec2-user /opt/etherpad-lite",
        "cd /opt",
        "git clone --depth 1 --branch 1.8.17 https://github.com/
➥ AWSinAction/etherpad-lite.git",
        "cd etherpad-lite",
        "./src/bin/installDeps.sh",
      ]
    }
}
```

Installs Node.js and Git

Fetches Etherpad from GitHub

Installs Etherpad dependencies

Before you proceed, make sure to install Packer on your local machine. The following commands install Packer on MacOS with brew:

```
$ brew tap hashicorp/tap
$ brew install hashicorp/tap/packer
```

You can also get the binaries from https://packer.io/downloads.html. Check out http://mng.bz/Zp8a for detailed instructions.

After you have installed Packer successfully, use the following command to initialize the tool:

```
$ packer init chapter15/
```

Next, build an AMI with Etherpad preinstalled like this:

```
$ packer build chapter15/etherpad.pkr.hcl
```

Watch Packer spinning up an EC2 instance, executing the provisioner shell script, stopping the EC2 instance, creating an AMI, and terminating the EC2 instance. At the end of the process, Packer will output the AMI ID, which is `ami-06beed8fa64e7cb68` in the following example:

```
==> Builds finished. The artifacts of successful builds are:
--> awsinaction-etherpad.amazon-ebs.etherpad: AMIs were created:
us-east-1: ami-06beed8fa64e7cb68
```

Note the AMI ID because you will need it to deploy Etherpad on AWS soon. As shown in figure 15.5, we are using a similar approach to roll out Etherpad as we did in the previous section. The Auto Scaling group orchestrates a rolling update. But instead of using a user data script to deploy Etherpad, the Auto Scaling group provisions EC2 instances based on the AMI with Etherpad preinstalled.

Use the following command to create the CloudFormation stack to roll out the Etherpad AMI. Replace `$AMI` with the AMI ID from Packer's output:

```
$ aws cloudformation deploy --stack-name etherpad-packer \
➥ --template-file packer.yaml
➥ --parameter-overrides AMI=$AMI \
➥ --capabilities CAPABILITY_IAM
```

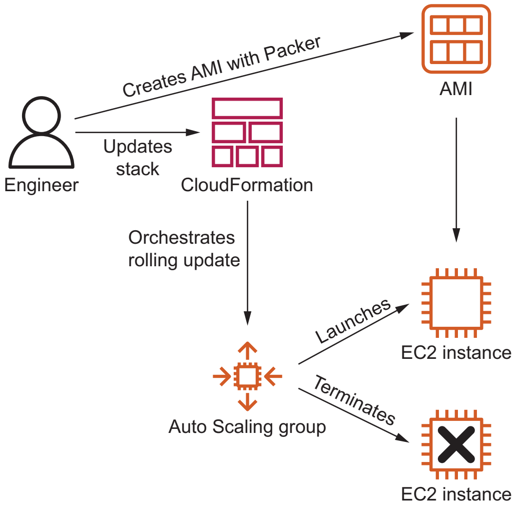


Figure 15.5 Rolling out an AMI with Etherpad preinstalled, built by Packer with the help of CloudFormation

It will take CloudFormation about 10 minutes to provision all the resources. After that, the command will return. Next, use the following command to get the URL to access the Etherpad application:

```
aws cloudformation describe-stacks --stack-name etherpad-packer \
➥ --query "Stacks[0].Outputs[0].OutputValue" --output text
```

Open the URL of your Etherpad application and create a new pad. Click the settings icon to check the current version of Etherpad as shown in the following listing. The version is c85ab49, which is the latest Git commit ID for version 1.8.17.

The CloudFormation template to deploy the Etherpad AMI built by Packer is very similar to that in the previous section, except a parameter allows you to hand over the ID of your Etherpad AMI. You'll find the Packer template to build an AMI for Etherpad at chapter15/packer.yaml in the book's code repository on GitHub at https://github.com/AWSinAction/code3.

Listing 15.9 Handing over the AMI ID to the CloudFormation template

```
# [...]
Parameters:
  AMI:                                         <- The parameter to set the AMI
    Type: 'AWS::EC2::Image::Id'
    Description: 'The AMI ID'
Resources:
  # [...]
  LaunchTemplate:
    Type: 'AWS::EC2::LaunchTemplate'
    Properties:
      # [...]
      LaunchTemplateData:
        ImageId: !Ref AMI
        UserData:
          'Fn::Base64': !Sub |
            #!/bin/bash -ex
            trap '/opt/aws/bin/cfn-signal -e 1 --stack ${AWS::StackName} \
➥ --resource AutoScalingGroup --region ${AWS::Region}' ERR
            cd /opt/etherpad-lite/
            echo "
            {
              \"title\": \"Etherpad\",
              \"dbType\": \"mysql\",
              \"dbSettings\": {
                \"host\": \"${Database.Endpoint.Address}\",
                \"port\": \"3306\",
                \"database\": \"etherpad\",
                \"user\": \"etherpad\",
                \"password\": \"etherpad\"
              },
              \"exposeVersion\": true
            }
            " > settings.json
            /opt/etherpad-lite/src/bin/fastRun.sh &
            /opt/aws/bin/cfn-signal -e 0 --stack ${AWS::StackName} \
➥ --resource AutoScalingGroup --region ${AWS::Region}
# [...]
```

The launch template references the AMI parameter.

Wait what? We are still using user data to inject a script into the boot process of the EC2 instance?

Yes, but only to create a settings file, which requires the database host name; this is not known when creating the AMI …

… and to start the Etherpad application.

There is only one important part missing: the Auto Scaling group. The following listing explains the details.

Listing 15.10 Orchestrating rolling updates with the Auto Scaling group

```
# [...]
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    TargetGroupARNs:
    - !Ref LoadBalancerTargetGroup
    LaunchTemplate:
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
    MinSize: 1
    MaxSize: 2
    HealthCheckGracePeriod: 300
    HealthCheckType: ELB
    VPCZoneIdentifier:
    - !Ref SubnetA
    - !Ref SubnetB
    Tags:
    - PropagateAtLaunch: true
      Value: etherpad
      Key: Name
  CreationPolicy:
    ResourceSignal:
      Timeout: PT10M
```

The Auto Scaling group ensures that an EC2 instance is running and is replaced in case of failure.

References the launch template explained in listing 15.9

Starts a single machine, because Etherpad cannot run on multiple machines in parallel

During a rolling update, the Auto Scaling group launches a new machine before terminating the old one; therefore, we need to set the maximum size to 2.

```
    UpdatePolicy:
      AutoScalingRollingUpdate:
        PauseTime: PT10M
        WaitOnResourceSignals: true
        MinInstancesInService: 1
# [...]
```

The update policy configures the rolling update.

The Auto Scaling group will wait 10 minutes for a new EC2 instance signal success with cfn-signal.

Enabling waiting for a signal from the EC2 instance during a rolling update

Indicates zero-downtime deployments by ensuring that at least one instance is running during a deployment

To deploy a new version, you need to build a new AMI with Packer and update the CloudFormation stack with the new AMI ID. Before you do, update the Packer template chapter15/etherpad.pkr.hcl and replace Etherpad version `1.8.17` with `1.8.18`, as shown next:

```
$ packer build chapter15/etherpad.pkr.hcl
$ aws cloudformation deploy --stack-name etherpad-packer \
➥ --template-file chapter15/packer.yaml \
➥ --parameter-overrides AMI=$AMI \
➥ --capabilities CAPABILITY_IAM
```

Hurray! You have deployed Etherpad on immutable servers launched from an image built by Packer. This approach is very reliable and allows you to deploy without any downtime. After updating the CloudFormation stack, reload the Etherpad web application. Check the version, which should be `4b96ff6`.

Cleaning up

Don't forget to clean up your AWS account before proceeding with the next step, as shown here:

```
$ aws cloudformation delete-stack --stack-name etherpad-packer
```

15.3.1 Tips and tricks for Packer and CloudFormation

Finally, we want to share the following tips and tricks for deploying an application of your choice with the help of Packer and CloudFormation:

- Launch an EC2 instance based on Amazon Linux 2 or the distribution you want to build on.
- Go through the steps necessary to get the application up and running manually.
- Transfer the manual steps into a shell script.
- Create a Packer template, and include the shell script.
- Run `packer build`, and launch an instance based on the AMI for testing.
- Roll out the AMI with CloudFormation and an Auto Scaling group.
- Use a user data script for dynamic configuration, as shown in our example.

In case you are looking for a way to ship your own source code, check out the file provisioner (see https://www.packer.io/docs/provisioners/file), which allows you to upload local files when building AMIs with Packer.

15.4 Comparing approaches

In this last section, we will compare the three different options to deploy applications on EC2 that you have learned about while reading through this chapter. We discussed the following three different methods to deploy applications on EC2 instances in this chapter:

- *AWS CodeDeploy*—Uses an agent running on the virtual machines to perform in-place deployments
- *AWS CloudFormation with user data*—Spins up new machines, which will execute a deployment script at the end of the boot process
- *Packer by HashiCorp*—Enables you to bundle the application into an AMI and launch immutable servers

All three options allow zero-downtime deployments, which is a game changer because it allows you to roll out changes without having to ask for a maintenance window in advance. The introduced tools also support deploying changes to a fleet of virtual machines gradually. But the approaches also have differences, as shown in table 15.2.

Table 15.2 Differences between CodeDeploy, CloudFormation and user data, and Packer

	AWS CodeDeploy	AWS CloudFormation and user data	Packer
Deployment speed	Fast	Slow	Medium
Agility	Medium	High	Low
Advantages	In-place deployments work for stateful machines as well.	Changing the user data script is a flexible way to deploy changes.	Machines are starting really fast, which is important if you want to scale based on demand. Also, you have low risk of failures during the boot process.
Disadvantages	Changes pile up on machines, which make it difficult to reproduce a deployment.	Potential failures during the boot process exist, for example, when third parties like GitHub are down.	Handling the life cycle of AMIs is tricky because you have to clean up unused and old AMIs to avoid storage costs.

We use all three options and choose the approach that best fits a particular scenario.

Summary

- AWS CodeDeploy is designed to automate deployments on EC2, Fargate, and Lambda. Even on-premises machines are supported.
- AWS CloudFormation is actually an Infrastructure as Code tool but comes with features to orchestrate rolling updates of EC2 instances as well.
- By configuring user data for an EC2 instance, you are able to inject a script that the machine will execute at the end of the boot process.
- Packer by HashiCorp is a tool to automate the process of creating Amazon Machine Images (AMIs).
- An immutable server is a server that you do not change after launch. Instead, to deploy changes you replace the old machine with a new one. This approach lowers the risk of side effects caused by former changes or third-party outages.

16 Designing for fault tolerance

This chapter covers

- What fault-tolerance is and why you need it
- Using redundancy to remove single points of failure
- Improving fault tolerance by retrying on failure
- Using idempotent operations to retry on failure
- AWS service guarantees

Failure is inevitable: hard disks, networks, power, and so on all fail from time to time. But failures do not have to affect the users of your system.

A fault-tolerant system provides the highest quality to your users. No matter what happens in your system, the user is never affected and can continue to go about their work, consume entertaining content, buy goods and services, or have conversations with friends. A few years ago, achieving fault tolerance was expensive and complicated, but with AWS, providing fault-tolerant systems is becoming an affordable standard. Nevertheless, building fault-tolerant systems is the top tier of cloud computing and might be challenging at the beginning.

Designing for fault tolerance means building for failure and building systems capable of resolving failure conditions automatically. An important aspect is avoiding single points of failures. You can achieve fault tolerance by introducing redundancy into your system. Instead of running your application on a single EC2 instance, you distribute the application among multiple machines. Also, decoupling the parts of your architecture such that one component does not rely on the uptime of the others is important. For example, the web server could deliver cached content if the database is not reachable.

The services provided by AWS offer different types of *failure resilience*. Resilience is the ability to deal with a failure with no or little effect on the user. You will learn about the resilience guarantees of major services in this chapter. But, in general, if you are unsure about the resilience capabilities of an AWS service, refer to the Resilience section of the official documentation for that service. A fault-tolerant system is very resilient to failure. We group AWS services into the following three categories:

- *No guarantees (single point of failure)*—No requests are served if failure occurs.
- *High availability*—In the case of failure, recovery can take some time. Requests might be interrupted.
- *Fault tolerant*—In the case of failure, requests are served as before without any availability problems.

The most convenient way to make your system fault tolerant is to build the architecture using only fault-tolerant services, which you will learn about in this chapter. If all your building blocks are fault tolerant, the whole system will be fault tolerant as well. Luckily, many AWS services are fault tolerant by default. If possible, use them. Otherwise, you'll need to deal with the consequences and handle failures yourself.

Unfortunately, one important service isn't fault tolerant by default: EC2 instances. Virtual machines aren't fault tolerant. This means an architecture that uses EC2 isn't fault tolerant by default. But AWS provides the building blocks to help you improve the fault tolerance of virtual machines. In this chapter, we will show you how to use Auto Scaling groups, Elastic Load Balancing (ELB), and Simple Queue Service (SQS) to turn EC2 instances into fault-tolerant systems.

First, however, let's look at the level of failure resistance of key services. Knowing which services are fault tolerant, which are highly available, and which are neither will help you create the kind of fault tolerance your system needs.

The following services provided by AWS are neither highly available nor fault tolerant. When using one of these services in your architecture, you are adding a *single point of failure* (SPOF) to your infrastructure. In this case, to achieve fault tolerance, you need to plan and build for failure as discussed during the rest of the chapter:

- *Amazon Elastic Compute Cloud (EC2) instance*—A single EC2 instance can fail for many reasons: hardware failure, network problems, availability zone (AZ) outage, and so on. To achieve high availability or fault tolerance, use Auto Scaling groups to set up a fleet of EC2 instances that serve requests in a redundant way.
- *Amazon Relational Database Service (RDS) single instance*—A single RDS instance could fail for the same reasons that an EC2 instance might fail. Use Multi-AZ mode to achieve high availability.

All the following services are *highly available* (HA) by default. When a failure occurs, the services will suffer from a short downtime but will recover automatically:

- *Elastic Network Interface (ENI)*—A network interface is bound to an AZ, so if this AZ goes down, your network interface will be unavailable as well. You can attach an ENI to another virtual machine in case of a smaller outage, however.
- *Amazon Virtual Private Cloud (VPC) subnet*—A VPC subnet is bound to an AZ, so if this AZ suffers from an outage, your subnet will not be reachable as well. Use multiple subnets in different AZs to remove the dependency on a single AZ.
- *Amazon Elastic Block Store (EBS) volume*—An EBS volume distributes data among multiple storage systems within an AZ. But if the whole AZ fails, your volume will be unavailable (you won't lose your data, though). You can create EBS snapshots from time to time so you can re-create an EBS volume in another AZ.
- *Amazon Relational Database Service (RDS) Multi-AZ instance*—When running in Multi-AZ mode, a short downtime (one minute) is expected if a problem occurs with the master instance while changing DNS records to switch to the standby instance.

The following services are *fault tolerant* by default. As a consumer of the service, you won't notice any failures:

- Elastic Load Balancing (ELB), deployed to at least two AZs
- Amazon EC2 security groups
- Amazon Virtual Private Cloud (VPC) with an ACL and a route table
- Elastic IP addresses (EIP)
- Amazon Simple Storage Service (S3)
- Amazon Elastic Block Store (EBS) snapshots
- Amazon DynamoDB
- Amazon CloudWatch
- Auto Scaling groups
- Amazon Simple Queue Service (SQS)
- AWS CloudFormation
- AWS Identity and Access Management (IAM, not bound to a single region; if you create an IAM user, that user is available in all regions)

Chapter requirements

To fully understand this chapter, you need to have read and understood the following concepts:

- EC2 (chapter 3)
- Autoscaling (chapter 13)
- Elastic Load Balancing (chapter 14)
- Simple Queue Service (chapter 14)

(continued)

On top of that, the example included in this chapter makes intensive use of the following:

- DynamoDB (chapter 12)
- Express, a Node.js web application framework

In this chapter, you'll learn everything you need to design a fault-tolerant web application based on EC2 instances (which aren't fault tolerant by default). During this chapter, you will build a fault-tolerant web application that allows a user to upload an image, apply a sepia filter to the image, and download the image. First, you will learn how to distribute a workload among multiple EC2 instances. Instead of running a single virtual machine, you will spin up multiple machines in different data centers, also known as availability zones. Next, you will learn how to increase the resilience of your code. Afterward, you will create an infrastructure consisting of a queue (SQS), a load balancer (ALB), EC2 instances managed by Auto Scaling groups, and a database (DynamoDB).

16.1 Using redundant EC2 instances to increase availability

Here are just a few reasons your virtual machine might fail:

- If the host hardware fails, it can no longer host the virtual machine on top of it.
- If the network connection to/from the host is interrupted, the virtual machine will lose the ability to communicate over the network.
- If the host system is disconnected from the power supply, the virtual machine will fail as well.

Additionally, the software running inside your virtual machine may also cause a crash for the following reasons:

- If your application contains a memory leak, the EC2 instance will run out of memory and fail. It may take a day, a month, a year, or more, but eventually it will happen.
- If your application writes to disk and never deletes its data, the EC2 instance will run out of disk space sooner or later, causing your application to fail.
- Your application may not handle edge cases properly and may instead crash unexpectedly.

Regardless of whether the host system or your application is the cause of a failure, a single EC2 instance is a single point of failure. If you rely on a single EC2 instance, your system will fail up eventually. It's merely a matter of time.

16.1.1 Redundancy can remove a single point of failure

Imagine a production line that makes fluffy cloud pies. Producing a fluffy cloud pie requires the following production steps (simplified!):

1. Produce a pie crust.
2. Cool the pie crust.
3. Put the fluffy cloud mass on top of the pie crust.
4. Cool the fluffy cloud pie.
5. Package the fluffy cloud pie.

The current setup is a single production line. The big problem with this process is that whenever one of the steps crashes, the entire production line must be stopped. Figure 16.1 illustrates what happens when the second step (cooling the pie crust) crashes. The steps that follow no longer work, because they no longer receive cool pie crusts.

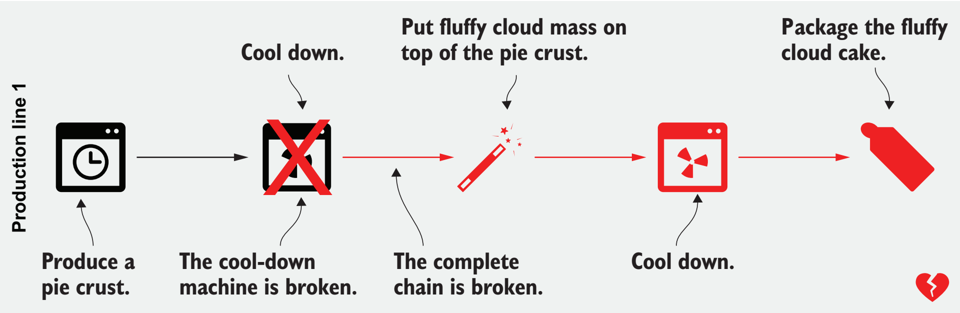

Production line 1
Cool down.
Put fluffy cloud mass on top of the pie crust.
Package the fluffy cloud cake.
Produce a pie crust.
The cool-down machine is broken.
The complete chain is broken.
Cool down.

Figure 16.1 A single point of failure affects not only itself but the entire system.

Why not have multiple production lines, each producing pies from pie crust through packaging? Instead of one line, suppose we have three. If one of the lines fails, the other two can still produce fluffy cloud pies for all the hungry customers in the world. Figure 16.2 shows the improvements; the only downside is that we need three times as many machines.

The example can be transferred to EC2 instances. Instead of having only one EC2 instance running your application, you can have three. If one of those instances fails, the other two will still be able to serve incoming requests. You can also minimize the cost of one versus three instances: instead of one large EC2 instance, you can choose three small ones. The problem that arises when using multiple virtual machines: how can the client communicate with the instances? The answer is *decoupling*: put a load balancer or message queue between your EC2 instances and the client. Read on to learn how this works.

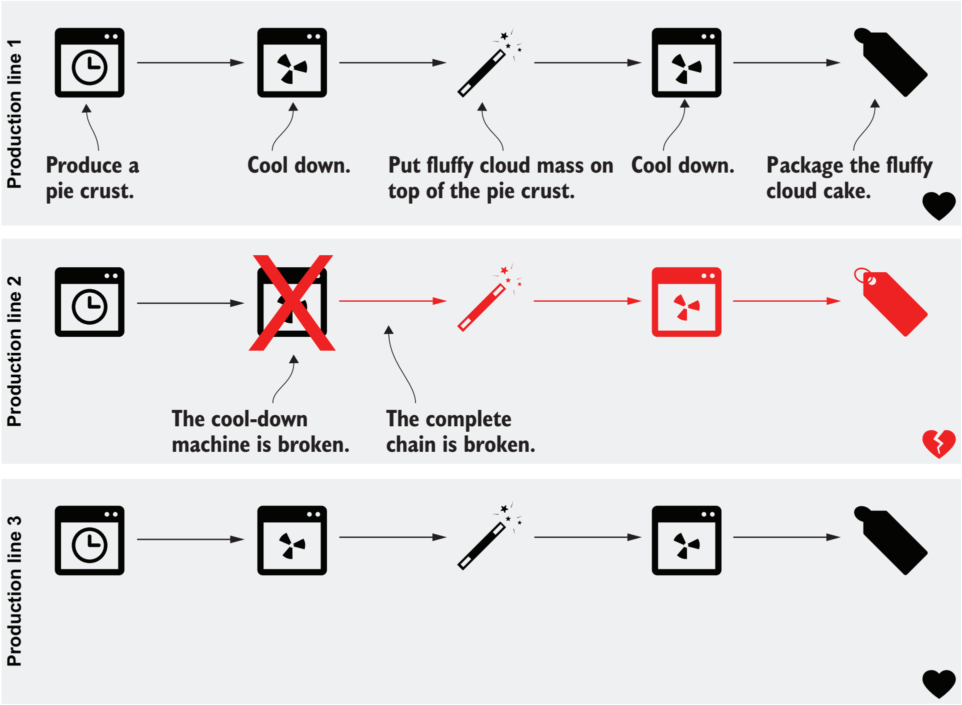

Production line 1
Produce a pie crust.
Cool down.
Put fluffy cloud mass on top of the pie crust.
Cool down.
Package the fluffy cloud cake.
Production line 2
The cool-down machine is broken.
The complete chain is broken.
Production line 3

Figure 16.2 Redundancy eliminates single points of failure and makes the system more stable.

16.1.2 Redundancy requires decoupling

In chapter 14, you learned how to use Elastic Load Balancing (ELB) and the Simple Queue Service (SQS) to decouple different parts of a system. You will apply both approaches to build a fault-tolerant system next.

First, figure 16.3 shows how EC2 instances can be made fault tolerant by using redundancy and synchronous decoupling. If one of the EC2 instances crashes, the load balancer stops routing requests to the crashed instances. Then, the Auto Scaling group replaces the crashed EC2 instance within minutes, and the load balancer begins to route requests to the new instance.

Take a second look at figure 16.3 and see what parts are redundant:

- *Availability zones (AZs)*—Two are used. If one AZ suffers from an outage, we still have instances running in the other AZ.
- *Subnets*—A subnet is tightly coupled to an AZ. Therefore, we need one subnet in each AZ.
- *EC2 instances*—Two subnets with one or more EC2 instances lead to redundancy among availability zones.
- *Load Balancer*—The load balancer spans multiple subnets and, therefore, multiple availability zones.

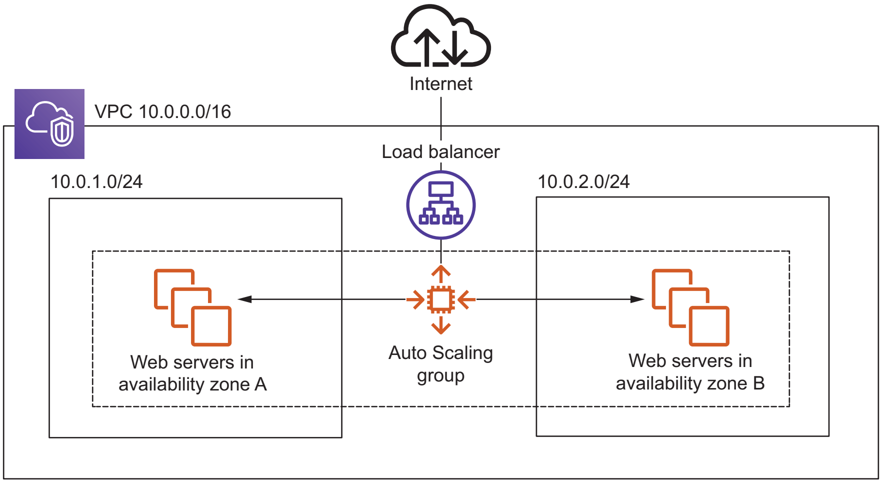


Figure 16.3 Fault-tolerant EC2 instances with an Auto Scaling group and an Elastic Load Balancer

Next, figure 16.4 shows a fault-tolerant system built with EC2 that uses the power of redundancy and asynchronous decoupling to process messages from an SQS queue.

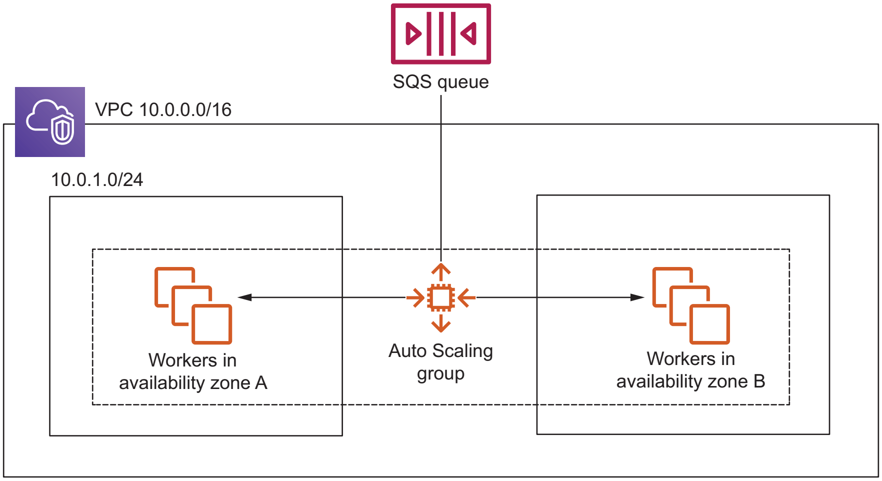

SQS queue
VPC 10.0.0.0/16
10.0.1.0/24
Workers in availability zone A
Auto Scaling group
Workers in availability zone B

Figure 16.4 Fault-tolerant EC2 instances with an Auto Scaling group and SQS

Second, in figures 16.3 and 16.4, the load balancer and the SQS queue appear only once. This doesn't mean that ELB or SQS are single points of failure; on the contrary, ELB and SQS are both fault tolerant by default.

You will learn how to use both models—synchronous decoupling with a load balancer and asynchronous decoupling with a queue—to build a fault-tolerant system in the following sections. But before we do so, let's have a look into some important considerations for making your code more resilient.

16.2 Considerations for making your code fault tolerant

If you want to achieve fault tolerance, you have to build your application accordingly. You can design fault tolerance into your application by following two suggestions:

- In the case of failure, let it crash, but also retry.
- Try to write idempotent code wherever possible.

16.2.1 Let it crash, but also retry

The Erlang programming language is famous for the concept of "let it crash." That means whenever the program doesn't know what to do, it crashes, and someone needs to deal with the crash. Most often people overlook the fact that Erlang is also famous for retrying. Letting it crash without retrying isn't useful—if you can't recover from a crash, your system will be down, which is the opposite of what you want.

You can apply the "let it crash" concept (some people call it "fail fast") to synchronous and asynchronous decoupled scenarios. In a synchronous decoupled scenario, the sender of a request must implement the retry logic. If no response is returned within a certain amount of time, or an error is returned, the sender retries by sending the same request again. In an asynchronous decoupled scenario, things are easier. If a message is consumed but not acknowledged within a certain amount of time, it goes back to the queue. The next consumer then grabs the message and processes it again. Retrying is built into asynchronous systems by default.

"Let it crash" isn't useful in all situations. If the program wants to respond to the sender but the request contains invalid content, this isn't a reason for letting the server crash: the result will stay the same no matter how often you retry. But if the server can't reach the database, it makes a lot of sense to retry. Within a few seconds, the database may be available again and able to successfully process the retried request.

Retrying isn't that easy. Imagine that you want to retry the creation of a blog post. With every retry, a new entry in the database is created, containing the same data as before. You end up with many duplicates in the database. Preventing this involves a powerful concept that's introduced next: idempotent retry.

16.2.2 Idempotent retry makes fault tolerance possible

How can you prevent a blog post from being added to the database multiple times because of a retry? A naive approach would be to use the title as the primary key. If the primary key is already used, you can assume that the post is already in the database and skip the step of inserting it into the database. Now the insertion of blog posts is idempotent, which means no matter how often a certain action is applied, the outcome must be the same. In the current example, the outcome is a database entry.

It continues with a more complicated example. Inserting a blog post is more complicated in reality, because the process might look something like this:

1. Create a blog post entry in the database.
2. Invalidate the cache because data has changed.
3. Post the link to the blog's Twitter feed.

Let's take a close look at each step.

1. Create a blog post entry in the database

We covered this step earlier by using the title as a primary key. But this time, we use a universally unique identifier (UUID) instead of the title as the primary key. A UUID like `550e8400-e29b-11d4-a716-446655440000` is a random ID that's generated by the client. Because of the nature of a UUID, it's unlikely that two identical UUIDs will be generated. If the client wants to create a blog post, it must send a request to the load balancer containing the UUID, title, and text. The load balancer routes the request to one of the backend servers. The backend server checks whether the primary key already exists. If not, a new record is added to the database. If it exists, the insertion continues. Figure 16.5 shows the flow.

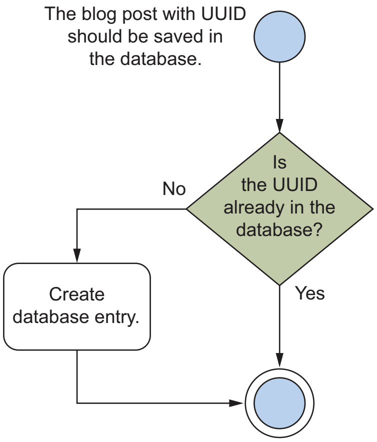

The blog post with UUID should be saved in the database.
Is the UUID already in the database?
No
Yes
Create database entry.

Figure 16.5 Idempotent database insert: Creating a blog post entry in the database only if it doesn't already exist

Creating a blog post is a good example of an idempotent operation that is guaranteed by code. You can also use your database to handle this problem. Just send an insert to your database. The next three things could happen:

- Your database inserts the data. The operation is successfully completed.
- Your database responds with an error because the primary key is already in use. The operation is successfully completed.
- Your database responds with a different error. The operation crashes.

Think twice about the best way to implement idempotence!

2. Invalidate the cache

This step sends an invalidation message to a caching layer. You don't need to worry about idempotence too much here: it doesn't hurt if the cache is invalidated more often than needed. If the cache is invalidated, then the next time a request hits the cache, the cache won't contain data, and the original source (in this case, the database) will be queried for the result. The result is then put in the cache for subsequent requests. If you invalidate the cache multiple times because of a retry, the worst thing that can happen is that you may need to make a few more calls to your database. That's easy.

3. Post to the blog's Twitter feed

To make this step idempotent, you need to use some tricks, because you interact with a third party that doesn't support idempotent operations. Unfortunately, no solution will guarantee that you post exactly one status update to Twitter. You can guarantee the creation of at least one (one or more than one) status update or at most one (one or none) status update. An easy approach could be to ask the Twitter API for the latest status updates; if one of them matches the status update that you want to post, you skip the step because it's already done.

But Twitter is an eventually consistent system: there is no guarantee that you'll see a status update immediately after you post it. Therefore, you can end up having your status update posted multiple times. Another approach would be to save whether you already posted the status update in a database. But imagine saving to the database that you posted to Twitter and then making the request to the Twitter API—but at that moment, the system crashes. Your database will state that the Twitter status update was posted, but in reality, it wasn't. You need to make a choice: tolerate a missing status update, or tolerate multiple status updates. Hint: it's a business decision. Figure 16.6 shows the flow of both solutions.

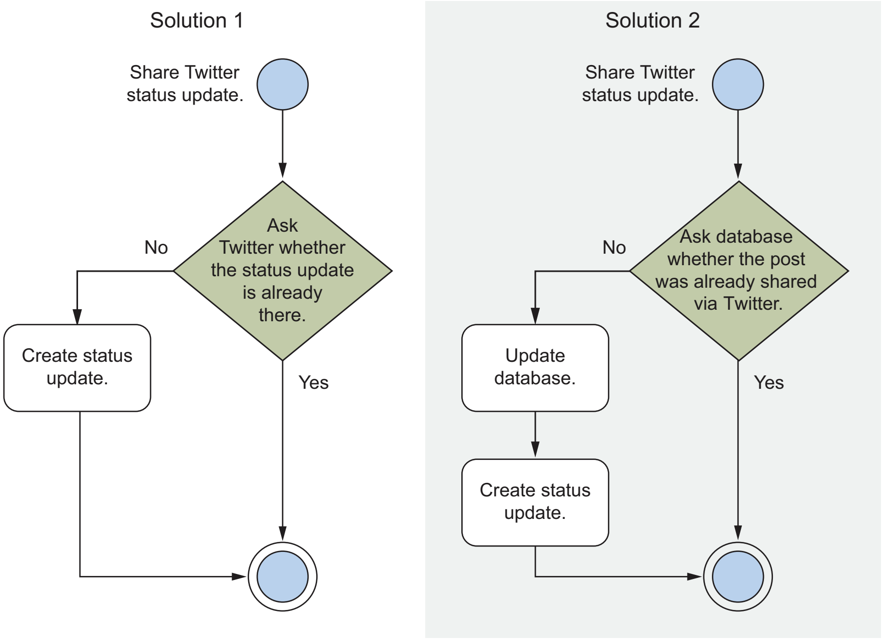

Solution 1
Share Twitter status update.
Ask Twitter whether the status update is already there.
No
Yes
Create status update.
Solution 2
Share Twitter status update.
Ask database whether the post was already shared via Twitter.
No
Yes
Update database.
Create status update.

Figure 16.6 Idempotent Twitter status update: Share a status update only if it hasn't already been done.

Now it's time for a practical example! You'll design, implement, and deploy a distributed, fault-tolerant web application on AWS. This example will demonstrate how distributed systems work and will combine most of the knowledge in this book.

16.3 Building a fault-tolerant web application: Imagery

Before you begin the architecture and design of the fault-tolerant Imagery application, we'll talk briefly about what the application should do. A user should be able to upload an image. This image is then transformed with a sepia filter so that it looks fancy. The user can then view the sepia image. Figure 16.7 shows the process.

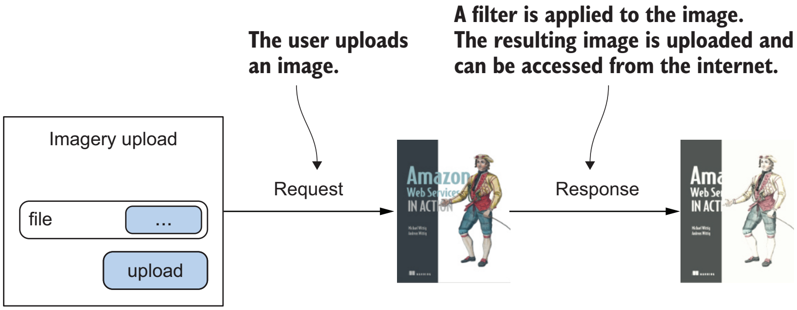

The user uploads an image.
A filter is applied to the image. The resulting image is uploaded and can be accessed from the internet.
Imagery upload
file
...
upload
Request
Response

Figure 16.7 The user uploads an image to Imagery, where a filter is applied.

The problem with the process shown in figure 16.7 is that it's synchronous. If the web server crashes during request and response, the user's image won't be processed. Another problem arises when many users want to use the Imagery app: the system becomes busy and may slow down or stop working. Therefore, the process should be turned into an asynchronous one. Chapter 14 introduced the idea of asynchronous decoupling by using an SQS message queue, as shown in figure 16.8.

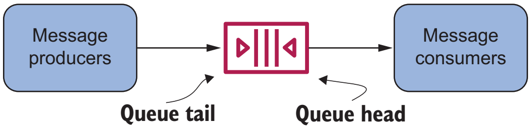


Figure 16.8 Producers send messages to a message queue while consumers read messages.

When designing an asynchronous process, it's important to keep track of the process. You need some kind of identifier for it. When a user wants to upload an image, the user creates a process first. This returns a unique ID. With that ID, the user can upload an image. If the image upload is finished, the worker begins to process the image in the background. The user can look up the process at any time with the process ID. While the image is being processed, the user can't see the sepia image, but as soon as the image is processed, the lookup process returns the sepia image. Figure 16.9 shows the asynchronous process.

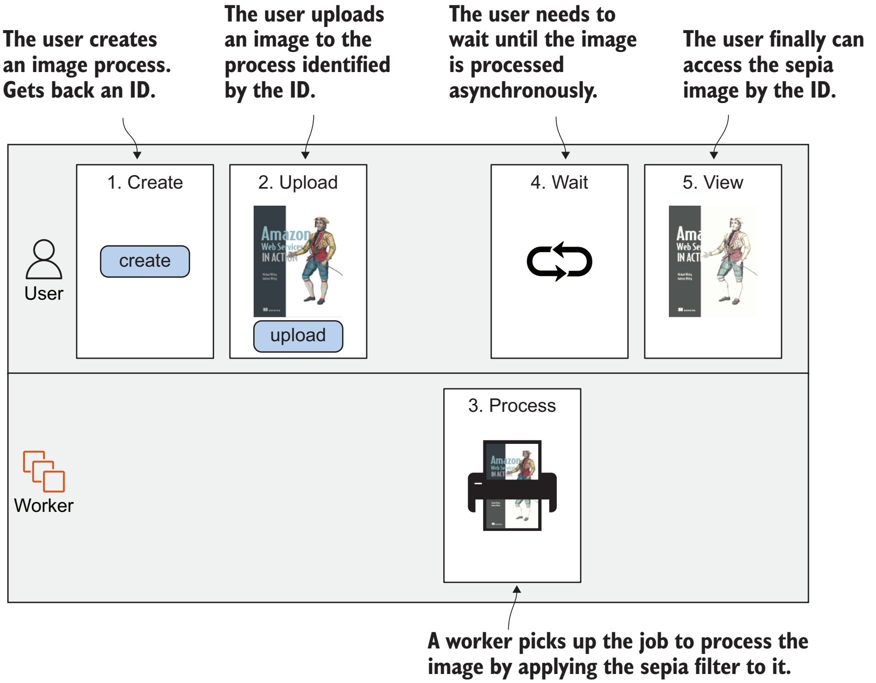

The user creates an image process. Gets back an ID.
The user uploads an image to the process identified by the ID.
The user needs to wait until the image is processed asynchronously.
The user finally can access the sepia image by the ID.
User
1. Create
create
2. Upload
upload
4. Wait
5. View
Worker
3. Process
A worker picks up the job to process the image by applying the sepia filter to it.

Figure 16.9 The user asynchronously uploads an image to Imagery, where a filter is applied.

Now that you have an asynchronous process, it's time to map that process to AWS services. Keep in mind that many services on AWS are fault tolerant by default, so it makes sense to pick them whenever possible. Figure 16.10 shows one way of doing it.

To make things as easy as possible, all the actions will be accessible via a REST API, which will be provided by EC2 instances. In the end, EC2 instances will provide the process and make calls to all the AWS services shown in figure 16.10.

You'll use many AWS services to implement the Imagery application. Most of them are fault tolerant by default, but EC2 isn't. You'll deal with that problem using an idempotent state machine, as introduced in the next section.

Example is 100% covered by the Free Tier

The example in this chapter is totally covered by the Free Tier. As long as you don't run the example 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 example within a few days, because you'll clean up your account at the end of the section.

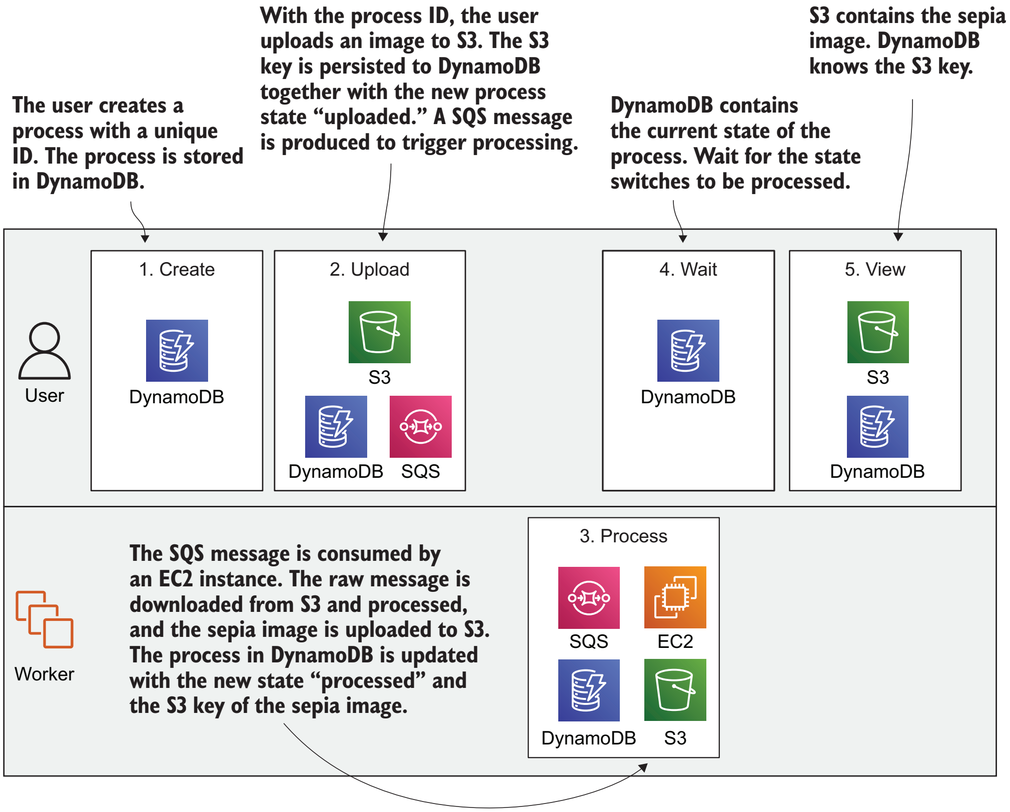

The user creates a process with a unique ID. The process is stored in DynamoDB.
With the process ID, the user uploads an image to S3. The S3 key is persisted to DynamoDB together with the new process state "uploaded." A SQS message is produced to trigger processing.
DynamoDB contains the current state of the process. Wait for the state switches to be processed.
S3 contains the sepia image. DynamoDB knows the S3 key.
User
1. Create
DynamoDB
2. Upload
S3
DynamoDB
SQS
4. Wait
DynamoDB
5. View
S3
DynamoDB
Worker
The SQS message is consumed by an EC2 instance. The raw message is downloaded from S3 and processed, and the sepia image is uploaded to S3. The process in DynamoDB is updated with the new state "processed" and the S3 key of the sepia image.
3. Process
SQS
EC2
DynamoDB
S3

Figure 16.10 Combining AWS services to implement the asynchronous Imagery process

16.3.1 The idempotent state machine

An idempotent state machine sounds complicated. We'll take some time to explain it because it's the heart of the Imagery application. Let's look at what a *state and machine* is and what idempotent means in this context.

The finite state machine

A finite state machine has at least one start state and one end state. Between the start and the end states, the state machine can have many other states. The machine also defines transitions between states. For example, a state machine with three states could look like this:

```
(A) -> (B) -> (C).
```

This means:

- State A is the start state.
- There is a transition possible from state A to B.

- There is a transition possible from state B to C.
- State C is the end state.

But there is no transition possible between (A) → (C) or (B) → (A). With this in mind, we apply the theory to our Imagery example. The Imagery state machine could look like this:

```
(Created) -> (Uploaded) -> (Processed)
```

Once a new process (state machine) is created, the only transition possible is to `Uploaded`. To make this transition happen, you need the S3 key of the uploaded raw image. The transition between `Created` → `Uploaded` can be defined by the function `uploaded(s3Key)`. Basically, the same is true for the transition `Uploaded` → `Processed`. This transition can be done with the S3 key of the sepia image: `processed(s3Key)`.

Don't be confused by the fact that the upload and the image filter processing don't appear in the state machine. These are the basic actions that happen, but we're interested only in the results; we don't track the progress of the actions. The process isn't aware that 10% of the data has been uploaded or 30% of the image processing is done. It cares only whether the actions are 100% done. You can probably imagine a bunch of other states that could be implemented, but we're skipping that for the purpose of simplicity in this example: resized and shared are just two examples.

IDEMPOTENT STATE TRANSITIONS

An idempotent state transition must have the same result no matter how often the transition takes place. If you can make sure that your state transitions are idempotent, you can do a simple trick: if you experience a failure during transitioning, you retry the entire state transition.

Let's look at the two state transitions you need to implement. The first transition `Created` → `Uploaded` can be implemented like this (pseudocode):

```
uploaded(s3Key) {
  process = DynamoDB.getItem(processId)
  if (process.state !== 'Created') {
    throw new Error('transition not allowed')
  }
  DynamoDB.updateItem(processId, {'state': 'Uploaded', 'rawS3Key': s3Key})
  SQS.sendMessage({'processId': processId, 'action': 'process'});
}
```

The problem with this implementation is that it's not idempotent. Imagine that `SQS.sendMessage` fails. The state transition will fail, so you retry. But the second call to `uploaded(s3Key)` will throw a "transition not allowed" error because `DynamoDB.updateItem` was successful during the first call.

To fix that, you need to change the `if` statement to make the function idempotent, like this (pseudocode):

```
uploaded(s3Key) {
  process = DynamoDB.getItem(processId)
  if (process.state !== 'Created' && process.state !== 'Uploaded') {
    throw new Error('transition not allowed')
  }
  DynamoDB.updateItem(processId, {'state': 'Uploaded', 'rawS3Key': s3Key})
  SQS.sendMessage({'processId': processId, 'action': 'process'});
}
```

If you retry now, you'll make multiple updates to Dynamo, which doesn't hurt. And you may send multiple SQS messages, which also doesn't hurt, because the SQS message consumer must be idempotent as well. The same applies to the transition `Uploaded` → `Processed`.

One little thing is still missing. So far, the code will fetch an item from DynamoDB and will update the item a few lines after that. In between, another process might have set the state to `Uploaded` already. Luckily, the database supports conditional updates, which allows us to reduce all the logic into a single DynamoDB request. DynamoDB will evaluate the condition before updating the item, as shown here (pseudocode):

```
uploaded(s3Key) {
  process = DynamoDB.getItem(processId)
  DynamoDB.updateItem(processId, {
    'state': 'Uploaded',
    'rawS3Key': s3Key,
    condition: 'NOT state IN(Created, Uploaded)'
  })
  SQS.sendMessage({'processId': processId, 'action': 'process'});
}
```

Next, you'll begin to implement the Imagery server.

16.3.2 Implementing a fault-tolerant web service

We'll split the Imagery application into two parts: the web servers and the workers. As illustrated in figure 16.11, the web servers provide the REST API to the user, and the workers process images.

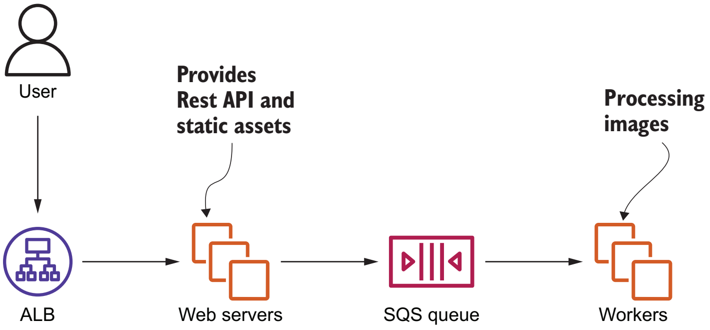


Figure 16.11 The Imagery application consists of two parts: the web servers and the workers.

Where is the code located?

As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code3. Imagery is located in /chapter16/.

The REST API will support the following routes:

- `POST /image`—A new image process is created when executing this route.
- `GET /image/:id`—This route returns the state of the process specified with the path parameter `:id`.
- `POST /image/:id/upload`—This route offers a file upload for the process specified with the path parameter `:id`.

To implement the web server, you'll again use Node.js and the Express web application framework. You'll use the Express framework, but don't feel intimidated because you won't need to understand it in depth to follow along.

SETTING UP THE WEB SERVER PROJECT

As always, you need some boilerplate code to load dependencies, initial AWS endpoints, and things like that. The next listing explains the code to do so.

Listing 16.1 Initializing the Imagery server (server/server.js)

```
const express = require('express');
const bodyParser = require('body-parser');
const AWS = require('aws-sdk');
const { v4: uuidv4 }  = require('uuid');
const multiparty = require('multiparty');

const db = new AWS.DynamoDB({});
const sqs = new AWS.SQS({});
const s3 = new AWS.S3({});

const app = express();
app.use(bodyParser.json());

// [...]

app.listen(process.env.PORT || 8080, function() {
  console.log('Server started. Open http://localhost:'
➥ + (process.env.PORT || 8080) + ' with browser.');
});
```

Loads the Node.js modules (dependencies)

Creates a DynamoDB endpoint

Creates an SQS endpoint

Creates an S3 endpoint

Creates an Express application

Tells Express to parse the request bodies

Starts Express on the port defined by the environment variable PORT, or defaults to 8080

Don't worry too much about the boilerplate code; the interesting parts will follow.

CREATING A NEW IMAGERY PROCESS

To provide a REST API to create image processes, a fleet of EC2 instances will run Node.js code behind a load balancer. The image processes will be stored in DynamoDB. Figure 16.12 shows the flow of a request to create a new image process.

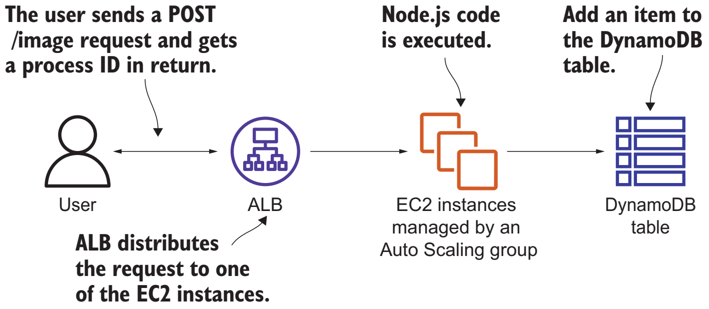

The user sends a POST /image request and gets a process ID in return.
Node.js code is executed.
Add an item to the DynamoDB table.
User
ALB
EC2 instances managed by an Auto Scaling group
DynamoDB table
ALB distributes the request to one of the EC2 instances.

Figure 16.12 Creating a new image process in Imagery

You'll now add a route to the Express application to handle `POST /image` requests, as shown in the following listing.

Listing 16.2 Creating an image process with `POST /image`

Uses the version for optimistic locking (explained in the following sidebar)

```
app.post('/image', function(request, response) {
   const id = uuidv4();
   db.putItem({
    'Item': {
      'id': {
        'S': id
      },
      'version': {
        'N': '0'
      },
      'created': {
        'N': Date.now().toString()
      },
      'state': {
        'S': 'created'
      }
    },
    'TableName': 'imagery-image',
    'ConditionExpression': 'attribute_not_exists(id)'
  }, function(err, data) {
    if (err) {
      throw err;
    } else {
      response.json({'id': id, 'state': 'created'});
    }
  });
});
```

Registers the route with Express

Creates a unique ID for the process

Invokes the putItem operation on DynamoDB

The id attribute will be the primary key in DynamoDB.

Stores the date and time when the process was created

The DynamoDB table will be created later in the chapter.

Prevents the item from being replaced if it already exists

Responds with the process ID

The process is now in the created state: this attribute will change when state transitions happen.

A new process can now be created.

Optimistic locking

To prevent multiple updates to an DynamoDB item, you can use a trick called *optimistic locking*. When you want to update an item, you must specify which version you want to update. If that version doesn't match the current version of the item in the database, your update will be rejected. Keep in mind that optimistic locking is your responsibility, not a default available in DynamoDB. DynamoDB only provides the features to implement optimistic locking.

Imagine the following scenario: an item is created in version 0. Process A looks up that item (version 0). Process B also looks up that item (version 0). Now process A wants to make a change by invoking the `updateItem` operation on DynamoDB. Therefore, process A specifies that the expected version is 0. DynamoDB will allow that modification, because the version matches; but DynamoDB will also change the item's version to 1 because an update was performed. Now process B wants to make a modification and sends a request to DynamoDB with the expected item version 0. DynamoDB will reject that modification because the expected version doesn't match the version DynamoDB knows of, which is 1.

To solve the problem for process B, you can use the same trick introduced earlier: retry. Process B will again look up the item, now in version 1, and can (you hope) make the change. There is one problem with optimistic locking, though: if many modifications happen in parallel, a lot of overhead results because of many retries. But this is a problem only if you expect a lot of concurrent writes to a single item, which you can solve by changing the data model. That's not the case in the Imagery application. Only a few writes are expected to happen for a single item: optimistic locking is a perfect fit to make sure you don't have two writes where one overrides changes made by another.

The opposite of optimistic locking is pessimistic locking. You can implement a pessimistic lock strategy by using a semaphore. Before you change data, you need to lock the semaphore. If the semaphore is already locked, you need to wait until the semaphore becomes free again.

The next route you need to implement is to look up the current state of a process.

LOOKING UP AN IMAGERY PROCESS

You'll now add a route to the Express application to handle `GET /image/:id` requests. Figure 16.13 shows the request flow.

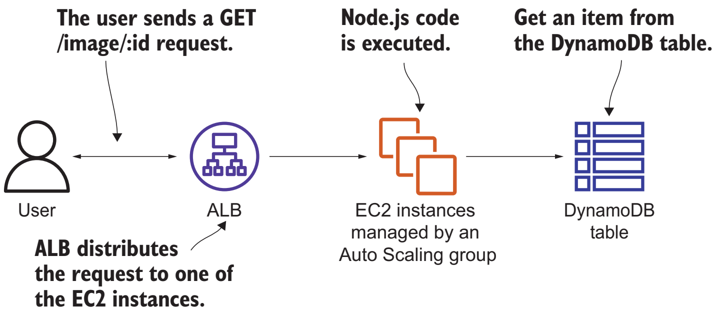

The user sends a GET /image/:id request.
Node.js code is executed.
Get an item from the DynamoDB table.
User
ALB
EC2 instances managed by an Auto Scaling group
DynamoDB table
ALB distributes the request to one of the EC2 instances.

Figure 16.13 Looking up an image process in Imagery to return its state

Express will take care of the path parameter `:id` by providing it within `request.params.id`. The implementation shown in the next listing needs to get an item from DynamoDB based on the path parameter ID.

Listing 16.3 `GET /image/:id` looks up an image process (server/server.js)

```
function mapImage = function(item) {
  return {
    'id': item.id.S,
    'version': parseInt(item.version.N, 10),
    'state': item.state.S,
    'rawS3Key': // [...]
    'processedS3Key': // [...]
    'processedImage': // [...]
  };
};

function getImage(id, cb) {
  db.getItem({
    'Key': {
      'id': {
        'S': id
      }
    },
    'TableName': 'imagery-image'
  }, function(err, data) {
    if (err) {
      cb(err);
    } else {
      if (data.Item) {
        cb(null, mapImage(data.Item));
      } else {
        cb(new Error('image not found'));
      }
    }
  });
}

app.get('/image/:id', function(request, response) {
  getImage(request.params.id, function(err, image) {
    if (err) {
      throw err;
    } else {
      response.json(image);
    }
  });
});
```

Helper function to map a DynamoDB result to a JavaScript object

Invokes the getItem operation on DynamoDB

id is the partition key.

Registers the route with Express

Responds with the image process

The only thing missing is the upload part, which comes next.

Uploading an image

Uploading an image via a POST request requires several steps:

1. Upload the raw image to S3.
2. Modify the item in DynamoDB.
3. Send an SQS message to trigger processing.

Figure 16.14 shows this flow.

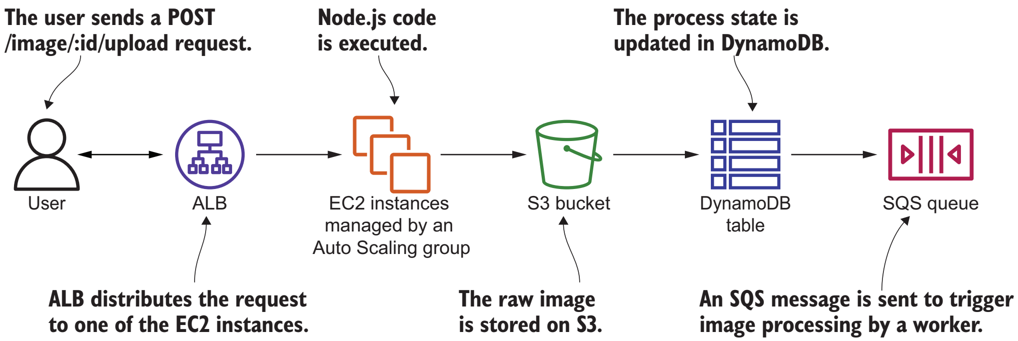

The user sends a POST /image/:id/upload request.
Node.js code is executed.
The process state is updated in DynamoDB.
User
ALB
EC2 instances managed by an Auto Scaling group
S3 bucket
DynamoDB table
SQS queue
ALB distributes the request to one of the EC2 instances.
The raw image is stored on S3.
An SQS message is sent to trigger image processing by a worker.

Figure 16.14 Uploading a raw image to Imagery and triggering image processing

The next code listing shows the implementation of these steps.

Listing 16.4 POST /image/:id/upload uploads an image (server/server.js)

```
function uploadImage(image, part, response) {
  const rawS3Key = 'upload/' + image.id + '-'
➥ + Date.now();
  s3.putObject({
    'Bucket': process.env.ImageBucket,
    'Key': rawS3Key,
    'Body': part,
    'ContentLength': part.byteCount
  }, function(err, data) {
    if (err) { /* [...] */ } else {
      db.updateItem({
        'Key': {'id': {'S': image.id}},
        'UpdateExpression': 'SET #s=:newState,
➥ version=:newVersion, rawS3Key=:rawS3Key',
        'ConditionExpression': 'attribute_exists(id)
➥ AND version=:oldVersion
➥ AND #s IN (:stateCreated, :stateUploaded)',
        'ExpressionAttributeNames': {'#s': 'state'},
        'ExpressionAttributeValues': {
          ':newState': {'S': 'uploaded'},
```

The S3 bucket name is passed in as an environment variable (the bucket will be created later in the chapter).

Creates a key for the S3 object

Calls the S3 API to upload an object

The body is the uploaded stream of data.

Calls the DynamoDB API to update an object

Updates the state, version, and raw S3 key

Updates only when the item exists. Version equals the expected version, and state is one of those allowed.

```
          ':oldVersion': {'N': image.version.toString()},
          ':newVersion': {'N': (image.version + 1).toString()},
          ':rawS3Key': {'S': rawS3Key},
          ':stateCreated': {'S': 'created'},
          ':stateUploaded': {'S': 'uploaded'}
        },
        'ReturnValues': 'ALL_NEW',
        'TableName': 'imagery-image'
      }, function(err, data) {
        if (err) { /* [...] */ } else {
          sqs.sendMessage({
            'MessageBody': JSON.stringify({
              'imageId': image.id, 'desiredState': 'processed'
            }),
            'QueueUrl': process.env.ImageQueue,
          }, function(err) {
            if (err) {
              throw err;
            } else {
              response.redirect('/#view=' + image.id);
              response.end();
            }
          });
        }
      });
    }
  });
}
app.post('/image/:id/upload', function(request,
➥ response) {
  getImage(request.params.id, function(err, image) {
    if (err) { /* [...] */ } else {
      const form = new multiparty.Form();
      form.on('part', function(part) {
        uploadImage(image, part, response);
      });
      form.parse(request);
    }
  });
});
```

Calls the SQS API to publish a message

Creates the message body containing the image's ID and the desired state

The queue URL is passed in as an environment variable.

Registers the route with Express

We are using the multiparty module to handle multipart uploads.

The server side is finished. Next, you'll continue to implement the processing part in the Imagery worker. After that, you can deploy the application.

16.3.3 Implementing a fault-tolerant worker to consume SQS messages

The Imagery worker processes images by applying a sepia filter asynchronously. The worker runs through the following steps in an endless loop. It is worth noting that multiple workers can run at the same time:

1. Poll the queue for new messages.
2. Fetch the process data from the database.
3. Download the image from S3.

4. Apply the sepia filter to the image.
5. Upload the modified image to S3.
6. Update the process state in the database.
7. Mark the message as done by deleting it from the queue.

Setting up the worker

To get started, you need some boilerplate code to load dependencies, initial AWS endpoints, and an endless loop to receive messages. The following listing explains the details.

Listing 16.5 Initializing the Imagery worker (worker/worker.js)

```
const AWS = require('aws-sdk');
const assert = require('assert-plus');
const Jimp = require('jimp');
const fs = require('fs/promises');

const db = new AWS.DynamoDB({});
const s3 = new AWS.S3({});
const sqs = new AWS.SQS({});

const states = {
  'processed': processed
};

async function processMessages() {
  let data = await sqs.receiveMessage({
    QueueUrl: process.env.ImageQueue,
    MaxNumberOfMessages: 1
  }).promise();
  if (data.Messages && data.Messages.length > 0) {
    var task = JSON.parse(data.Messages[0].Body);
    var receiptHandle = data.Messages[0].ReceiptHandle;
    assert.string(task.imageId, 'imageId');
    assert.string(task.desiredState, 'desiredState');
    let image = await getImage(task.imageId);
    if (typeof states[task.desiredState] === 'function') {
      await states[task.desiredState](image);
      await sqs.deleteMessage({
        QueueUrl: process.env.ImageQueue,
        ReceiptHandle: receiptHandle
      }).promise();
    } else {
      throw new Error('unsupported desiredState');
    }
  }
}

async function run() {
  while (true) {
    try {
    await processMessages();
    await new Promise(resolve => setTimeout(resolve,
 ➥ 10000));
```

Loads the Node.js modules (dependencies)

Configures the clients to interact with AWS services

This function reads messages from the queue, processes them, and finally deletes the message from the queue.

Reads one message from the queue; might return an empty result if there are no messages in the queue

Makes sure the message contains all the required properties

Gets the process data from the database

Triggers the state machine

If the message was processed successfully, deletes the message from the queue

A loop running endlessly

Sleeps for 10 seconds

```
    } catch (e) {
      console.log('ERROR', e);
    }
  }
}

run();
```

Catches all exceptions, ignores them, and tries again

The Node.js module `jimp` is used to create sepia images. You'll wire that up next.

Handling SQS messages and processing the image

The SQS message to trigger the image processing is handled by the worker. Once a message is received, the worker starts to download the raw image from S3, applies the sepia filter, and uploads the processed image back to S3. After that, the process state in DynamoDB is modified. Figure 16.15 shows the steps.

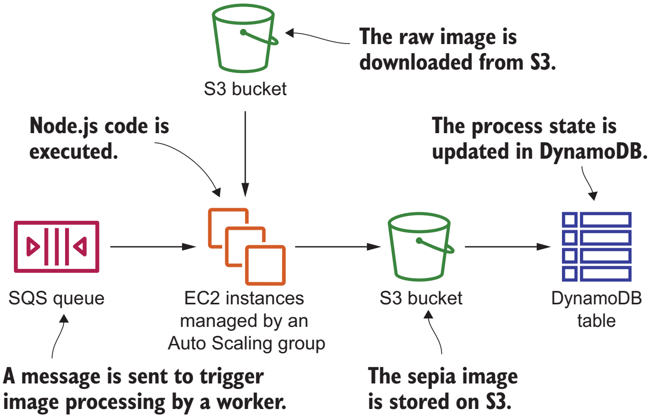

The raw image is downloaded from S3.
S3 bucket
Node.js code is executed.
The process state is updated in DynamoDB.
SQS queue
EC2 instances managed by an Auto Scaling group
S3 bucket
DynamoDB table
A message is sent to trigger image processing by a worker.
The sepia image is stored on S3.

Figure 16.15 Processing a raw image to upload a sepia image to S3

The code to process an image appears next.

Listing 16.6 Imagery worker: Handling SQS messages (worker/worker.js)

```
async function processImage(image) {
  let processedS3Key = 'processed/' + image.id + '-' + Date.now() + '.png';
  let rawFile = './tmp_raw_' + image.id;
  let processedFile = './tmp_processed_' + image.id;
  let data = await s3.getObject({
    'Bucket': process.env.ImageBucket,
    'Key': image.rawS3Key
  }).promise();
  await fs.writeFile(rawFile, data.Body,
➥ {'encoding': null});
  let lenna = await Jimp.read(rawFile);
```

Fetches the original image from S3

Writes the original image to a temporary folder on disk

Reads the file with the image manipulation library

```
  await lenna.sepia().write(processedFile);
  await fs.unlink(rawFile);
  let buf = await fs.readFile(processedFile,
➥ {'encoding': null});
  await s3.putObject({
    'Bucket': process.env.ImageBucket,
    'Key': processedS3Key,
    'ACL': 'public-read',
    'Body': buf,
    'ContentType': 'image/png'
  }).promise();
  await fs.unlink(processedFile);
  return processedS3Key;
}

async function processed(image) {
  let processedS3Key = await processImage(image);
  await db.updateItem({
    'Key': {
      'id': {
        'S': image.id
      }
    },
    'UpdateExpression':
      'SET #s=:newState, version=:newVersion,
➥ processedS3Key=:processedS3Key',
    'ConditionExpression':
      'attribute_exists(id) AND version=:oldVersion
➥ AND #s IN (:stateUploaded, :stateProcessed)',
    'ExpressionAttributeNames': {
      '#s': 'state'
    },
    'ExpressionAttributeValues': {
      ':newState': {'S': 'processed'},
      ':oldVersion': {'N': image.version.toString()},
      ':newVersion': {'N': (image.version + 1).toString()},
      ':processedS3Key': {'S': processedS3Key},
      ':stateUploaded': {'S': 'uploaded'},
      ':stateProcessed': {'S': 'processed'}
    },
    'ReturnValues': 'ALL_NEW',
    'TableName': 'imagery-image'
  }).promise();
}
```

Applies the sepia filter and writes the processed image to disk

Deletes the original image from the temporary folder

Reads the processed image

Uploads the processed image to S3

Deletes the processed file from the temporary folder

Updates the database item by calling the updateItem operation

Updates the state, version, and processed S3 key

Updates only when an item exists, the version equals the expected version, and the state is one of those allowed

The worker is ready to manipulate your images. The next step is to deploy all that code to AWS in a fault-tolerant way.

16.3.4 Deploying the application

As before, you'll use CloudFormation to deploy the application. The infrastructure consists of the following building blocks:

- An S3 bucket for raw and processed images
- A DynamoDB table, `imagery-image`

- An SQS queue and dead-letter queue
- An Application Load Balancer (ALB)
- Two Auto Scaling groups to manage EC2 instances running the server and worker
- IAM roles for the server and worker instances

It takes quite a while to create that CloudFormation stack; that's why you should do so now. After you've created the stack, we'll look at the template. After that, the application should be ready to use.

To help you deploy Imagery, we have created a CloudFormation template located at http://s3.amazonaws.com/awsinaction-code3/chapter16/template.yaml. Create a stack based on that template. The stack output `EndpointURL` returns the URL that you can access from your browser to use Imagery. Here's how to create the stack from the terminal:

```
$ aws cloudformation create-stack --stack-name imagery \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code3/chapter16/template.yaml \
➥ --capabilities CAPABILITY_IAM
```

Next, let's have a look what is going on behind the scenes.

Bundling runtime and application into a machine image (AMI)

In chapter 15, we introduced the concept of immutable machines. The idea is to create an Amazon Machine Image (AMI) containing the runtime, all required libraries, and the application's code or binary. The AMI is then used to launch EC2 instances with everything preinstalled. To deliver a new version of your application, you would create a new image, launch new instances, and terminate the old instances. We used Packer by HashiCorp to build AMIs. Check out chapter 15 if you want to recap the details. All we want to show here is the configuration file we used to prebuild and share AMIs containing the Imagery worker and server with you.

Listing 16.7 explains the configuration file we used to build the AMIs for you. Please note: you do not need to run Packer to build your own AMIs. We have done so already and shared the AMIs publicly.

Find the Packer configuration file at chapter16/imagery.pkr.hcl in our source code repository at https://github.com/AWSinAction/code3.

Listing 16.7 Configuring Packer to build an AMI containing the Imagery app

```
packer {
  required_plugins {
    amazon = {
      version = ">= 0.0.2"
      source  = "github.com/hashicorp/amazon"
    }
  }
}
```

Initializes and configures Packer

Adds the plug-in required to build AMIs

```
source "amazon-ebs" "imagery" {
  ami_name      = "awsinaction-imagery-{{timestamp}}"
  tags = {
    Name = "awsinaction-imagery"
  }
  instance_type = "t2.micro"
  region        = "us-east-1"
  source_ami_filter {
    filters = {
      name                = "amzn2-ami-hvm-2.0.*-x86_64-gp2"
      root-device-type    = "ebs"
      virtualization-type = "hvm"
    }
    most_recent = true
    owners      = ["137112412989"]
  }
  ssh_username = "ec2-user"
  ami_groups = ["all"]
  ami_regions = [
    "us-east-1",
    # [...]
  ]
}

build {
  name    = "awsinaction-imagery"
  sources = [
    "source.amazon-ebs.imagery"
  ]

  provisioner "file" {
    source = "./"
    destination = "/home/ec2-user/"
  }

  provisioner "shell" {
    inline = [
      "curl -sL https://rpm.nodesource.com/setup_14.x | sudo bash -",
      "sudo yum update",
      "sudo yum install -y nodejs cairo-devel
➥ libjpeg-turbo-devel",
      "cd server/ && npm install && cd -",
      "cd worker/ && npm install && cd -"
    ]
  }
}
```

Configures how Packer will create the AMI

The name for the AMI created by Packer

The tags for the AMI created by Packer

The region used by Packer to create the AMI

The instance type used by Packer when spinning up a virtual machine to build the AMI

The filter describes how to find the base AMI—the latest version of Amazon Linux 2—from which to start.

Allows anyone to access the AMI

The username required to connect to the build instance via SSH

Copies the AMI to all commercial regions

Configures the steps Packer executes while building the image

The name for the build

The sources for the build (references source from above)

Copies all files and folders from the current directory …

… to the home directory of the EC2 instance used to build the AMI

Executes a shell script on the EC2 instance used to build the AMI

Adds a repository for Node.js 14, the runtime for the Imagery server and worker

Installs Node.js and the libraries needed to manipulate images

Installs Node.js packages for the server and worker

Next, you will learn how to deploy the infrastructure with the help of CloudFormation.

Deploying S3, DynamoDB, and SQS

The next code listing describes the VPC, S3 bucket, DynamoDB table, and SQS queue.

Listing 16.8 Imagery CloudFormation template: S3, DynamoDB, and SQS

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 16'
Mappings:
  RegionMap:
    'us-east-1':
      AMI: 'ami-0ad3c79dfb359f1ba'
    # [...]
Resources:
  VPC:
    Type: 'AWS::EC2::VPC'
    Properties:
      CidrBlock: '172.31.0.0/16'
      EnableDnsHostnames: true
  # [...]
  Bucket:
    Type: 'AWS::S3::Bucket'
    Properties:
      BucketName: !Sub 'imagery-${AWS::AccountId}'
      WebsiteConfiguration:
        ErrorDocument: error.html
        IndexDocument: index.html
  Table:
    Type: 'AWS::DynamoDB::Table'
    Properties:
      AttributeDefinitions:
      - AttributeName: id
        AttributeType: S
      KeySchema:
      - AttributeName: id
        KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 1
        WriteCapacityUnits: 1
      TableName: 'imagery-image'
  SQSDLQueue:
    Type: 'AWS::SQS::Queue'
    Properties:
      QueueName: 'imagery-dlq'
  SQSQueue:
    Type: 'AWS::SQS::Queue'
    Properties:
      QueueName: imagery
      RedrivePolicy:
        deadLetterTargetArn: !Sub '${SQSDLQueue.Arn}'
        maxReceiveCount: 10
# [...]
Outputs:
  EndpointURL:
```

The map contains key-value pairs mapping regions to AMIs built by us including the Imagery server and worker.

The CloudFormation template contains a typical public VPC configuration.

A S3 bucket for uploaded and processed images, with web hosting enabled

The bucket name contains the account ID to make the name unique.

DynamoDB table containing the image processes

The id attribute is used as the partition key.

The SQS queue that receives messages that can't be processed

The SQS queue to trigger image processing

If a message is received more than 10 times, it's moved to the dead-letter queue.

Visit the output with your browser to use Imagery.

```
    Value: !Sub 'http://${LoadBalancer.DNSName}'
    Description: Load Balancer URL
```

The concept of a *dead-letter queue (DLQ)* needs a short introduction here as well. If a single SQS message can't be processed, the message becomes visible again on the queue after reaching its visibility timeout for other workers. This is called a *retry*. But if for some reason every retry fails (maybe you have a bug in your code), the message will reside in the queue forever and may waste a lot of resources because of all the retries. To avoid this, you can configure a dead-letter queue. If a message is retried more than a specific number of times, it's removed from the original queue and forwarded to the DLQ. The difference is that no worker listens for messages on the DLQ. You should create a CloudWatch alarm that triggers if the DLQ contains more than zero messages, because you need to investigate this problem manually by looking at the message in the DLQ. Once the bug is fixed, you can move the messages from the dead letter queue back to the original queue to process them again.

Now that the basic resources have been designed, let's move on to the more specific resources.

IAM roles for server and worker EC2 instances

Remember that it's important to grant only the privileges that are necessary. All server instances must be able to do the following:

- `sqs:SendMessage` to the SQS queue created in the template to trigger image processing
- `s3:PutObject` to the S3 bucket created in the template to upload a file to S3 (You can further limit writes to the `upload/` key prefix.)
- `dynamodb:GetItem`, `dynamodb:PutItem`, and `dynamodb:UpdateItem` to the DynamoDB table created in the template

All worker instances must be able to do the following:

- `sqs:DeleteMessage`, and `sqs:ReceiveMessage` to the SQS queue created in the template
- `s3:PutObject` to the S3 bucket created in the template to upload a file to S3 (You can further limit writes to the `processed/` key prefix.)
- `dynamodb:GetItem` and `dynamodb:UpdateItem` to the DynamoDB table created in the template

Both servers and workers need to grant access for the AWS Systems Manager to enable access via Session Manager as follows:

- `ssmmessages:*`
- `ssm:UpdateInstanceInformation`
- `ec2messages:*`

If you don't feel comfortable with IAM roles, take a look at the book's code repository on GitHub at https://github.com/AWSinAction/code3. The template with IAM roles can be found in /chapter16/template.yaml.

Now it's time to deploy the server.

Deploying the server with a load balancer and an Auto Scaling group

The Imagery server allows the user to upload images, monitor the processing, and show the results. An Application Load Balancer (ALB) acts as the entry point into the system. Behind the load balancer, a fleet of servers running on EC2 instances answers incoming HTTP requests. An Auto Scaling group ensures EC2 instances are up and running and replaces instances that fail the load balancer's health check.

The following listing shows how to create the load balancer with the help of CloudFormation.

Listing 16.9 CloudFormation template: Load balancer for the Imagery server

```
LoadBalancer:
  Type: 'AWS::ElasticLoadBalancingV2::LoadBalancer'
  Properties:
    Subnets:
    - Ref: SubnetA
    - Ref: SubnetB
    SecurityGroups:
    - !Ref LoadBalancerSecurityGroup
    Scheme: 'internet-facing'
  DependsOn: VPCGatewayAttachment
LoadBalancerListener:
  Type: 'AWS::ElasticLoadBalancingV2::Listener'
  Properties:
    DefaultActions:
    - Type: forward
      TargetGroupArn: !Ref LoadBalancerTargetGroup
    LoadBalancerArn: !Ref LoadBalancer
    Port: 80
    Protocol: HTTP
LoadBalancerTargetGroup:
  Type: 'AWS::ElasticLoadBalancingV2::TargetGroup'
  Properties:
    HealthCheckIntervalSeconds: 5
    HealthCheckPath: '/'
    HealthCheckPort: 8080
    HealthCheckProtocol: HTTP
    HealthCheckTimeoutSeconds: 3
    HealthyThresholdCount: 2
    UnhealthyThresholdCount: 2
    Matcher:
      HttpCode: '200,302'
    Port: 8080
    Protocol: HTTP
    VpcId: !Ref VPC
LoadBalancerSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'awsinaction-elb-sg'
    VpcId: !Ref VPC
    SecurityGroupIngress:
```

The load balancer distributes incoming requests among a fleet of virtual machines.

Configures a listener for the load balancer

The HTTP listener forwards all requests to the default target group defined below.

The listener will listen for HTTP requests on port 80/TCP.

The default target group

The target group will check the health of registered EC2 instances by sending HTTP requests on port 8080/TCP.

By default, the target group will forward requests to port 8080/TCP of registered virtual machines.

A security group for the load balancer

```
    - CidrIp: '0.0.0.0/0'
      FromPort: 80
      IpProtocol: tcp
      ToPort: 80
```

Allows incoming traffic on port 80/TCP from anywhere

Next, creating an Auto Scaling group to launch EC2 instances and registering them at the load balancer is illustrated in the following listing.

Listing 16.10 CloudFormation template: Auto Scaling group for the Imagery server

```
ServerSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'imagery-worker'
    VpcId: !Ref VPC
    SecurityGroupIngress:
    - FromPort: 8080
      IpProtocol: tcp
      SourceSecurityGroupId: !Ref LoadBalancerSecurityGroup
      ToPort: 8080
ServerLaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'
  Properties:
    LaunchTemplateData:
      IamInstanceProfile:
        Name: !Ref ServerInstanceProfile
      ImageId: !FindInMap [RegionMap, !Ref
➥ 'AWS::Region', AMI]
      Monitoring:
        Enabled: false
      InstanceType: 't2.micro'
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeviceIndex: 0
        Groups:
        - !Ref ServerSecurityGroup
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1
➥ --region ${AWS::Region} --stack ${AWS::StackName}
➥ --resource ServerAutoScalingGroup' ERR
          cd /home/ec2-user/server/
          sudo -u ec2-user ImageQueue=${SQSQueue} ImageBucket=${Bucket}
➥ nohup node server.js > server.log &
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
➥ --resource ServerAutoScalingGroup --region ${AWS::Region}
ServerAutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    LaunchTemplate:
```

A security group for the EC2 instances running the server

Allows incoming traffic on port 8080/TCP but only from the load balancer

The launch template used as a blueprint for spinning up EC2 instances

Looks up the AMI with the Imagery server preinstalled from the region map (see listing 16.9)

Launches virtual machines of type t2.micro to run examples under the Free Tier

Configures a network interface (ENI) with a public IP address and the security group of the server

Each virtual machine will execute this script at the end of the boot process. The script starts the Node.js server.

Creates an Auto Scaling group that manages the virtual machines running the Imagery server

References the launch template

```
      LaunchTemplateId: !Ref ServerLaunchTemplate
      Version: !GetAtt 'ServerLaunchTemplate.LatestVersionNumber'
    MinSize: 1
    MaxSize: 2
    DesiredCapacity: 1
    TargetGroupARNs:
    - !Ref LoadBalancerTargetGroup
    HealthCheckGracePeriod: 120
    HealthCheckType: ELB
    VPCZoneIdentifier:
    - !Ref SubnetA
    - !Ref SubnetB
    # [...]
  DependsOn: VPCGatewayAttachment
  # [...]
```

The Auto Scaling group will spin up at least one and no more than two EC2 instances.

The Auto Scaling group will register and deregister virtual machines at the target group.

The Auto Scaling group will replace EC2 instances that fail the target group's health check.

Spins up EC2 instances distributed among two subnets and, therefore, two AZs

That's it for the server. Next, you need to deploy the worker.

Deploying the worker with an Auto Scaling group

Deploying the worker works similar to the process for the server. Instead of a load balancer, however, the queue is used for decoupling. Please note that we already explained how to create a SQS in listing 16.8. Therefore, all that's left is the Auto Scaling group and a launch template. The next listing shows the details.

Listing 16.11 Load balancer and Auto Scaling group for the Imagery worker

```
WorkerLaunchTemplate:
  Type: 'AWS::EC2::LaunchTemplate'
  Properties:
    LaunchTemplateData:
      IamInstanceProfile:
        Name: !Ref WorkerInstanceProfile
      ImageId: !FindInMap [RegionMap, !Ref
➥ 'AWS::Region', AMI]
      Monitoring:
        Enabled: false
      InstanceType: 't2.micro'
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeviceIndex: 0
        Groups:
        - !Ref WorkerSecurityGroup
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -ex
          trap '/opt/aws/bin/cfn-signal -e 1 --region ${AWS::Region}
➥ --stack ${AWS::StackName} --resource WorkerAutoScalingGroup' ERR
          cd /home/ec2-user/worker/
          sudo -u ec2-user ImageQueue=${SQSQueue} ImageBucket=${Bucket}
➥ nohup node worker.js > worker.log &
```

The launch template used as a blueprint for spinning up EC2 instances

Attaches an IAM role to the EC2 instances to allow the worker to access SQS, S3, and DynamoDB

Looks up the AMI with the Imagery worker preinstalled from the region map (see listing 16.10)

Disables detailed monitoring of EC2 instances to avoid costs

Launches virtual machines of type t2.micro to run examples under the Free Tier

Configures a network interface (ENI) with a public IP address and the security group of the worker

Each virtual machine will execute this script at the end of the boot process. The script starts the Node.js worker.

```
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
➥ --resource WorkerAutoScalingGroup --region ${AWS::Region}
WorkerAutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    LaunchTemplate:
      LaunchTemplateId: !Ref WorkerLaunchTemplate
      Version: !GetAtt 'WorkerLaunchTemplate.LatestVersionNumber'
    MinSize: 1
    MaxSize: 2
    DesiredCapacity: 1
    HealthCheckGracePeriod: 120
    HealthCheckType: EC2
    VPCZoneIdentifier:
    - !Ref SubnetA
    - !Ref SubnetB
    Tags:
    - PropagateAtLaunch: true
      Value: 'imagery-worker'
      Key: Name
  DependsOn: VPCGatewayAttachment
  # [...]
```

Creates an Auto Scaling group that manages the virtual machines running the Imagery worker

References the launch template

The Auto Scaling group will spin up at least one and no more than two EC2 instances.

The Auto Scaling group will replace failed EC2 instances.

Spins up EC2 instances distributed among two subnets: AZs

Adds a Name tag to each instance, which will show up at the Management Console, for example

After all that YAML reading, the CloudFormation stack should be created. Verify the status of your stack like this:

```
$ aws cloudformation describe-stacks --stack-name imagery
{
  "Stacks": [{
    [...]
    "Description": "AWS in Action: chapter 16",
    "Outputs": [{
      "Description": "Load Balancer URL",
      "OutputKey": "EndpointURL",
      "OutputValue":
➥ "http://....us-east-1.elb.amazonaws.com"
    }],
    "StackName": "imagery",
    "StackStatus": "CREATE_COMPLETE"
  }]
}
```

Copy this output into your web browser.

Waits until CREATE_COMPLETE is reached

The `EndpointURL` output of the stack contains the URL to access the Imagery application. When you open Imagery in your web browser, you can upload an image as shown in figure 16.16.

Go ahead and upload some images and enjoy watching the images being processed.

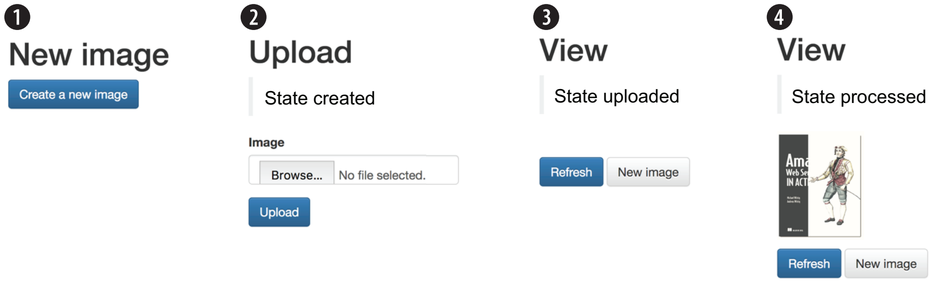


Figure 16.16 The Imagery application in action

Cleaning up

To get the name of the S3 bucket used by Imagery, run the following command in your terminal:

```
$ aws cloudformation describe-stack-resource --stack-name imagery \
➥ --logical-resource-id Bucket \
➥ --query "StackResourceDetail.PhysicalResourceId" \
➥ --output text
imagery-000000000000
```

Delete all the files in your S3 bucket `imagery-000000000000` as follows. Don't forget to replace `$bucketname` with the output from the previous command:

```
$ aws s3 rm --recursive s3://$bucketname
```

Execute the following command to delete the CloudFormation stack:

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

Stack deletion will take some time.

Congratulations! You have accomplished a big milestone: building a fault-tolerant application on AWS. You are only one step away from the end game, which is scaling your application dynamically based on load.

Summary

- Fault tolerance means expecting that failures happen and designing your systems in such a way that they can deal with failure.
- To create a fault-tolerant application, you can use idempotent actions to transfer from one state to the next.
- State shouldn't reside on the EC2 instance (a stateless server) as a prerequisite for fault tolerance.

- AWS offers fault-tolerant services and gives you all the tools you need to create fault-tolerant systems. EC2 is one of the few services that isn't fault tolerant right out of the box.
- You can use multiple EC2 instances to eliminate the single point of failure. Redundant EC2 instances in different availability zones, started with an Auto Scaling group, are how to make EC2 fault tolerant.

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.

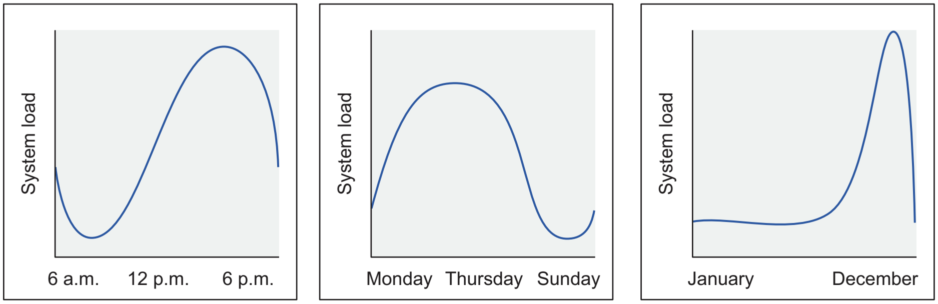


Figure 17.1 Typical traffic patterns for a web shop

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

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.

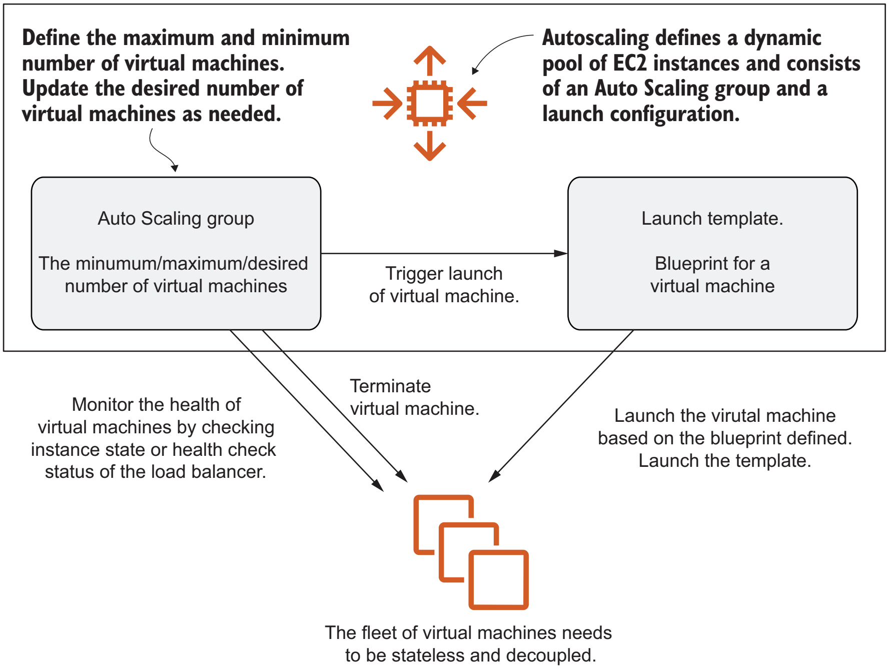

Define the maximum and minimum number of virtual machines. Update the desired number of virtual machines as needed.
Autoscaling defines a dynamic pool of EC2 instances and consists of an Auto Scaling group and a launch configuration.
Auto Scaling group
The minumum/maximum/desired number of virtual machines
Trigger launch of virtual machine.
Launch template.
Blueprint for a virtual machine
Monitor the health of virtual machines by checking instance state or health check status of the load balancer.
Terminate virtual machine.
Launch the virutal machine based on the blueprint defined. Launch the template.
The fleet of virtual machines needs to be stateless and decoupled.

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

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 instance) 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 spinning up virtual machines	ID and version of launch template
`TargetGroupARNs`	The target groups of a load balancer, where autoscaling registers new instances automatically	List of target group ARNs
`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'
      InstanceType: 't2.micro'
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeviceIndex: 0
        Groups:
        - !Ref WebServerSecurityGroup
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -x
          yum -y install httpd
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    TargetGroupARNs:
    - !Ref LoadBalancerTargetGroup
    LaunchTemplate:
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
    MinSize: 2
    MaxSize: 4
    HealthCheckGracePeriod: 300
    HealthCheckType: ELB
    VPCZoneIdentifier:
    - !Ref SubnetA
    - !Ref SubnetB
# [...]
```

Image (AMI) from which to launch new virtual machines

Instance type for new EC2 instances

Associates a public IP address with new virtual machines

Attaches these security groups when launching new virtual machines

The script executed during the bootstrap of virtual machines

References the launch template

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

Minimum number of EC2 instances

Maximum number of EC2 instances

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

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

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.

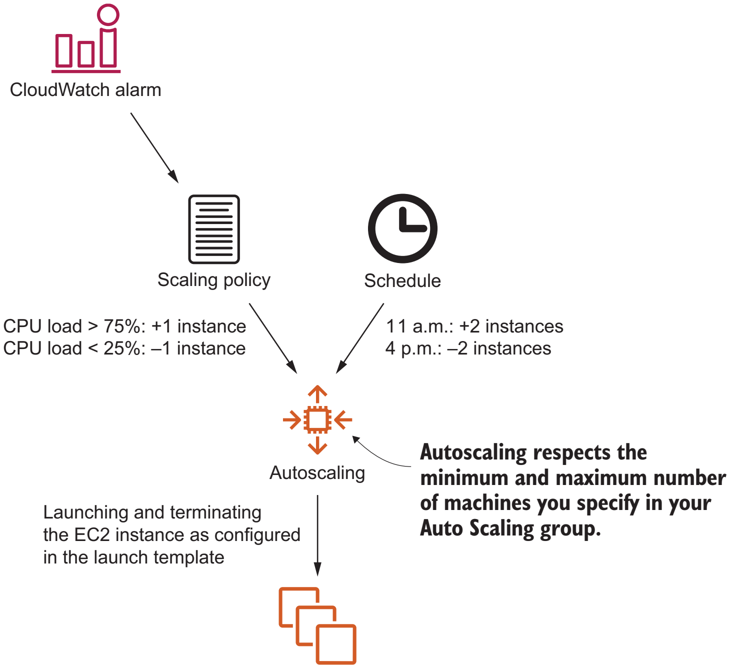

CloudWatch alarm
Scaling policy
Schedule
CPU load > 75%: +1 instance
CPU load < 25%: –1 instance
11 a.m.: +2 instances
4 p.m.: –2 instances
Autoscaling
Autoscaling respects the minimum and maximum number of machines you specify in your Auto Scaling group.
Launching and terminating the EC2 instance as configured in the launch template

Figure 17.3 Triggering autoscaling based on CloudWatch alarms or schedules

- *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).

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'
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup
    DesiredCapacity: 4
    StartTime: '2025-01-01T12:00:00Z'
```

Defining a scheduled action

Name of the Auto Scaling group

Sets the desired capacity to 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'
  Properties:
```

Defining a scheduled action

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

Sets the desired capacity to 4

Increases the capacity at 08:00 UTC every day

Sets the desired capacity to 2

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.

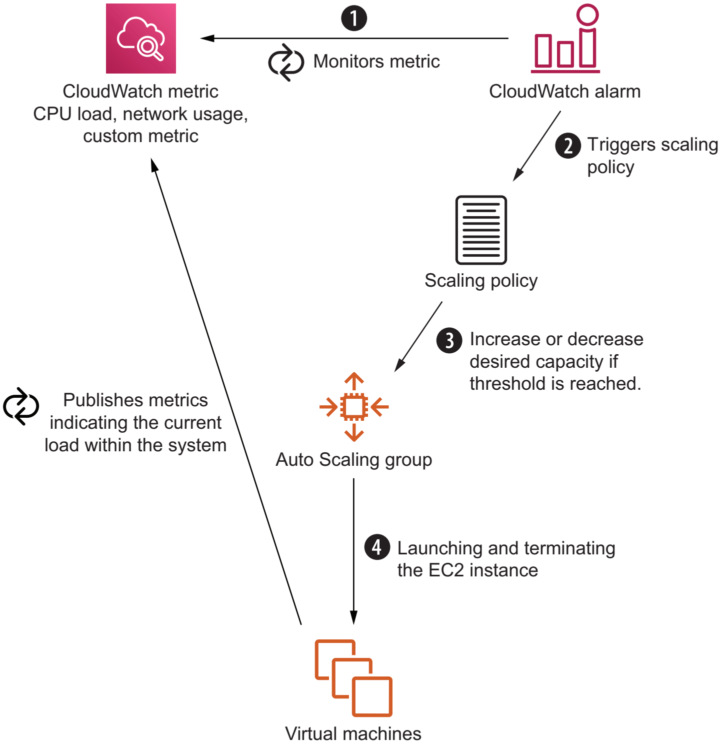

1
Monitors metric
CloudWatch metric
CPU load, network usage,
custom metric
CloudWatch alarm
2 Triggers scaling policy
Scaling policy
3 Increase or decrease desired capacity if threshold is reached.
Publishes metrics indicating the current load within the system
Auto Scaling group
4 Launching and terminating the EC2 instance
Virtual machines

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.

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.

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.

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.

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)

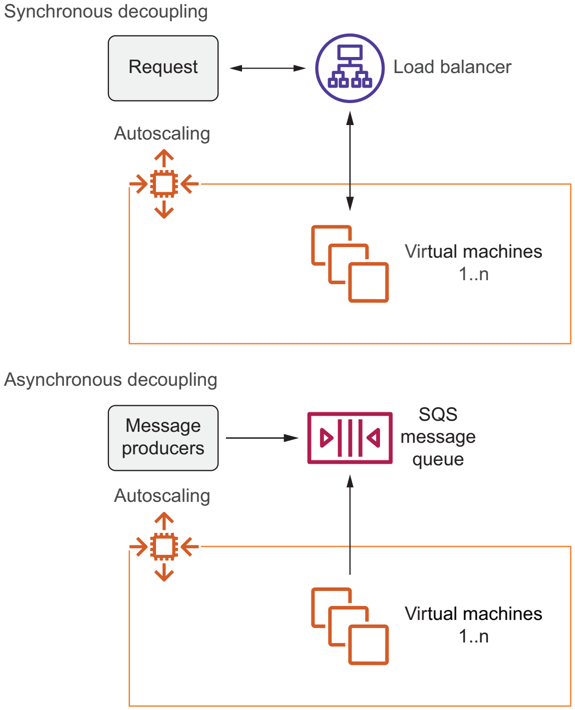


Figure 17.5 Decoupling allows you to scale the number of virtual machines dynamically.

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

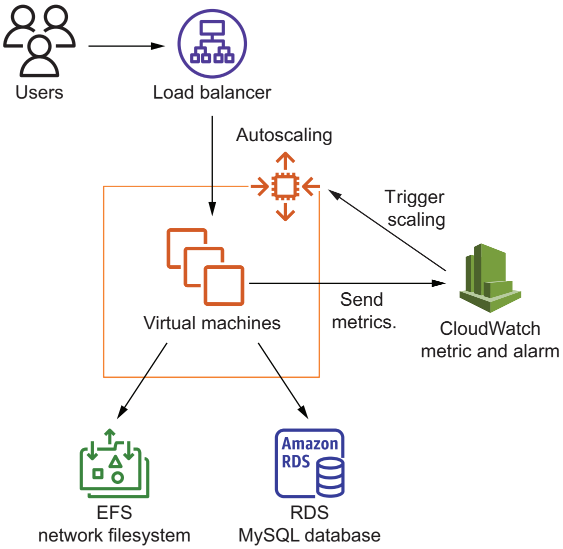


Figure 17.6 Autoscaling web servers running WordPress, storing data on RDS and EFS, decoupled with a load balancer scaling based on load

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'
    Metadata:  # [...]
    Properties:
      LaunchTemplateData:
        IamInstanceProfile:
          Name: !Ref InstanceProfile
        ImageId: !FindInMap [RegionMap,
➥ !Ref 'AWS::Region', AMI]
        Monitoring:
          Enabled: false
        InstanceType: 't2.micro'
        NetworkInterfaces:
        - AssociatePublicIpAddress: true
          DeviceIndex: 0
          Groups:
          - !Ref WebServerSecurityGroup
        UserData: # [...]
```

Contains the configuration for the EC2 instance applied during bootstrapping

Creates a launch template for autoscaling

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

Selects the image from a map with AMIs organized by region

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

Configures the instance type

Defines the network interface for the virtual machine

Enables a public IP address for the virtual machine

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

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'
    DependsOn:
    - EFSMountTargetA
    - EFSMountTargetB
    Properties:
      TargetGroupARNs:
      - !Ref LoadBalancerTargetGroup
```

Creates an Auto Scaling group

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

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

```
    LaunchTemplate:
      LaunchTemplateId: !Ref LaunchTemplate
      Version: !GetAtt 'LaunchTemplate.LatestVersionNumber'
    MinSize: 2
    MaxSize: 4
    HealthCheckGracePeriod: 300
    HealthCheckType: ELB
    VPCZoneIdentifier:
    - !Ref SubnetA
    - !Ref SubnetB
    Tags:
    - PropagateAtLaunch: true
      Value: wordpress
      Key: Name
```

References the latest version of the launch template

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

Launches no more than four machines

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

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

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

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'
  Properties:
```

Creates a scaling policy

```
    AutoScalingGroupName: !Ref AutoScalingGroup
    PolicyType: TargetTrackingScaling
    TargetTrackingConfiguration:
      PredefinedMetricSpecification:
        PredefinedMetricType: ASGAverageCPUUtilization
      TargetValue: 70
    EstimatedInstanceWarmup: 60
```

Adjusts the desired capacity of the Auto Scaling group

Creates a scaling policy tracking a specified target

Configures the target tracking

Uses a predefined scaling metric

Average CPU usage across all EC2 instances of the ASG

Defines the target at 70% CPU usage

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.

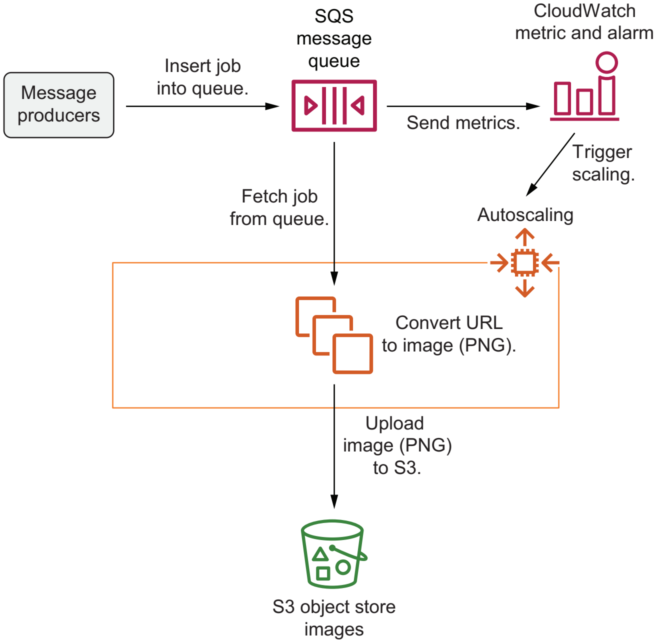


Figure 17.7 Autoscaling virtual machines that convert URLs into images, decoupled by an SQS queue

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.

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 `url2png`.

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
    Statistic: Sum
    Threshold: 5
    AlarmDescription: 'Alarm if queue length is higher than 5.'
    Period: 300
    AlarmActions:
    - !Ref ScalingUpPolicy
    Namespace: 'AWS/SQS'
    Dimensions:
    - Name: QueueName
```

Number of time periods to evaluate when checking for an alarm

Sums up all values in a period

Alarms if the threshold of five is reached

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

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

The metric is published by the SQS service.

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

```
      Value: !Sub '${SQSQueue.QueueName}'
    ComparisonOperator: GreaterThanThreshold
    MetricName: ApproximateNumberOfMessagesVisible
# [...]
```

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

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'
  Properties:
    AdjustmentType: 'ChangeInCapacity'
    AutoScalingGroupName: !Ref AutoScalingGroup
    PolicyType: 'StepScaling'
    MetricAggregationType: 'Average'
    EstimatedInstanceWarmup: 60
    StepAdjustments:
    - MetricIntervalLowerBound: 0
      ScalingAdjustment: 1
# [...]
```

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

Creates a scaling policy

The scaling policy increases the capacity by an absolute number.

Attaches the scaling policy to the Auto Scaling group

Creates a scaling policy of type step scaling

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

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

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

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 example:

```
$ URL2PNG_BUCKET=aws cloudformation describe-stacks --stack-name url2png \
➥ --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.

18 Building modern architectures for the cloud: ECS, Fargate, and App Runner

This chapter covers

- Deploying a web server with App Runner, the simplest way to run containers on AWS
- Comparing Elastic Container Service (ECS) and Elastic Kubernetes Service (EKS)
- An introduction into ECS: cluster, task definition, task, and service
- Running containers with Fargate without the need for managing virtual machines
- Building a modern architecture based on ALB, ECS, Fargate, and S3

When working with our consulting clients, we handle two types of projects:

- Brownfield projects, where the goal is to migrate workloads from on-premises to the cloud. Sooner or later, these clients also ask for ways to modernize their legacy systems.
- Greenfield projects, where the goal is to develop a solution from scratch with the latest technology available in the cloud.

Both types of projects are interesting and challenging. This chapter introduces a modern architecture, which you could use to modernize a legacy system as well as to build something from scratch. In recent years, there has hardly been a technology that has spread as rapidly as containers. In our experience, containers fit well for both brownfield and greenfield projects. You will learn how to use containers to deploy your workloads on AWS in this chapter.

We want to focus on the cutting-edge aspects of deploying containers on AWS. Therefore, we skip the details on how to create container images or start a container with Docker. We recommend *Docker in Action* (Manning, 2019; https://www.manning.com/books/docker-in-action-second-edition) if you want to learn about the fundamentals of Docker and containers.

Examples not covered by Free Tier

The examples in this chapter are not covered by the Free Tier. Deploying the examples to AWS will cost you less than $1 per day. You will find information on how to delete all resources at the end of each example or at the end of the chapter. Therefore, we recommend you complete the chapter within a few days.

Chapter requirements

This chapter assumes that you have a basic understanding of the following components:

- Running software in containers (*Docker in Action*, second edition; http://mng.bz/512a)
- Storing data on Simple Storage Service (chapter 7)
- Distributing requests with Elastic Load Balancing (chapter 14)

On top of that, the example included in this chapter makes extensive use of the following:

- Automating cloud infrastructure with CloudFormation (chapter 4)

18.1 Why should you consider containers instead of virtual machines?

Containers and virtual machines are similar concepts. This means you can apply your knowledge gained from previous chapters to the world of containers. As shown in figure 18.1, both approaches start with an image to spin up a virtual machine or container. Of course, differences exist between the technologies, but we will not discuss them here. As a mental model, it helps to think of containers as lightweight virtual machines.

How often do you hear "but it works on my machine" when talking to developers? It is not easy to create an environment providing the libraries, frameworks, and runtime environments required by an application. Since 2013, Docker has made the concept of containers popular. As in logistics, a container in software development is a

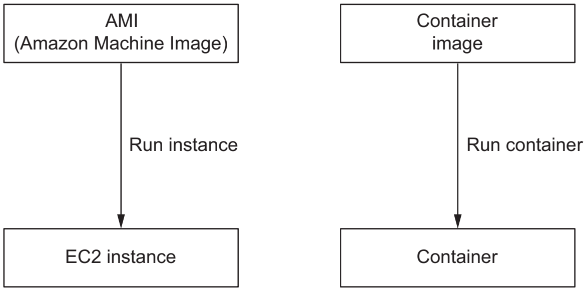


Figure 18.1 From a high-level view, virtual machines and containers are similar concepts.

standardized unit that can be easily moved and delivered. In our experience, this method simplifies the development process significantly, especially when aiming for continuous deployment, which means shipping every change to test or production systems automatically.

In theory, you spin up a container based on the same image on your local machine, an on-premises server, and in the cloud. Boundaries exist only between UNIX/Linux and Windows, as well as Intel/AMD and ARM processors. In contrast, it is much more complicated to launch an Amazon Machine Image (AMI) on your local machine.

Containers also increase portability. In our opinion, it is much easier to move a containerized workload from on-premises to the cloud or to another cloud provider. But beware of the marketing promises by many vendors: it is still a lot of work to integrate your system with the target infrastructure.

We have guided several organizations in the adoption of containers. In doing so, we have observed that containers promote an important competency: building and running immutable servers. An immutable server is a server that you do not change once it is launched from an image. But what if you need to roll out a change? Create a new image and replace the old servers with servers launched from the new image. In theory, you could do the same thing with EC2 instances as well, and we highly recommend you do so. But because you are typically not able to log in to a running container to make changes, following the immutable server approach is your only option. The keyword here is *Dockerfile*, a configuration file containing everything needed to build a container image.

18.2 Comparing different options to run containers on AWS

Next, let's answer the question of how best to deploy containers on AWS. To impress you, let's start with a simple option: AWS App Runner. Type the code in listing 18.1 into your terminal to launch containers running a simple web server from a container image.

AWS CLI

Is the AWS CLI not working on your machine? Go to chapter 4 to learn how to install and configure the command-line interface.

Listing 18.1 Creating an App Runner service

```
aws apprunner create-service \
➥ --service-name simple \
➥ --source-configuration '{"ImageRepository": \
➥ {"ImageIdentifier": "public.ecr.aws/
➥ s5r5a1t5/simple:latest", \
➥ "ImageRepositoryType": "ECR_PUBLIC"}}'
```

Creates an App Runner service that will spin up containers

Defines a name for the service

Configures the source of the container image

Chooses a public or private container registry hosted by AWS

It will take about five minutes until a simple web server is up and running. Use the following code to get the status and URL of your service, and open the URL in your browser. On a side note, App Runner even supports custom domains, in case that's a crucial feature to you.

Listing 18.2 Fetching information about App Runner services

```
$ aws apprunner list-services
{
  "ServiceSummaryList": [
    {
      "ServiceName": "simple",
      "ServiceId": "5e7ffd09c13d4d6189e99bb51fc0f230",
      "ServiceArn": "arn:aws:apprunner:us-east-1:...",
      "ServiceUrl":
➥   "bxjsdpnnaz.us-east-1.awsapprunner.com",
      "CreatedAt": "2022-01-07T20:26:48+01:00",
      "UpdatedAt": "2022-01-07T20:26:48+01:00",
      "Status": "RUNNING"
    }
  ]
}
```

The ARN of the service, needed to delete the service later

Opens this URL in your browser

Waits until the status reaches RUNNING

App Runner is a Platform as a Service (PaaS) offering for container workloads. You provide a container image bundling a web application, and App Runner takes care of everything else, as illustrated in figure 18.2:

- Runs and monitors containers
- Distributes requests among running containers
- Scales the number of containers based on load

You pay for memory but not CPU resources during times when a running container does not process any requests. Let's look at pricing with an example. Imagine a web application with minimal resource requirements only used from 9 a.m. to 5 p.m.,

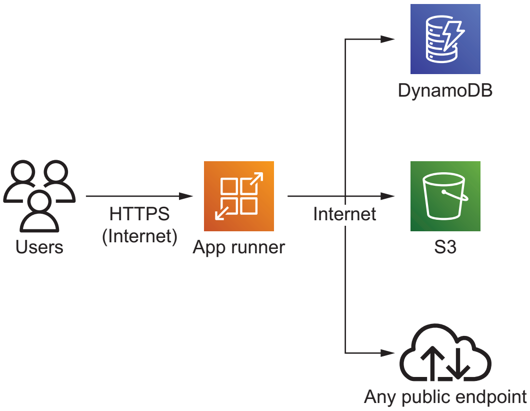


Figure 18.2 App Runner provides a simple way to host containerized web applications.

which is eight hours per day. The minimal configuration on App Runner is 1 vCPU and 2 GB memory:

- Active hours (= hours in which requests are processed)
  - 1 vCPU: $0.064 * 8 * 30 = $15.36 per month
  - 2 GB memory: 2 * $0.007 * 8 * 30 = $3.36 per month
- Inactive hours (= hours in which no requests are processed)
  - 2 GB memory: 2 * $0.007 * 16 * 30 = $6.72 per month

In total, that's $25.44 per month for the smallest configuration supported by App Runner. See "AWS App Runner Pricing" at https://aws.amazon.com/apprunner/pricing/ for more details.

By the way, don't forget to delete your App Runner service to avoid unexpected costs. Replace `$ServiceArn` with the ARN you noted after creating the service, as shown here.

Listing 18.3 Fetching information about App Runner services

```
$ aws apprunner delete-service \
➥ --service-arn $ServiceArn
```

That was fun, wasn't it? But simplicity comes with limitations. Here are two reasons App Runner might not be a good fit to deploy your application:

- App Runner does not come with an SLA yet.
- Also, comparing costs between the different options is tricky, because different dimensions are used for billing. Roughly speaking, App Runner should be cheap for small workloads with few requests but rather expensive for large workloads with many requests.

That's why we will introduce two other ways to deploy containers on AWS next. The two main services to manage containers on AWS are Elastic Container Service (ECS) and Elastic Kubernetes Services (EKS).

> **WHAT IS KUBERNETES?** Kubernetes (K8s) is an open source container orchestration system. Originally, Google developed Kubernetes, but nowadays, the Cloud Native Computing Foundation maintains the project. Kubernetes can run on your local machine and on-premises, and most cloud providers offer a fully managed service.

The discussion about which of the two services is better is often very heated and reminiscent of the discussions about the editors vim and emacs. When viewed unemotionally, the functional scope of ECS and EKS is very similar. The both handle the following:

- Monitoring and replacing failed containers
- Deploying new versions of your containers
- Scaling the number of containers to adapt to load

Of course, we would also like to highlight the differences, which we have summarized in table 18.1.

Table 18.1 Launch configuration parameters

Category	ECS	EKS
Portability	ECS is available on AWS. ECS Anywhere is an extension to use ECS for on-premises workloads. Other cloud providers do not support ECS.	EKS is available on AWS. For on-premises workloads, you have EKS Anywhere, which is supported by AWS but requires VMware vSphere and offers the option to deploy and manage Kubernetes yourself. Also, most other cloud providers come with a Kubernetes offering.
License	Proprietary service but free of charge	Open source license (Apache License 2.0)
Ecosystem	Works very well together with many AWS services (e.g., ALB, IAM, and VPC)	Comes with a vibrant open source ecosystem (e.g., Prometheus, Helm). Integration with AWS services exists but is not always mature.
Costs	A cluster is free. Of course, you pay for the compute infrastructure.	AWS charges about $72 per month for each cluster. Also, AWS recommends *not* to deploy workloads that require isolation to the same cluster. On top of that, you are paying for the compute infrastructure.

We observe that Kubernetes is very popular especially, but not only, among developers. Even though we are software developers ourselves, we prefer ECS for most workloads. The most important arguments for us are monthly costs per cluster and integration with other AWS services. On top of that, CloudFormation comes with full support for ECS.

Next, you will learn about the basic concepts behind ECS.

18.3 The ECS basics: Cluster, service, task, and task definition

When working with ECS, you need to create a cluster first. A cluster is a logical group for all the components we discuss next. It is fine to create multiple clusters to isolate workloads from each other. For example, we typically create different clusters for test and production environments. The cluster itself is free, and by default, you can create up to 10,000 clusters—which you probably do not need, by the way.

To run a container on ECS, you need to create a task definition. The task definition includes all the information required to run a container, as shown here. See figure 18.3 for more details:

- The container image URL
- Provisioned baseline and limit for CPU
- Provisioned baseline and limit for memory
- Environment variables
- Network configuration

Please note: a task definition might describe one or multiple containers.

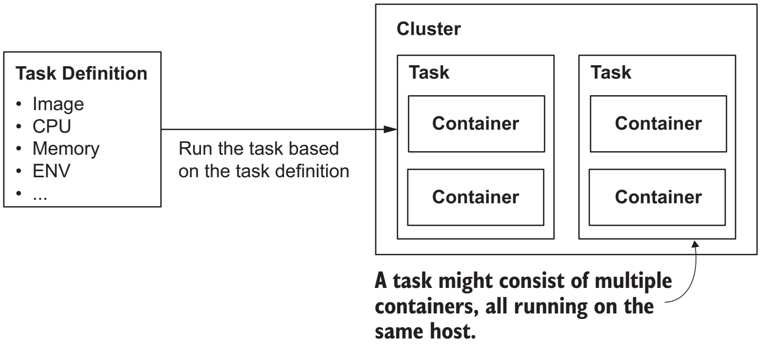

Task Definition
• Image
• CPU
• Memory
• ENV
• ...
Run the task based on the task definition
Cluster
Task
Container
Container
Task
Container
Container
A task might consist of multiple containers, all running on the same host.

Figure 18.3 A task definition defines all the details needed to create a task, which consists of one or multiple containers.

Next, you are ready to create a task. To do so, you need to specify the cluster as well as the task definition. After you create the task, ECS will try to run the containers as specified. Note that all containers defined in a task definition will run on the same host. This is important if you have multiple containers that need to share local resources—the local network, for example. Figure 18.3 shows how to run tasks based on a task definition.

Luckily, you can, but do not have to, create tasks manually. Suppose you want to deploy a web server on ECS. In this case, you need to ensure that at least two containers of the same kind are running around the clock to spread the workload among two availability zones. In the case of high load, even more containers should be started for a short time. You need an ECS service for that.

Think of an ECS service as similar to an Auto Scaling group. An ECS service, as shown in figure 18.4, performs the following tasks:

- Runs multiple tasks of the same kind
- Scales the number of tasks based on load
- Monitors and replaces failed tasks
- Spreads tasks across availability zones
- Orchestrates rolling updates

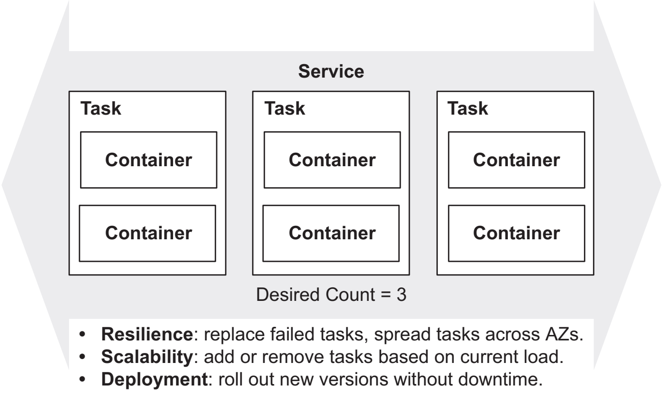

Service
Task
Container
Container
Task
Container
Container
Task
Container
Container
Desired Count = 3
• Resilience: replace failed tasks, spread tasks across AZs.
• Scalability: add or remove tasks based on current load.
• Deployment: roll out new versions without downtime.

Figure 18.4 An ECS service manages multiple tasks of the same kind.

Equipped with the knowledge of the most important components for ECS, we move on.

18.4 AWS Fargate: Running containers without managing a cluster of virtual machines

Let's take a little trip down AWS history lane. ECS has been generally available since 2015. Since its inception, ECS has been adding another layer to our infrastructures. With ECS you had to manage, maintain, and scale not only containers but also the underlying EC2 instances. This increased complexity significantly.

In November 2017, AWS introduced an important service: AWS Fargate. As shown in figure 18.5, Fargate provides a fully managed container infrastructure, allowing you to spin up containers in a similar way to launching EC2 instances. This was a game changer! Since then, we have deployed our workloads with ECS and Fargate whenever possible, and we advise you to do the same.

By the way, Fargate is available not only for ECS but for EKS as well. Also, Fargate offers Amazon Linux 2 and Microsoft Windows 2019 Server Full and Core editions as a platform for your containers.

With EC2 instances, you choose an instance type, which specifies the available resources like CPU and memory. In contrast, Fargate requires you to configure the provisioned CPU and memory capacity per task. Table 18.2 shows the available options.

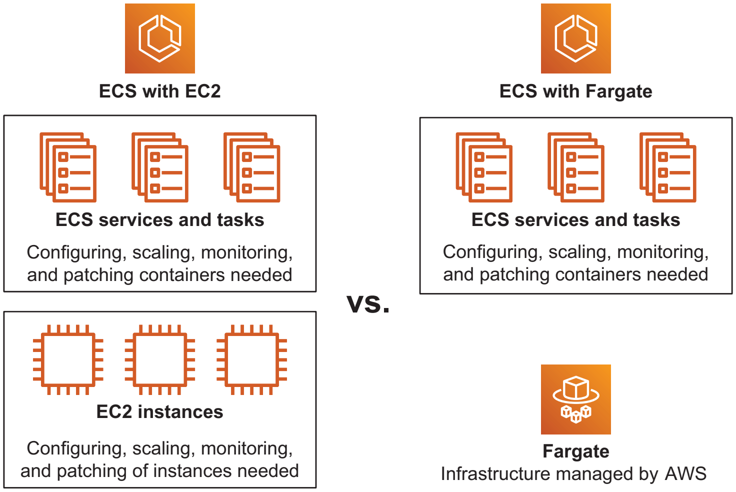

ECS with EC2
ECS services and tasks
Configuring, scaling, monitoring, and patching containers needed
EC2 instances
Configuring, scaling, monitoring, and patching of instances needed
vs.
ECS with Fargate
ECS services and tasks
Configuring, scaling, monitoring, and patching containers needed
Fargate
Infrastructure managed by AWS

Figure 18.5 With Fargate, you do not need to manage a fleet of EC2 instances to deploy containers with ECS.

Table 18.2 Provisioning CPU and memory for Fargate

CPU	Memory
0.25 vCPU	0.5 GB, 1 GB, or 2 GB
0.5 vCPU	Minimum 1 GB, Maximum 4 GB, 1 GB increments
1 vCPU	Minimum 2 GB, Maximum 8 GB, 1 GB increments
2 vCPU	Minimum 4 GB, Maximum 16 GB, 1 GB increments
4 vCPU	Minimum 8 GB, Maximum 30 GB, 1 GB increments
8 vCPU	Minimum 16 GB, Maximum 60 GB, 4 GB increments
16 vCPU	Minimum 32 GB, Maximum 120 GB, 8 GB increments

Fargate is billed for every second a task is running, from downloading the container image until the task terminates. What does a Fargate task with 1 vCPU and 4 GB memory cost per month? It depends on the region and architecture (Linux/X86, Linux/ARM, Windows/X86). Let's do the math for Linux/ARM in us-east-1:

- 1 vCPU: $0.04048 * 24 * 30 = $29.15 per month
- 4 GB memory: 4 * $0.004445 * 24 * 30 = $12.80 per month

In total, that's $41.95 per month for a Fargate task with 1 vCPU and 2 GB memory. See "AWS Fargate Pricing" at https://aws.amazon.com/fargate/pricing/ for more details.

When comparing the costs for CPU and memory, it is noticeable that EC2 is cheaper compared to Fargate. For example, a `m6g.medium` instance with 1 vCPU and 4 GB memory costs $27.72 per month. But when scaling EC2 instances for ECS yourself, fragmentation and overprovisioning will add up as well. Besides that, the additional complexity will consume working time. In our opinion, Fargate is worth it in most scenarios.

It is important to mention that Fargate comes with a few limitations. Most applications are not affected by those limitations, but you should double-check before starting with Fargate. A list of the most important—but not all—limitations follows. See "Amazon ECS on AWS Fargate" at http://mng.bz/19Wn for more details:

- A maximum of 16 vCPU and 120 GB memory.
- Container cannot run in `privileged` mode.
- Missing GPU support.
- Attaching EBS volumes is not supported.

Now it's finally time to see ECS in action.

18.5 Walking through a cloud-native architecture: ECS, Fargate, and S3

We take notes all the time: when we're on the phone with a customer, when we're thinking through a new chapter for a book, when we're looking at a new AWS service in detail. Do you do this too? Imagine you want to host your notes in the cloud. In this example, you will deploy Notea, a privacy-first, open source note-taking application to AWS. Notea is a typical modern web application that uses React for the user interface and Next.js for the backend. All data is stored on S3.

The cloud-native architecture, as shown in figure 18.6, consists of the following building blocks:

- The Application Load Balancer (ALB) distributes incoming requests among all running containers.
- An ECS service spins up containers and scales based on CPU load.
- Fargate provides the underlying compute capacity.
- The application stores all data on S3.

You may have noticed that the concepts from the previous chapters can be transferred to a modern architecture based on ECS easily.

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 Notea example is located in /chapter18/notea.yaml.

Execute the following command to create a CloudFormation stack that spins up Notea. Don't forget to replace `$ApplicationId` with a unique character sequence (e.g., your name abbreviation) and `$Password` with a password for protecting your notes.

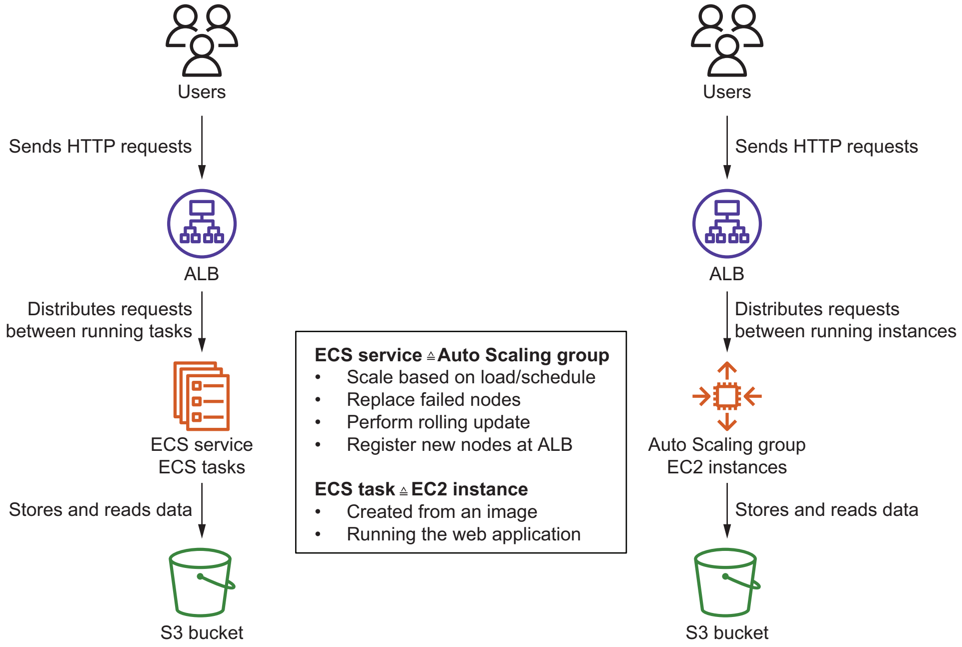


Figure 18.6 ECS is for containers what an Auto Scaling group is for EC2 instances.

Please note: your password will be transmitted unencrypted over HTTP, so you should use a throwaway password that you are not using anywhere else:

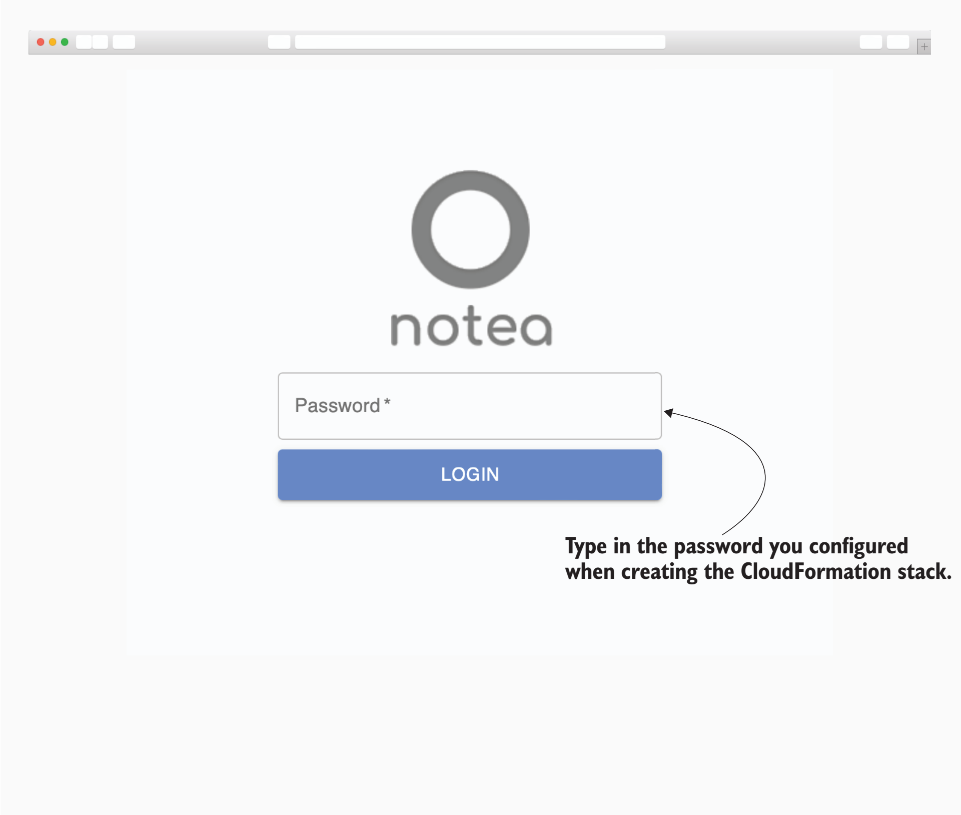
```
$ aws cloudformation create-stack --stack-name notea \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code3/chapter18/notea.yaml --parameters \
➥ "ParameterKey=ApplicationID,ParameterValue=$ApplicationId" \
➥ "ParameterKey=Password,ParameterValue=$Password" \
➥ --capabilities CAPABILITY_IAM
```

It will take about five minutes until your note-taking app is up and running (figure 18.7). Use the following command to wait until the stack was created successfully and fetch the URL to open in your browser:

```
$ aws cloudformation wait stack-create-complete \
➥ --stack-name notea && aws cloudformation describe-stacks \
➥ --stack-name notea --query "Stacks[0].Outputs[0].OutputValue" \
➥ --output text
```

Congratulations! You have launched a modern web application with ECS and Fargate. Happy note-taking!

Next, we highly recommend you open the AWS Management Console and go to the ECS service to explore the cluster, the service, the tasks, and tasks definition. Use https://console.aws.amazon.com/ecs/ to jump right into the ECS service.



Type in the password you configured when creating the CloudFormation stack.

Figure 18.7 Notea is up and running!

What we like about ECS is that we can deploy all components with CloudFormation. Therefore, let's dive into the code. First, you need to create a task definition. For a better understanding, figure 18.8 shows the different configuration parts of the task definition.

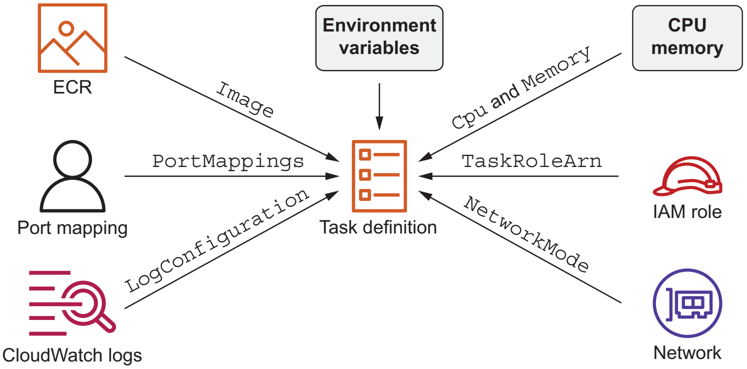


Figure 18.8 Configuring a task definition with CloudFormation

The code in the next listing shows the details.

Listing 18.4 Configuring a task definition

```
TaskDefinition:
  Type: 'AWS::ECS::TaskDefinition'
  Properties:
    ContainerDefinitions:
    - Name: app
      Image: 'public.ecr.aws/s5r5a1t5/notea:latest'
      PortMappings:
      - ContainerPort: 3000
        Protocol: tcp
      Essential: true
      LogConfiguration:
        LogDriver: awslogs
        Options:
          'awslogs-region': !Ref 'AWS::Region'
          'awslogs-group': !Ref LogGroup
          'awslogs-stream-prefix': app
      Environment:
      - Name: 'PASSWORD'
        Value: !Ref Password
      - Name: 'STORE_REGION'
        Value: !Ref 'AWS::Region'
      - Name: 'STORE_BUCKET'
        Value: !Ref Bucket
      - Name: COOKIE_SECURE
        Value: 'false'
    Cpu: 512
    ExecutionRoleArn: !GetAtt 'TaskExecutionRole.Arn'
    Family: !Ref 'AWS::StackName'
    Memory: 1024
    NetworkMode: awsvpc
    RequiresCompatibilities: [FARGATE]
    TaskRoleArn: !GetAtt 'TaskRole.Arn'
```

We will reference the container named app later.

Remember that a task definition describes one or multiple containers? In this example, there is only one container, called app.

The URL points to a publicly hosted container image bundling the Notea app.

The container starts a server on port 3000.

The log configuration tells the container to ship logs to CloudWatch, which is the default for ECS and Fargate.

The notea container expects a few environment variables for configuration. Those environment variables are configured here.

Tells Fargate to provision 0.5 vCPUs for our task

The IAM role is used by Fargate to fetch container images, ship logs, and similar tasks.

Tells Fargate to assign 1024 MB memory to our task

Specifies that the task definition should be used with Fargate only

The IAM role used by the application to access S3

Fargate supports only the networking mode awsvpc, which will attach an Elastic Network Interface (ENI) to each task. You learned about the ENI in chapter 16 already.

The task definition configures two IAM roles for the tasks. An IAM role is required to authenticate and authorize when accessing any AWS services. The IAM role defined by `ExecutionRoleArn` is not very interesting—the role grants Fargate access to basic services for downloading container images or publishing logs. However, the IAM role `TaskRoleArn` is very important because it grants the containers access to AWS services. In our example, Notea requires read and write access to S3. And that's exactly what the IAM role in listing 18.5 is all about.

Listing 18.5 Granting the container access to objects in an S3 bucket

```
TaskRole:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Statement:
      - Effect: Allow
        Principal:
          Service: 'ecs-tasks.amazonaws.com'
        Action: 'sts:AssumeRole'
    Policies:
    - PolicyName: S3AccessPolicy
      PolicyDocument:
        Statement:
        - Effect: Allow
          Action:
          - 's3:GetObject'
          - 's3:PutObject'
          - 's3:DeleteObject'
          Resource: !Sub '${Bucket.Arn}/*'
        - Effect: Allow
          Action:
          - 's3:ListBucket'
          Resource: !Sub '${Bucket.Arn}'
Bucket:
  Type: 'AWS::S3::Bucket'
  Properties:
    BucketName: !Sub 'awsinaction-notea-${ApplicationID}'
```

The IAM role is used by ECS tasks only; therefore, we need to allow the ECS tasks service access to assume the role.

Authorizes the role to read data from S3

Authorizes the role to write data to S3

Authorizes the role to delete data from S3

The S3 bucket used by Notea to store data

Read and write access is granted only to Notea's S3 bucket.

Allows listing all the objects in the bucket

Next, you need to create an ECS service that launches tasks and, with them, containers. The most important configuration details shown in the next listing are:

- `DesiredCount`—Defines the number of tasks the service will launch. The `DesiredCount` will be changed by autoscaling later.
- `LoadBalancers`—The service registers and unregisters tasks at the ALB out of the box with this configuration.

Listing 18.6 Creating an ECS service to spin up tasks running the web app

```
Service:
  DependsOn: HttpListener
  Type: 'AWS::ECS::Service'
  Properties:
    Cluster: !Ref 'Cluster'
    CapacityProviderStrategy:
    - Base: 0
      CapacityProvider: 'FARGATE'
      Weight: 1
    DeploymentConfiguration:
      MaximumPercent: 200
      MinimumHealthyPercent: 100
```

Runs tasks on Fargate. Alternatively, you could switch to FARGATE_SPOT to reduce costs, similar to EC2 Spot Instances, as discussed in chapter 3.

A service belongs to a cluster.

During a deployment, ECS is allowed to double the number of tasks.

During a deployment, ECS ensures that the number of running containers does not decrease.

```
      DeploymentCircuitBreaker:
        Enable: true
        Rollback: true
    DesiredCount: 2
    HealthCheckGracePeriodSeconds: 30
    LoadBalancers:
    - ContainerName: 'app'
      ContainerPort: 3000
      TargetGroupArn: !Ref TargetGroup
    NetworkConfiguration:
      AwsvpcConfiguration:
        AssignPublicIp: 'ENABLED'
        SecurityGroups:
        - !Ref ServiceSecurityGroup
        Subnets: [!Ref SubnetA, !Ref SubnetB]
    PlatformVersion: '1.4.0'
    TaskDefinition: !Ref TaskDefinition
```

The ECS service registers and unregisters tasks at the load balancer.

By enabling the deployment circuit breaker, you ensure that ECS will not try forever to deploy a broken version.

ECS will run or stop tasks to make sure two tasks are up and running.

When a new task starts, ECS will wait for 30 seconds for the task to pass the health check. You need to increase this period for applications that start slowly.

The application is listening on port 3000.

The target group of the load balancer to register or deregister tasks

When deploying to a public subnet, assigning public IP addresses is required to ensure outbound connectivity.

A list of subnets in which to start tasks. You should use at least two different subnets and a desired count greater than two to achieve high availability.

Each tasks comes with its own ENI. The security group defined here is used to filter traffic.

From time to time, AWS releases a new Fargate plattform with additional features. We highly recommend specifying a platform version instead of using LATEST to avoid problems in production.

To be more precise, a specific container is registered at the load balancer.

Being able to scale workloads is one of the superpowers of cloud computing. Of course, our container-based infrastructure should be able to scale out and scale in based on load as well. The next listing shows how to configure autoscaling. In the example, we are using a target-tracking scaling policy. The trick is that we need to define the target value only for the CPU utilization. The Application Auto Scaling service will take care of the rest and will increase or decrease the desired count of the ECS service automatically.

Listing 18.7 Configuring autoscaling based on CPU utilization for the ECS service

```
ScalableTarget:
  Type: AWS::ApplicationAutoScaling::ScalableTarget
  Properties:
    MaxCapacity: '4'
    MinCapacity: '2'
    RoleARN: !GetAtt 'ScalableTargetRole.Arn'
```

The upper limit for scaling tasks

The lower limit for scaling tasks

The IAM role is required to grant Application Auto Scaling access to CloudWatch metrics and ECS.

Application Auto Scaling supports all kinds of services. You want to scale ECS in this example.

Scales by increasing or decreasing the desired count of the ECS service

```
    ServiceNamespace: ecs
    ScalableDimension: 'ecs:service:DesiredCount'
    ResourceId: !Sub
    - 'service/${Cluster}/${Service}'
    - Cluster: !Ref Cluster
      Service: !GetAtt 'Service.Name'
CPUScalingPolicy:
  Type: AWS::ApplicationAutoScaling::ScalingPolicy
  Properties:
    PolicyType: TargetTrackingScaling
    PolicyName: !Sub 'awsinaction-notea-${ApplicationID}'
    ScalingTargetId: !Ref ScalableTarget
    TargetTrackingScalingPolicyConfiguration:
      TargetValue: 50.0
      ScaleInCooldown: 180
      ScaleOutCooldown: 60
      PredefinedMetricSpecification:
        PredefinedMetricType: ECSServiceAverageCPUUtilization
```

References the ECS service in the ECS cluster created above

A simple way to scale ECS services is by using target-tracking scaling, which requires minimal configuration.

References the scalable target resource from above

After terminating tasks, waits three minutes before reevaluating the situation

In this example, we scale based on CPU utilization. ECSServiceAverageMemoryUtilization is another predefined metric.

The target is to keep the CPU utilization at 50%. You might want to increase that to 70–80% in real-world scenarios.

After starting tasks, waits one minute before reevaluating the situation

That's it, you have learned how to deploy a modern web application on ECS and Fargate.

Don't forget to delete the CloudFormation stack and all data on S3, as shown in the following code snippet. Replace `$ApplicationId` with a unique character sequence you chose when creating the stack:

```
$ aws s3 rm s3://awsinaction-notea-${ApplicationID} --recursive
$ aws cloudformation delete-stack --stack-name notea
$ aws cloudformation wait stack-delete-complete \
➥ --stack-name notea
```

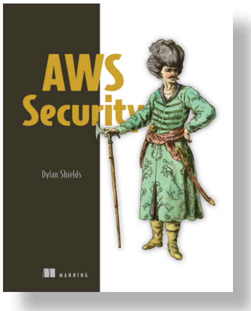
What a ride! You have come a long way from AWS basics, to advanced cloud architecture principles, to modern containerized architectures. Now only one thing remains to be said: go build!

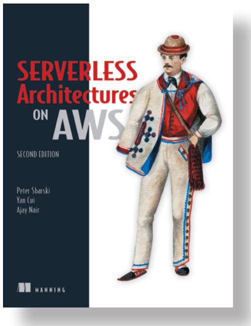
Summary

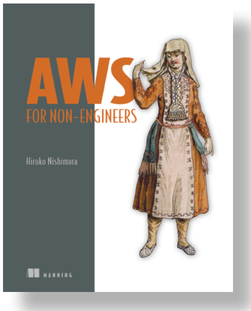
- App Runner is the simplest way to run containers on AWS. However, to achieve simplicity, App Runner comes with limitations. For example, the containers aren't running in your VPC.

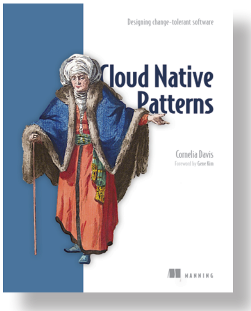
- The Elastic Container Service (ECS) and the Elastic Kubernetes Service (EKS) are both orchestrating container clusters. We recommend ECS for most use cases because of cost per cluster, integration into all parts of AWS, and CloudFormation support.
- With Fargate, you no longer have to maintain an EC2 instance to run your containers. Instead, AWS provides a fully managed compute layer for containers.
- The main components of ECS are cluster, task definition, task, and service.
- The concepts from EC2-based architectures apply to container-based architectures as well. For example, an ECS service is the equivalent of an Auto Scaling group.

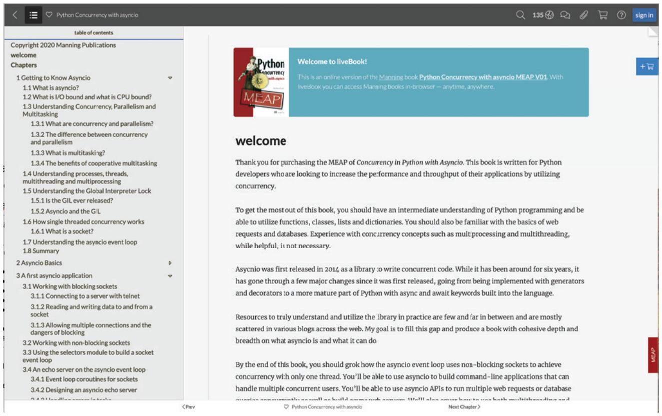
index


AWS
Security
Dylan Shields
MANNING


SERVERLESS
Architectures
ON AWS
SECOND EDITION
Peter Sbarski
Yan Cui
Ajay Nair
MANNING


AWS
FOR NON-ENGINEERS
Hiroko Nishimura
MANNING


Designing change-tolerant software
Cloud Native
Patterns
Cornelia Davis
Foreword by Gene Kim
MANNING



AWS Services Explained in the Book

Storage and Database

Abbr.	Name	Description	Section
S3	Amazon Simple Storage Service	Object store to save data without any size restrictions	7
Glacier	Amazon S3 Glacier	Inexpensive data archive solution	7.4
EBS	Amazon Elastic Block Store	Network attached block-level storage for EC2 instances	8.1
	Amazon EC2 Instance Store	Block-level storage for EC2 instances	8.2
EFS	Amazon Elastic File System	Scalable network filesystem based on NFSv4	9
RDS	Amazon Relational Database Service	MySQL, Oracle Database, Microsoft SQL Server, or PostgreSQL	10
	Amazon ElastiCache	In-memory data store based on Redis or Memcached, typically used to cache data	11
	Amazon DynamoDB	Proprietary NoSQL key-value database with limited but powerful options for queries	12

Architecting on AWS

Abbr.	Name	Description	Section
AZ	Availability Zone	Group of isolated data centers within a region	13.2
ASG	Amazon EC2 Auto Scaling Group	Observes a fleet of servers: replaces faulty servers and increases/decreases the size of the fleet based on external triggers, like CPU usage	17
	Amazon CloudWatch	Keeps track of metrics and triggers alarms when a threshold is reached	13.1, 17
ELB	Elastic Load Balancing	Load balancer for EC2 instances	14.1
ALB	Application Load Balancer	Layer 7 load balancer with HTTP and HTTPS support	14.1
SQS	Amazon Simple Queue Service	Message queue	14.2

CLOUD / AMAZON WEB SERVICES

Amazon Web Services IN ACTION

Third Edition

Andreas Wittig • Michael Wittig

Amazon Web Services, the leading cloud computing platform, offers customers APIs for on-demand access to computing services. Rich in examples and best practices of how to use AWS, this Manning bestseller is now released in its third, revised, and improved edition.

In **Amazon Web Services in Action, Third Edition: An in-depth guide to AWS**, the Wittig brothers give you a comprehensive, practical introduction to deploying and managing applications on the AWS cloud platform. With a sharp focus on the most important AWS tasks and services, they will save you hours of unproductive time. You'll learn hands-on as you complete real-world projects like hosting a WordPress site, setting up a private cloud, and deploying an app on containers.

What's Inside

- Leverage globally distributed data centers to launch virtual machines
- Enhance performance with caching data in-memory
- Secure workloads running in the cloud with VPC and IAM
- Build fault-tolerant web applications with ALB and SQS

Written for mid-level developers, DevOps or platform engineers, architects, and system administrators.

Andreas Wittig and **Michael Wittig** are software engineers and consultants focused on AWS. Together, they migrated the first bank in Germany to AWS in 2013.

For print book owners, all ebook formats are free:
https://www.manning.com/freebook

“Up-to-date coverage. Code examples and configurations are all excellent. Even containerization is very well explained. This is the bible for Amazon Web Services.”

—Mohammad Shahnawaz Akhter, Bank of America

“It has never been so easy to learn AWS.”

—Jorge Ezequiel Bo, TravelX

“Essential for those who decide to embark on the Amazon cloud journey.”

—Matteo Rossi
UnipolSai Assicurazioni

“A complete introduction to the most important AWS Services with very useful practical examples.”

—Matteo Battista, GamePix

Free eBook
See first page

ISBN-13: 978-1-63343-916-0

90000

9 781633 439160

MANNING