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.

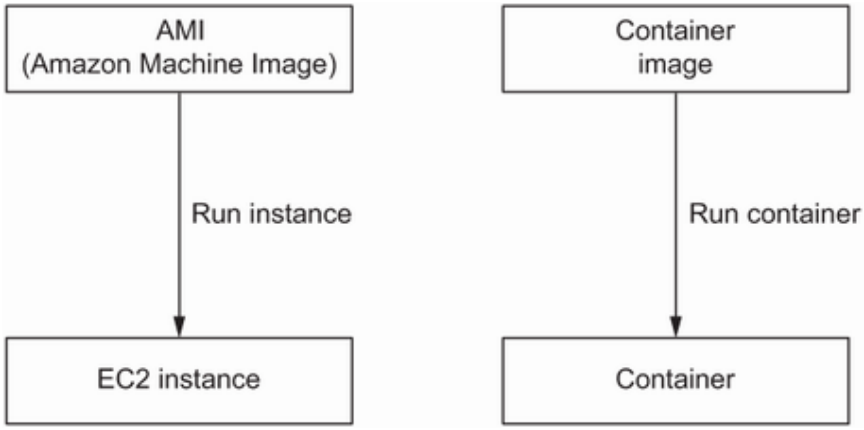

Figure 18.1 From a high-level view, virtual machines and containers are similar concepts.

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 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 ap-

proach 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 \                                    ①
  --service-name simple \                                         ②
  --source-configuration '{"ImageRepository": \
  {"ImageIdentifier": "public.ecr.aws/
  s5r5a1t5/simple:latest", \                                      ③
  "ImageRepositoryType": "ECR_PUBLIC"}}'                          ④
```

① Creates an App Runner service that will spin up containers

② Defines a name for the service

③ Configures the source of the container image

④ 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:...",   ①
      "ServiceUrl":
→ "bxjsdpnnaz.us-east-1.awsapprunner.com",              ②
      "CreatedAt": "2022-01-07T20:26:48+01:00",
      "UpdatedAt": "2022-01-07T20:26:48+01:00",
      "Status": "RUNNING"                                ③
    }
  ]
}
```

① The ARN of the service, needed to delete the service later

② Opens this URL in your browser

③ 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

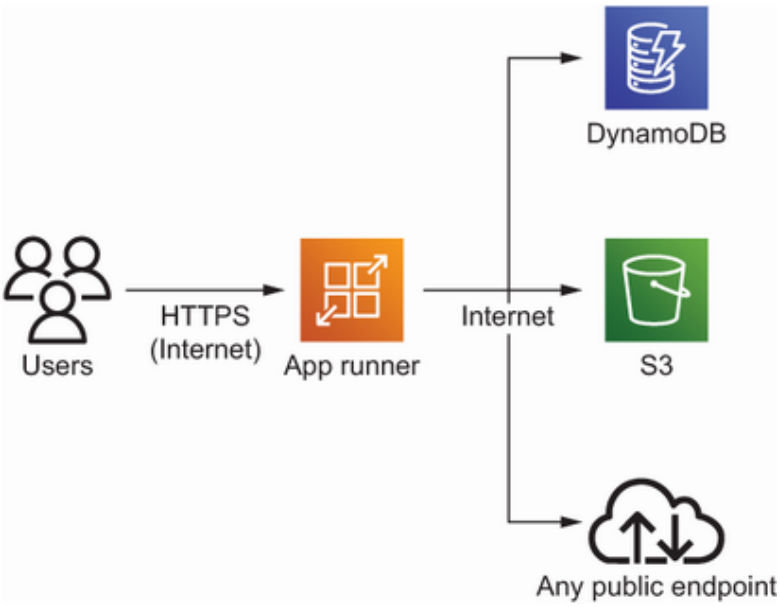

Figure 18.2 App Runner provides a simple way to host containerized web applications.

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 ex-

ample. Imagine a web application with minimal resource requirements only used from 9 a.m. to 5 p.m., 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.

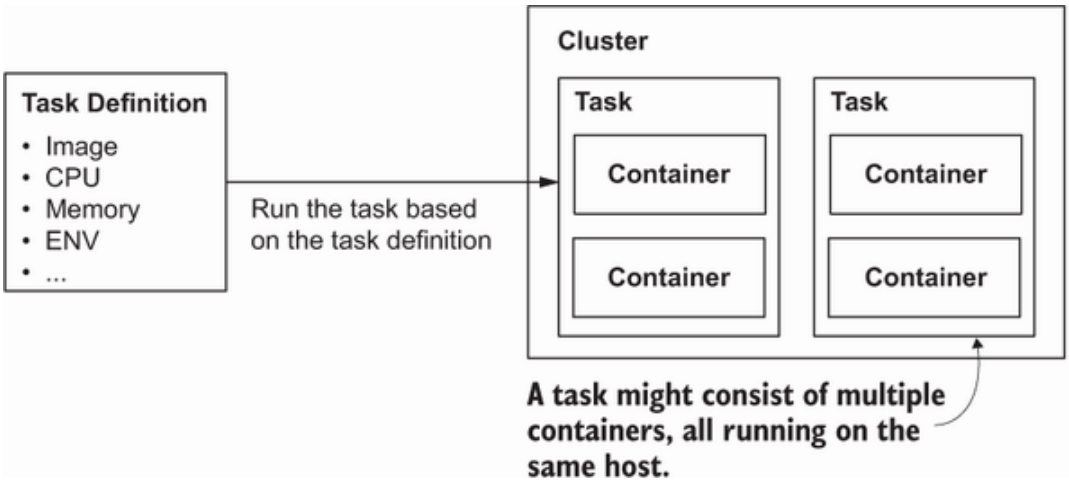

Figure 18.3 A task definition defines all the details needed to create a task, which consists of one or multiple containers.

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 multi-

ple 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

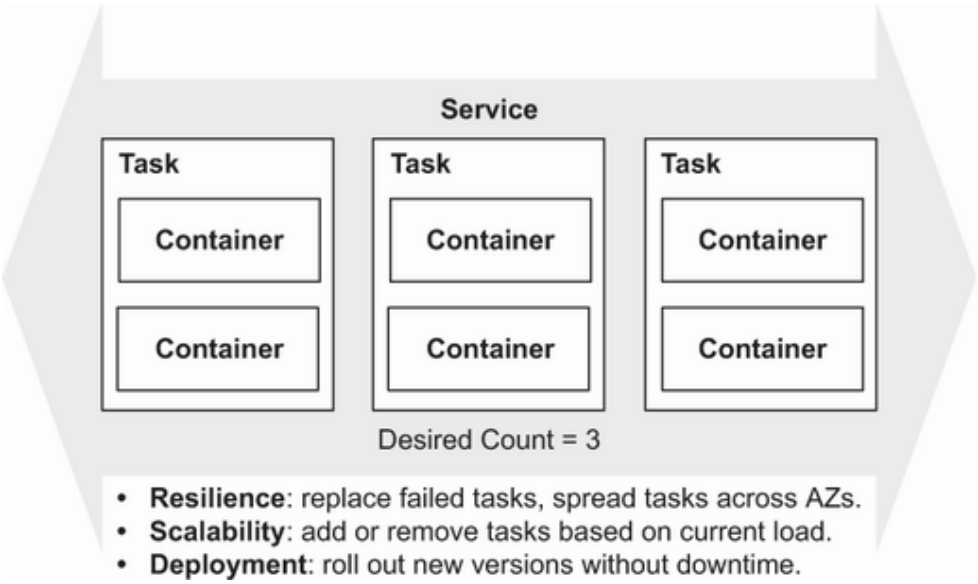

Figure 18.4 An ECS service manages multiple tasks of the same kind.

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.

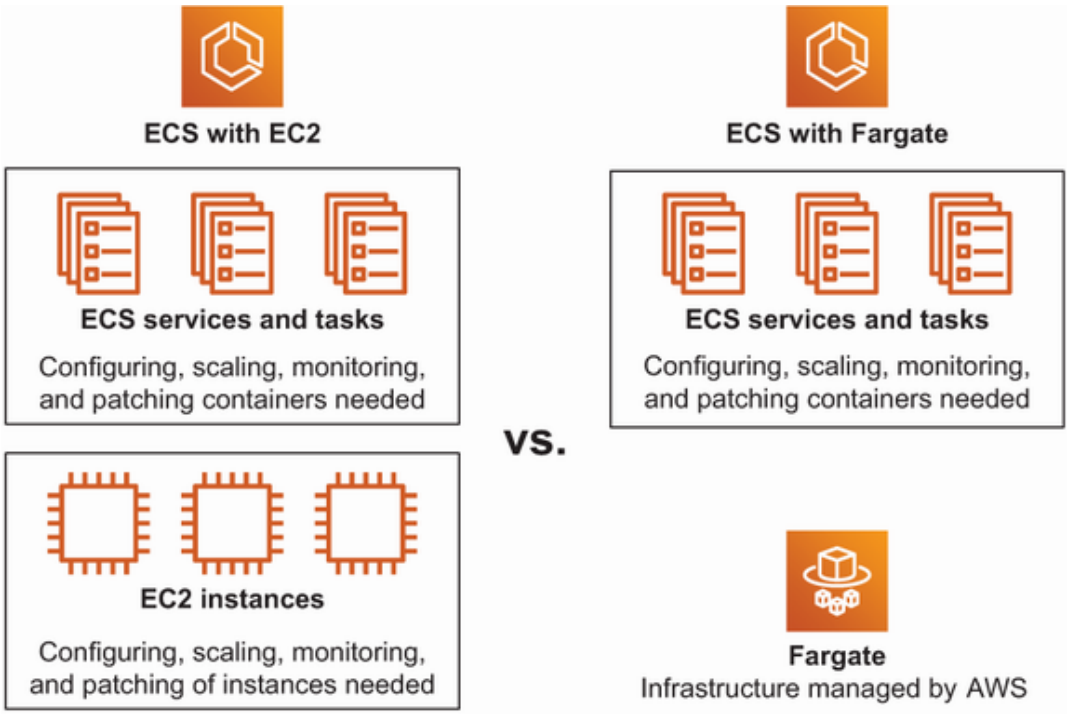

Figure 18.5 With Fargate, you do not need to manage a fleet of EC2 instances to deploy containers with ECS.

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.

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.

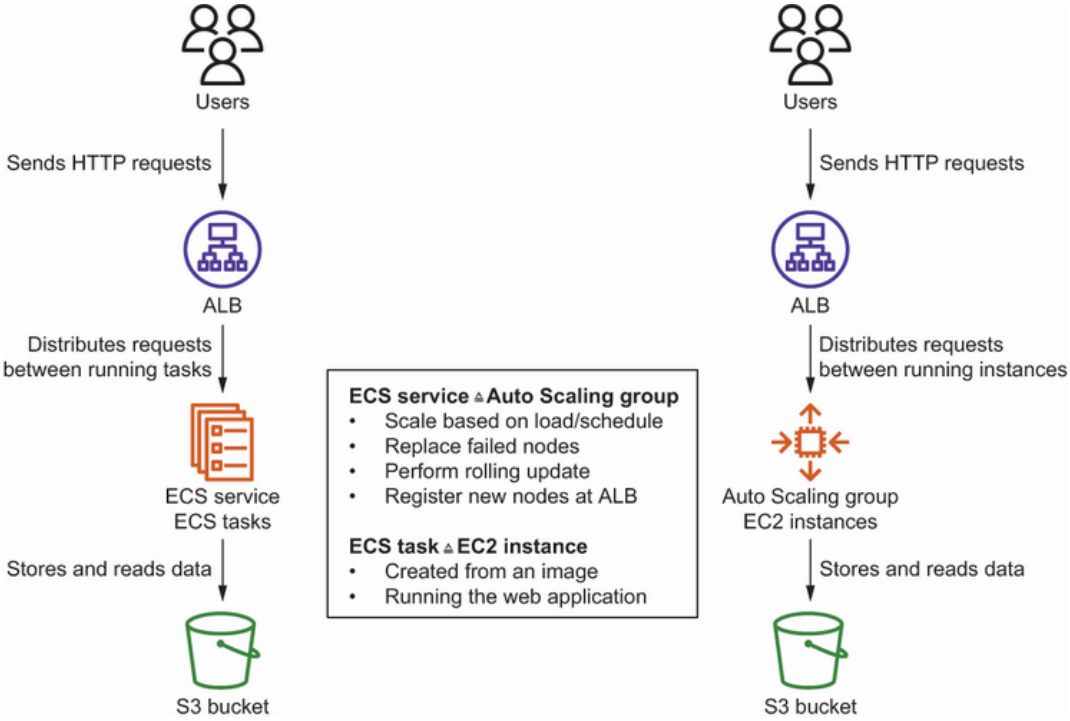

Figure 18.6 ECS is for containers what an Auto Scaling group is for EC2 instances.

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. 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!

Figure 18.7 Notea is up and running!

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.

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:                                    ①
    - Name: app                                              ②
      Image: 'public.ecr.aws/s5r5a1t5/notea:latest'          ③
      PortMappings:
      - ContainerPort: 3000                                  ④
        Protocol: tcp
      Essential: true
      LogConfiguration:                                      ⑤
        LogDriver: awslogs
        Options:
          'awslogs-region': !Ref 'AWS::Region'
```

```
            'awslogs-group': !Ref LogGroup
            'awslogs-stream-prefix': app
        Environment:                                                    ⑥
        - 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                                                          ⑦
      ExecutionRoleArn: !GetAtt 'TaskExecutionRole.Arn'                 ⑧
      Family: !Ref 'AWS::StackName'
      Memory: 1024                                                      ⑨
      NetworkMode: awsvpc                                               ⑩
      RequiresCompatibilities: [FARGATE]                                ⑪
      TaskRoleArn: !GetAtt 'TaskRole.Arn'                               ⑫
```

① Remember that a task definition describes one or multiple containers? In this example, there is only one container, called app.

② We will reference the container named app later.

③ The URL points to a publicly hosted container image bundling the Notea app.

④ The container starts a server on port 3000.

⑤ The log configuration tells the container to ship logs to CloudWatch, which is the default for ECS and Fargate.

⑥ The notea container expects a few environment variables for configuration. Those environment variables are configured here.

⑦ Tells Fargate to provision 0.5 vCPUs for our task

⑧ The IAM role is used by Fargate to fetch container images, ship logs, and similar tasks.

⑨ Tells Fargate to assign 1024 MB memory to our task

⑩ 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.

⑪ Specifies that the task definition should be used with Fargate only

⑫ The IAM role used by the application to access S3

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'      ①
        Action: 'sts:AssumeRole'
    Policies:
    - PolicyName: S3AccessPolicy
      PolicyDocument:
        Statement:
        - Effect: Allow
          Action:
          - 's3:GetObject'                        ②
          - 's3:PutObject'                        ③
          - 's3:DeleteObject'                     ④
          Resource: !Sub '${Bucket.Arn}/*'        ⑤
        - Effect: Allow
            Action:
            - 's3:ListBucket'                     ⑥
            Resource: !Sub '${Bucket.Arn}'
Bucket:                                           ⑦
  Type: 'AWS::S3::Bucket'
  Properties:
    BucketName: !Sub 'awsinaction-notea-${ApplicationID}'
```

① The IAM role is used by ECS tasks only; therefore, we need to allow the ECS tasks service access to assume the role.

② Authorizes the role to read data from S3

③ Authorizes the role to write data to S3

④ Authorizes the role to delete data from S3

⑤ Read and write access is granted only to Notea's S3 bucket.

⑥ Allows listing all the objects in the bucket

⑦ The S3 bucket used by Notea to store data

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'                          ①
    CapacityProviderStrategy:
    - Base: 0
      CapacityProvider: 'FARGATE'                    ②
      Weight: 1
    DeploymentConfiguration:
      MaximumPercent: 200                            ③
      MinimumHealthyPercent: 100                     ④
      DeploymentCircuitBreaker:                      ⑤
        Enable: true
        Rollback: true
    DesiredCount: 2                                  ⑥
    HealthCheckGracePeriodSeconds: 30                ⑦
    LoadBalancers:                                   ⑧
    - ContainerName: 'app'                           ⑨
      ContainerPort: 3000                            ⑩
      TargetGroupArn: !Ref TargetGroup               ⑪
    NetworkConfiguration:
      AwsvpcConfiguration:
```

```
          AssignPublicIp: 'ENABLED'                    ⑫
          SecurityGroups:
          - !Ref ServiceSecurityGroup                  ⑬
          Subnets: [!Ref SubnetA, !Ref SubnetB]        ⑭
      PlatformVersion: '1.4.0'                         ⑮
      TaskDefinition: !Ref TaskDefinition
```

① A service belongs to a cluster.

② Runs tasks on Fargate. Alternatively, you could switch to FARGATE_SPOT to reduce costs, similar to EC2 Spot Instances, as discussed in chapter 3.

③ During a deployment, ECS is allowed to double the number of tasks.

④ During a deployment, ECS ensures that the number of running containers does not decrease.

⑤ By enabling the deployment circuit breaker, you ensure that ECS will not try forever to deploy a broken version.

⑥ ECS will run or stop tasks to make sure two tasks are up and running.

⑦ 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.

⑧ The ECS service registers and unregisters tasks at the load balancer.

⑨ To be more precise, a specific container is registered at the load balancer.

⑩ The application is listening on port 3000.

⑪ The target group of the load balancer to register or deregister tasks

⑫ When deploying to a public subnet, assigning public IP addresses is required to ensure outbound connectivity.

⑬ Each tasks comes with its own ENI. The security group defined here is used to filter traffic.

⑭ 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.

⑮ 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.

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'
    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
```

① The upper limit for scaling tasks

② The lower limit for scaling tasks

③ The IAM role is required to grant Application Auto Scaling access to CloudWatch metrics and ECS.

④ Application Auto Scaling supports all kinds of services. You want to scale ECS in this example.

⑤ Scales by increasing or decreasing the desired count of the ECS service

⑥ References the ECS service in the ECS cluster created above

⑦ A simple way to scale ECS services is by using target-tracking scaling, which requires minimal configuration.

⑧ References the scalable target resource from above

⑨ The target is to keep the CPU utilization at 50%. You might want to increase that to 70–80% in real-world scenarios.

⑩ After terminating tasks, waits three minutes before reevaluating the situation

⑪ After starting tasks, waits one minute before reevaluating the situation

⑫ In this example, we scale based on CPU utilization. ECSServiceAverageMemoryUtilization is another predefined metric.

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.