Apply the same changes for the movies-store, movies-marketplace, and movies-parser Jenkinsfiles.

> **NOTE** Chapter 11 covers how to use the Jenkins Slack Notification plugin to send a notification with a changelog as an attachment.

10.4 Handling code promotion with Jenkins

Maintaining multiple Swarm cluster environments makes sense to avoid breaking things while promoting code to production. Also, having a production-like environment can help you keep a mirror of your application running in production and reproducing issues in the staging environment without impacting your clients. But this comes at a price.

> **NOTE** You can reduce the costs of the sandbox and staging environments by shutting down instances outside of regular business hours.

With that being said, create a new Swarm cluster for the staging environment in a dedicated staging VPC with a 10.2.0.0/16 CIDR block, or deploy it within the same management VPC where Jenkins is deployed, as shown in figure 10.37.

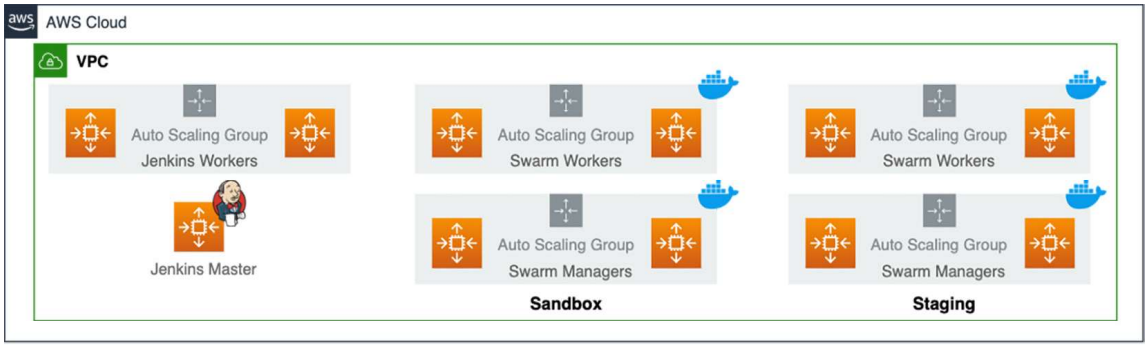

Figure 10.37 Deployment of sandbox and staging Swarm clusters and Jenkins within the same VPC

Create a preprod branch on the watchlist-deployment GitHub repository by running this command:

```
git checkout -b preprod
```

Create a docker-compose.yml file that uses the `preprod` tag, and update the SQS URL to use the staging queue, as shown in the following listing.

Listing 10.24 Docker Compose for staging deployment

```
version: "3.3"
services:
  movies-loader:
    image: ID.dkr.ecr.REGION.amazonaws.com/USER/movies-loader:preprod
```

```
    environment:
      - AWS_REGION=eu-west-3
      - SQS_URL=https://sqs.REGION.amazonaws.com/ID/movies_to_parse_staging
  movies-parser:
    image: ID.dkr.ecr.REGION.amazonaws.com/USER/movies-parser:preprod
```

Create a Jenkins credential of type SSH Username with Private Key with the SSH key pair used to deploy the Swarm staging cluster. Give it a name of `swarm-staging`, as shown in figure 10.38.

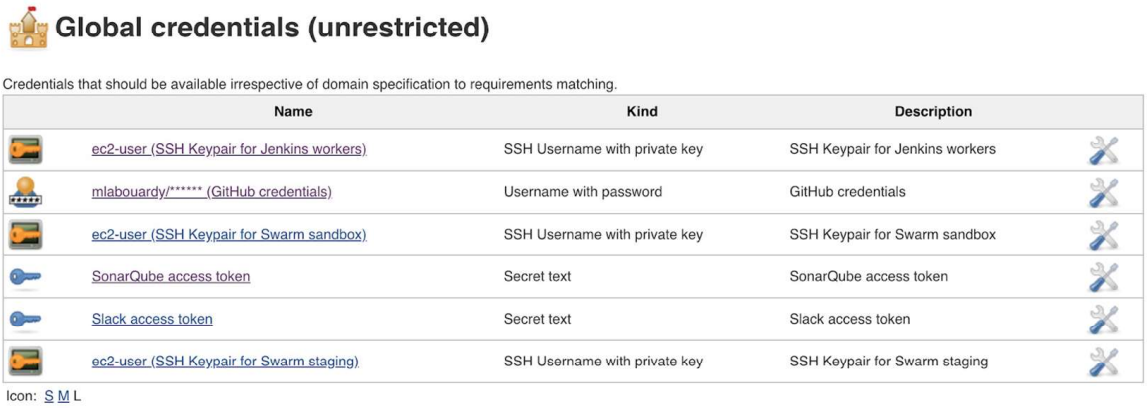

Figure 10.38 Swarm staging cluster SSH credentials

Create a Jenkinsfile similar to the one in the develop branch, as shown in the following listing. Update the `swarmManager` variable to reference the manager staging the IP or DNS record instead. Also update the SSH agent credentials to use the Swarm staging credential.

Listing 10.25 Jenkinsfile for staging deployment

```
def swarmManager = 'manager.staging.domain.com'
def region = 'AWS REGION'

node('master'){
    stage('Checkout'){
        checkout scm
    }

     sshagent (credentials: ['swarm-staging']){
         stage('Copy'){
            sh "scp -o StrictHostKeyChecking=no
docker-compose.yml ec2-user@${swarmManager}:/home/ec2-user"
        }

        stage('Deploy stack'){
            sh "ssh -oStrictHostKeyChecking=no
ec2-user@${swarmManager}
```

Swarm manager DNS alias record or private IP address

AWS region where the ECR repositories are created

Copies docker-compose.yml to the Swarm manager instance over SSH

```
'\$(\$(aws ecr get-login --no-include-email --region ${region}))'
|| true"
            sh "ssh -oStrictHostKeyChecking=no
ec2-user@${swarmManager}
docker stack deploy --compose-file
docker-compose.yml --with-registry-auth watchlist"
        }
    }
}
```

Authenticates with ECR and redeploys the application stack over SSH

Push the changes to the preprod branch. A new preprod nested job should be triggered on the watchlist-deployment item on Jenkins upon the push event, as shown in figure 10.39.

Figure 10.39 Stack deployment on staging

At the end of the pipeline, the application stack will be deployed to Swarm staging. Similarly, to access the application, use Terraform to deploy a public load balancer for the marketplace and the store API.

Finally, to trigger autodeployment on preprod, we need to update the Jenkinsfile for each project to trigger the watchlist-deployment on preprod—for example, for movies-loader Jenkinsfile. We build and push a Docker image with the `preprod` tag, as shown in the next listing.

Listing 10.26 Tagging a Docker image based on the Git branch

```
stage('Push'){
    sh "\$(aws ecr get-login --no-include-email --region ${region}) || true"
    docker.withRegistry("https://${registry}") {
        docker.image(imageName).push(commitID())
        if (env.BRANCH_NAME == 'develop') {
            docker.image(imageName).push('develop')
        }
        if (env.BRANCH_NAME == 'preprod') {
            docker.image(imageName).push('preprod')
        }
    }
}
```

Authenticates with ECR by using AWS CLI

Tags the image with the current Git commit ID and stores it in ECR

Based on the current Git branch name, the Docker image is tagged with a unique tag.

In the following listing, we update the `Deploy` stage's `if` clause condition to trigger the deployment of the external job if the branch name is preprod.

Listing 10.27 Triggering external deployment job

```
stage('Deploy'){
    if(env.BRANCH_NAME == 'develop' || env.BRANCH_NAME == 'preprod'){
        build job: "watchlist-deployment/${env.BRANCH_NAME}"
    }
}
```

Push the changes to the develop branch. Then create a pull request to merge develop to the preprod branch after Jenkins posts the build status regarding develop changes (figure 10.40).

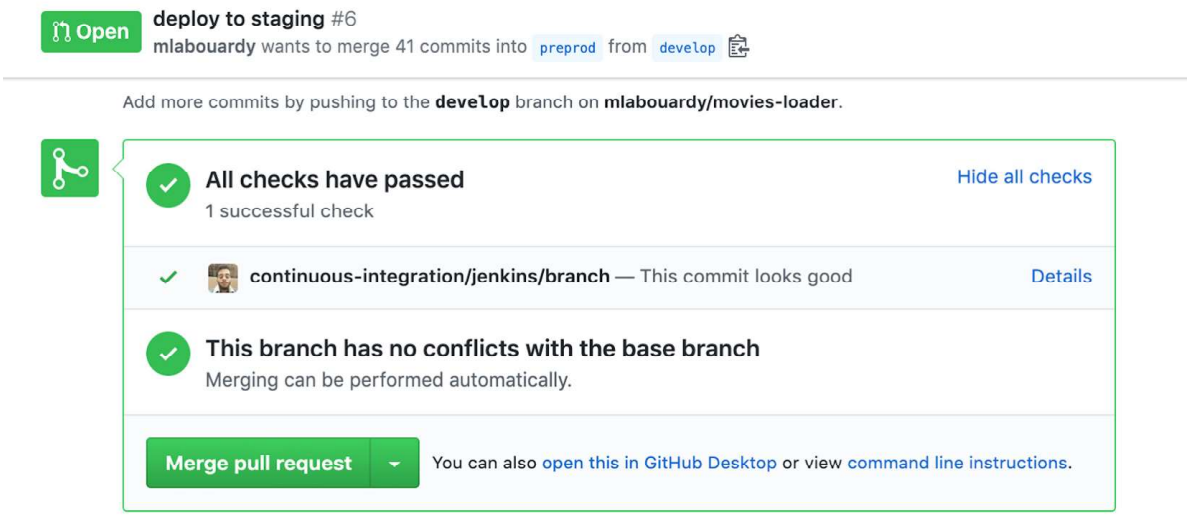

Figure 10.40 Pull request build status

When the merge occurs, a new build should be triggered on the preprod branch, as you can see in the Blue Ocean view in figure 10.41.

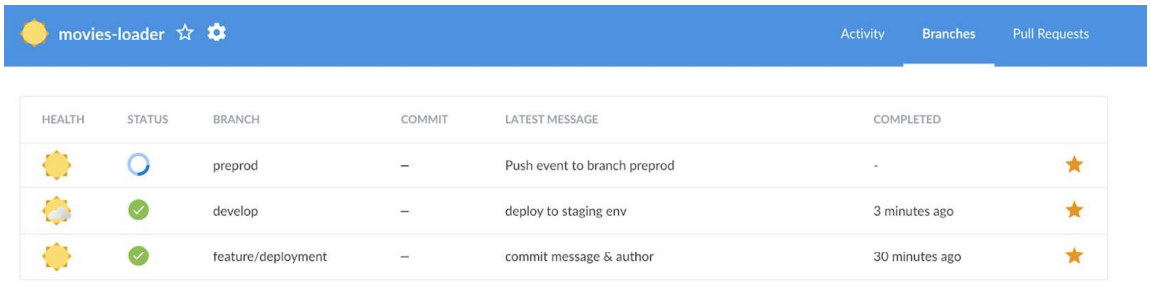

Figure 10.41 Build trigger on preprod branch

Once the `Push` stage is executed, a new image with a `preprod` tag should be pushed to the Docker registry (figure 10.42).

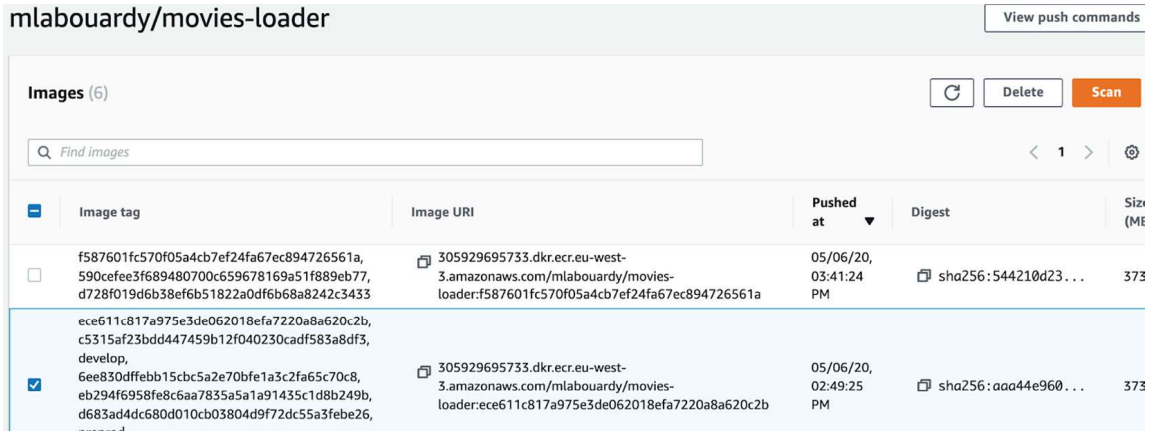

Figure 10.42 Docker image with `preprod` tag stored in ECR

Then, the deployment job on the preprod branch will be executed to deploy the changes on the Docker Swarm staging environment (figure 10.43).

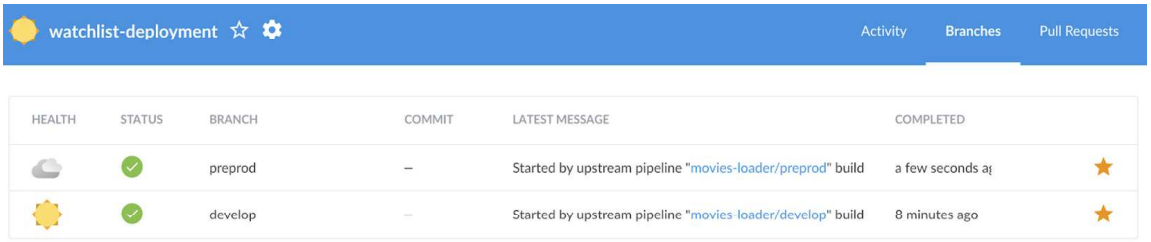

Figure 10.43 Staging deployment triggered automatically

Make the same changes for other microservices, except for movies-marketplace. For movies-marketplace, we need to update the build stage, as shown in the following listing, to inject the appropriate environment and point the frontend to the right API URL.

Listing 10.28 Injecting API URL during build

```
stage('Build'){
    switch(env.BRANCH_NAME){
        case 'develop':
            docker.build(imageName, '--build-arg ENVIRONMENT=sandbox .')
            break
        case 'preprod':
            docker.build(imageName, '--build-arg ENVIRONMENT=staging .')
            break
        default:
```

If the branch name is develop, we set the environment to sandbox, so the sandbox settings are loaded.

```
                docker.build(imageName)
        }
}
```

If the branch name doesn't match develop or preprod, the sandbox settings will be loaded by default.

Push the changes to GitHub. This time, the Docker build process will be executed with the `ENVIRONMENT` argument set to `staging` (when the current branch is preprod), as shown in figure 10.44. This will replace the environment.ts file with environment .staging.ts values.

Figure 10.44 Docker build with the environment as an argument

10.5 Implementing the Jenkins delivery pipeline

Finally, to deploy our application stack to production, you need to spin up a new Swarm cluster for the production environment. Once again, I opted to isolate the production workload in a dedicated production VPC with the 10.3.0.0/16 CIDR block and to set up a VPC peering between the management VPC (where Jenkins is located) and production VPC (where Swarm production is deployed). Figure 10.45 summarizes the deployed architecture.

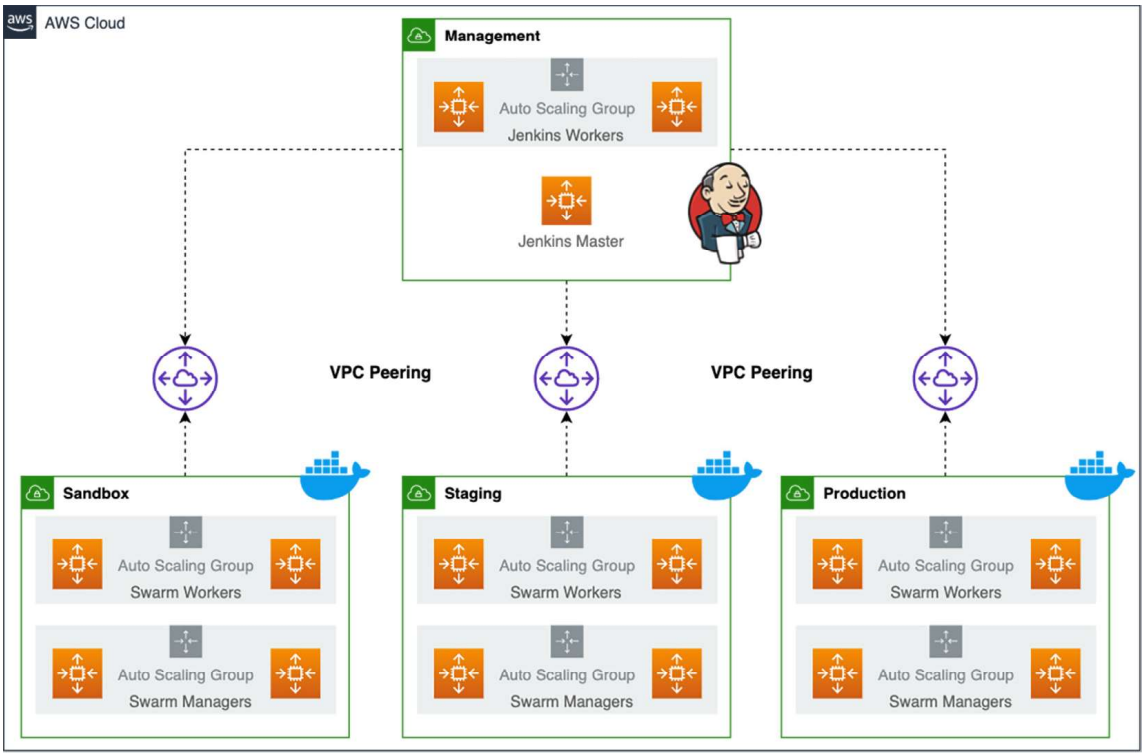

Figure 10.45 VPC peering with multiple Swarm cluster VPCs. The management VPC where the Jenkins cluster is deployed has access to the sandbox, staging, and production VPCs.

NOTE VPC peering doesn't support transitive peering. The production, staging, and sandbox environments are fully isolated, and packets cannot be routed directly from sandbox to production, for example, through the management VPC.

On the master branch of the watchlist-deployment repository, create a docker-compose .yml file. This time, we use the `latest` tag for services running in production, as shown in the next listing.

Listing 10.29 Docker Compose for production deployment

```
version: "3.3"
services:
  movies-loader:
    image: ID.dkr.ecr.REGION.amazonaws.com/USER/movies-loader:latest
    environment:
      - AWS_REGION=eu-west-3
      - SQS_URL=https://sqs.REGION.amazonaws.com/ID/
     movies_to_parse_production
  movies-parser:
    image: ID.dkr.ecr.REGION.amazonaws.com/USER/movies-parser:latest
```

Create a Jenkins credential with the SSH key used to deploy the Swarm cluster for the production environment and call it `swarm-production`, as shown in figure 10.46.

Credentials that should be available irrespective of domain specification to requirements matching.

Name	Kind	Description
ec2-user (SSH Keypair for Jenkins workers)	SSH Username with private key	SSH Keypair for Jenkins workers
mlabouardy/****** (GitHub credentials)	Username with password	GitHub credentials
ec2-user (SSH Keypair for Swarm sandbox)	SSH Username with private key	SSH Keypair for Swarm sandbox
SonarQube access token	Secret text	SonarQube access token
Slack access token	Secret text	Slack access token
ec2-user (SSH Keypair for Swarm staging)	SSH Username with private key	SSH Keypair for Swarm staging
ec2-user (SSH Keypair for Swarm production)	SSH Username with private key	SSH Keypair for Swarm production

Icon: S M L

Figure 10.46 Swarm production cluster SSH credentials

Then, create a Jenkinsfile, shown in the following listing, to remotely upload the docker-compose.yml file to the manager machine. Execute the `docker stack deploy` command to deploy the application.

Listing 10.30 Jenkinsfile for production deployment

```
def swarmManager = 'manager.production.domain.com'
def region = 'AWS REGION'
node('master'){
    stage('Checkout'){...}
```

Clones the GitHub repository—refer to listing 10.25 for instructions.

```
        sshagent (credentials: ['swarm-production']){
            stage('Copy'){...}

            stage('Deploy stack'){...}
        }
}
```

Copies docker-compose.yml to the Swarm manager over SSH—refer to listing 10.25 for instructions

Redeploys the Docker Compose stack over SSH—refer to listing 10.25 for instructions

Push the changes to the master branch. The GitHub repository should look like figure 10.47.

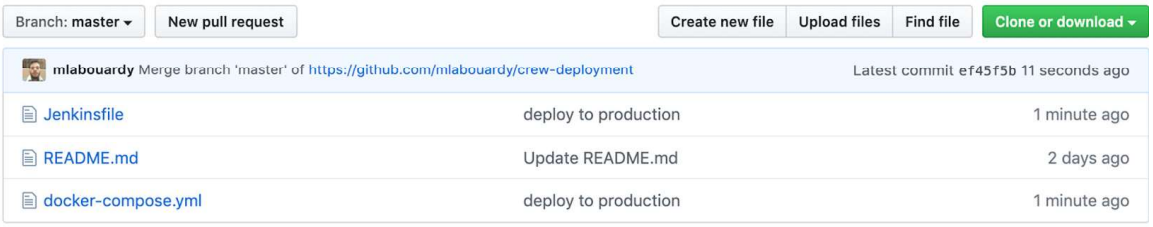

Figure 10.47 Deployment files stored in the GitHub repository

The Jenkins pipeline will be triggered on the master branch. Once the pipeline is finished, the application stack will be deployed to the production environment, as you can see in figure 10.48.

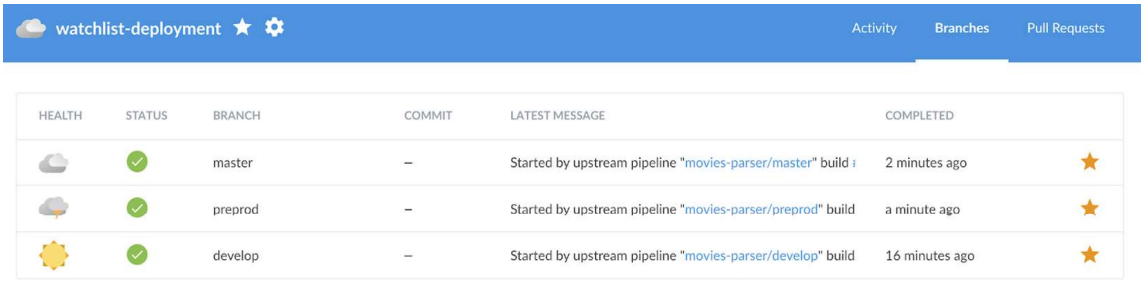

Figure 10.48 Deployment triggered in the master branch

To trigger the deployment of production at the end of the CI pipeline, update the GitHub repository to trigger the deployment job if the current branch is master. For instance, update the movies-loader's Jenkinsfile to build the image for production and push the result to the Docker registry with the `latest` tag, as shown in the following listing.

Listing 10.31 Tagging the production image

```
stage('Push'){
    sh "\$(aws ecr get-login --no-include-email --region ${region}) || true"
    docker.withRegistry("https://${registry}") {
```

```
        docker.image(imageName).push(commitID())
        if (env.BRANCH_NAME == 'develop') {
            docker.image(imageName).push('develop')
        }
        if (env.BRANCH_NAME == 'preprod') {
            docker.image(imageName).push('preprod')
        }
        if (env.BRANCH_NAME == 'master') {
            docker.image(imageName).push('latest')
        }
    }
}
```

For the deployment part, we can simply update the `if` clause to support deployment on the master branch too:

```
stage('Deploy'){
            if(env.BRANCH_NAME == 'develop'
|| env.BRANCH_NAME == 'preprod'
|| env.BRANCH_NAME == 'master'){
                build job: "watchlist-deployment/${env.BRANCH_NAME}"
            }
}
```

However, we want to require manual validation before deploying to production to simulate the product/business validation (or QA team running tests before approving for production) before deploying releases to production.

To do so, you can use the Input Step plugin to pause the pipeline execution and allow the user to interact and control the deployment process to production, as shown in the following listing.

Listing 10.32 Requiring user approval before production deployment

```
stage('Deploy'){
   if(env.BRANCH_NAME == 'develop' || env.BRANCH_NAME == 'preprod'){
      build job: "watchlist-deployment/${env.BRANCH_NAME}"
   }
   if(env.BRANCH_NAME == 'master'){
      timeout(time: 2, unit: "HOURS") {
         input message: "Approve Deploy?", ok: "Yes"
      }
      build job: "watchlist-deployment/master"
   }
}
```

Here, we set the time-out to be 2 hours to give developers enough time to validate the release. When the 2-hour time-out is reached, the pipeline will be aborted.

NOTE To avoid having a Jenkins worker doing nothing for 2 hours, you can move the `Deploy` stage outside a node block. You can also send a Slack reminder when waiting for user input.

Push the changes to a feature branch, and raise a pull request to merge changes to the develop branch after the feature branch is successfully built and approved by Jenkins (figure 10.49).

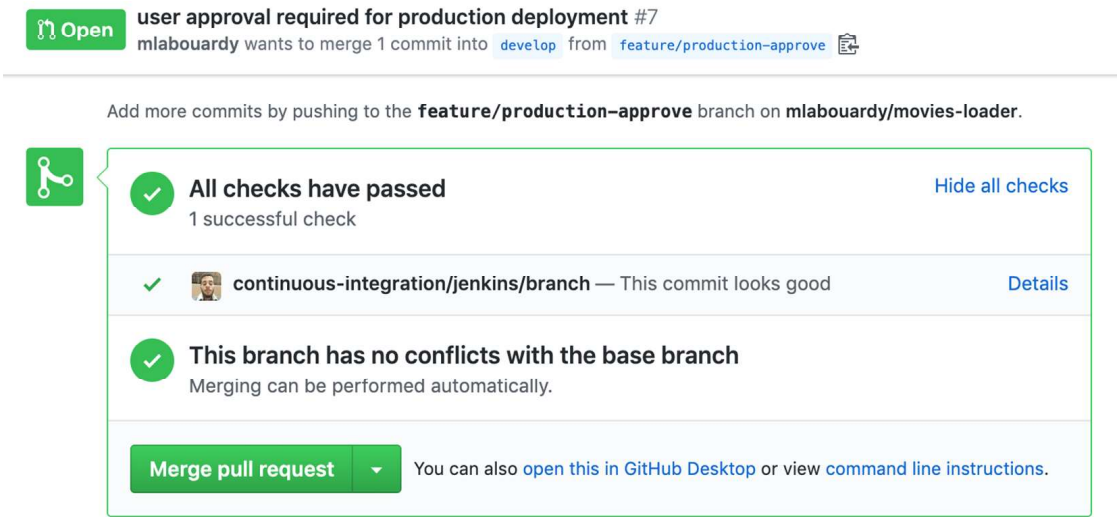

Figure 10.49 Merging the feature branch into develop

Merge the changes to the develop branch and delete the feature branch. A new build should be triggered on the develop branch, which will deploy the image to the Swarm sandbox cluster; see figure 10.50.

Figure 10.50 Deployment to sandbox triggered

Next, raise a pull request to merge develop into the preprod branch (figure 10.51).

Once the PR is merged, a new build will be triggered on the preprod branch, at the end of the CI/CD pipeline. The changes will be deployed into the Swarm staging cluster, as shown in figure 10.52.

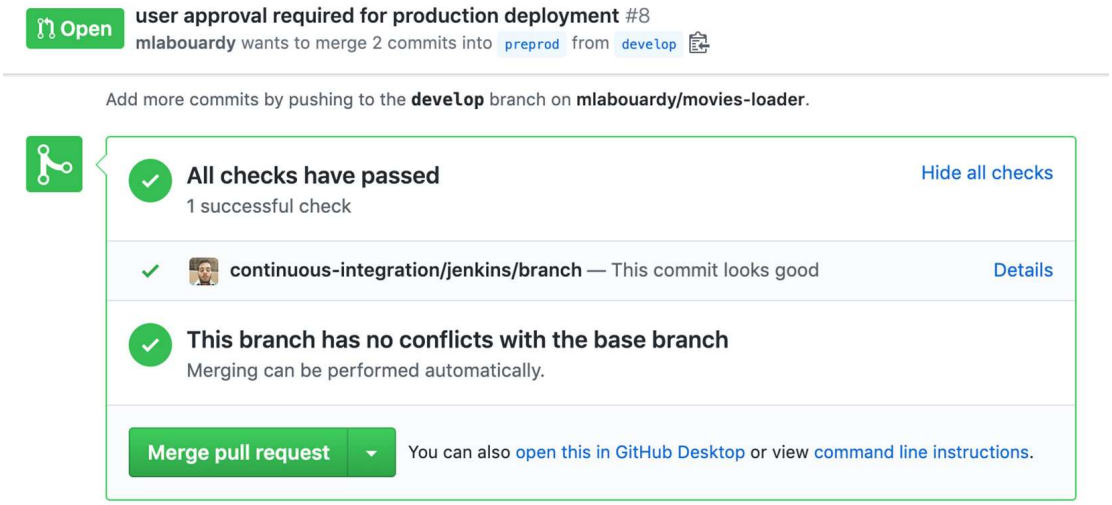

Figure 10.51 Merging the develop branch into preprod

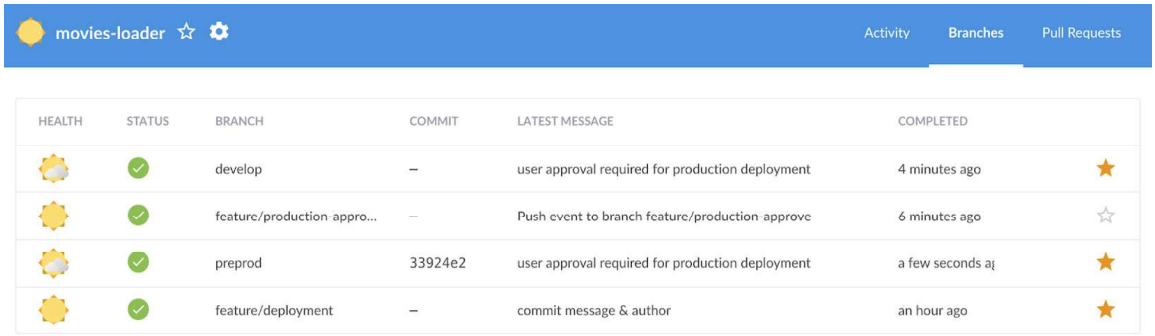

Figure 10.52 Deployment to staging cluster triggered

Finally, create a pull request to merge preprod into the master branch (figure 10.53).

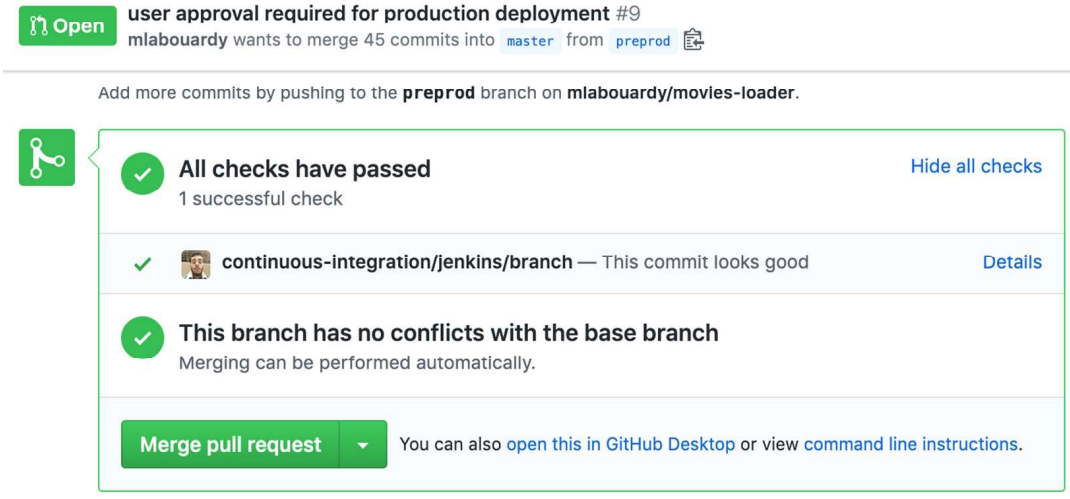

Figure 10.53 Merging the preprod branch into master

When the merge occurs, Jenkins will trigger a build on the master branch of the movies-loader service, as illustrated in figure 10.54. However, this time, once it reaches the deploy stage, an input dialog will pop up for deployment confirmation.

Figure 10.54 CI/CD pipeline execution on the master branch

As you can see in figure 10.55, the interactive input will ask whether we approve the deployment.

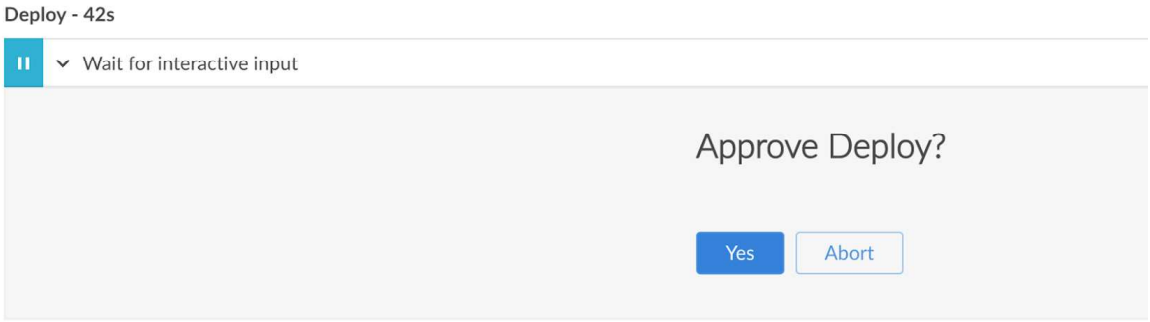

Figure 10.55 Deployment user input dialog

If we click Yes, the pipeline will be resumed, and the deployment job will be triggered on the master, as shown in figure 10.56.

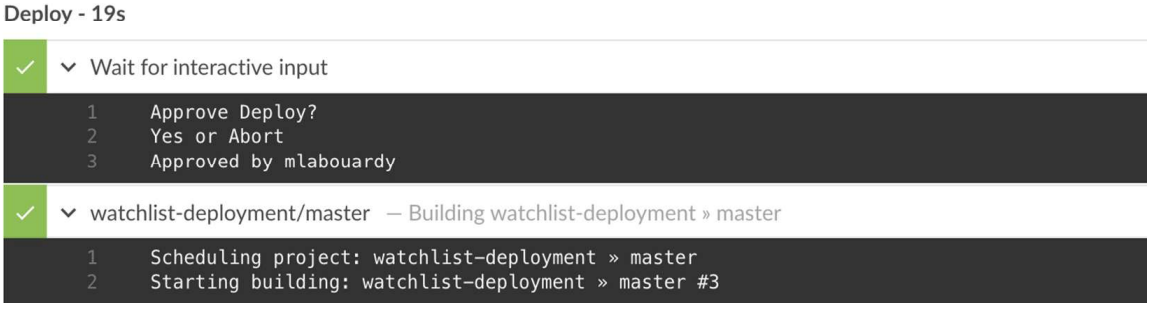

Figure 10.56 Production deployment approval

At the end of the deployment process, the new stack will be deployed to Swarm production, and a Slack notification will be sent to the configured Slack channel (figure 10.57).

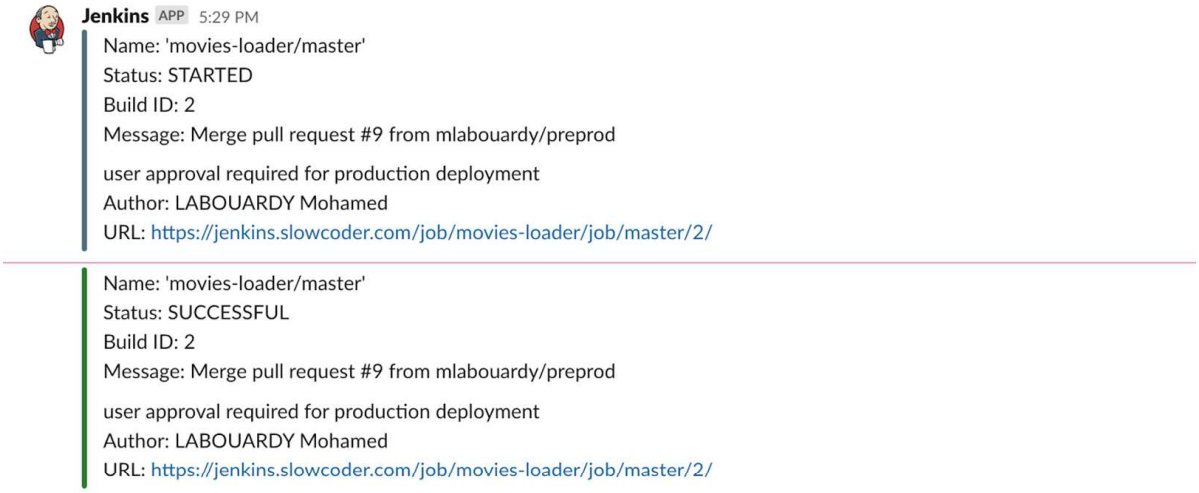

Figure 10.57 Production deployment success notification

With the production deployment covered, you have seen how to deploy containerized microservice applications to multiple environments and how to handle code promotion within a CI/CD pipeline. However, because we're managing only three environments (sandbox, staging, and production), we will limit the discovering behavior of the deployment job to the three main branches by defining a regular expression, as shown in figure 10.58.

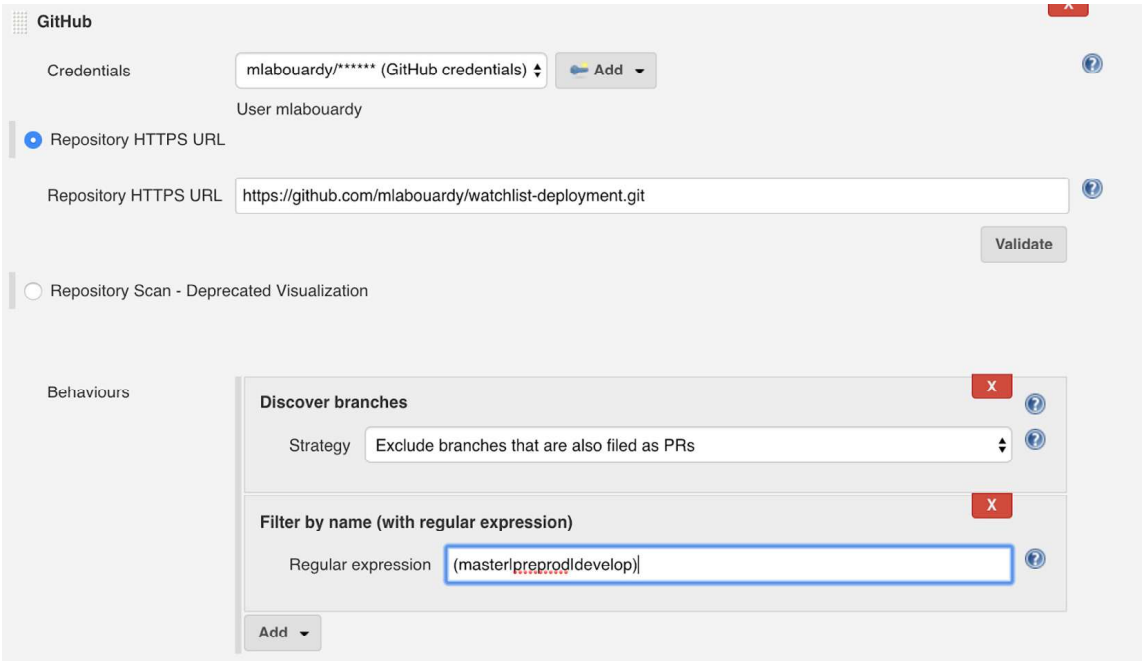

Figure 10.58 Jenkins discovery behavior based on a regular expression

As a result, Jenkins will discover and be triggered only if one of the three main branches has changed; see figure 10.59.

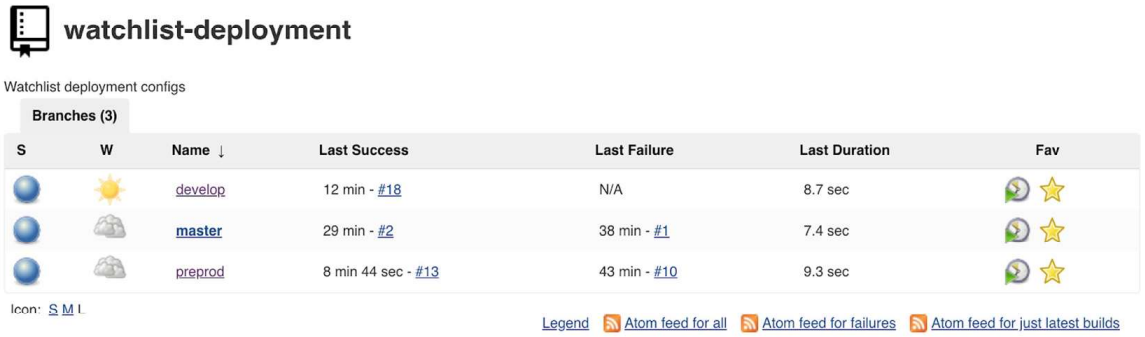

Figure 10.59 Deployment multibranch job

So now if we make any change to our application, CI/CD pipelines will be triggered and `docker stack deploy` will be executed, which will update any services that were changed from the previous version.

> **NOTE** If the deployment target is one single host, a swarm is not needed. The same docker-compose.yml and procedure explained in this chapter should be sufficient to continuously deploy your application on a single-host deployment environment.

Summary

- An S3 bucket or distributed consistent key-value store such as etcd, Consul, or ZooKeeper can be used as service discovery to make the nodes autojoin a Swarm cluster.
- Continuous deployment of containers on a Swarm cluster can be reached by executing `docker stack deploy` over SSH on a Swarm manager.
- Adding Slack notifications within CI/CD pipelines makes the product delivery faster. The sooner the team members are aware of a build, integration, or deployment failure, the quicker they can act.
- To simulate business/product validation before deploying a production release, the Jenkins Input Step plugin can prompt the user for manual validation before deployment.

11 Dockerized microservices on K8s

This chapter covers

- Setting up a Kubernetes cluster on AWS with Terraform
- Automating application deployment on Kubernetes with Jenkins pipelines
- Packaging and versioning Kubernetes Helm charts
- Converting Compose files to Kubernetes manifests with Kompose
- Running post-deployment tests and health checks within CI/CD pipelines
- Discovering Jenkins X and setting up serverless CI/CD pipelines

The preceding chapter covered how to set up a CI/CD pipeline from scratch for containerized applications running in Docker Swarm (figure 11.1). This chapter covers how to deploy the same application in Kubernetes (K8s) and automate the deployment. In addition, you'll learn how to use Jenkins X to simplify the workflow of cloud-native applications running in Kubernetes.

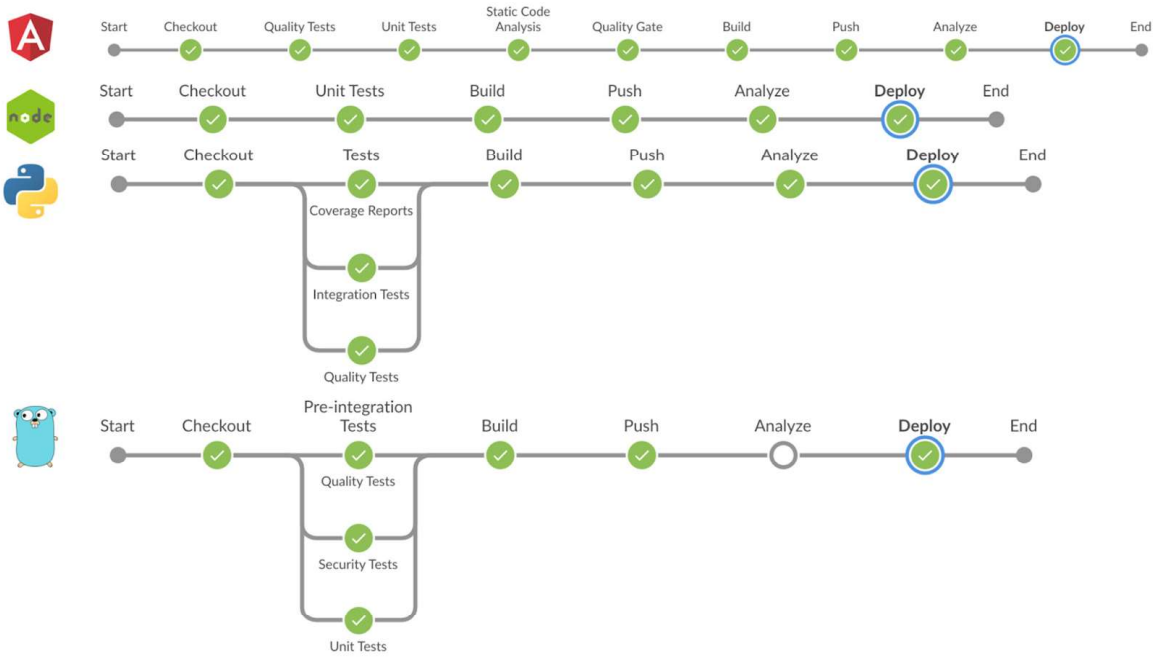

Figure 11.1 Current CI/CD pipeline workflow

Docker Swarm might be a good solution for beginners and smaller workloads. However, for large deployment and at a certain scale, you might want to consider shifting to Kubernetes.

For those of you who are AWS power users, Amazon Elastic Kubernetes Service (EKS) is a natural fit. Other cloud providers offer managed Kubernetes solutions, including Azure Kubernetes Service (AKS) and Google Kubernetes Engine (GKE).

11.1 Setting up a Kubernetes cluster

As I've said, AWS offers the Amazon Elastic Kubernetes Service (https://aws.amazon.com/eks). The EKS cluster will be deployed in a custom VPC within multiple private subnets. EKS runs the Kubernetes control plane for you across multiple AWS availability zones to eliminate a single point of failure, as shown in figure 11.2.

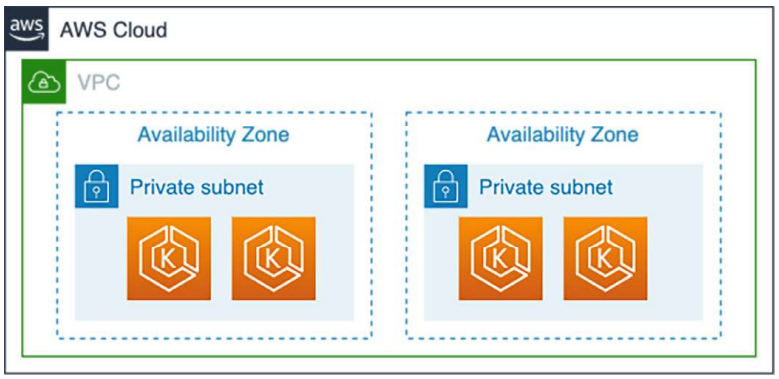

Figure 11.2 The AWS EKS architecture consists of node groups deployed in private subnets.

A few tools (including AWS CloudFormation, eksctl, and kOps) allow you to get up and running quickly on EKS. In this chapter, we picked Terraform because we were already using it to manage our Jenkins cluster on AWS.

To get started, provision a new VPC to host the sandbox environment and divide it into two private subnets. Amazon EKS requires subnets in at least two availability zones. The VPC is created to isolate the Kubernetes workload. For EKS to discover the VPC subnets and manage network resources, we tag them with `kubernetes.io/cluster/<cluster-name>`. The `<cluster-name>` value matches the EKS cluster's name, which is `sandbox`. Create a file called vpc.tf with the content in the following listing.

Listing 11.1 Kubernetes custom VPC

```
resource "aws_vpc" "sandbox" {
  cidr_block           = var.cidr_block
  enable_dns_hostnames = true
  tags = {
    Name    = var.vpc_name
    Author = var.author
    "kubernetes.io/cluster/${var.cluster_name}" = "shared"
  }
}
```

Then, define the subnets and set up the appropriate route tables. Refer to chapter11/eks/vpc.tf for the full source code, or head back to chapter 10 for a step-by-step guide on how to deploy a custom VPC on AWS.

Next, we create a new eks_masters.tf file and define the `sandbox` EKS cluster, which is a managed K8s control plane, as shown in the following listing.

Listing 11.2 EKS sandbox cluster

```
resource "aws_eks_cluster" "sandbox" {
  name            = var.cluster_name
  role_arn        = aws_iam_role.cluster_role.arn
  vpc_config {
    security_group_ids = [aws_security_group.cluster_sg.id]
    subnet_ids         = [for subnet in aws_subnet.private_subnets :
     subnet.id]
  }
  depends_on = [
    aws_iam_role_policy_attachment.cluster_policy,
    aws_iam_role_policy_attachment.service_policy,
  ]
}
```

The managed control plane uses an IAM role with the AmazonEKSClusterPolicy and AmazonEKServicePolicy policies. These attachments grant the cluster the permissions it needs to take care of itself.

Now it's time to spin up some worker nodes. A node is a simple EC2 instance that runs the Kubernetes objects (pods, deployments, services, and so forth). The master's

automatic scheduling takes into account the available resources on each node. Define an EKS node group resource within eks_workers.tf as shown in the following listing.

Listing 11.3 Kubernetes node group resource

```
resource "aws_eks_node_group" "workers_node_group" {
  cluster_name    = aws_eks_cluster.sandbox.name
  node_group_name = "${var.cluster_name}-workers-node-group"
  node_role_arn   = aws_iam_role.worker_role.arn
  subnet_ids      = [for subnet in aws_subnet.private_subnets : subnet.id]
  scaling_config {
    desired_size = 2
    max_size     = 5
    min_size     = 2
  }
  depends_on = [
    aws_iam_role_policy_attachment.worker_node_policy,
    aws_iam_role_policy_attachment.cni_policy,
    aws_iam_role_policy_attachment.ecr_policy,
  ]
}
```

We also create an IAM role that the worker nodes are going to assume. We grant the AmazonEKSWorkerNodePolicy, AmazonEKS_CNI_Policy, and AmazonEC2ContainerRegistryReadOnly policies. Refer to chapter11/eks/eks_workers.tf for the full source code.

> **NOTE** This section assumes that you are familiar with the usual Terraform plan/apply workflow; if you're new to Terraform, refer first to chapter 5.

Lastly, define the variables listed in table 11.1 in the variables.tf file.

Table 11.1 EKS Terraform variables

Variable	Type	Value	Description
`region`	String	None	The name of the region, such as `eu-central-1`, in which to deploy the EKS cluster
`shared_credentials_file`	String	`~/.aws/credentials`	The path to the shared credentials file. If this is not set and a profile is specified, `~/.aws/credentials` will be used.
`aws_profile`	String	`profile`	The AWS profile name as set in the shared credentials file
`author`	String	None	Name of the owner of the EKS cluster. It's optional, but recommended, to tag your AWS resources to track the monthly costs by owner or environment.
`availability_zones`	List	None	Availability zone for spinning up the VPC subnets

Table 11.1 EKS Terraform variables *(continued)*

Variable	Type	Value	Description
vpc_name	String	sandbox	The name of the VPC
cidr_block	String	10.1.0.0/16	The VPC CIDR block
cluster_name	String	sandbox	The EKS cluster's name
public_subnets_count	Number	2	The number of public subnets to create
private_subnets_count	Number	2	The number of private subnets to create

Then, issue the `terraform init` command to initialize a working directory and download the AWS provider plugin. In your initialized directory, run `terraform plan` to review the planned actions. Your terminal output should indicate that the plan is running and the resources that will be created. This should include the EKS cluster, VPC, and IAM roles.

If you're comfortable with the execution plan, confirm the run with `terraform apply`. This provisioning process should take a few minutes. Upon successful deployment, a new EKS cluster for the sandbox environment will be deployed and available in the AWS EKS console, as shown in figure 11.3.

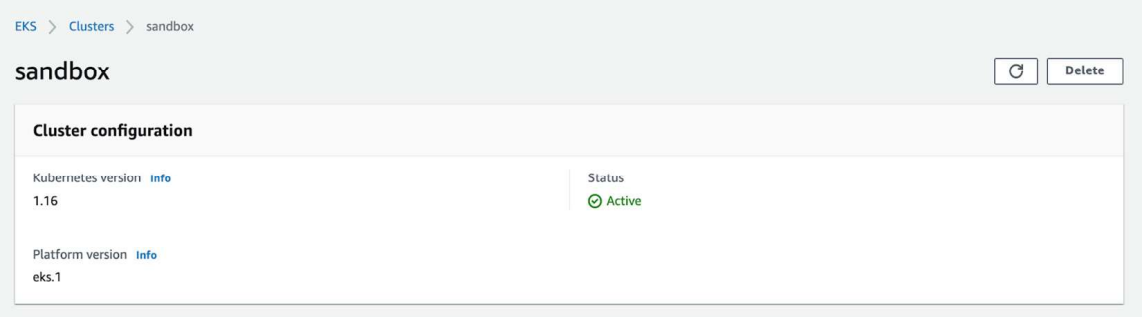

Figure 11.3 EKS sandbox cluster

Now that you've provisioned your EKS cluster, you need to configure kubectl. This is a command-line utility for communicating with the cluster API server. At the time of writing this book, I'm using version v1.18.3.

> **NOTE** The kubectl tool is available in many operating system package managers; refer to the official documentation (https://kubernetes.io/docs/tasks/tools/) for installation instructions.

To grant kubectl access to the K8s API, we need to generate a kubeconfig file (located under .kube/config in your home directory). You can create or update a kubeconfig

file with the AWS CLI `update-kubeconfig` command. Issue this command to get the access credentials for your cluster:

```
aws eks update-kubeconfig --name sandbox --region AWS_REGION
```

To verify that your cluster is configured correctly and running, execute the following command:

```
kubectl get nodes
```

The output will list all of the nodes in a cluster and the status of each node:

```
[jenkins:eks mlabouardy$ kubectl get nodes
NAME                                          STATUS   ROLES    AGE   VERSION
ip-10-1-0-25.eu-west-3.compute.internal    Ready    <none>   73s   v1.15.10-eks-bac369
ip-10-1-2-225.eu-west-3.compute.internal   Ready    <none>   69s   v1.15.10-eks-bac369
```

> **NOTE** To optimize K8s costs, you can use EC2 Spot instances, as they cost about 30–70% less than their on-demand counterparts. However, this requires some special considerations, as they could be terminated with only a 2-minute warning.

At this point, you should be able to use Kubernetes. In the next section, we will automate the deployment of the Watchlist application described in chapter 7 into the K8s cluster with Jenkins following the PaC approach.

11.2 Automating continuous deployment flow with Jenkins

To complete a Kubernetes deployment from Jenkins, all we need are K8s deployment files, which will contain references to the Docker images, along with the configuration settings (for example, port, network name, labels, and constraints). To run this file, we will need to execute the `kubectl apply` command.

On the develop branch of the watchlist-deployment GitHub repository, create a deployments folder. Inside it, create a movies-loader-deploy.yaml file by using your favorite text editor or IDE, with the content in the following listing. The deployment instructs Kubernetes on how to create and update the movies-loader service.

Listing 11.4 Movie loader deployment resource

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: movies-loader
  namespace: watchlist
spec:
  selector:
    matchLabels:
      app: movies-loader
  template:
    metadata:
      labels:
        app: movies-loader
```

```
    spec:
      containers:
      - name: movies-loader
        image: ID.dkr.ecr.REGION.amazonaws.com/USER/movies-loader:develop
        env:
        - name: AWS_REGION
          value: REGION
        - name: SQS_URL
          value: https://sqs.REGION.amazonaws.com/ID/movies_to_parse_sandbox
```

NOTE As a reminder, the movies-loader and movies-store services are using Amazon SQS to load and consume movie items, respectively. To grant those services permission to interact with SQS, you need to assign the AmazonSQS-FullAccess policy to the EKS node group.

The movies-loader service can be deployed to Kubernetes through a deployment resource. The deployment definition uses the `develop` tag of the movies-loader Docker image and defines a set of environment variables, such as the SQS URL and AWS region. The MongoDB resource can also be deployed with the mongodb-deploy.yaml file in the following listing.

Listing 11.5 MongoDB deployment resource

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongodb
  namespace: watchlist
spec:
  selector:
    matchLabels:
      app: mongodb
  template:
    metadata:
      labels:
        app: mongodb
    spec:
      containers:
      - name: mongodb
        image: bitnami/mongodb:latest
        env:
        - name: MONGODB_USERNAME
          valueFrom:
            secretKeyRef:
              name: mongodb-access
              key: username
        - name: MONGODB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mongodb-access
              key: password
        - name: MONGODB_DATABASE
          valueFrom:
            secretKeyRef:
              name: mongodb-access
              key: database
```

The most interesting thing about this deployment definition is the environment variables part. Instead of hardcoding the MongoDB credentials, we are using K8s secrets. We're creating secret store authentication credentials so only Kubernetes can access them.

Before we create a Kubernetes secret, we need to maintain a space in the Kubernetes cluster where we can get a view on the list of pods, services, and deployments we use to build and run the application. We will create a dedicated namespace to associate all of our Kubernetes objects with the following command:

```
kubectl create namespace watchlist
```

Then, invoke the following Kubernetes command on your local machine to create MongoDB credentials secrets:

```
kubectl create secret generic mongodb-access --from-
     literal=database='watchlist'
--from-literal=username='root'
--from-literal=password='PASSWORD' -n watchlist
```

```
jenkins:chapter11 mlabouardy$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-wbzr7   kubernetes.io/service-account-token   3      59m
mongodb-access        Opaque                                4      5s
```

Create deployment files for the rest of the services: movies-store, movies-parser, and movies-marketplace. The deployments folder structure should look like this:

```
mongodb-deploy.yaml
movies-store-deploy.yaml
movies-loader-deploy.yaml
movies-parser-deploy.yaml
movies-marketplace-deploy.yaml
```

All the source code can be downloaded from the GitHub repository, under the chapter11/deployment/kubectl/deployments folder.

To deploy the application with Jenkins, create a Jenkinsfile.eks file at the top-level directory of the watchlist-deployment project, as shown in the following listing. The Jenkinsfile will configure kubectl with the `aws eks update-kubeconfig` command. Then it issues a `kubectl apply` command to deploy the deployment resources. The `kubectl apply` command takes as an argument the deployments folder.

Listing 11.6 Jenkinsfile deployment stages

```
def region = 'AWS REGION'
def accounts = [master:'production', preprod:'staging', develop:'sandbox']

node('master'){
    stage('Checkout'){
        checkout scm
    }
```

AWS region where the EKS cluster is deployed

```
    stage('Authentication'){
        sh "aws eks update-kubeconfig --name ${accounts[env.BRANCH_NAME]} --
     region ${region}"
    }

    stage('Deploy'){
        sh 'kubectl apply -f deployments/'
    }
}
```

Configures kubectl so that you can connect to an Amazon EKS cluster

Deploys the new changes to EKS

Before pushing the Jenkinsfile and deployment files to the Git remote repository, we need to install the kubectl command line on the Jenkins master. Also, we need to provide access to EKS with IAM roles. To grant Jenkins master permissions to interact with the K8s cluster, we must edit the `aws-auth` ConfigMap within Kubernetes. On your local machine, run the following command:

```
kubectl edit -n kube-system configmap/aws-auth
```

A text editor will open; add the Jenkins instance's IAM role to the `mapRoles` section. Then, save the file and exit the text editor. Check whether the ConfigMap is properly configured with the following command:

```
kubectl describe -n kube-system configmap/aws-auth
```

```
Name:         aws-auth
Namespace:    kube-system
Labels:       <none>
Annotations:  <none>

Data
====
mapRoles:
----
- groups:
  - system:bootstrappers
  - system:nodes
  rolearn: arn:aws:iam::305929695733:role/terraform-eks-demo-node
  username: system:node:{{EC2PrivateDNSName}}
- groups:
  - system:masters
  rolearn: arn:aws:iam::305929695733:role/JenkinsMasterRole
  username: system:node:{{EC2PrivateDNSName}}

Events:  <none>
```

Once the ConfigMap is configured, install aws-iam-authenticator, which is a tool to manage AWS IAM credentials for Kubernetes access. Refer to the AWS documentation at http://mng.bz/AOWW for the installation guide. Then, generate a kubeconfig with the AWS CLI `update-kubeconfig` command. The command should create a /home/ec2-user/.kube/config file with no warning. Now we can issue the `kubectl get nodes` command:

```
[ec2-user@ip-10-0-0-216 ~]$ kubectl get nodes
NAME                                          STATUS   ROLES    AGE    VERSION
ip-10-1-0-43.eu-west-3.compute.internal       Ready    <none>   4h8m   v1.16.8-eks-e16311
ip-10-1-2-182.eu-west-3.compute.internal      Ready    <none>   4h8m   v1.16.8-eks-e16311
```

Now, we're ready to push the Jenkinsfile and Kubernetes deployment files to the Git repository under the develop branch:

```
git add .
git commit -m "k8s deployment files"
git push origin develop
```

The GitHub repository content should look similar to figure 11.4 after pushing K8s deployment files.

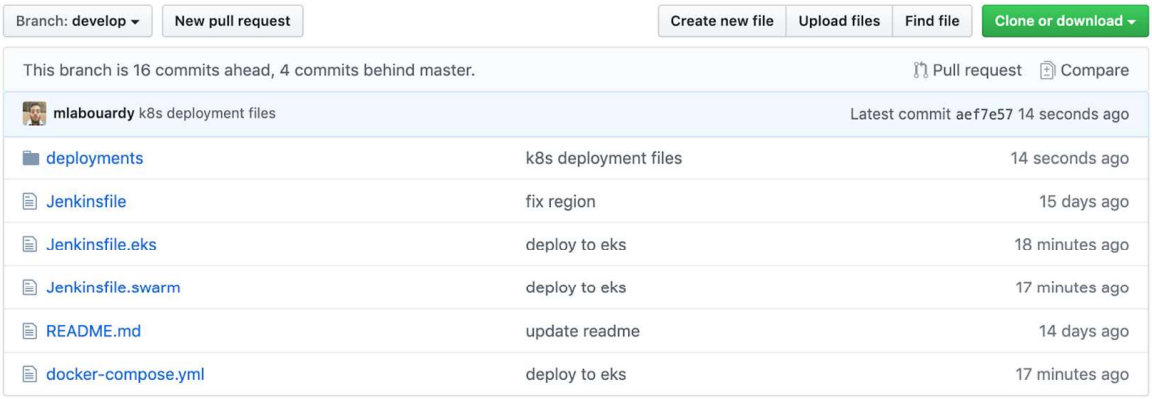

Figure 11.4 Kubernetes deployment files in the Git repository

Once the changes are committed, the GitHub webhook we created in section 7.6 will trigger a build on the watchlist-deployment multibranch job on the develop branch's nested job; see figure 11.5.

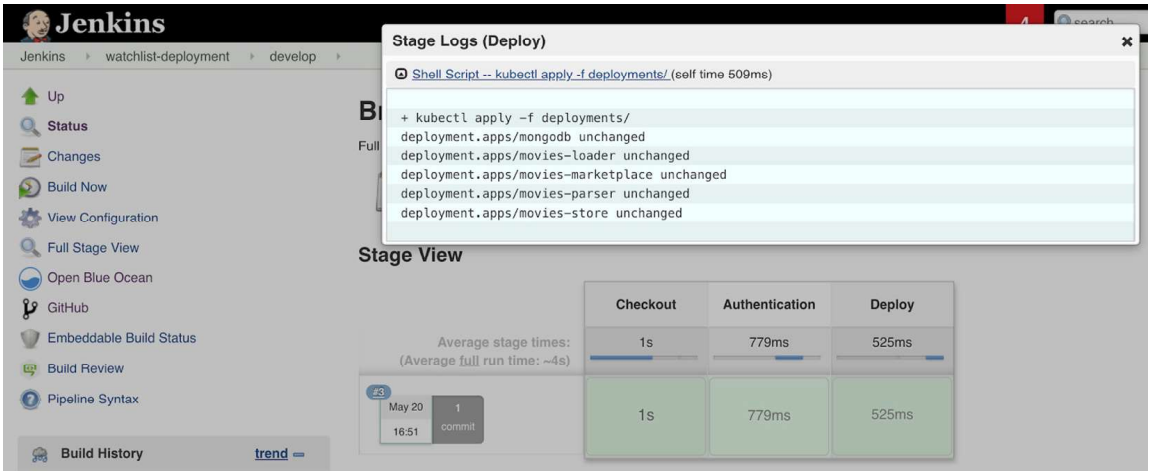

Figure 11.5 The `kubectl apply` command's output

At the `Deploy` stage, the `kubectl apply` command will be executed to deploy the application deployment resources. On your local machine, run this command to list deployments running in the sandbox K8s cluster:

```
kubectl get deployments --namespace=watchlist
```

The four components (loader, parser, store, and marketplace) of our application will be deployed alongside a MongoDB server:

```
NAME                                  READY   STATUS    RESTARTS   AGE
mongodb-7b647bdd54-rdczs              1/1     Running   0          33s
movies-loader-7895fcc9cc-vgpj4        1/1     Running   1          7s
movies-marketplace-7749dc4fd8-wtgf8   1/1     Running   0          16m
movies-parser-7d4fd8f7-9lxkk          1/1     Running   1          3m9s
movies-store-584658766b-b5b2c         1/1     Running   0          16m
```

These deployment resources are referencing Docker images stored in Amazon ECR. At the time of deploying the EKS cluster, we have granted permissions to the K8s cluster to interact with ECR. However, if your Docker images are hosted on a remote repository that requires username/password authentication, you need to create a Docker Registry secret with the following command:

```
kubectl create secret docker-registry registry
--docker-username=USERNAME
--docker-password=PASSWORD
--namespace watchlist
```

Then, you need to reference this secret in your deployment file under the `spec` section as follows:

```
spec:
  containers:
  - name: movies-loader
    image: REGISTRY_URL/USER/movies-loader:develop
  imagePullSecrets:
  - name: registry
```

Our application is deployed. To access it, we need to create a K8s service for both the marketplace and store, as shown in the following listing. Create a services directory in the root repository, and then create a service for movies-store called movies-store.svc.yaml. The service creates a cloud network load balancer (for instance, AWS Elastic Load Balancer). This provides an externally accessible IP address for accessing the Movies Store API.

Listing 11.7 Movie store service resource

```
apiVersion: v1
kind: Service
metadata:
  name: movies-store
```

```
  namespace: watchlist
spec:
  ports:
  - port: 80
    targetPort: 3000
  selector:
    app: movies-store
  type: LoadBalancer
```

Additionally, we create another service to expose the Movies Marketplace (UI). Add the content in the following listing to movies-marketplace.svc.yaml.

Listing 11.8 Movies Marketplace service resource

```
apiVersion: v1
kind: Service
metadata:
  name: movies-marketplace
  namespace: watchlist
spec:
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: movies-marketplace
  type: LoadBalancer
```

The movies-store and movies-parser services store the movie metadata in a MongoDB service. Therefore, we need to expose the MongoDB deployment through a Kubernetes service to allow MongoDB to receive incoming operations. The service is exposed to an internal IP in the cluster. The `ClusterIP` keyword makes the service reachable from only within the cluster. The MongoDB pod targeted by the service is determined by `LabelSelector`. Add the following YAML block to mongodb-svc.yaml.

Listing 11.9 Movies Marketplace service resource

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb
  namespace: watchlist
spec:
  ports:
    - port: 27017
  selector:
    app: mongodb
    tier: mongodb
  clusterIP: None
```

Finally, we update the Jenkinsfile in listing 11.6 to deploy the Kubernetes services by providing the services folder as a parameter to the `kubectl apply` command:

```
stage('Deploy'){
        sh 'kubectl apply -f deployments/'
        sh 'kubectl apply -f services/'
}
```

Push the changes to the develop branch. A new build will be triggered, and the services will be deployed, as shown in figure 11.6.

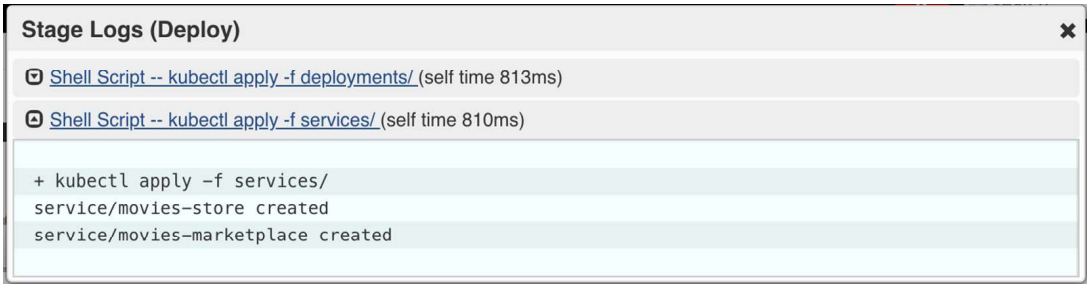

Figure 11.6 The `kubectl apply` output

Type the following command on your local machine:

```
kubectl get svc -n watchlist
```

It should show the load balancers for the three K8s services:

```
NAME                TYPE           CLUSTER-IP       EXTERNAL-IP
mongodb             ClusterIP      None             <none>
movies-marketplace  LoadBalancer   172.20.140.150   a5149491400274ac2a7b15f0140133ed-1717030020.eu-west-3.elb.amazonaws.com
movies-store        LoadBalancer   172.20.68.115    a9b485440ea404dc68a60e0973954b70-1525632852.eu-west-3.elb.amazonaws.com
```

On AWS Management Console, two public-facing load balancers should be created in the EC2 dashboard (http://mng.bz/Zx7Z), as shown in figure 11.7.

Figure 11.7 Movies Store and Marketplace ELBs

NOTE Make sure to set the load balancer FQDN in the environment.sandbox.tf file of the movies-marketplace project. The API URL will be injected while building the marketplace Docker image. Refer to section 9.1.2 for more details.

To secure access to the Store API, we can enable an HTTPS listener on the public load balancer by updating the movies-store service with the changes detailed in the following listing.

Listing 11.10 HTTPS listener configuration

```
apiVersion: v1
kind: Service
metadata:
  name: movies-store
  namespace: watchlist
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http
    service.beta.kubernetes.io/aws-load-balancer-ssl-cert:
     arn:aws:acm:{region}:{user id}:certificate/{id}
    service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "https"
spec:
  ports:
  - name: http
    port: 80
    targetPort: 3000
  - name: https
    port: 443
    targetPort: 3000
  selector:
    app: movies-store
  type: LoadBalancer
```

Used on the service to specify the protocol spoken by the backend (pod) behind a listener

Exposes port 443 (HTTPS) and forwards requests internally to port 3000 of the movies-store pod

Push the changes to the remote repository. Jenkins will deploy the changes and update the load balancer listener configuration to accept incoming traffic on port 443 (HTTPS), as shown in figure 11.8.

Load Balancer Protocol	Load Balancer Port	Instance Protocol	Instance Port	Cipher	SSL Certificate
HTTP	80	HTTP	31123	N/A	N/A
HTTPS	443	HTTP	30757	Change	fecce01b-9c10-41ae-8a1a-345d9f89efad (ACM) Change

Figure 11.8 Load balancer HTTP/HTTPS listeners

It's optional, but you can create an A record in Amazon Route 53 pointing to the load balancer FQDN and update environment.sandbox.ts to use the friendly domain name instead of the load balancer FQDN; see the following listing.

Listing 11.11 Marketplace Angular environment variables

```
export const environment = {
  production: false,
  apiURL: 'https://api.sandbox.domain.com',
};
```

If you point your browser to the marketplace URL, it should call the Movies Store API and list the movies crawled from IMDb pages, as shown in figure 11.9. It might take several minutes for DNS to propagate and for the marketplace to show up.

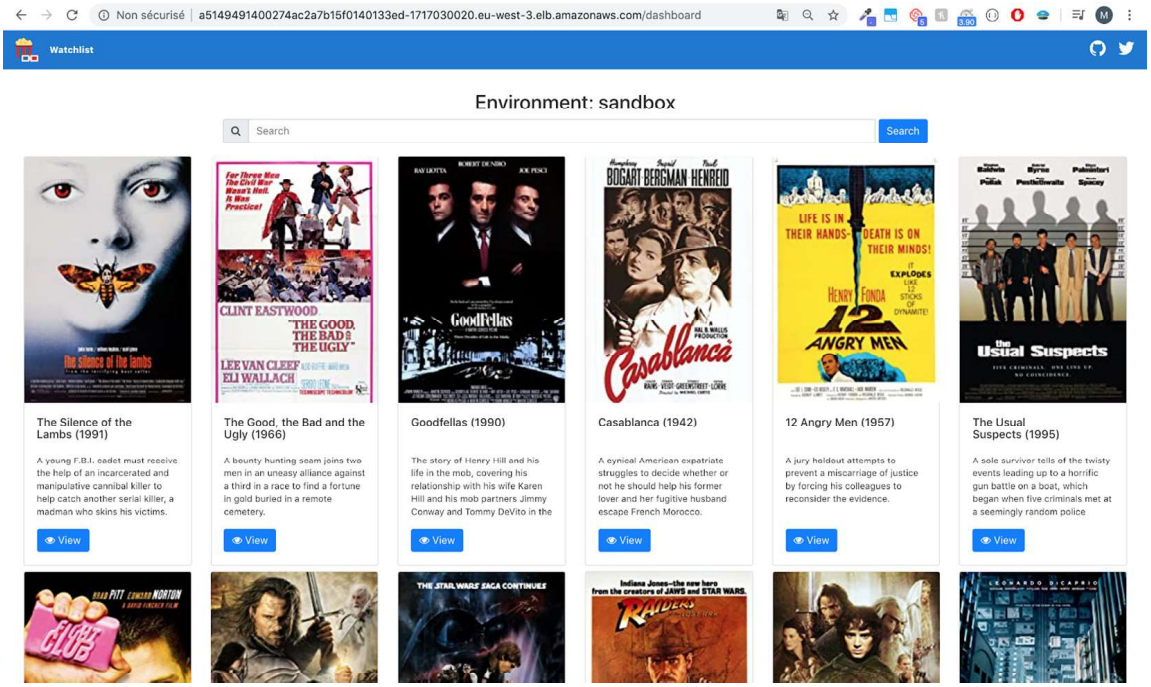

Figure 11.9 Watchlist Marketplace application

Now, every time you change the source code of any of the four microservices, the pipeline will be triggered, and the changes will be deployed to the sandbox Kubernetes cluster, as shown in figure 11.10.

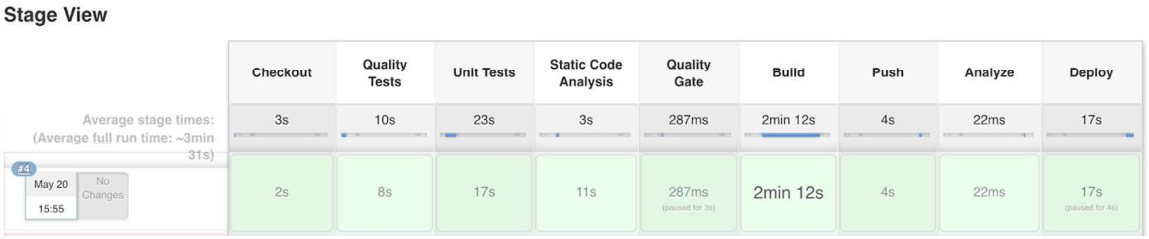

Figure 11.10 Movies Marketplace CI/CD workflow

Finally, to visualize our application, we can deploy the Kubernetes dashboard by issuing the following commands in a terminal session:

```
kubectl apply -f https://github.com/kubernetes-sigs/
metrics-server/releases/latest/download/components.yaml
```

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/
dashboard/v2.0.5/aio/deploy/recommended.yaml
```

These commands will deploy the metrics-server and K8s dashboard v2.0.5 under the kube-system namespace. The metrics-server, which collects resource metrics from Kubelet, has to be running in the cluster for the metrics and graphs to be available in the Kubernetes dashboard.

To grant access to cluster resources from the K8s dashboard, we need to create an eks-admin service account and cluster role binding to securely connect to the dashboard with admin-level permissions. Create an eks-admin.yaml file with the content in the following listing (`apiVersion` of the `ClusterRoleBinding` resource may differ between Kubernetes versions).

Listing 11.12 Kubernetes dashboard service account

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: eks-admin
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: eks-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: eks-admin
  namespace: kube-system
```

Then, create a service account with the following command:

```
kubectl apply -f eks-admin.yaml
```

Now, create a proxy server that will allow you to navigate to the dashboard from the browser on your local machine. This will continue running until you stop the process by pressing Ctrl-C. Issue the `kubectl proxy` command, and the dashboard should be accessible from http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/#/login.

Opening this URL will take us to the account authentication page for the Kubernetes dashboard. To get access to the dashboard, we need to authenticate our account. Retrieve an authentication token for the eks-admin service account with the following command:

```
kubectl -n kube-system describe secret
$(kubectl -n kube-system get secret
```

```
| grep eks-admin
| awk '{print $1}')
```

Now copy the token and paste it into the Enter Token field on the login screen. Click the Sign In button, and that's it. You are now logged in as an admin.

The Kubernetes dashboard, shown in figure 11.11, provides user-friendly features to manage and troubleshoot the deployed application. Awesome! You have successfully built a CI/CD pipeline for a cloud-native application in K8s.

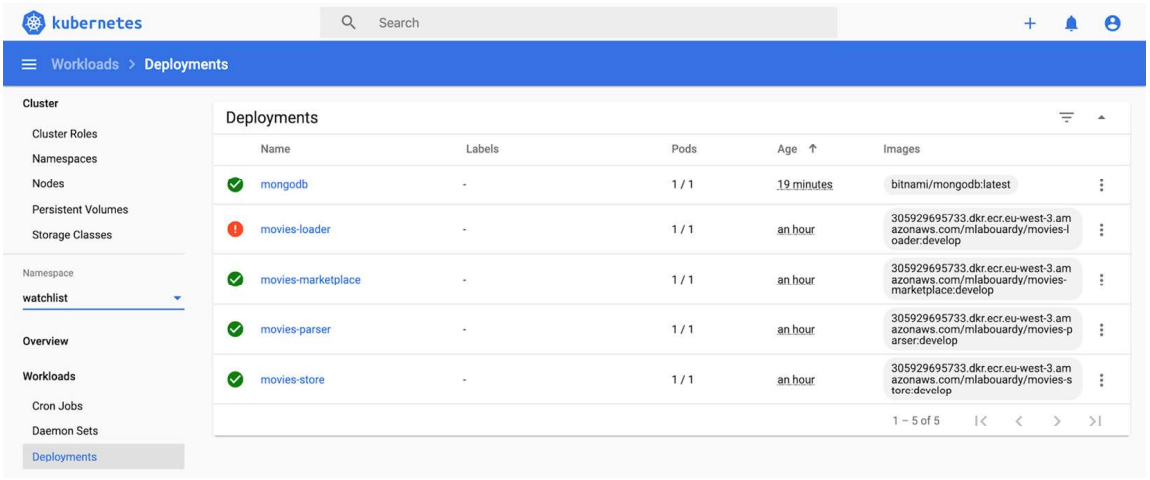

Figure 11.11 Kubernetes dashboard

11.2.1 Migrating Docker Compose to K8s manifests with Kompose

Another way of creating deployment files is by converting the docker-compose.yml file defined in chapter 10's listing 10.12 with an open source tool called Kompose. Refer to the project's official GitHub repository (https://github.com/kubernetes/kompose) for an installation guide.

Once Kompose is installed, run the following command against the docker-compose.yml file provided in chapter 10 (chapter10/deployment/sandbox/docker-compose.yml):

```
kompose convert -f docker-compose.yml
```

This should create the Kubernetes deployments and services based on the settings and network topology specified in docker-compose.yml:

```
INFO Kubernetes file "movies-marketplace-service.yaml" created
INFO Kubernetes file "movies-store-service.yaml" created
INFO Kubernetes file "mongodb-deployment.yaml" created
INFO Kubernetes file "movies-loader-deployment.yaml" created
INFO Kubernetes file "movies-marketplace-deployment.yaml" created
INFO Kubernetes file "movies-parser-deployment.yaml" created
INFO Kubernetes file "movies-store-deployment.yaml" created
```

You can push those files to the remote Git repository, and Jenkins will issue the `kubectl apply -f` command to deploy the services and deployments.

However, writing and maintaining Kubernetes YAML manifests for all the required Kubernetes objects can be a time-consuming and tedious task. For the simplest of deployments, you would need at least three YAML manifests with duplicated and hard-coded values. That's where a tool like Helm (https://helm.sh/) comes into play to simplify this process and create a single package that can be advertised to your cluster.

11.3 Walking through continuous delivery steps

Helm is a useful package manager for Kubernetes. It has two parts: the client (CLI) and the server (which is called Tiller and was removed in Helm 3). The client lives on your local machine, and the server lives on the Kubernetes cluster to execute what is needed.

To fully grasp Helm, you need to become familiar with these three concepts:

- *Chart*—A package of preconfigured Kubernetes resources
- *Release*—A specific instance of a chart that has been deployed to the cluster by using Helm
- *Repository*—A group of published charts that can be made available to others through a remote registry

Check out the getting started page for instructions on downloading and installing Helm: https://helm.sh/docs/intro/install/.

> **NOTE** Helm is assumed to be compatible with *n*-3 versions of Kubernetes. Refer to the Helm Version Support Policy documentation to determine which version of Helm is compatible with your K8s cluster.

At the time of writing this book, Helm v3.6.1 is being used. After installing Helm, create a new chart for the application called `watchlist` in the top-level directory of the watchlist-deployment project:

```
helm create watchlist
```

This should create a directory called watchlist with the following files and folders:

- *Values.yaml*—Defines all values we want to inject into Kubernetes templates
- *Chart.yaml*—Can be used to describe the version of the chart we're packaging
- *.helmignore*—Similar to .gitignore and .dockerignore, contains a list of files and folders to exclude while packaging the Helm chart
- *templates/*—Contains the actual manifest such as Deployments, Services, ConfigMaps, and Secrets

Next, define template files inside the templates folder for each microservice. The template file describes how to deploy each service on Kubernetes:

```
.
├── Chart.yaml
├── charts
├── templates
│   ├── movies-loader
│   │   ├── configmap.yaml
│   │   └── deployment.yaml
│   ├── movies-marketplace
│   │   ├── deployment.yaml
│   │   └── service.yaml
│   ├── movies-parser
│   │   ├── configmap.yaml
│   │   └── deployment.yaml
│   ├── movies-store
│   │   ├── deployment.yaml
│   │   └── service.yaml
│   ├── namespace.yaml
│   └── secret.yaml
└── values.yaml
```

For instance, the movies-loader template folder uses the same deployment files we defined in listing 11.4, except it references variables defined in values.yaml.

The deployment.yaml file is responsible for deploying a deployment object based on the movies-loader Docker image. This definition pulls the built Docker image from the Docker Registry and creates a new deployment with it in Kubernetes; see the following listing.

Listing 11.13 Movie loader deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: movies-loader
  namespace: {{ .Values.namespace }}
  labels:
    app: movies-loader
    tier: backend
spec:
  selector:
    matchLabels:
      app: movies-loader
  template:
    metadata:
      name: movies-loader
      labels:
        app: movies-loader
        tier: backend
      annotations:
        jenkins/build: {{ .Values.metadata.jenkins.buildTag | quote }}
        git/commitId: {{ .Values.metadata.git.commitId | quote }}
    spec:
      containers:
        - name: movies-loader
          image: "{{ .Values.services.registry.uri }}/
mlabouardy/movies-loader:{{ .Values.deployment.tag }}"
          imagePullPolicy: Always
```

```
envFrom:
  - configMapRef:
      name: {{ .Values.namespace }}-movies-loader
  - secretRef:
      name: {{ .Values.namespace }}-secrets
{{- if .Values.services.registry.secret }}
imagePullSecrets:
- name: {{ .Values.services.registry.secret }}
{{- end }}
```

Helm charts use `{{}}` for templating, which means that whatever is inside will be interpreted to provide an output value. We can also use a piping mechanism to combine two or more commands for scripting and filtering.

The movies-loader container reference environment variables like `AWS_REGION` and `SQS_URL` are defined in configmap.yaml, as shown in the following listing.

Listing 11.14 Movie loader ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Values.namespace }}-movies-loader
  namespace: {{ .Values.namespace }}
  labels:
    app: {{ .Values.namespace }}-movies-loader
data:
  AWS_REGION: {{ .Values.services.aws.region }}
  SQS_URL: https://sqs.{{ .Values.services.aws.region }}
.amazonaws.com/{{ .Values.services.aws.account }}/
movies_to_parse_{{ .Values.environment }}
```

The deployment file also references sensitive information such as MongoDB credentials. These credentials are stored securely in Kubernetes secrets, which are provided in the following listing.

Listing 11.15 Application secrets

```
apiVersion: v1
kind: Secret
metadata:
  name: {{ .Values.namespace }}-secrets
  namespace: {{ .Values.namespace }}
data:
  MONGO_URI: {{ .Values.services.mongodb.uri | b64enc }}
  MONGO_DATABASE : {{ .Values.mongodb.mongodbDatabase | b64enc }}
  MONGODB_USERNAME : {{ .Values.mongodb.mongodbUsername | b64enc }}
  MONGODB_PASSWORD : {{ .Values.mongodb.mongodbPassword | b64enc }}
```

Helm charts make it easy to set overridable defaults in the values.yaml file, allowing us to define a base setting. We can move as many variables as we want out of the template

and into the values.yaml file. This way, we can easily update and inject new values at installation time:

```
namespace: 'watchlist'
services:
  registry:
    uri: ''
    secret: ''
deployment:
  tag: ''
  workers:
    replicas: 2
```

This allows us to create a portable package that can be customized during runtime by overriding the values.

Also, note the use of custom annotations or metadata in the deployment file. We will inject the Jenkins build ID and Git commit ID during the build of the Helm chart. This can be useful for debugging and troubleshooting running Kubernetes deployments:

```
annotations:
        jenkins/build: {{ .Values.metadata.jenkins.buildTag | quote }}
        git/commitId: {{ .Values.metadata.git.commitId | quote }}
```

MongoDB offers a stable and official Helm chart that can be used for straightforward installation and configuration on Kubernetes. We define the MongoDB chart as a dependency in Chart.yaml under the `dependencies` section:

```
dependencies:
  - name: mongodb
    version: 7.8.10
    repository: https://charts.bitnami.com/bitnami
    alias: mongodb
```

Now that our chart is defined, on your terminal session, issue the following command to install the watchlist application via the Helm chart we just created:

```
helm install watchlist ./watchlist -f values.override.yaml
```

The command takes the values.override.yaml file, which contains the values to override at runtime, such as the environment name and MongoDB username and password:

```
environment: 'sandbox'
mongodb:
  mongodbUsername: 'watchlist'
  mongodbPassword: 'watchlist'
deployment:
  tag: 'develop'
  workers:
    replicas: 2
```

Check installation progress by checking the status of deployments and pods. Type `kubectl get pods -n watchlist` to show the running pods:

```
NAME                                  READY   STATUS      RESTARTS   AGE
movies-loader-748c544c6b-l7cl5        0/1     Completed   0          4s
movies-marketplace-57659fbcc-b5jh5    1/1     Running     0          32m
movies-parser-84df877c4-7mn6h         1/1     Running     0          32m
movies-parser-84df877c4-mr5md         1/1     Running     0          32m
movies-store-76d74646bc-v7rsx         1/1     Running     0          32m
```

NOTE To check the generated manifests of a release without installing the chart, use the `--dry-run` flag to return rendered templates.

We can now update the Jenkinsfile (chapter11/Jenkinsfile.eks) to use the Helm command line instead of kubectl. Since our application chart is already installed, we will use the `helm upgrade` command to upgrade the chart. This command takes as a parameter values to override, and sets the annotation values from the Jenkins environment variable `BUILD_TAG` and the `commitID()` method, as shown next.

Listing 11.16 Helm upgrade within the Jenkins pipeline

```
stage('Deploy'){
        sh """
            helm upgrade --install watchlist
./watchlist -f values.override.yaml \
                --set metadata.jenkins.buildTag=${env.BUILD_TAG} \
                --set metadata.git.commitId=${commitID()}
        """
}
```

Helm tries to perform the least invasive upgrade. It will update only things that have changed since the last release.

Push the changes to the develop branch. The GitHub repository should look similar to figure 11.12.

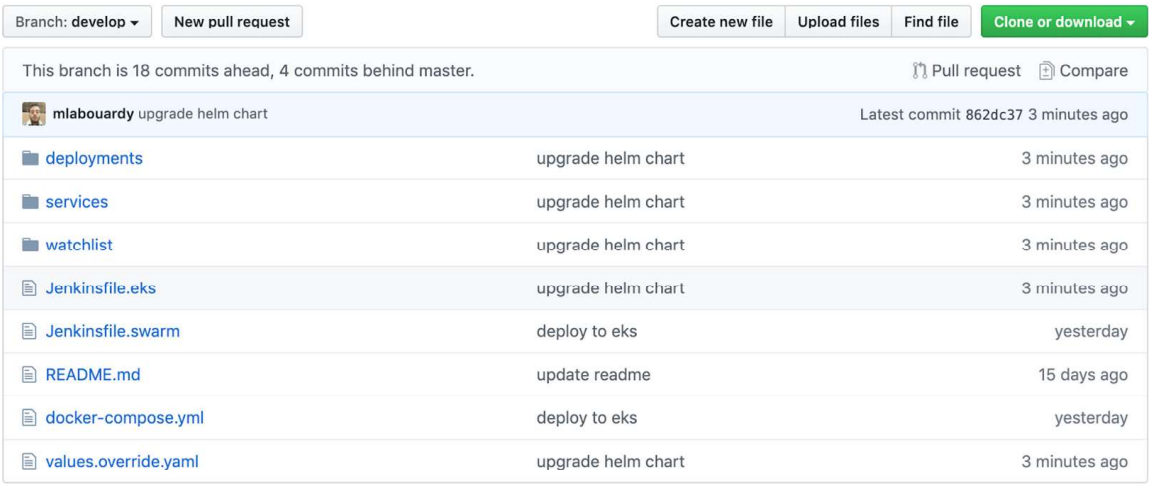

Figure 11.12 Watchlist Helm chart

On Jenkins, a new build will be triggered. At the end of the `Deploy` stage, the `helm upgrade` command will be executed; the output is shown in figure 11.13.

Console Output

```
+ helm upgrade --install watchlist ./watchlist -f values.override.yaml --set metadata.jenkins.buildTag=jenkins-watchlist-deployment-develop-8 --
set metadata.git.commitId=58a23e4fbd3a22ba9c9a7f58f5e780747b63f75e
Release "watchlist" has been upgraded. Happy Helming!
NAME: watchlist
LAST DEPLOYED: Thu May 21 14:51:29 2020
NAMESPACE: default
STATUS: deployed
REVISION: 2
TEST SUITE: None
```

Figure 11.13 Helm upgrade output

Now every change on the develop branch will build a new Helm chart and create a new release on the sandbox cluster. If the Docker image has been changed, Kubernetes rolling updates provide the functionality to deploy changes with 0% downtime.

> **NOTE** If something does not go as planned during a release, rolling back to a previous release is easy by using the `helm rollback` command.

For code promotion to the staging environment, we just need to update the values .override.yaml file to set the environment value to `staging` and use the `preprod` image tag, as shown in the following listing.

Listing 11.17 Staging variables

```
environment: 'staging'
mongodb:
  mongodbUsername: 'watchlist'
  mongodbPassword: 'watchlist'
deployment:
  tag: 'preprod'
  workers:
    replicas: 2
```

If you push the changes to the preprod branch, the application will be deployed to the Kubernetes staging cluster, as shown in figure 11.14.

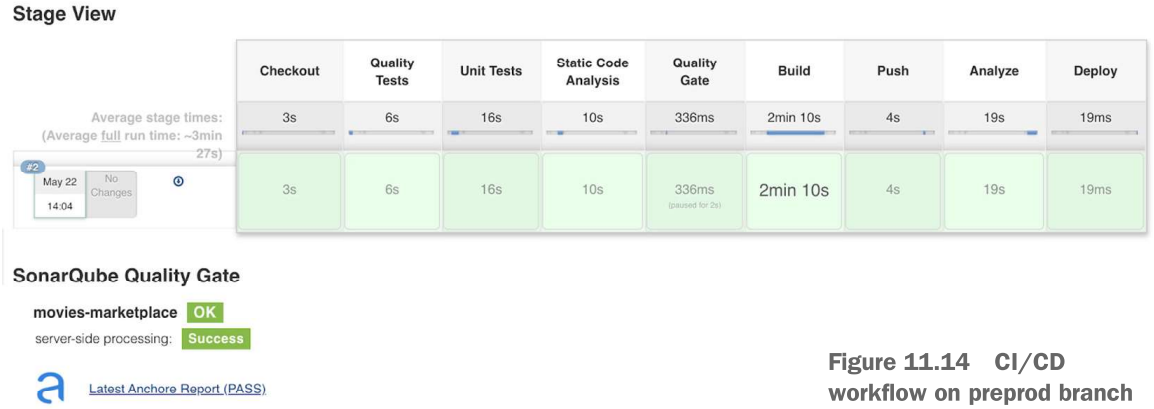

Figure 11.14 CI/CD workflow on preprod branch

We can verify that the preprod version has been deployed by typing the following command:

```
kubectl describe deployment movies-marketplace -n watchlist
```

The movies-marketplace deployment has annotations with git/commitId equal to the GitHub commit ID responsible for triggering the Jenkins job, and the jenkins/build annotation's value is the name of the Jenkins job that triggered the deployment (figure 11.15).

```
Name:                   movies-marketplace
Namespace:              watchlist
CreationTimestamp:      Fri, 22 May 2020 14:32:51 +0200
Labels:                 app=movies-marketplace
                        app.kubernetes.io/managed-by=Helm
                        tier=frontend
Annotations:            deployment.kubernetes.io/revision: 3
                        meta.helm.sh/release-name: watchlist
                        meta.helm.sh/release-namespace: default
Selector:               app=movies-marketplace
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:       app=movies-marketplace
                tier=frontend
  Annotations:  git/commitId: b82100da404cf03670124001cecbbd72d8f365b3
                jenkins/build: jenkins-watchlist-deployment-preprod-4
  Containers:
   movies-marketplace:
    Image:        305929695733.dkr.ecr.eu-west-3.amazonaws.com/mlabouardy/movies-marketplace:preprod
    Port:         <none>
    Host Port:    <none>
```

Figure 11.15 Movies Marketplace deployment description

For production deployment, update values.override.yaml with proper values, as shown in the following listing. In this example, we set the image tag to `latest`, the environment to `production`, and we configure five replicas of the movies-parser service.

Listing 11.18 Production variables

```
environment: production
mongodb:
  mongodbUsername: 'watchlist'
  mongodbPassword: 'watchlist'
deployment:
  tag: 'latest'
  workers:
    replicas: 5
```

Push the new files to the master branch. At the end of the pipeline, the stack will be deployed to the K8s production cluster.

Now if a push event occurs on the master branch on any of the four microservices, the CI/CD pipeline will be triggered, and user approval will be requested, as shown in figure 11.16.

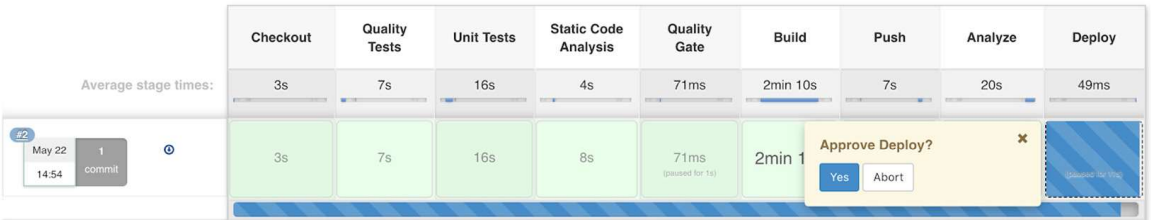

Figure 11.16 User approval for production deployment

If the deployment is approved, the watchlist-deployment job will be triggered, and the master nested job will be executed. As a result, a new Helm release of the watchlist application will be created in production, as shown in figure 11.17.

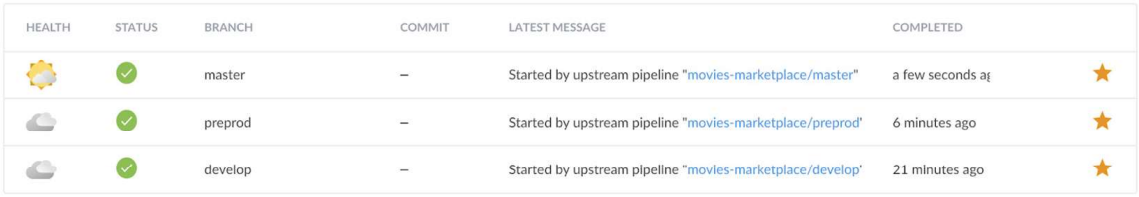

Figure 11.17 Application deployment in production

Upon the completion of the deployment process, a Slack notification will be sent to a preconfigured Slack channel, as shown in figure 11.18.

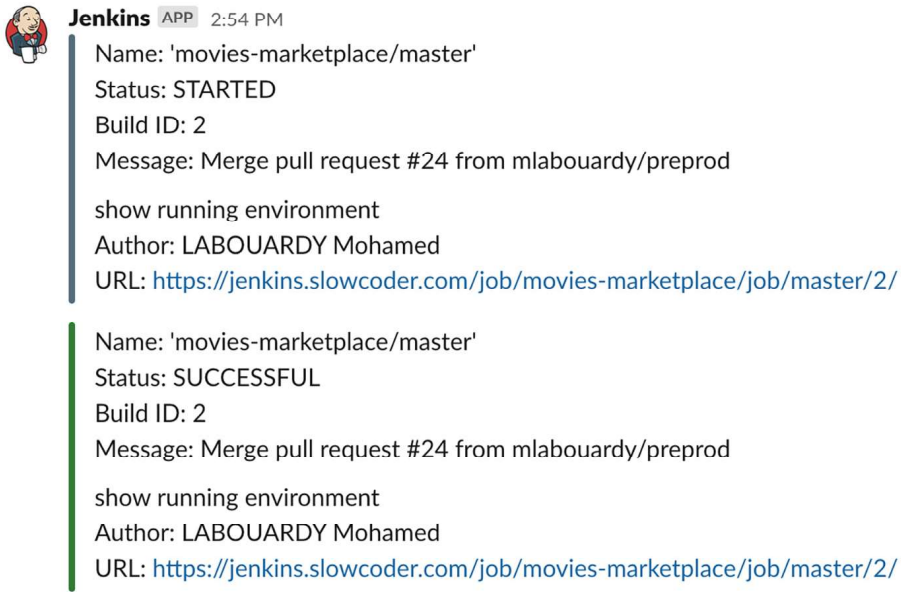

Figure 11.18 Production deployment Slack notification

Run the `kubectl get pods` command. This should display five pods based on the movies-parser Docker image:

```
NAME                                 READY   STATUS      RESTARTS   AGE
movies-loader-5fc5b6847b-7zrsg       0/1     Completed   1          12s
movies-marketplace-6b7898d567-xvhrh  1/1     Running     0          4m6s
movies-parser-7fd8c9498d-nv427       1/1     Running     5          4m6s
movies-parser-7fd8c9498d-slp7t       1/1     Running     5          4m6s
movies-parser-7fd8c9498d-tpw44       1/1     Running     5          4m6s
movies-parser-7fd8c9498d-x87qv       1/1     Running     5          4m6s
movies-parser-7fd8c9498d-xdlvs       1/1     Running     5          4m6s
movies-store-58d9ffc7d9-9ml5c        1/1     Running     0          4m6s
```

To view the marketplace dashboard, locate the external IP of the load balancer in the EXTERNAL-IP column of the `kubectl get services -n watchlist` output:

```
NAME                TYPE           CLUSTER-IP      EXTERNAL-IP
movies-marketplace  LoadBalancer   172.20.93.31    a3b35d67b360f4a5e9dd8ebc81bdf8ec-167487368.eu-west-3.elb.amazonaws.com
movies-store        LoadBalancer   172.20.230.116  aa8091de87549426cac6f128b0e73512-633343171.eu-west-3.elb.amazonaws.com
```

Navigate to that address in your browser, and the Movies Marketplace UI should be displayed, as you can see in figure 11.19.

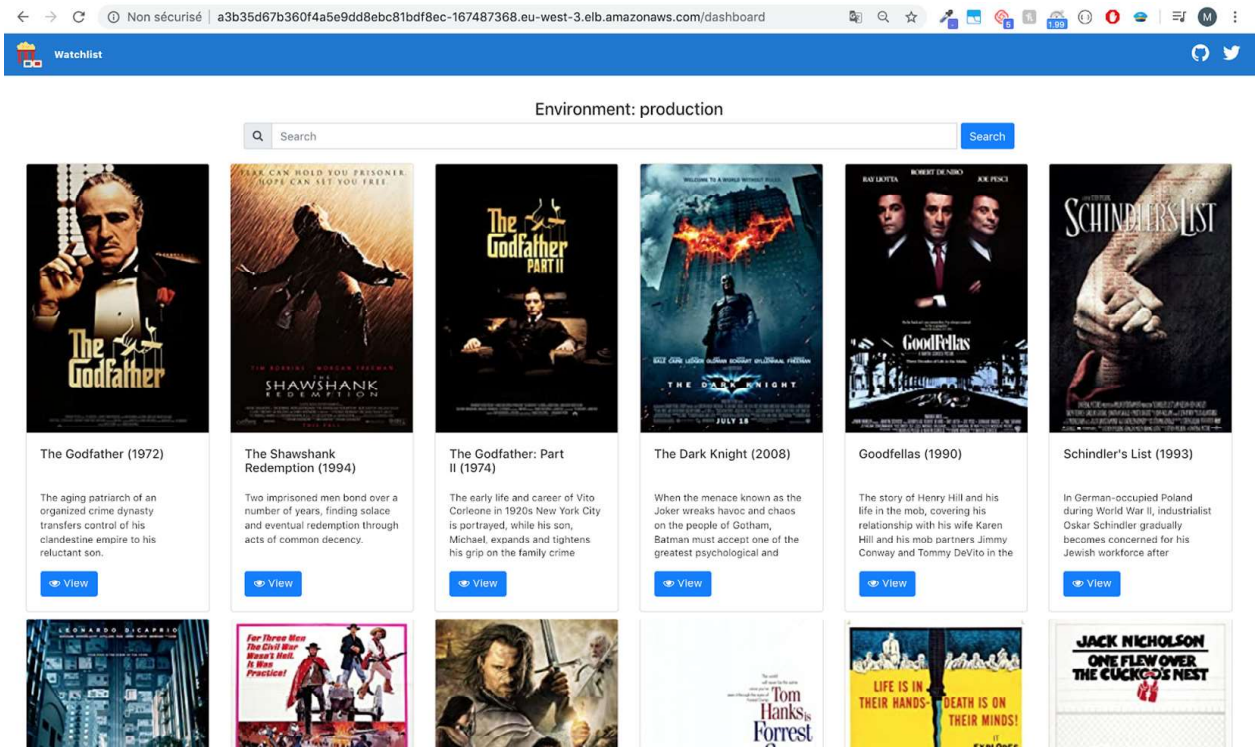

Figure 11.19 Marketplace production environment