Part 2

Operating a self-healing Jenkins cluster

You've read through part 1 and now feel comfortable with some of the core concepts and principles of pipeline as code. It's time to get your hands dirty and deploy a Jenkins cluster from scratch with infrastructure-as-code tools on the cloud, including Amazon Web Services, Google Cloud Platform, Microsoft Azure, and DigitalOcean.

Along the way, you'll discover how to scale Jenkins workers dynamically and how to architect Jenkins for scale with distributed build mode. We'll then look at Jenkins essential plugins and how to provision a preconfigured Jenkins cluster with all needed dependencies and configurations using Packer and Groovy scripts.

3 Defining Jenkins architecture

This chapter covers

- Understanding how Jenkins distributed builds work
- Understanding the roles of Jenkins master and worker nodes
- Architecting Jenkins in the cloud for scale
- Configuring multiple Jenkins masters
- Preparing an AWS environment and CLI configuration

In a distributed microservices architecture, you may have multiple services to build, test, and deploy regularly. Hence, having multiple build machines makes sense. While you can always run Jenkins in a standalone mode, running all builds on a central machine may not be the best option and will result in having a single point of failure (a single Jenkins server cannot handle the entire load for larger and heavier projects). Fortunately, Jenkins can also be configured to run distributed builds across a fleet of machines/nodes by setting up a master/worker cluster, as shown in figure 3.1.

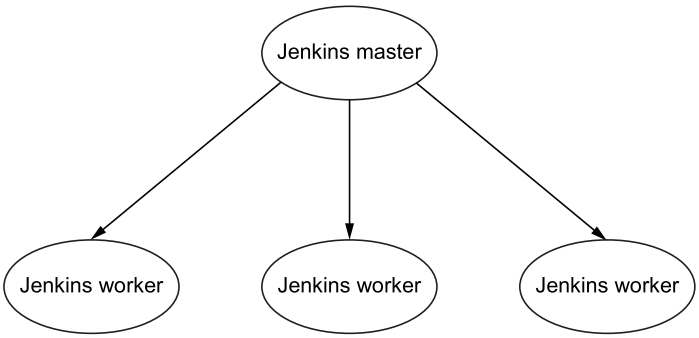

Figure 3.1 Distributed master-worker architecture

Jenkins uses a master-worker architecture to manage distributed builds. Each component has a specific role:

- *Jenkins master*—Responsible for scheduling build jobs and distributing builds to the workers for the actual execution. It also monitors the workers' states, and collects and aggregates the build results in the web dashboard.
- *Jenkins worker*—Also known as a *slave* or *build agent*, this is a Java executable that runs on a remote machine, listens for requests coming from the Jenkins master, and executes build jobs. You can have as many workers as you want (up to 100+ nodes). Workers can be added and removed on the fly. Therefore, the workload will be distributed to them automatically, and the workers will take the load off the master Jenkins server.

NOTE In 2016, the Jenkins community decided to start removing offensive terminology within the project. The *slave* term was deprecated in Jenkins 2.0 and replaced by *agent*.

To sum up, Jenkins can be deployed in a standalone mode. However, when you want to run multiple build jobs regularly in different environments to meet the requirements of the build environment for different projects, then a single Jenkins server cannot simply handle the workload. That's why in this book, we will be focusing on *master-worker architecture*.

3.1 Understanding master-worker architecture

In a master-worker architecture, the web dashboard is running on the Jenkins master instance. The master's role is to handle scheduling build jobs, dispatching and delegating builds to the workers for the actual execution, monitoring the workers' state (online or offline), and recording and presenting the build results. Even in a distributed architecture, a master instance of Jenkins can also execute build jobs directly.

Jenkins workers can be added and configured on the Jenkins dashboard or through a Jenkins RESTful API. The worker's role is to execute build jobs assigned by

the master. You can configure a project to always run on a particular node by assigning labels to nodes. Labels are a powerful feature; they are virtual group names. You can assign multiple labels to a worker node while configuring it. Labels can also be used to restrict the build job to run on a worker node associated with a specific label name—for instance, to restrict a job to be built on a CPU-optimized instance.

To add a worker, you can click Manage Jenkins in the admin page menu, and then click Manage Nodes and Add New Node. Fill in the configuration information, including a name for the node, the workspace name, and the IP address of the node. Then, enter a label like `workers` (you can assign multiple labels in the Labels entry box by separating them with spaces). Figure 3.2 shows how to add a new worker to Jenkins.

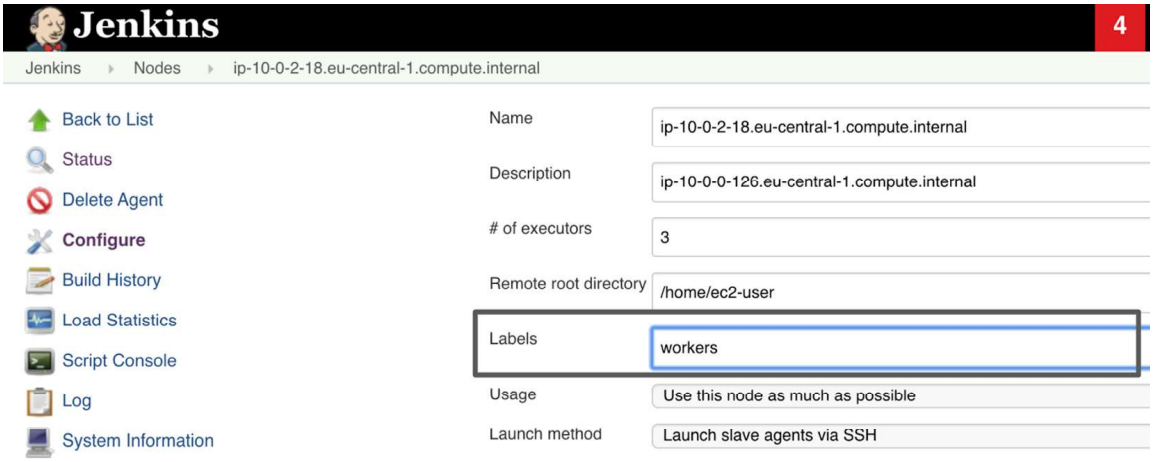

Figure 3.2 Using labels for Jenkins jobs assignments

By assigning the `workers` label to the node, you can reference it easily in your Jenkinsfile. In a declarative pipeline, you can restrict the pipeline to run on nodes with the `workers` label by setting up the `agent` directive as follows:

```
pipeline{
    agent{
        label 'workers'
    }
    stages{
       stage('Checkout'){}
    }
}
```

The scripted pipeline, however, uses the `node` block wrapper with the label name as a parameter to define the execution environment for the pipeline:

```
node('workers'){
    stage('Checkout'){}
}
```

If more build jobs are requested for the same node, Jenkins will automatically create a job queue. By default, each node can execute one job; however, you can increase the node's capacity for running jobs by setting the field labeled # of Executors. In the previous example, the node is configured with three executors, which means up to three jobs can be executed at once. If four jobs are started, the first three will execute, and the fourth will be added to the build queue. Once nodes become available, Jenkins will execute the remaining jobs in the order they were requested.

To be able to add a worker to the Jenkins cluster, the workers and master need to establish bidirectional communication through TCP/IP. Another requirement is that Java should be installed on the worker machine. Because Java is a platform-agnostic programming language, a Jenkins cluster might consist of workers that run on a variety of OS platforms such as Windows, Linux, or macOS. This architecture comes with multiple benefits, such as having a heterogeneous build farm that supports all of the environments that you might need to run builds/tests with a different OS or CPU architecture.

In the example in figure 3.3, using a worker to represent each of your required environments results in having several environments and configurations to test, build, and deploy your projects. The delegation behavior of build jobs depends on the configuration of each project; some projects may choose to "stick" to a particular machine for a build using labels, while others may choose to roam freely among available workers.

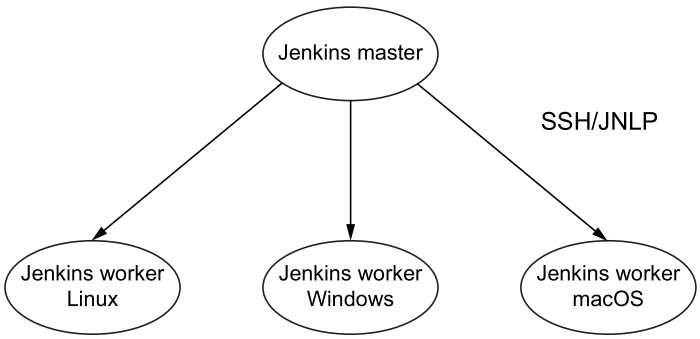

Figure 3.3 You can set up multiple workers running different operating systems by using SSH or Java Network Launch Protocol (JNLP)

3.2 Managing Jenkins workers

Several strategies are available when it comes to managing Jenkins workers, depending on your target operating systems and other architectural considerations. These strategies affect the way you configure your workers, so we need to consider each separately.

3.2.1 SSH

If you are working in a UNIX environment, the most convenient way to start a Jenkins worker is undoubtedly to use Secure Shell (SSH). Jenkins has its own built-in SSH client, and almost all UNIX environments support SSH (usually `sshd`) out of the box.

The worker needs to be reachable from the master server, and you will have to supply the hostname, login, and password. You can also provide a path to the SSH private key file on the master instance to use public/private key authentication, as shown in figure 3.4.

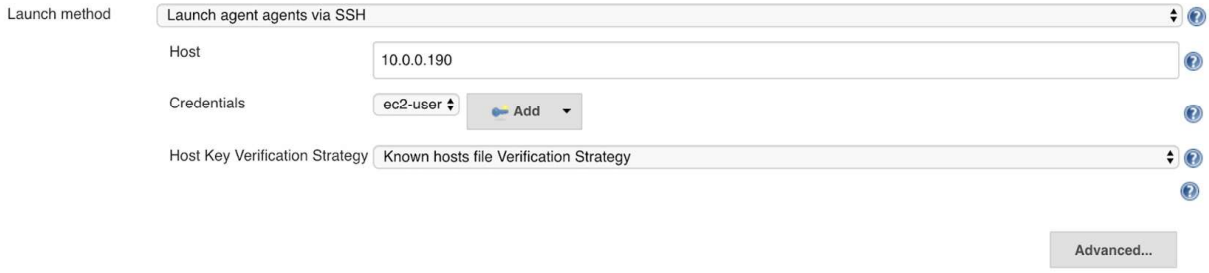

Figure 3.4 Launching a Jenkins worker via SSH

NOTE In chapter 5, we will use the SSH launch method to set up a Jenkins cluster.

3.2.2 Command line

You can add a worker by having Jenkins execute a command from the master, as shown in figure 3.5. Use this approach when the master is capable of remotely executing a process on another machine. However, the remoting mode has been deprecated since Jenkins 2.54 (so it might not a valid option in the newest version of Jenkins).

Figure 3.5 Launching a Jenkins worker via the command line

3.2.3 JNLP

Another option is to start an agent from the worker machine itself by using Java Web Start (JWS). This approach is useful if the master cannot reach the worker—for example, if the worker machine is running on the other side of a firewall. It works no matter what operating system your worker is running on. However, it is more suitable for managing Windows workers.

This approach does suffer from a few major drawbacks: the worker machine cannot be started or restarted automatically by Jenkins. If the worker goes down, the master instance cannot restart it. When you do this on a Windows machine, you need to start the Jenkins worker manually at least once. This requires opening a browser on the machine, opening the worker node page on the Jenkins master, and launching

the worker using a very visible JNLP icon. However, once you have launched the worker, you can install it as a Windows service.

3.2.4 Windows service

Jenkins can also manage a remote Windows worker as a Windows service, using the Windows DCOM Server Process Launcher service, which is installed out of the box on Windows. When you choose this option, you need to provide a Windows hostname, username, and password, as you can see in figure 3.6.

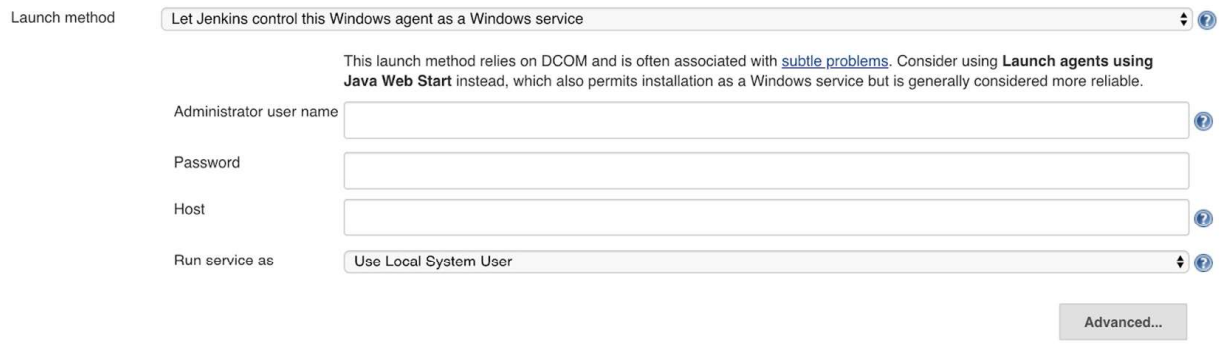

Figure 3.6 Starting a Windows worker

This launching mode is convenient, as it does not require you to physically connect to the Windows machine to set it up. However, it does have limitations—in particular, you cannot run any applications requiring a graphical interface.

Once the workers are added to the Jenkins cluster, the master will proactively monitor their statuses and take a worker offline if it considers the worker incapable of safely executing a build job. You can fine-tune exactly what Jenkins monitors on the Manage Nodes page, shown in figure 3.7.

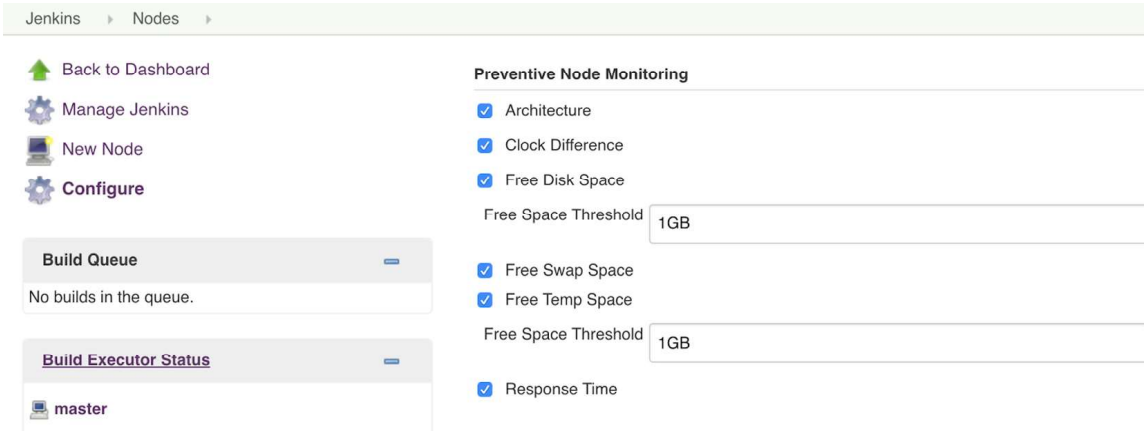

Figure 3.7 Defining node-monitoring thresholds

Jenkins monitors the available disk space of $JENKINS_HOME on each worker, as well as the disk space of the temporary directory and swap space. It also keeps tabs on the system clock difference between the master and workers. Finally, it monitors the round-trip network response time from the master to the worker. If any of these criteria is below a certain threshold, the worker will be marked offline.

Finally, it's worth mentioning that by default Jenkins uses the workers as much as possible. Whenever a build can be executed by a specific worker, Jenkins will use it.

To control how Jenkins is scheduling builds on available workers, you can configure the Usage field, shown in figure 3.8, to use the Only Build Jobs with Label Expressions Matching This Node option to restrict jobs to a worker that matches its name and/or label. This can become handy if you want to reserve a worker for a certain kind of Jenkins job. Furthermore, if you set the # of Executors field's value to 1, you can ensure that only one job will be executed at any given time. As a result, no other builds will interfere.

Figure 3.8 Configuring Jenkins worker usage

3.3 Architecting Jenkins for scale in AWS

So far, we have covered how Jenkins distributed builds work. This section covers how to architect Jenkins for scale on AWS. Therefore, you will need an AWS account to follow the examples. With a new AWS account, the Free Tiers should cover all the examples at no cost to you. For more information on the AWS Free Tier, and a step-by-step guide on how to create a new AWS account, visit https://aws.amazon.com/free/.

> **NOTE** Although this section focuses on AWS, this content can also be used to help set up a Jenkins cluster in other cloud providers. Chapter 6 provides a step-by-step guide.

The simple architecture you can deploy is a standalone or single-node setup. You simply need to deploy a Jenkins server on an Amazon Elastic Compute Cloud (EC2) instance from the AWS Marketplace (https://aws.amazon.com/marketplace), shown in figure 3.9.

The AWS Marketplace contains preconfigured Amazon Machine Images (AMIs) from popular categories such as security, networking, storage, machine learning, business intelligence, database, and DevOps. You can quickly launch a Jenkins server with

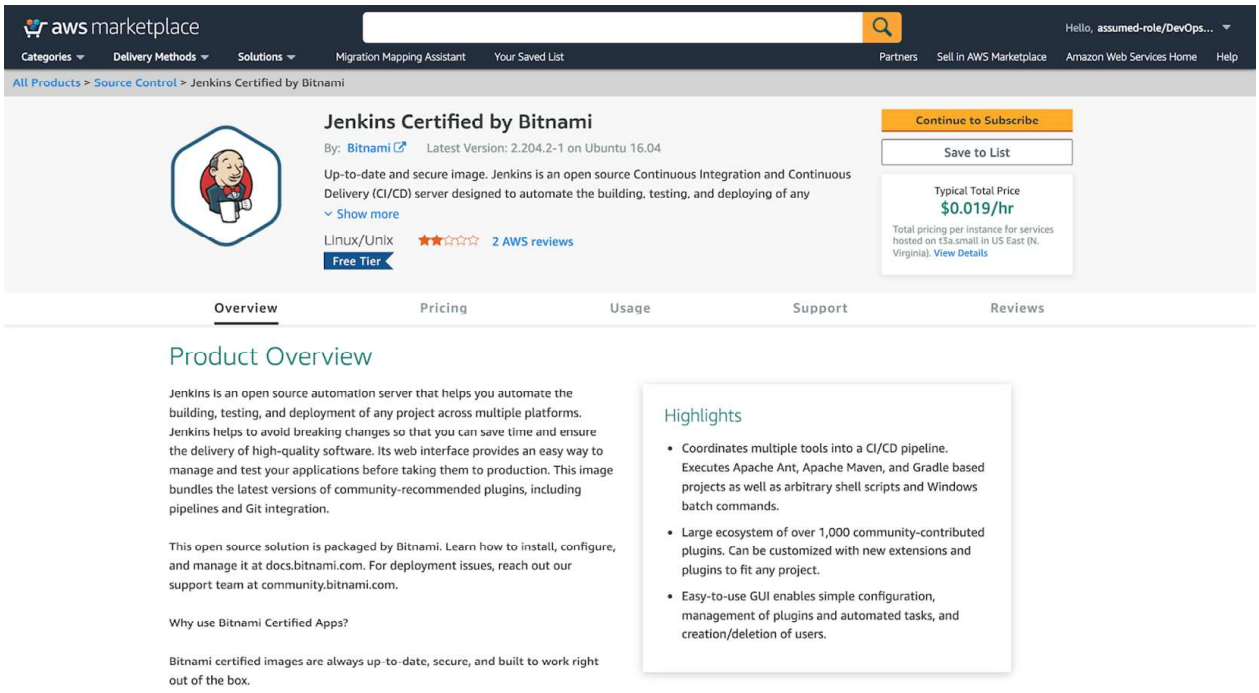

Figure 3.9 Jenkins Amazon Machine Image available on AWS Marketplace

just a few clicks, by selecting the Jenkins Long-Term Support (LTS) release and the machine instance type (based on resource requirements).

You can also install Jenkins on a base machine image by using a package manager (for example, APT or Yum). Jenkins installers are available for several Linux distributions as well as Windows and macOS. Otherwise, you can set up a Jenkins playground with a Jenkins official Docker image.

> **NOTE** Chapter 4 covers how to create your own Jenkins machine image from scratch with HashiCorp Packer.

Once you have installed Jenkins on an EC2 instance, you will need to configure the security group attached to the instance to allow traffic on port 8080. This is the port where the Jenkins dashboard is exposed to.

A *security group* acts as a firewall that controls the traffic allowed to reach the EC2 instances (figure 3.10). To control traffic, we create rules in the security group. For this case, the following security rules need to be added:

- Allow inbound (ingress) traffic on port 8080 (Jenkins dashboard port number).

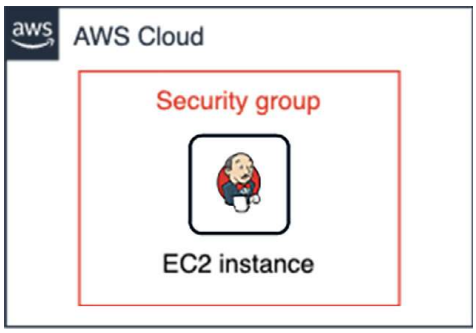

Figure 3.10 The Jenkins standalone architecture on AWS consists of an EC2 instance behind a security group.

- (Optional) Allow inbound SSH traffic from your computer's public address so that you can connect to your Jenkins instance for debugging or maintenance.
- By default, a security group includes an outbound rule that allows all outbound (egress) traffic.

You might set up a network access-control list (ACL) with rules similar to your security group to add an additional layer of security to your instance. The security group acts as a firewall for your Amazon EC2 instance, controlling both inbound and outbound traffic at the instance level. ACL acts as a firewall for associated subnets, controlling both inbound and outbound traffic at the subnet level.

NOTE While you can scale the Jenkins master vertically to absorb the loading pike of build jobs, there is a limit to how much an instance can be scaled.

While this architecture works for smaller projects, it can't scale for larger and complex projects. Therefore, we will deploy a Jenkins cluster to share the load across multiple workers. Instead of scheduling builds jobs on a Jenkins master instance, they will be assigned to Jenkins workers. As a result, additional EC2 instances (figure 3.11) will be deployed as build servers or Jenkins agents.

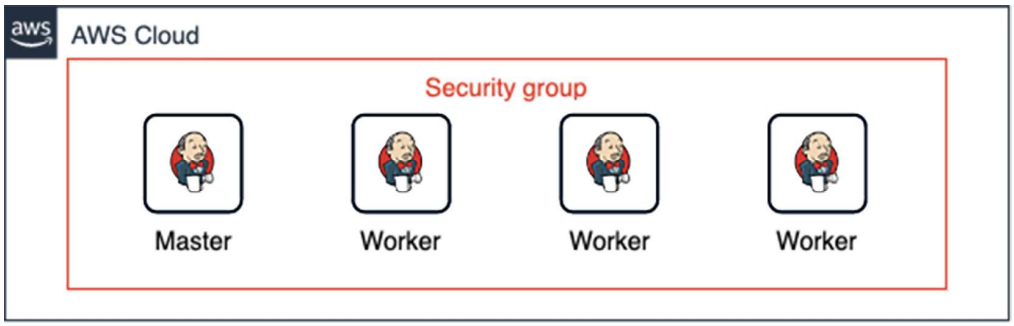

Figure 3.11 Jenkins distributed architecture on AWS

This architecture is much better. However, distributed builds are generally used to absorb extra load (for example, in build activity) by dynamically adding extra machines as required. Hence, the number of workers shouldn't be fixed in advance. We want to add or remove workers based on the number of jobs waiting in the queue or the CPU utilization of the worker's cluster. That's why, instead of deploying workers independently, we will deploy them inside an AWS Auto Scaling group (ASG); see https://aws.amazon.com/autoscaling/.

The ASG feature comes with EC2 and allows you to deploy a group of EC2 instances that are treated as a logical grouping for the purpose of automatic scaling. In addition, Amazon EC2 Auto Scaling helps to ensure that you have the correct number of instances by specifying the minimum and maximum number of instances at any given time.

To create and terminate Jenkins workers on demand based on build jobs, we can create scaling policies. A *scaling policy* is a set of instructions for adjusting the size of instances in the ASG in response to an Amazon CloudWatch alarm (http://mng.bz/g1rl).

An Amazon CloudWatch alarm will monitor the CPU usage of the EC2 instances, for example. Then it will trigger a scale-out or scale-in event to add or remove a worker to the Jenkins cluster automatically. For instance, if the average CPU utilization of the Jenkins workers is over 80%, a scale-out event will be triggered, and a new worker will be deployed and added to the Jenkins cluster. Similarly, if the average CPU utilization of the Jenkins workers is less than 20%, a scale-in event will be triggered, and unused workers will be removed (providing infrastructure cost optimization).

NOTE When creating an alarm on the Auto Scaling group, the alarm uses aggregated metrics across all Jenkins worker instances (average CPU utilization). This way, it won't add instances just because one worker is too busy.

When the CPU utilization is less than 20%, the scale-in policy takes effect, and the ASG terminates on the available instances. If you did not assign a specific termination policy to the ASG, it uses the default termination policy. This means the ASG selects the instance to terminate based on the following factors:

- The instance that is closed to the next billing hour.
- Longest/oldest running EC2 instance.
- Oldest launch configuration. The *launch configuration* is the blueprint or template that describes what a Jenkins worker instance should look like.

However, you can use Amazon EC2 termination protection to protect a Jenkins worker from being accidentally terminated. Refer to the official guide for instructions: http://mng.bz/ePwz.

We can also configure the scaling policies based on memory utilization. However, memory utilization is one of the metrics not available by default in CloudWatch. Since AWS does not have access to the instance at the OS level, only metrics that can be monitored through the hypervisor layer (such as CPU and network utilization) are recorded.

We have various ways to solve this problem. The most used one is to install a metrics collector agent on the EC2 instances. For more details on how to fetch the memory utilization, check out chapter 13.

NOTE To be able to add workers automatically, the worker machine will run a shell script at boot time and use the Jenkins RESTful API to autoregister to the cluster with the machine's private IP address (known as *cluster discovery*). Chapters 4 and 5 explain this part in depth.

Figure 3.12 illustrates how to dynamically scale Jenkins workers by using CloudWatch scaling policies.

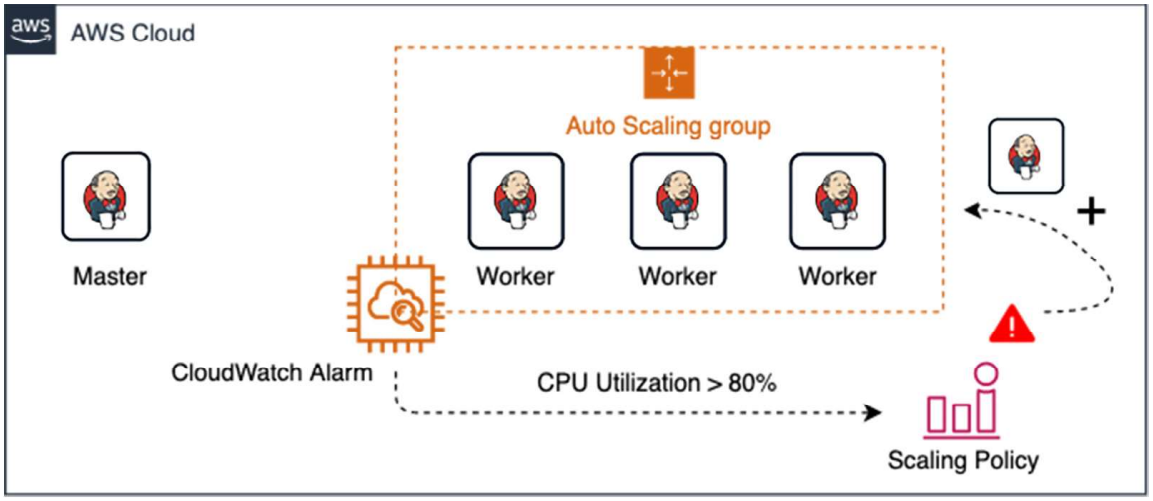

Figure 3.12 Jenkins workers belong to an AWS autoscaling group and will be scaled dynamically based on the average CPU utilization of the group.

We can also use custom metrics such as the number of jobs waiting in the build queue to trigger scaling policies. To get this information, you can use an open source solution such as Prometheus (https://prometheus.io/docs/introduction/overview/) to export Jenkins cluster metrics and make a Lambda function to consume/scrape those metrics. From the Lambda function, you can trigger scale-out or scale-in events on the Jenkins worker autoscaling group by using the AWS API/SDK.

> **NOTE** Chapter 13 covers how to monitor a Jenkins cluster's health and how to use the Prometheus exporter plugin on Jenkins to expose server-side metrics.

Figure 3.13 demonstrates how to scale Jenkins workers dynamically based on a custom metric.

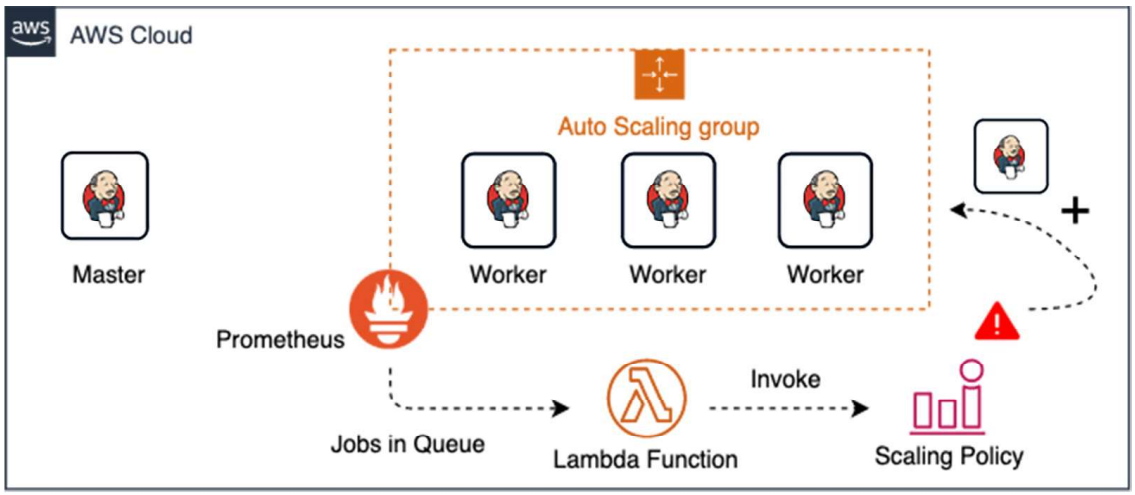

Figure 3.13 You can scale Jenkins workers dynamically based on the number of jobs waiting in the build queue by integrating Prometheus and AWS Lambda.

So far, the architecture is promising. However, it's not secure and resilient. To secure our Jenkins cluster, we will deploy the architecture inside a virtual private cloud (VPC) and within a private subnet precisely. In reality, by default, any EC2 instance is deployed in the AWS default VPC. But we will create a nondefault VPC that suits our specific requirements, using specific Classless Inter-Domain Routing (CIDR) block range and subnet sizes.

Amazon VPC (https://aws.amazon.com/vpc) lets you provision a logically isolated section of the AWS cloud where you can launch AWS resources in a virtual network that you can define. You have complete control over your virtual networking environment, including a selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways.

An important point to note here is that a VPC is still a part of the AWS cloud. It is not physically separate hosting provided by AWS; it is a logically isolated part of the EC2 infrastructure. This isolation is done at the network layer and is similar to a traditional datacenter's network isolation; it's just that we, as end users, are shielded from the complexities of it. Figure 3.14 shows the network topology of AWS VPC.

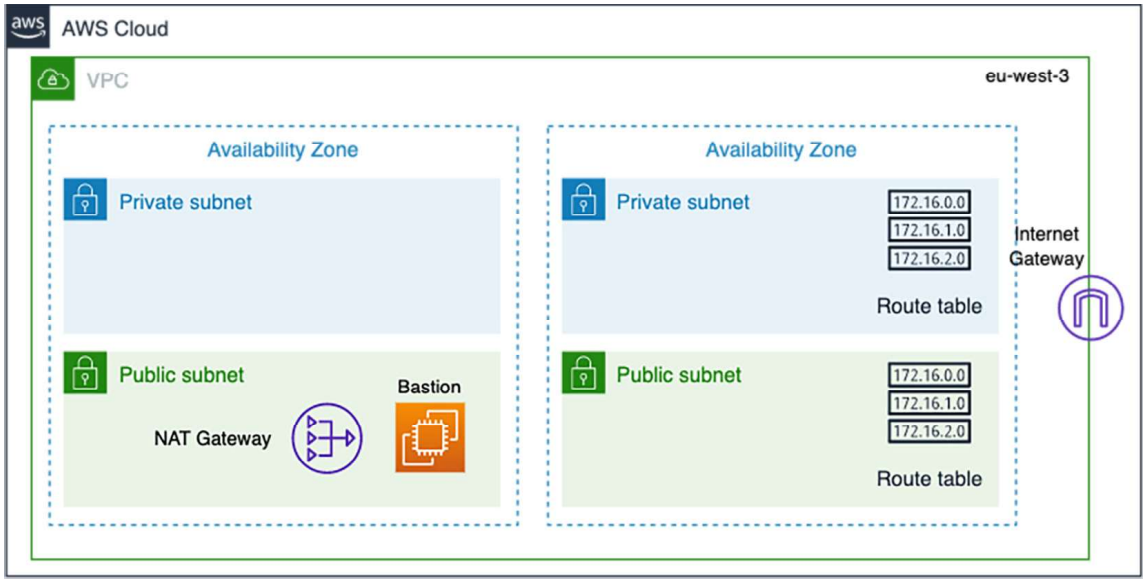

Figure 3.14 The virtual private cloud consists of private and public subnets.

We will create an AWS VPC with multiple subnets. A *subnet* is nothing more than a range of valid IP addresses. For resiliency, these subnets will be deployed in different availability zones in the selected AWS region.

Next, we deploy an internet gateway (IGW) and attach it to the VPC. The IGW will be used primarily to provide internet connectivity to Jenkins instances (this might be needed if your build jobs running in Jenkins workers require downloading external

packages from the internet). Plus, the IGW maps the instance's private IP address with an associated public or Elastic IP address (http://mng.bz/p9QG) and then routes traffic outside the subnet to the internet. Finally, we create a public route table with rules to direct network traffic from public subnets to the IGW, as shown in table 3.1.

Table 3.1 Public route table

Destination	Target	Remark
10.0.0.0/16	local	Allow traffic to flow with this particular subnet (10.0.0.0/16)
0.0.0.0/0	IGW ID	Allow subnet traffic to flow through the internet.

But what about instances in the private subnets? That's where a Network Address Translation (NAT) instance or gateway comes into play. The NAT gateway/instance will be created inside a public subnet and will forward the outbound traffic and not allow any traffic from the internet to reach the private subnets. This means instances will have access to the internet without being exposed to the public (no public IP address is given). Once the NAT gateway is deployed, we need to add an entry to the private subnets route table to point to the NAT gateway; see table 3.2.

Table 3.2 Private route table

Destination	Target	Remark
10.0.0.0/16	local	Allow traffic to flow with this particular subnet (10.0.0.0/16)
0.0.0.0/0	NAT ID	Allow subnet traffic to flow through the NAT gateway/instance

Because Jenkins instances will be deployed into private subnets that are isolated from the internet, we cannot SSH directly to them from local desktops. A basic solution is to deploy a special instance that acts as a proxy you can use to SSH into your Jenkins instances. This special instance is called a *bastion host*, or *jump box*. This instance will be deployed in your public subnet and will basically route only SSH traffic from your local network over the Jenkins instances by setting up a secure SSH tunnel/bridge.

> **NOTE** An advanced solution is to deploy OpenVPN to establish a secure TLS VPN session to securely access your private Jenkins instances. Refer to "Setting Up OpenVPN Access Server in Amazon VPC" at http://mng.bz/OQVn for instructions.

Once the VPC is configured, we can go ahead and deploy a dedicated EC2 instance running the Jenkins server on a private subnet. Alongside, an ASG of Jenkins workers will be deployed across multiple private subnets. We configure scaling policies with CloudWatch alarms to dynamically scale Jenkins workers based on the build activity. Figure 3.15 summarizes the current deployment architecture.

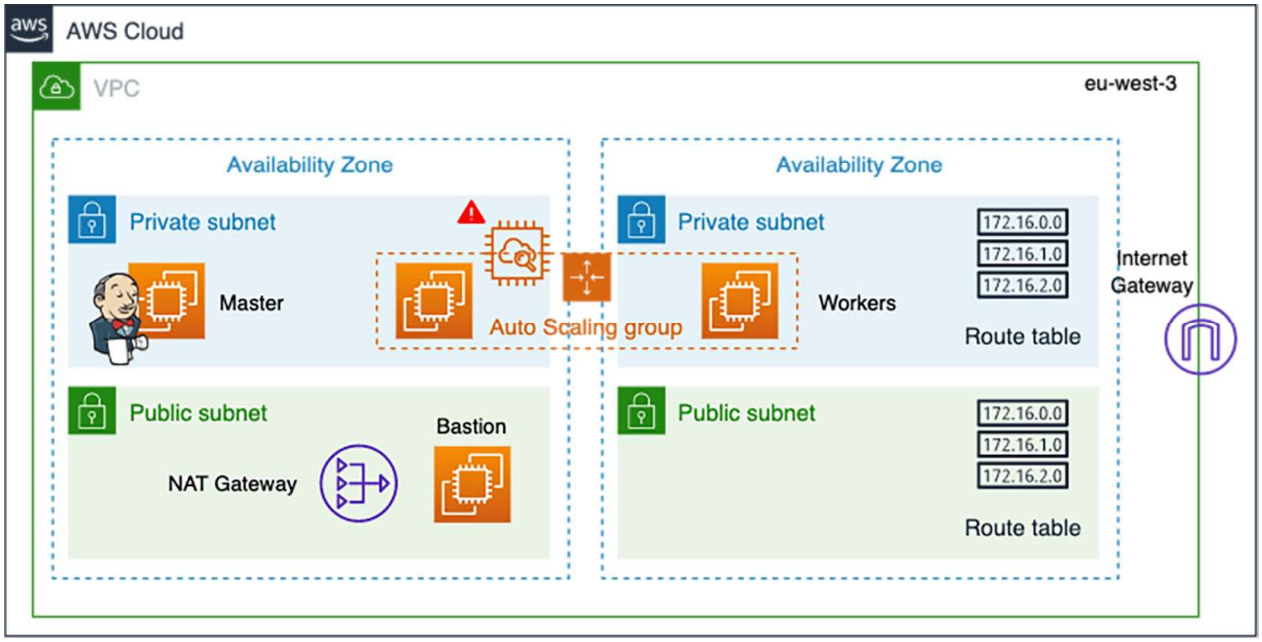

Figure 3.15 This Jenkins cluster deployed in private subnets consists of an ASG of workers and an EC2 instance holding the Jenkins dashboard.

We can take this architecture further, and configure a public-facing Elastic load balancer in front of the Jenkins instance to access the Jenkins web dashboard. This way, your Jenkins instance does not have to be directly exposed to the internet.

> **NOTE** It's possible to have multiple Jenkins instances even though Jenkins core doesn't support multiple masters by default. Then, use the load balancer to fetch requests and distribute them among multiple Jenkins masters.

The load balancer will listen on both the HTTP (80) and HTTPS (443) ports and send incoming requests to the instance on port 8080. That way, it uses an encrypted connection to communicate with the Jenkins instance. Table 3.3 summarizes the port configurations.

Table 3.3 Load balancer listener configuration

Load balancer protocol	Load balancer port	Instance protocol	Instance port
HTTP	80	HTTP	8080
HTTPS	443	HTTP	8080

If you specify the HTTPS listener, you will need to select a private Secure Sockets Layer (SSL) certificate. The load balancer uses the certificate to terminate the connection and then decrypt requests from clients before sending them to the Jenkins

instance. You can get a free SSL certificate with AWS Certificate Manager (ACM); you can also import your own certificate.

The load balancer has a publicly resolvable DNS name, so it can route requests from clients over the internet to a Jenkins instance that is registered with the load balancer. Also, it will be useful while setting up a GitHub webhook for continuously triggering Jenkins builds upon push events.

> **NOTE** If you plan to stick with a private Jenkins instance, chapter 7 explains how to set up a GitHub webhook for a Jenkins instance running behind a firewall.

Finally, if you would like to use a friendly DNS name to access your load balancer, instead of the default DNS name automatically assigned to your load balancer, you can create a custom domain name and associate it with the DNS name for your load balancer. The DNS configuration can be done on Amazon Route 53 (https://aws.amazon.com/route53/). Figure 3.16 shows the final architecture diagram.

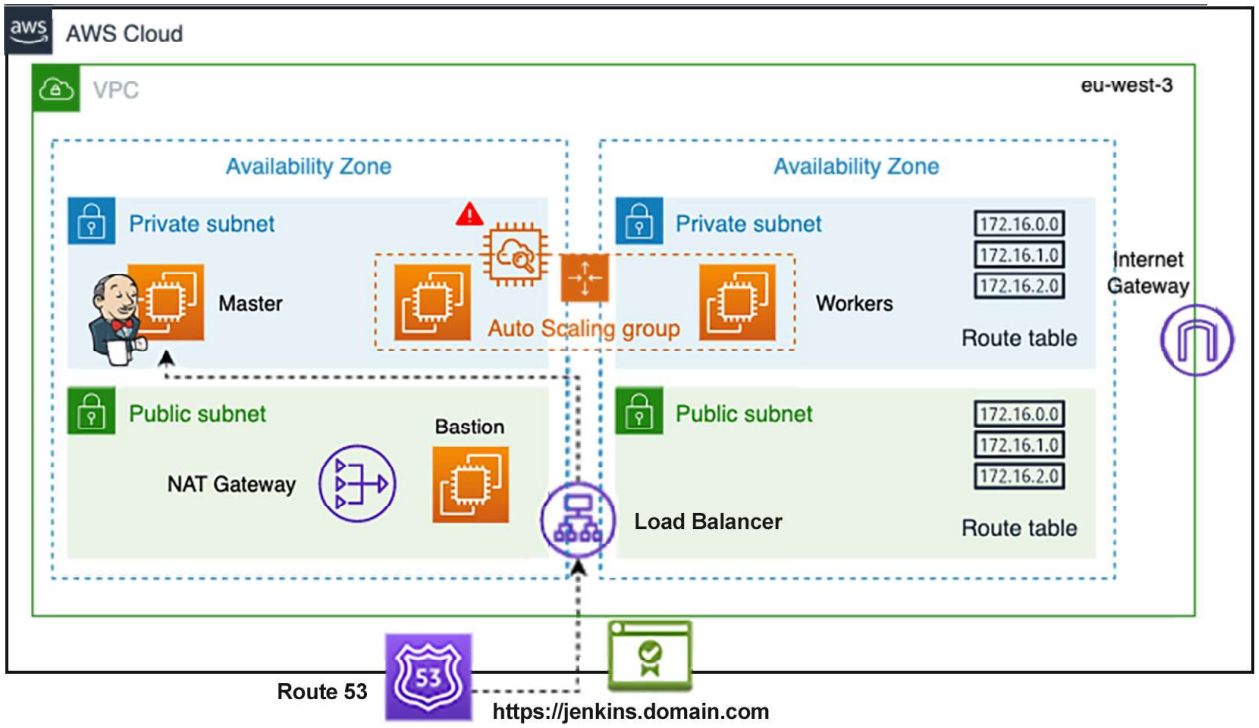

Figure 3.16 Jenkins cluster deployment on a custom VPC

Adding workers to a Jenkins cluster is the typical way to scale Jenkins. However, you can set up multiple Jenkins masters with a proxy (typically, HAProxy or NGINX) to actively monitor the primary master and reroute requests to backup masters if the

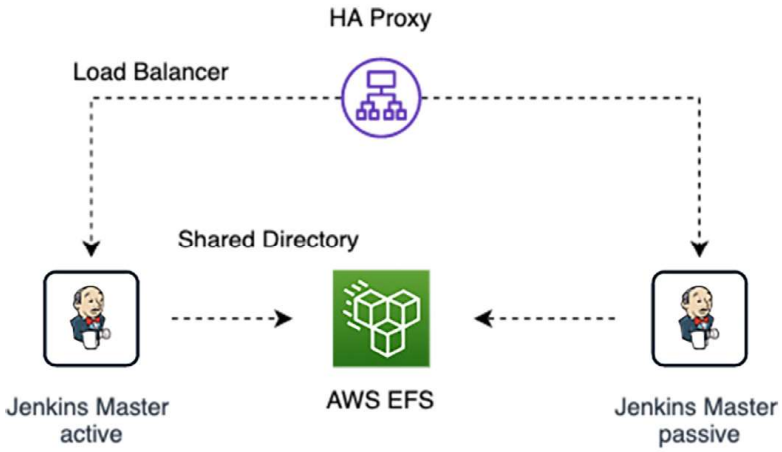

Figure 3.17 The Jenkins master HA setup uses Amazon Elastic File System to persist the Jenkins home directory.

active master goes down. The Jenkins architecture for master instances will look like figure 3.17.

As you can see, the first tier is the reverse proxy. Whenever an incoming request for the build occurs, it will first reach the proxy. Then, the proxy will decide the instance to which the request can be routed. Here, one of the masters will be in the active state to serve requests, and the other one will be passive. Whenever a problem exists with the active master and it goes down, the other master will become active, and requests will resume. (We also can deploy Jenkins masters inside an ASG to ensure that a minimum number of masters is always available for backup). These requests will then be served by the master that has become active.

The second tier is Amazon Elastic File System, or EFS (https://aws.amazon.com/efs/), which is used as a storage solution to persist the Jenkins home directory $JENKINS_HOME so both Jenkins masters can access and store Jenkins jobs. This storage solution can be mounted on multiple Jenkins instances concurrently. Amazon EFS, like any Network File System (NFS) server, supports full filesystem access semantics such as strong consistency and file locking.

EFS can also be used if you plan to deploy Jenkins on a Kubernetes cluster or Docker-based orchestration platforms like AWS ECS or Fargate. As the Jenkins master container can be launched on any node in the cluster, EFS can be used to persist the Jenkins data directory to preserve its state.

> **NOTE** Chapter 14 covers how to mount EFS in the $JENKINS_HOME directory to ensure that 100% of data is shared and can't be lost in case of failure.

Now that the Jenkins architecture is clear, next we will prepare our AWS environment, and then install and configure the tools needed for upcoming chapters.

3.3.1 Preparing the AWS environment

This section will walk you through installing and configuring the AWS command line. The command-line interface (CLI) is a solid and mandatory tool that we'll use in

upcoming chapters. It will save us substantial time by automating the deployment and configuration of a Jenkins cluster on AWS with HashiCorp Terraform and Packer as well as defining CI/CD steps for cloud-native applications.

3.3.2 Configuring the AWS CLI

The AWS CLI (https://aws.amazon.com/cli/) is a powerful tool for managing your AWS services and resources from a terminal session. It was built on top of the AWS API, and hence everything that can be done through the AWS Management Console (https://console.aws.amazon.com/console/home) can be done with the CLI; this makes it a handy tool that can be used to automate and control your AWS infrastructure through scripts. Later chapters provide information on the use of the CLI with Jenkins to manage cloud-native applications in AWS.

Let's go through the installation process for the AWS CLI; you can find information on its configuration and testing in the AWS Management Console section. To get started, refer to the official documentation and follow the instructions to install the AWS CLI based on your operating system (http://mng.bz/Yw8N).

Once the AWS CLI is installed, you need to add the AWS CLI binary path to the `PATH` environment variable as follows:

- For Windows, press the Windows key and type `Environment Variables`. In the Environment Variables window, highlight the `PATH` variable in the System Variables section. Edit it and add a path by placing a semicolon right after the last path, and then enter the complete path to the folder where the CLI binary is installed.
- For Linux, Mac, or any UNIX system, open your shell's profile script (.bash_profile, .profile, or .bash_login) and add the following line to the end of the file:

```
export PATH=~/.local/bin:$PATH
```

Finally, load the profile into your current session:

```
source ~/.bash_profile
```

Verify that the CLI is correctly installed by opening a new terminal session and typing the following command:

```
aws --version
```

You should be able to see the AWS CLI version; in my case, 2.0.0 is installed. Let's test it out and list Amazon S3 buckets in the Frankfurt region as an example:

```
aws s3 ls --region eu-central-1
```

The previous command displays the following output:

```
jenkins:~ mlabouardy$ aws s3 ls --region eu-central-1
Unable to locate credentials. You can configure credentials by running "aws configure".
jenkins:~ mlabouardy$
```

When using the CLI, you'll generally need your AWS credentials to authenticate with AWS services. You can configure AWS credentials in multiple ways:

- *Environment credentials*—Use the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_KEY` variables. They can be useful for scripting or temporarily setting a named profile as the default.

NOTE If you set the environment variables at the terminal prompt, the values are saved for only the duration of the current session. To make the environment variable settings persistent across all terminal sessions, store them under /etc/profile or in ~/.bash_profile for the current user.

- *Shared Credentials file*—The AWS CLI stores the credentials in a local file named *credentials* under the .aws folder in your home directory. You can specify a non-default location for the credentials file by setting the `AWS_SHARED_CREDENTIALS_FILE` environment variable to another local path.
- *IAM roles*—If you're using the CLI in an EC2 instance, this removes the need to manage credential files in production. Each Amazon EC2 instance contains metadata that the AWS CLI can directly query for temporary credentials.

In the next section, I will show you how to create a new user for the AWS CLI with the AWS Identity and Access Management (IAM) service.

3.3.3 Creating and managing the IAM user

IAM (https://aws.amazon.com/iam/) is a service that allows you to manage users, groups, and their level of access to AWS services. It's strongly recommended that you do not use the AWS root account for any task except billing tasks, as it has the ultimate authority to create and delete IAM users, change billing, close the account, and perform all other actions on your AWS account. Therefore, we will create a new IAM user and grant it the permissions it needs to access the right AWS resources following the principle of least privilege.

NOTE The *principle of least privilege* (PoLP) works by giving a given user only the minimum levels of access—or permissions—needed to perform the required task.

Sign in to AWS Management Console by using your AWS email address and password. Then, open the IAM console from the Security, Identity & Compliance section or type `IAM` in the search bar; figure 3.18 shows the console.

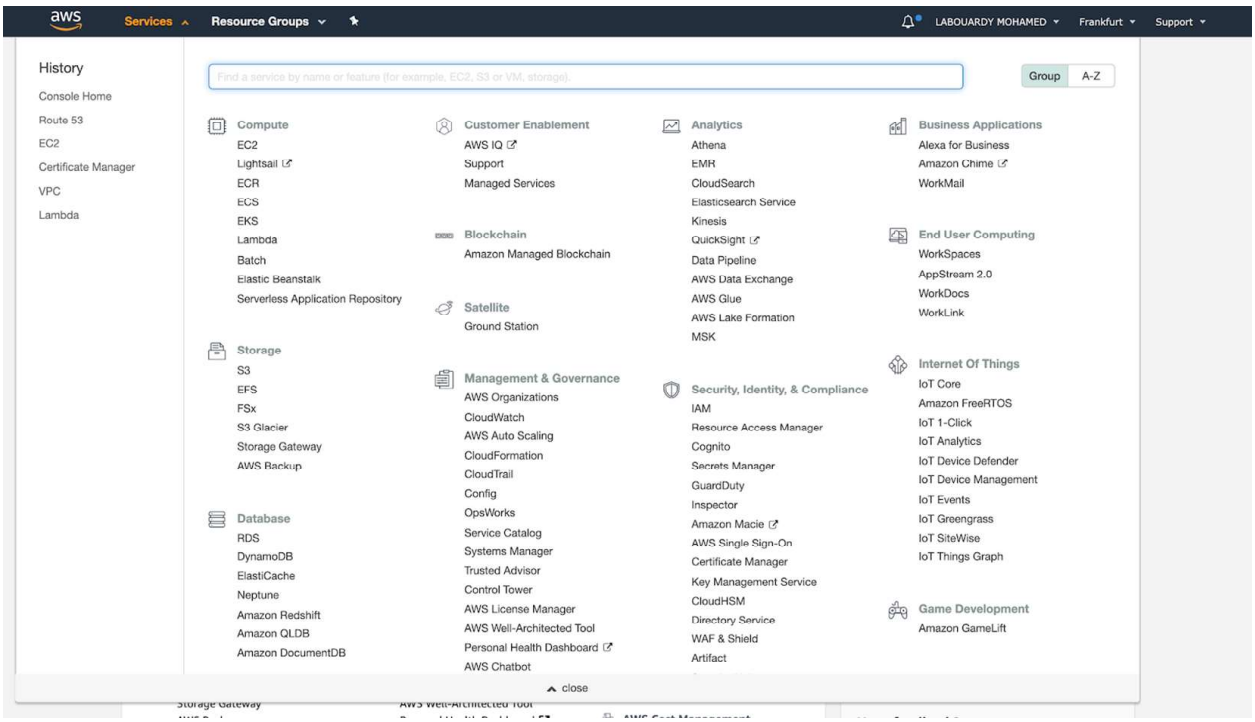

Figure 3.18 AWS Management Console

From the navigation pane, choose Users. Click the Add User button. Then set a name for the user and select Programmatic Access (also select AWS Management Console access if you want the same user to have access to the console), as shown in figure 3.19.

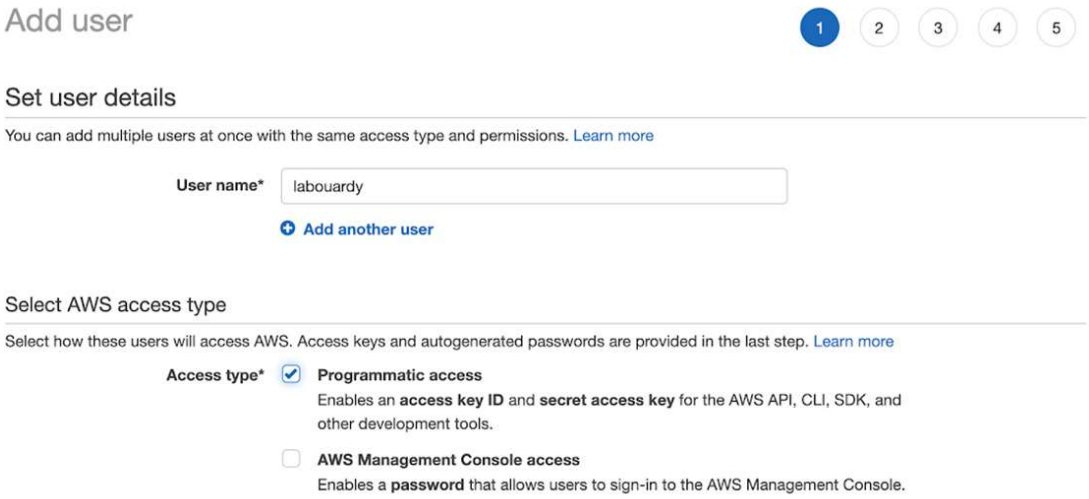

Figure 3.19 Creating a new IAM user

In the Set Permissions section, assign the AmazonS3FullAccess policy to the user, as shown in figure 3.20.

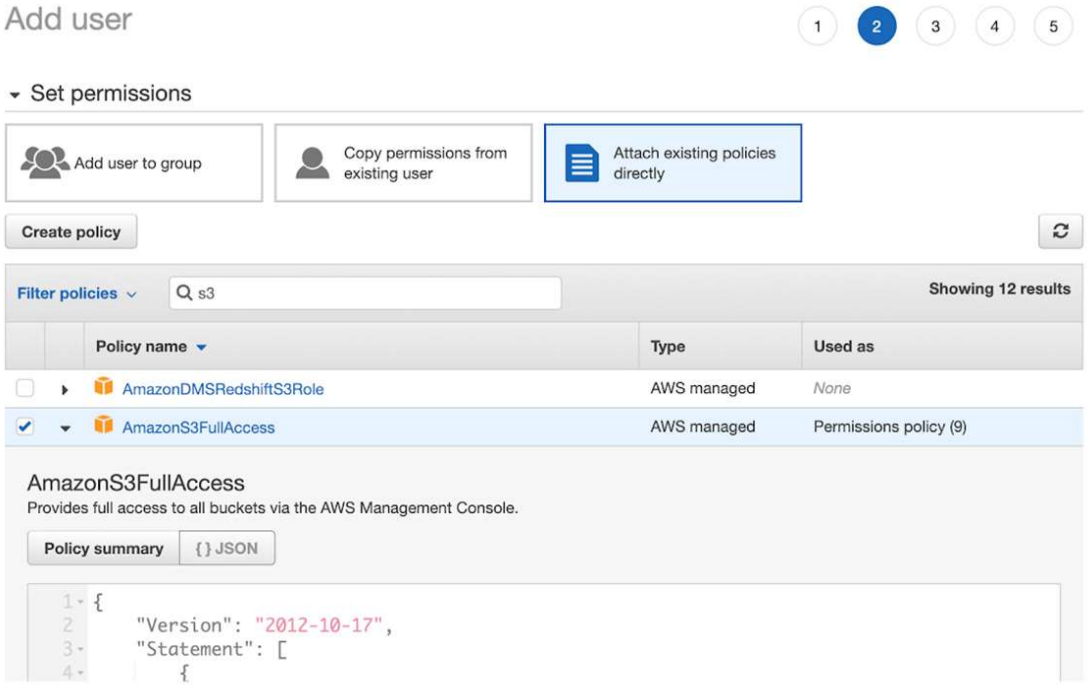

Figure 3.20 Attaching IAM policies to the user

> **NOTE** It's better to be granular and specify only permissions that are needed to get the job done (leave privilege access). Start with a minimum set of permissions and add more permissions only if necessary.

On the final page, you should see the user's AWS credentials (figure 3.21). Make sure you save the access keys in a safe location, as you won't be able to see them again.

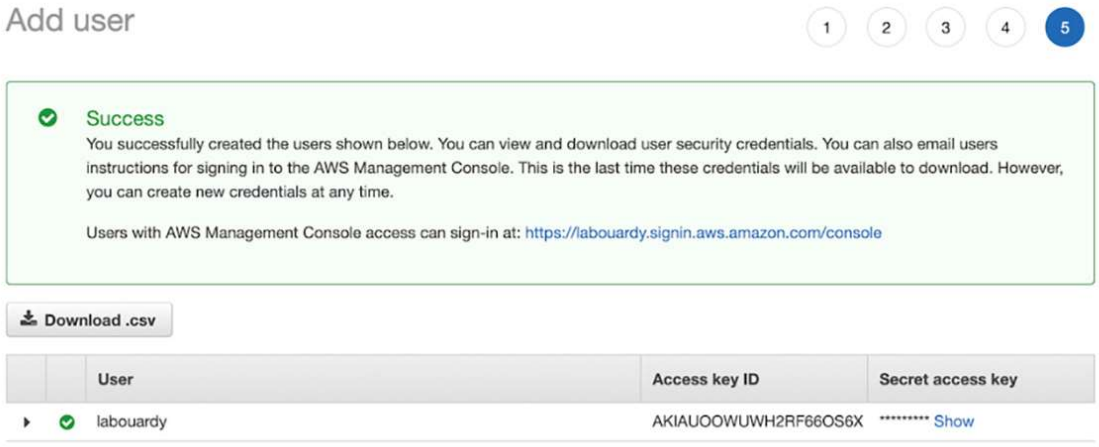

Figure 3.21 AWS credentials generation

NOTE You can create IAM users to represent users, applications, or services. In the next chapter, we will create dedicated IAM users for HashiCorp Terraform and Packer tools.

Next, configure the AWS CLI by using the `aws configure` command. The CLI will store credentials specified in the preceding command in a local file under ~/.aws/credentials (or in %UserProfile%\.aws\credentials on Windows) with the following content (substitute `eu-central-1` with your AWS region):

```
[default]
region=eu-central-1
aws_access_key_id=ACCESS KEY ID
aws_secret_access_key=SECRET ACCESS KEY
```

NOTE You can override the region in which your AWS resources are located by using the `AWS_DEFAULT_REGION` environment variable of the `--region` command-line option.

That should be it; try out the following command and, if you have an S3 bucket, you should be able to see the credentials listed. Otherwise, the command will return no results:

```
aws s3 ls
```

Now that the AWS environment is set up, let's get down to business and deploy a Jenkins cluster on AWS.

Summary

- Deploying Jenkins in distributed builds mode allows for decoupling orchestration, build executions, and better performance.
- Jenkins is a crucial component of the DevOps chain, and its downtime may have adverse effects on the DevOps environment. To overcome these, you need a high-availability setup for Jenkins.
- AWS CloudWatch provides a rich set of metrics to monitor the health of EC2 instances. The metrics collected can be used to set up alarms and trigger scaling policies upon alarm firing such as scaling Jenkins workers.
- Delegating the workload of building projects to worker nodes is referred to as distributed builds.
- You can configure a build to run on a particular worker machine by using Jenkins labels.
- It's highly recommended to launch your Jenkins deployment within a private subnet in a VPC for security purposes.
- By assigning labels to nodes, you can specify the resources you want to use for specific jobs, and set up graceful queuing for your tests.