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.

4 Baking machine images with Packer

This chapter covers

- Overview of immutable infrastructure
- Baking Jenkins machine images with Packer
- Discovering Jenkins essentials plugins
- Executing Jenkins Groovy scripts
- Using Packer provisioners to automate Jenkins settings

In the previous chapter, you learned how Jenkins distributed mode architecture works. In this one, we will get our hands dirty and deploy a Jenkins cluster on AWS. As a quick reminder, you learned that the Jenkins cluster is divided into two main components: master and worker. Before diving into the implementation of the distributed builds architecture, we will deploy the standalone mode, shown in figure 4.1, to cover some basics.

To deploy this architecture, we need to provision a server (for example, an EC2 instance in AWS). Then we'll install and configure Jenkins on the machine. While this manual process works, it's not efficient when we want to deploy Jenkins to

Figure 4.1 Jenkins standalone architecture on AWS

scale. Plus, updating or upgrading Jenkins can be lengthy and painful, and things can easily go wrong—breaking your CI/CD pipelines and impacting your product release as a result.

So instead of installing Jenkins after infrastructure creation (EC2 instance deployment) and applying updates on an existing Jenkins instance (in case of upgrades or maintenance), all changes must be packaged in a new machine image. A new Jenkins instance should be deployed based on the new image, and then the old server will be destroyed. This process creates what is known as an *immutable infrastructure*.

4.1 Immutable infrastructure

Immutable infrastructure is all about immutable components that are re-created and replaced instead of updated after infrastructure creation. This immutable infrastructure reduces the number of places where things can go wrong. This helps reduce inconsistency and improves reliability in the deployment process.

When an update is necessary for immutable infrastructure, new servers are provisioned with a preconfigured image, and old servers are destroyed. We create a new machine image that is built for deployment and use it for creating new servers. In immutable infrastructure, we are moving the configuration setup after the server creation process to the build process. As all deployments are done by new images, we can keep the history of previous releases in case of reverting to an old build. This allows us to reduce deployment time and the chance of configuration failure, and to scale deployments. Figure 4.2 illustrates the differences between immutable and mutable infrastructures.

Notice that the new Instance B, generated from a "golden" machine image, is provisioned upon the destruction of Instance A in the immutable pattern. Note, too, that there is no Jenkins downtime during instance replacement with well-architected immutable patterns that have multiple instances in service at a given time. By contrast, in the mutable pattern, Instance A isn't replaced. The same instance is modified manually or by using a script or tool, with the Jenkins updated from v1.0 to v2.0.

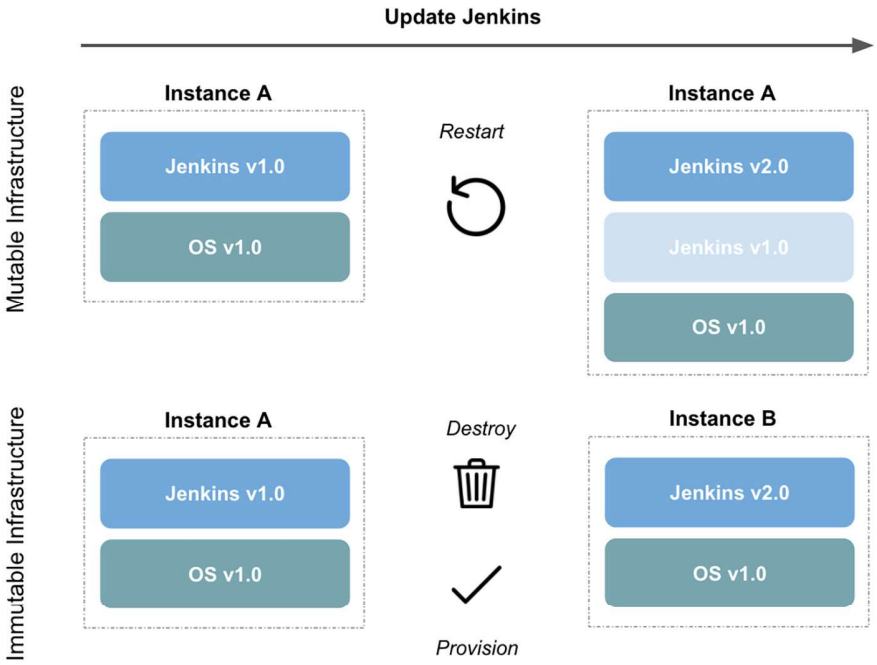

Figure 4.2 Updating via mutable and immutable infrastructures

In this era of cloud computing, many companies are adopting immutable infrastructure to simplify configuration management and improve reliability by using infrastructure as code. With immutable infrastructure, instead of making changes on a running server, we create a new server. Creating immutable infrastructure is hard and needs a sophisticated process for building and testing. The best way to implement immutable infrastructure is to use a well-tested and tried tool.

Multiple tools and frameworks allow you to build immutable infrastructure. The most famous ones are HashiCorp Packer, HashiCorp Vagrant, and Docker. In this book, we will keep our focus on machine images by using Packer. The goal is to illustrate the workflow for building immutable infrastructure and show how it can be fully automated using Packer. However, the same workflow can be applied while using other alternatives.

4.2 Introducing Packer

HashiCorp Packer (www.packer.io) is a lightweight and easy-to-use open source tool that automates the creation of any type of machine image for multiple platforms. Packer is not a replacement for configuration management tools like Ansible, Puppet, or Chef. Packer works with these tools to install and configure software and dependencies while creating images.

Packer uses a configuration file to create a machine image. Then it uses builders to spin up an instance on the target platform, and runs provisioners to configure applications or services. Once setup is done, it shuts down the instance and saves the new baked machine instance with any needed post-processing.

Using Packer has many advantages. Here are a few:

- *Fast infrastructure deployment*—Machine images allow us to more quickly launch provisioned and configured machines.
- *Scalable*—Packer installs and configures all needed software and dependencies for a machine during the image-creation process. The same image can be used to spawn any number of instances without doing extra configuration. (The same image can be used to deploy multiple Jenkins workers, for instance.)
- *Multiprovider support*—Packer can be used to create images for multiple cloud providers like AWS, GCP, and Microsoft Azure.

Figure 4.3 illustrates a typical machine image build process with Packer.

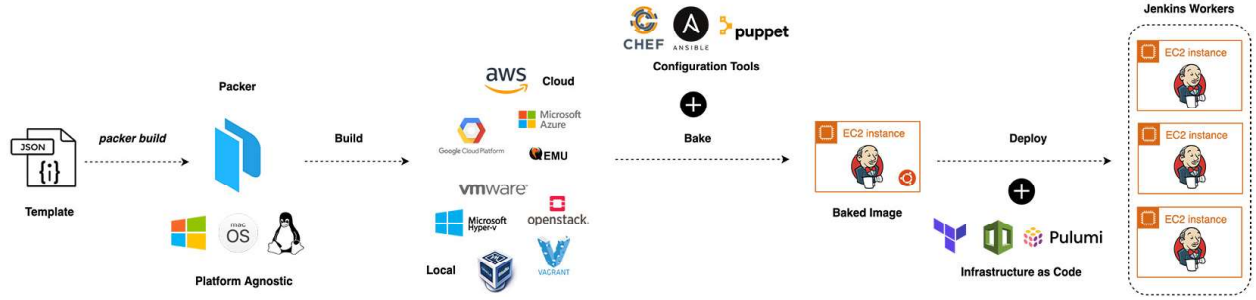

Figure 4.3 Building Jenkins machine images with Packer

The drawback of using Packer is managing existing images: you need to manage them yourself by using tags or versions and keep deleting old, unused images (in AWS, you're charged for the storage of the bits that make up your machine image, or AMI).

4.2.1 How does it work?

Figure 4.4 illustrates the process Packer uses to bake machine images.

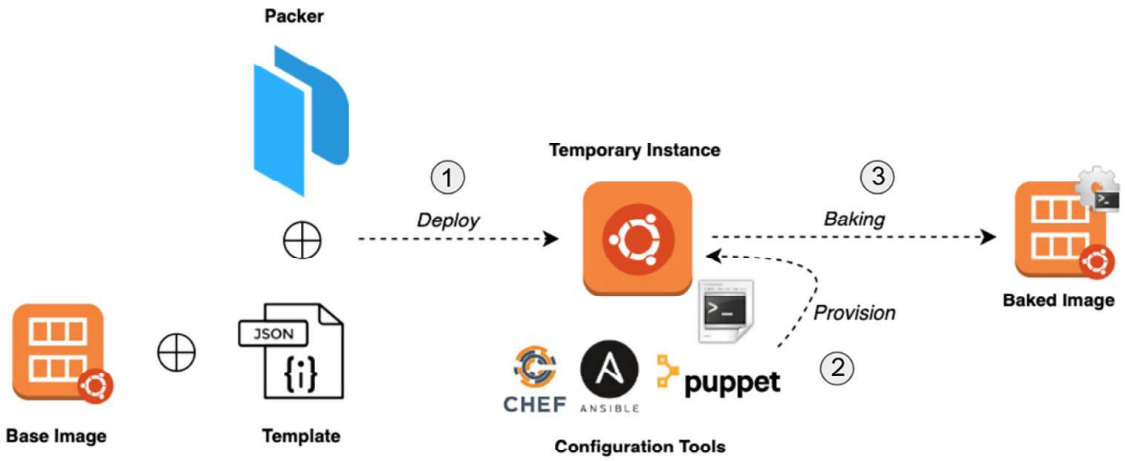

Figure 4.4 Packer baking workflow

Here are the steps in the process:

1. Boot a temporary instance using the base image defined in the template file.
2. Provision the instance by using configuration management tools like Ansible, Chef, or Puppet, or with a simple automated script to configure the instance into the desired state.
3. Create a new machine image from the temporary running instance and shut down the temporary instance after the image is baked.

Once a new machine image is created, booting a new server from this new image will give the same configuration that was already done on the temporary instance. This helps provide a smooth deployment process. This also helps scale our services fast.

The Packer configuration, also known as a template file, can be written in JSON or YAML format. It consists of the following three main components:

- *User variables*—This section is used to parameterize the Packer template file so we can keep secret, environment variables and other parameters out of the template. The section helps with the portability of the template file and helps in separating out the part that can be modified in our template. Variables can be passed through command lines, environment variables, HashiCorp Vault (www.vaultproject.io), or files. The section is a key-value mapping with the variable name assigned to a default value.
- *Builders*—This section contains a list of builders that Packer uses to generate a machine image. Builders are responsible for creating an instance and generating machine images from them. A builder maps to a single machine image. This section contains information including the type (which is the name of the builder), access keys, and credentials required to connect to the platform (AWS, for instance).
- *Provisioners*—This section, which is optional, contains a list of provisioners that Packer uses to install and configure software within a running instance before creating a machine image. The type specifies the name of a provisioner such as Shell, Chef, or Ansible.

NOTE For a full list of supported builders, refer to the official documentation at www.packer.io/docs/builders/. For a full list of supported provisioners, see www.packer.io/docs/provisioners/.

Packer helps bake configuration into the machine image during image creation time. This helps in creating identical servers in case things go wrong.

4.2.2 Installation and configuration

Packer is written in Go, which is a compiled language. Hence, installing Packer is straightforward; you just need to download the appropriate binary for your system and architecture from www.packer.io/downloads/. Figure 4.5 shows the download page.

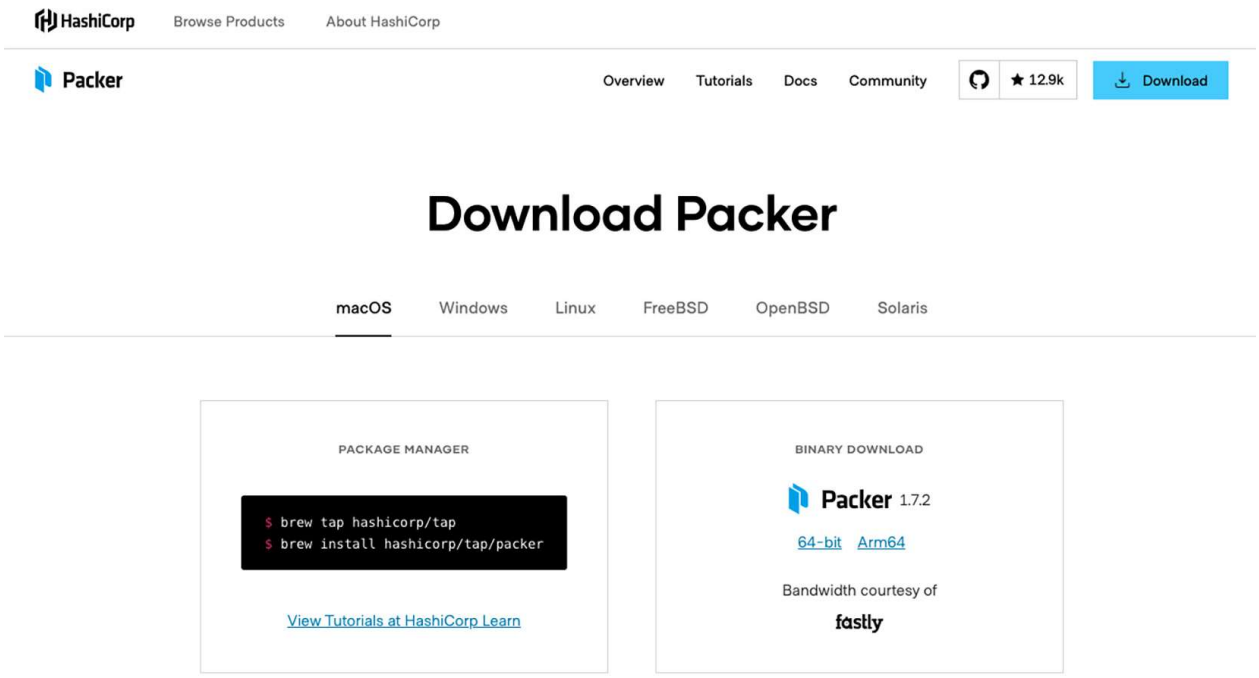

Figure 4.5 Packer download page

NOTE Make sure the directory where you installed the Packer binary is on the PATH variable.

After installing Packer, verify that the installation is working by opening a new terminal session and checking that Packer is available by issuing the following command:

```
jenkins:~ mlabouardy$ packer
Usage: packer [--version] [--help] <command> [<args>]

Available commands are:
    build       build image(s) from template
    console     creates a console for testing variable interpolation
    fix         fixes templates from old versions of packer
    inspect     see components of a template
    validate    check that a template is valid
    version     Prints the Packer version
```

NOTE At the time of writing this book, the latest stable version of Packer is 1.7.2.

If you get an error that Packer could not be found, your PATH environment variable was not set up properly. Otherwise, Packer is installed, and you're ready to go!

4.2.3 Baking a machine image

With Packer installed, let's dive right into it and build our first image. Our first machine image will be an Amazon EC2 AMI with Jenkins pre-installed. To create this AMI, we need to write a Packer configuration file.

NOTE The following Packer template file has been cropped for brevity. The full template is available in the GitHub repository under the chapter4 folder: http://mng.bz/GO8q.

Create a template.json file and fill it with the following content.

Listing 4.1 Packer template for standalone Jenkins server

```
{
    "variables" : {
        "region" : "AWS REGION",
        "aws_profile": "AWS PROFILE",
        "source_ami" : "AMAZON LINUX AMI ID",
        "instance_type": "EC2 INSTANCE TYPE"
    },
    "builders" : [
        {
            "type" : "amazon-ebs",
            "profile" : "{{user `aws_profile`}}",
            "region" : "{{user `region`}}",
            "instance_type" : "{{user `instance_type`}}",
            "source_ami" : "{{user `source_ami`}}",
            "ssh_username" : "ec2-user",
            "ami_name" : "jenkins-master-2.204.1",
            "ami_description" : "Amazon Linux Image with Jenkins Server",
    ],
    "provisioners" : [            {
            "type" : "shell",
            "script" : "./setup.sh",
            "execute_command" : "sudo -E -S sh '{{ .Path }}'"
        }
    ]
}
```

This template file consists of three main sections:

- `variables`
- `builders`
- `provisioners`

Instead of hardcoding values in the template file, we are using variables that can be overridden at the Packer runtime. In our example, we have defined the variables in table 4.1.

Substitute the value of `source_ami` with the appropriate Amazon Linux AMI ID. The Amazon Linux AMI ID can be found by heading to AWS Management Console and navigating to the EC2 dashboard. Click Launch EC2 Instance. On the Choose AMI tab, type `Amazon Linux AMI` in the search bar, shown in figure 4.6.

Table 4.1 Packer variables

Variable	Description
`region`	The name of the AWS region, such as `eu-central-1`, in which to launch the EC2 instance to create the AMI. While you can always copy an AMI from one region to another, for simplicity the AMI location will be the same as the region where the Jenkins EC2 instance will be deployed to.
`aws_profile`	The AWS profile used. Check chapter 3 for details about AWS CLI configuration. You can also provide AWS credentials through environment variables or with EC2 metadata if you plan to run Packer inside an EC2 instance. If you plan to use AWS access and secrets keys, keep them out of the template and provide them only during runtime by using the `-var` flag.
`instance_type`	The EC2 instance type to use while building the AMI, such as a `t2.micro`. A list of supported instance types can be found at https://aws.amazon.com/ec2/instance-types/.
`source_ami`	The base AMI to use to boot the temporary EC2 instance. In the previous example, we're using the official Amazon Linux image. You may need to change the source AMI ID based on what images exist when this template is run and the AWS region you're using.

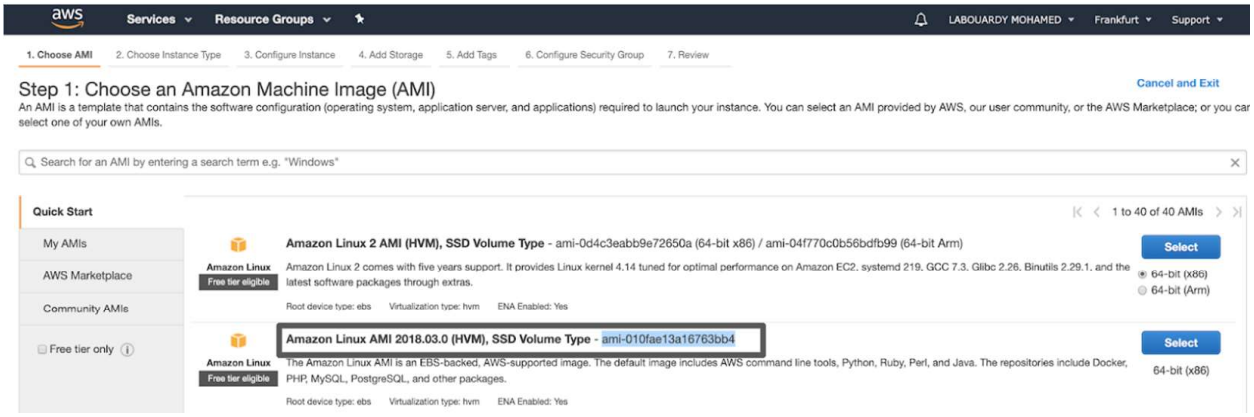

Figure 4.6 Amazon Linux image identifier

You can also find the ID programmatically with Packer by using the `source_ami_filter` attribute in the Packer template file. This attribute will automatically populate the `source_ami` attribute based on the defined filters. For instance, the following snippet selects the most recent Amazon Linux AMI (the full template file can be copied from chapter4/standalone/template-with-filter.json):

```
"builders" : [
        {
            "ami_name" : "jenkins-master-2.204.1",
            "ami_description" : "Amazon Linux Image with Jenkins Server",
```

```
            "source_ami_filter": {
                "filters": {
                  "virtualization-type": "hvm",
                  "name": "Amazon Linux AMI-*",
                  "root-device-type": "ebs"
                },
                "owners": ["amazon"],
                "most_recent": true
            }
        }
]
```

If multiple AMIs meet all of the filtering criteria provided in `source_ami_filter`, the `most_recent` attribute will select the newest Amazon Linux image.

Because the target machine image is an Amazon Machine Image, we are using the `amazon-ebs` builder. This is the Amazon EC2 AMI builder that ships with Packer. This builder builds an EBS-backed AMI by launching a source AMI, provisioning on top of that, and repackaging it into a new AMI. Multiple builders are available based on the target platform. Separate builders are available for EC2, VMware, VirtualBox, and others. Packer comes with many builders by default and can also be extended to add new builders.

The `ami_name` attribute in the `builder` section is the name of the resulting AMI that will appear when managing AMIs in the AWS console. The name must be unique. To help make this unique, I have added it as a prefix to the version of the installed Jenkins server, but you can also use the current timestamp with the following format:

```
"ami_name" : "jenkins-master-2.204.1-{{timestamp}}"
```

`{{timestamp}}` will be replaced by the Packer template engine to generate the current UNIX timestamp in Coordinated Universal Time (UTC).

The `provisioners` stage is responsible for installing and configuring all needed dependencies. Packer fully supports multiple modern configuration management tools such as Ansible, Chef, and Puppet. Bash scripts are also supported. To simplify the baking process for the Jenkins AMI, we have defined a bash script called setup.sh with the following content.

Listing 4.2 Bash script to install Jenkins LTS

```
#!/bin/bash
yum remove -y java
yum install -y java-1.8.0-openjdk
wget -O /etc/yum.repos.d/jenkins.repo
http://pkg.jenkins-ci.org/redhat-stable/jenkins.repo
rpm --import https://jenkins-ci.org/redhat/jenkins-ci.org.key
yum install -y jenkins
chkconfig jenkins on
service jenkins start
```

The script is self-explanatory: it installs the Java Development Kit (JDK), which is mandatory to run Jenkins, and then it installs the latest stable version of Jenkins. Here we install the Jenkins LTS release. Although it might lag behind in terms of new features, it provides more stability than weekly releases. The weekly Jenkins releases deliver bug fixes and new features rapidly to users and plugin developers who need them. But for more conservative users, it's preferable to stick to a release line that changes less often and receives only important bug fixes.

Once the Jenkins package is installed with the Yum package manager, the script configures Jenkins to start automatically if the machine has been restarted with the `chkconfig` command.

Now that our template file is defined, we can execute the following command to verify the syntax of the template file:

```
packer validate template.json
```

The command will return a zero exit status to indicate that the template.json syntax is valid.

Before we take this template and build an image from it, we need to assign the AmazonEC2FullAccess policy to the IAM user created in chapter 3 for Packer to be able to deploy an EC2 instance and create a machine image out of it.

Head back to AWS Console, navigate to the IAM dashboard, and jump to the Users section. Then, select the Packer user and attach the policy in listing 4.3, as shown in figure 4.7.

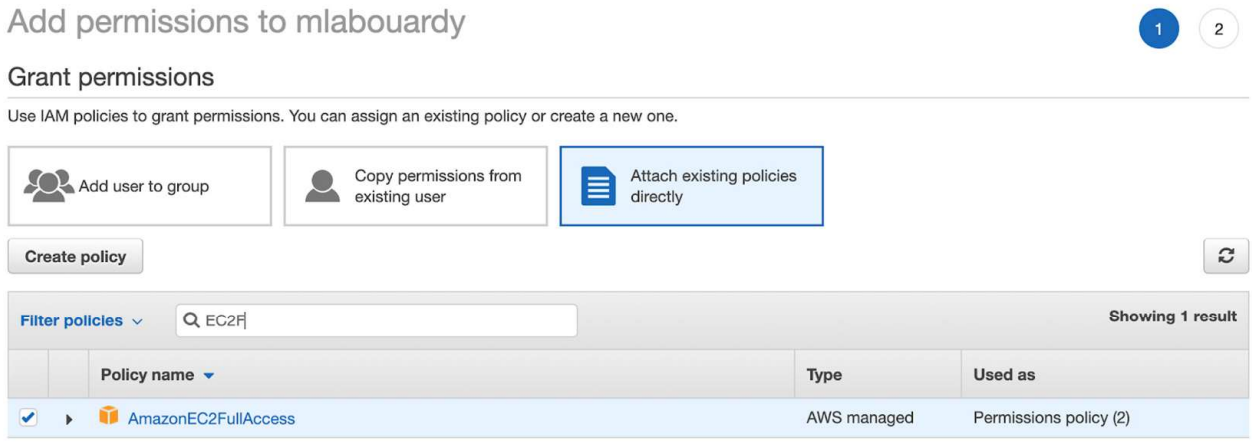

Figure 4.7 Attaching the EC2 policy to an IAM user

NOTE A preferred approach is to provide the minimal set of permissions necessary for Packer to work. The following listing is an IAM policy with the minimal set permissions necessary for the Amazon plugin to work.

Listing 4.3 AWS IAM policy for Packer

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action" : [
          "ec2:AttachVolume",
          "ec2:AuthorizeSecurityGroupIngress",
          "ec2:CopyImage",
          "ec2:CreateImage",
          "ec2:CreateKeypair",
          "ec2:CreateSecurityGroup",
          "ec2:CreateSnapshot",
          "ec2:CreateTags",
          "ec2:CreateVolume",
          "ec2:DeleteKeyPair",
          "ec2:DeleteSecurityGroup",
          "ec2:DeleteSnapshot",
          "ec2:DeleteVolume",
          "ec2:DeregisterImage",
          "ec2:DescribeImageAttribute",
          "ec2:DescribeImages",
          "ec2:DescribeInstances",
          "ec2:DescribeInstanceStatus",
          "ec2:DescribeRegions",
          "ec2:DescribeSecurityGroups",
          "ec2:DescribeSnapshots",
          "ec2:DescribeSubnets",
          "ec2:DescribeTags",
          "ec2:DescribeVolumes",
          "ec2:DetachVolume",
          "ec2:GetPasswordData",
          "ec2:ModifyImageAttribute",
          "ec2:ModifyInstanceAttribute",
          "ec2:ModifySnapshotAttribute",
          "ec2:RegisterImage",
          "ec2:RunInstances",
          "ec2:StopInstances",
          "ec2:TerminateInstances"
        ],
        "Resource" : "*"
    }]
  }
```

With a properly configured IAM user, it is time to build your first image. This is done by calling the `packer build` command with the template file as an argument:

```
packer build template.json
```

Packer will deploy an EC2 instance based on the configuration specified in the template file, and then execute the bash script on the deployed instance. The

output should look similar to the following. Note that this process typically takes a few minutes.

```
    amazon-ebs: Total download size: 60 M
    amazon-ebs: Installed size: 61 M
    amazon-ebs: Downloading packages:
    amazon-ebs: Running transaction check
    amazon-ebs: Running transaction test
    amazon-ebs: Transaction test succeeded
    amazon-ebs: Running transaction
    amazon-ebs:   Installing : jenkins-2.204.1-1.1.noarch                    1/1
    amazon-ebs:   Verifying  : jenkins-2.204.1-1.1.noarch                    1/1
    amazon-ebs:
    amazon-ebs: Installed:
    amazon-ebs:   jenkins.noarch 0:2.204.1-1.1
    amazon-ebs:
    amazon-ebs: Complete!
    amazon-ebs: Starting Jenkins [  OK  ]
==> amazon-ebs: Stopping the source instance...
    amazon-ebs: Stopping instance
==> amazon-ebs: Waiting for the instance to stop...
==> amazon-ebs: Creating AMI jenkins-master-2.204.1 from instance i-04ce242efa89ee5cd
    amazon-ebs: AMI: ami-051933c5e0fc71592
==> amazon-ebs: Waiting for AMI to become ready...
==> amazon-ebs: Modifying attributes on AMI (ami-051933c5e0fc71592)...
    amazon-ebs: Modifying: description
==> amazon-ebs: Modifying attributes on snapshot (snap-0ddcaf514891ce646)...
==> amazon-ebs: Adding tags to AMI (ami-051933c5e0fc71592)...
==> amazon-ebs: Tagging snapshot: snap-0ddcaf514891ce646
==> amazon-ebs: Creating AMI tags
    amazon-ebs: Adding tag: "Tool": "Packer"
    amazon-ebs: Adding tag: "Author": "mlabouardy"
==> amazon-ebs: Creating snapshot tags
==> amazon-ebs: Terminating the source AWS instance...
==> amazon-ebs: Cleaning up any extra volumes...
==> amazon-ebs: No volumes to clean up, skipping
==> amazon-ebs: Deleting temporary security group...
==> amazon-ebs: Deleting temporary keypair...
Build 'amazon-ebs' finished.

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs: AMIs were created:
eu-central-1: ami-051933c5e0fc71592
```

At the end of running the `packer build` command, Packer outputs the artifacts that were created as part of the build. Artifacts are the results of a build and typically represent the AMI ID. (Your ID will surely be different from the preceding one.) In this example, we have only a single artifact: the AMI was created in the Frankfurt region (`eu-central-1`).

You can use the same template file to create Jenkins machine images for different platforms, all from the same specification. This is a nice feature that allows you to create machine images of different types of providers without repetitive coding. For example, we can modify the template to add Google Compute Cloud and Microsoft Azure builders to it, as shown in the following listing. The full template is available on the GitHub repository (chapter4/standalone/template-multiple-builders.json).

Listing 4.4 Jenkins multiplatform machine image builds

```
{
    "builders": [
        {
            "type": "amazon-ebs",
            "profile": "{{user `aws_profile`}}",
            "region": "{{user `region`}}",
            "instance_type": "{{user `instance_type`}}",
            "source_ami": "{{user `source_ami`}}",
            "ssh_username": "ec2-user",
            "ami_name": "jenkins-master-2.204.1",
            "ami_description": "Amazon Linux Image with Jenkins Server",
        },
        {
            "type": "azure-arm",
            "subscription_id": "{{user `subscription_id`}}",
            "client_id": "{{user `client_id`}}",
            "client_secret": "{{user `client_secret`}}",
            "tenant_id": "{{user `tenant_id`}}",
            "managed_image_resource_group_name": "{{user `resource_group`}}",
            "managed_image_name": "jenkins-master-v22041",
            "os_type": "Linux",
            "image_publisher": "OpenLogic",
            "image_offer": "CentOS",
            "image_sku": "8.0",
            "location": "{{user `location`}}",
            "vm_size": "Standard_B1ms"
        },
        {
            "type": "googlecompute",
            "image_name": "jenkins-master-v22041",
            "account_file": "{{user `service_account`}}",
            "project_id": "{{user `project`}}",
            "source_image_family": "centos-8",
            "ssh_username": "packer",
            "zone": "{{user `zone`}}"
        }
    ]
}
```

Packer will create multiple Jenkins images for multiple platforms in parallel, all configured from a single template. In this example, Packer can make an Amazon Machine Image, Azure image, and Google Compute Engine image in parallel, provisioned with the same script, resulting in a near-identical Jenkins image.

> **NOTE** For a step-by-step guide on how to bake machine images for Azure virtual machines and Google Compute Engine instances, refer to chapter 6.

Once the AMI is created, the temporary EC2 instance will be terminated by Packer, and the baked AMI will be available in the AMIs section under Images on the EC2 dashboard, as shown in figure 4.8.

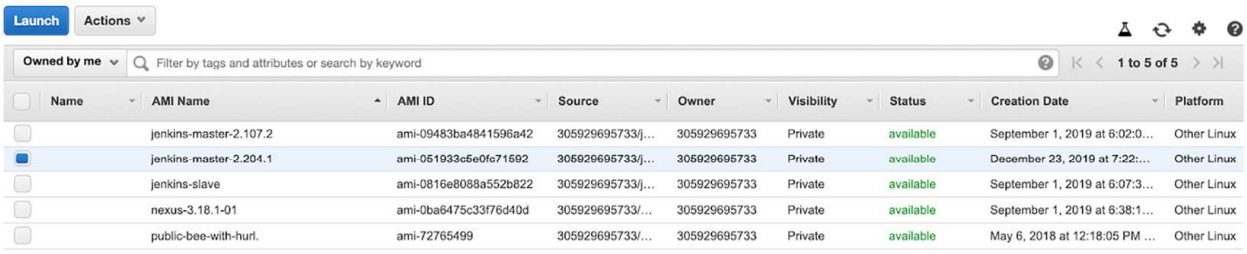

Figure 4.8 A new baked image is available on the Images section.

Now that our Jenkins AMI has been created, let's test it out and see if Jenkins has been properly installed. Jump to Instances and click the Launch Instance button. Then, select the AMI built by Packer from the My AMIs section, as shown in figure 4.9.

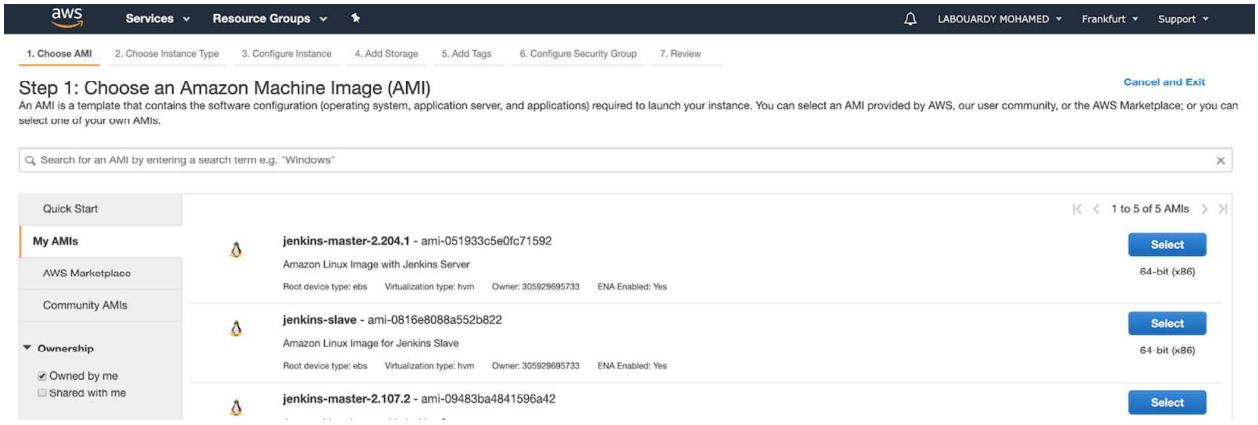

Figure 4.9 The new AMI can be selected from the My AMIs section.

For the instance type, select a general-purpose instance such as `t2.micro`, which is Free Tier eligible. We will cover Jenkins resource requirements in the next chapter.

For now, leave all the other values at their default settings. Navigate to the Add Tags section and type a name for your EC2 instance in the value box. This name, more correctly known as a *tag*, will appear in the console when the instance launches. This makes it easy to keep track of the running Jenkins instance.

Configure the security group (firewall that controls traffic to the instance) to allow traffic on port 8080 from anywhere. Port 8080 is the default port to which the Jenkins web dashboard is exposed.

NOTE The instance will be deployed inside the default VPC. In chapter 5, we will deploy the Jenkins cluster on a custom VPC from scratch and go through advanced network configurations.

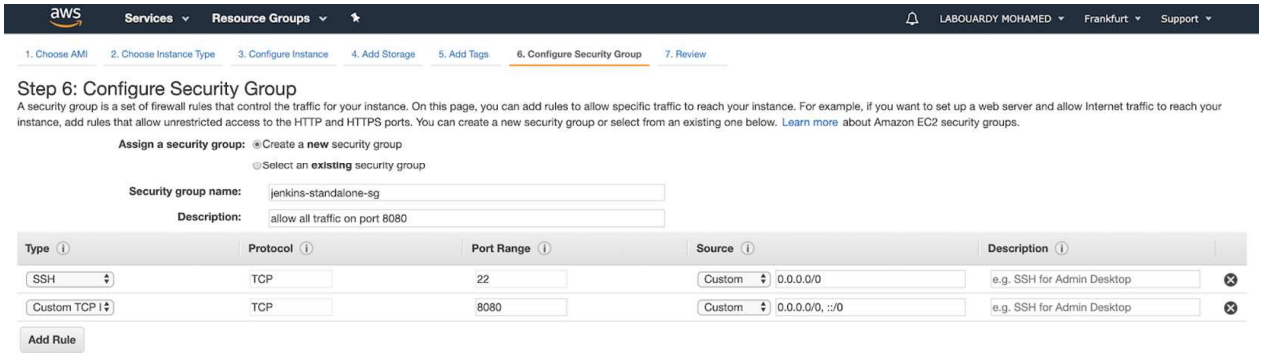

Figure 4.10 Allowing traffic on port 8080

The EC2 instance security group rules should look similar to figure 4.10.

Make sure to allow inbound traffic on port 22 in order to authorize SSH traffic from your computer's public IPv4 address. It's mandatory; otherwise, you won't be able to unlock the Jenkins dashboard later.

Finally, verify the configuration details in the Review section and select an SSH key pair, or create a new one if it's the first time you're launching an EC2 instance. This configuration will allow you to connect to your instance via SSH.

Once the instance is running, point your browser to the instance's public IP address and specify port 8080. The Jenkins setup wizard should pop up on the screen, as shown in figure 4.11. Congrats—you have successfully deployed a Jenkins instance from a custom AMI built with Packer.

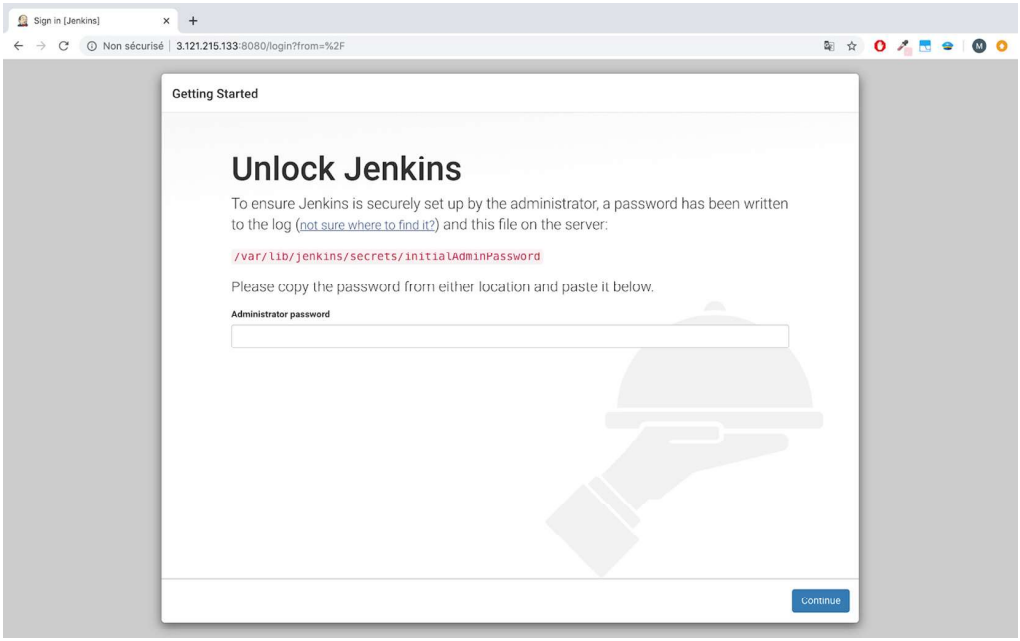

Figure 4.11 Jenkins setup wizard

You will be asked to unlock Jenkins by using an initial password. You can find this password inside the file /var/lib/jenkins/secrets/initialAdminPassword. (The following sections cover how to create a custom admin account for Jenkins.)

So far, we have deployed Jenkins in standalone mode. Figure 4.12 summarizes the currently deployed architecture.

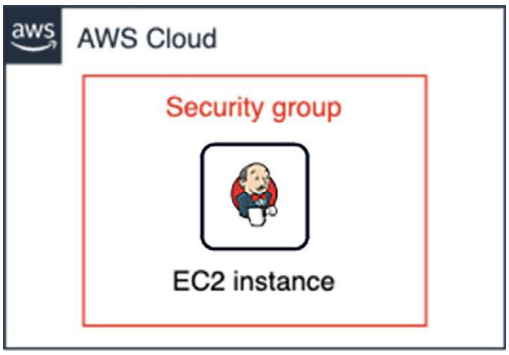

Figure 4.12 Jenkins standalone mode in AWS

NOTE Make sure to terminate the instance when you no longer need it, to stop incurring charges for that instance.

Next, you will learn how to use Groovy scripts to customize and configure Jenkins settings while baking the Jenkins master AMI. Furthermore, we will create another image for Jenkins workers to deploy Jenkins at scale.

4.3 Baking the Jenkins master AMI

We can use the AMI built in the previous section, but the ending Jenkins instance will still have many settings requiring manual configuration, including Jenkins admin credentials, needed plugins to set up CI/CD pipelines, and security checks. While you can configure those manually, the purpose of this book is to avoid operational overhead as much as possible. We want to automate the tedious tasks while deploying a highly available and fault-tolerant Jenkins cluster on your favorite cloud provider with few commands by using automation tools like HashiCorp Packer and Terraform.

NOTE When I say *high availability*, I am referring to a Jenkins cluster that can operate continuously without failure.

To fully automate a Jenkins master instance, we will use Jenkins post-initialization scripts. We will leverage the power of Groovy scripts and place them in the $JENKINS_HOME/init.groovy.d directory. This directory will be consumed by Jenkins upon startup. Therefore, it can be used to preconfigure Jenkins to the target desired state.

4.3.1 Configuring Jenkins upon startup

These scripts are written in Groovy and are executed inside the same Java Virtual Machine (JVM) as Jenkins, allowing full access to the domain model of Jenkins (we can access classes in Jenkins and all its plugins).

NOTE Another alternative to Groovy scripts is the Jenkins Configuration as Code (JCasC) plugin. For more details, refer to the official guide on GitHub: http://mng.bz/zEJa.

The basic-security.groovy script in listing 4.5 creates a Jenkins user with full admin access. (You need to replace the `USERNAME` and `PASSWORD` attributes with your own values.) Furthermore, by default, the anonymous read access is disabled by default, which means Jenkins requires authentication to access the web dashboard. However, you can enable anonymous read access by adding the `strategy.setAllowAnonymousRead(true)` instruction before the `instance.save()` statement.

Listing 4.5 basic-security.groovy script

```
#!groovy

import jenkins.model.*
import hudson.security.*

def instance = Jenkins.getInstance()
def hudsonRealm = new HudsonPrivateSecurityRealm(false)
hudsonRealm.createAccount('USERNAME','PASSWORD')
instance.setSecurityRealm(hudsonRealm)

def strategy = new FullControlOnceLoggedInAuthorizationStrategy()
instance.setAuthorizationStrategy(strategy)
instance.save()
```

Gets an instance of the Jenkins model

Creates a new user account by registering a password to the user

Gives full access to logged-in users

In addition to user management, we will also set some basic configurations for hardening Jenkins to protect against CSRF attacks. With CSRF protection enabled, all issued tokens should include a web session to prevent external attackers from obtaining web sessions. However, if your automation script uses a CSRF token for authentication, you can install the Strict Crumb Issuer plugin (available in the list of plugins installed while baking the Jenkins image) to exclude the web session ID from the validation criteria. We will enable CSRF protection with the csrf-protection.groovy script in the following listing.

Listing 4.6 csrf-protection.groovy script

```
#!groovy

import hudson.security.csrf.DefaultCrumbIssuer
import jenkins.model.Jenkins

def instance = Jenkins.getInstance()
instance.setCrumbIssuer(new DefaultCrumbIssuer(true))
instance.save()
```

Enables CSRF protection by setting up a crumb issuer

This option is enabled by default in new installations, starting with Jenkins 2.*x*. You can also enable CSRF by updating `JENKINS_JAVA_OPTIONS`. Add the following argument:

```
JENKINS_JAVA_OPTIONS="-Dhudson.security.csrf.DefaultCrumbIssuer=true"
```

NOTE If you're using the Jenkins linter feature to validate Jenkinsfiles against a Jenkins server protected from CSRF, you need to use an API token that doesn't require a CSRF token (crumb) since Jenkins 2.96.

Jenkins has a built-in CLI that allows users and administrators to access Jenkins from a script or a shell environment. The use of the CLI is not recommended for security reasons (to prevent remote access). Hence, we will disable it through the disable-cli.groovy script in the following listing.

Listing 4.7 disable-cli.groovy script

```
#!groovy

import jenkins.model.Jenkins

Jenkins jenkins = Jenkins.getInstance()
jenkins.CLI.get().setEnabled(false)
jenkins.save()
```

Gets an instance of Jenkins and disabled CLI access

We will also disable the JNLP and old unencrypted protocols (JNLP-connect, JNLP2-connect, JNLP3-connect, and CLI-connect) to get rid of the warning messages in the web dashboard. The script disable-jnlp.groovy is in the following listing.

Listing 4.8 disable-jnlp.groovy script

```
#!groovy

import jenkins.model.Jenkins
import jenkins.security.s2m.*

Jenkins jenkins = Jenkins.getInstance()
jenkins.setSlaveAgentPort(-1)
HashSet<String> newProtocols = new HashSet<>(jenkins.getAgentProtocols());
newProtocols.removeAll(Arrays.asList(
        "JNLP3-connect", "JNLP2-connect", "JNLP-connect", "CLI-connect"
));
jenkins.setAgentProtocols(newProtocols);
jenkins.save()
```

Sets 0 to indicate random available TCP port, -1 to disable this service

Initializes HashSet structure with available agent protocols, removes old unencrypted protocols from the structure, and saves the new list

Adding credentials to a new, local Jenkins server for development or troubleshooting can be a daunting task. However, with Groovy scripts and the right setup, developers can automate adding the required credentials into the new Jenkins server.

The Groovy script in listing 4.9 creates SSH credentials based on the AWS key pair we will use to deploy Jenkins worker instances. The SSH credentials object is created by using the `BasicSSHUserPrivateKey` constructor, which takes as parameters the credentials scope, username, SSH private key, and passphrase. The use of these SSH credentials will be illustrated in chapter 5.

Listing 4.9 node-agent.groovy script

```
import jenkins.model.*
import com.cloudbees.plugins.credentials.*
import com.cloudbees.plugins.credentials.common.*
import com.cloudbees.plugins.credentials.domains.*
import com.cloudbees.plugins.credentials.impl.*
import com.cloudbees.jenkins.plugins.sshcredentials.impl.*
import hudson.plugins.sshslaves.*;

domain = Domain.global()
store = Jenkins.instance
.getExtensionList('com.cloudbees.plugins.credentials \
                  .SystemCredentialsProvider')[0].getStore()

slavesPrivateKey = new BasicSSHUserPrivateKey(CredentialsScope.GLOBAL,
         "Jenkins-workers",
         "Ec2-user",
         new BasicSSHUserPrivateKey.UsersPrivateKeySource(),
         "", "")
store.addCredentials(domain, slavesPrivateKey)
```

Creates a Jenkins credential of type "SSH Username with private key." The constructor takes the username, private key, passphrase, and description as arguments.

NOTE Now every time the Jenkins server is restarted, the scripts will run and apply configuration for you. You don't need to worry about executing these settings manually every time the server restarts.

You can use Groovy init scripts to customize Jenkins and enforce the desired state. Although writing Groovy scripts requires knowing Jenkins internals and API, you've seen how to configure the common tasks and settings with Groovy scripts upon Jenkins initialization. We still need to install plugins to extend Jenkins functionalities in order to be able to build CI/CD pipelines.

4.3.2 Discovering Jenkins plugins

Plugins can be easily installed from the Jenkins dashboard. However, the purpose of this section is to build a fully automated Jenkins AMI, because if you want to install many plugins, this manual process can be fairly long and boring. Therefore, we will use a script provided by the Jenkins community to install plugins, including their dependencies. The scripts take, as a parameter, a file containing the list of Jenkins plugins to be installed.

Table 4.2 lists some of the most useful plugins that help developers save time, as well as making their lives easier. The full list is in the GitHub repository at chapter4/distributed/master/config/plugins.txt.

Table 4.2 Essential Jenkins plugins

Plugin	Description
blueocean	Provides the new Jenkins user experience with sophisticated visualizations of CI/CD pipelines and a bundled pipeline editor that makes automating CI/CD workflows approachable by guiding the user through an intuitive and visual process to create a pipeline. Refer to chapter 2 to explore the key features of Blue Ocean mode.
git	Provides access to any Git server with support for fundamental Git operations within Jenkins pipelines. It can pull, fetch, check out, branch, list, merge, tag, and push Git repositories.
ssh-agent	Allows you to provide SSH credentials to builds via ssh-agent in Jenkins. The ssh-agent is a helper program to hold private keys used for public-key authentication.
ssh-credentials	Allows you to store SSH credentials in Jenkins. It is used to launch Jenkins workers via SSH and execute Docker commands on a Kubernetes cluster remotely over SSH.
slack	Provides Jenkins notification integration with Slack. It can be used to send Slack notifications with Jenkins job build status upon the completion of a CI/CD pipeline. This plugin does require some straightforward setup on the Slack side in order to connect and post messages.
credentials-binding	Allows credentials to be bound to environment variables for use from miscellaneous build steps. It gives you an easy way to package up all of a job's secret files and passwords, and access them using environment variables during the build.
github-pullrequest	Fundamental for integrating Jenkins with GitHub repositories, it supports GitHub pull requests, branches, and custom webhooks. GitHub will trigger a new hook each time a pull request is opened, and once Jenkins receives the hook, it will run the associated job.
job-dsl	Allows jobs to be defined in a programmatic form in a human-readable file. It can be used to create complex pipelines for Jenkins freestyle jobs.
jira	Does pretty much what it says on the tin. It allows developers to integrate Jira (www.atlassian.com/software/jira) into Jenkins to update Jira open issues within CI/CD pipelines. It also associates build and deployment information with relevant Jira tickets and exposes key information about the pipeline across Jira boards.
htmlpublisher	Useful for publishing HTML reports that your builds generate at build time. It can be used to generate code coverage HTML reports and track the percentage of tests covering your application source code in a user-friendly way.
email-ext	Can be used to send email notifications. It's highly customizable: you can configure notifications triggers, content, and recipients. Plus, it supports both plaintext and HTML for the email body.
sonar	Allows easy integration of SonarQube (www.sonarqube.org), the open source platform for continuous inspection of code quality and code security.
embeddable-build-status	Generates badges for all your Jenkins jobs that display, in real time, their build status. You can add these badges to your Git repository README.md file.

NOTE These are just some of the plugins we will use, and upcoming chapters offer dozens more to explore.

More than a thousand plugins are available to support almost every solution, tool, and process for building, deploying, and automating your projects within Jenkins pipelines. The Jenkins Plugins Index, shown in figure 4.13, has over more than 1,800 plugins at https://plugins.jenkins.io/, free for download and use.

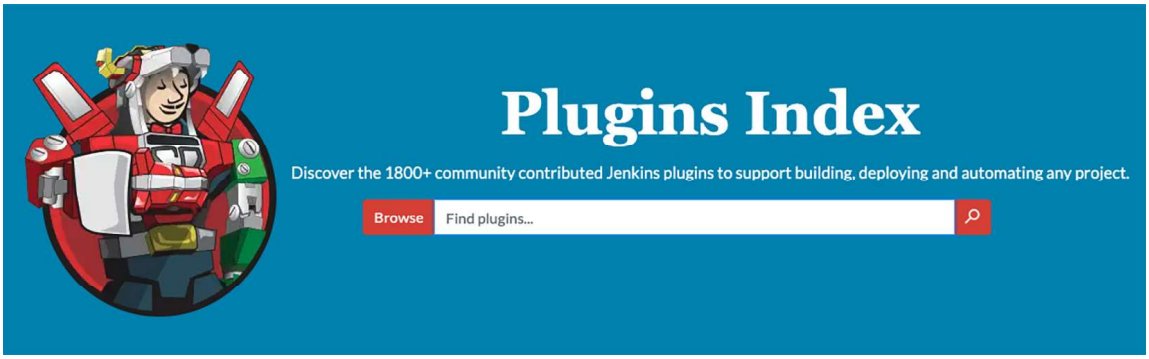

Figure 4.13 Jenkins plugins

NOTE Before installing a Jenkins plugin, make sure to review the changelog in the plugin's description page, as not all plugins may be safe to use. Also, always pick the latest stable version available.

Now you are more familiar with the essential Jenkins plugins. Let's go ahead and install them.

The script in listing 4.10 will go through the file containing a list of Jenkins plugins line by line, and then issue a cURL command to download the plugin from the Jenkins Plugins Index. Finally, the script will copy the downloaded plugin file to the /var/lib/jenkins/plugins folder. The listing illustrates the main function, and the full script can be downloaded from the GitHub repository at chapter4/distributed/master/config/install-plugins.sh.

Listing 4.10 install-plugins.sh script

```
#!/bin/bash
installPlugin() {
  if [ -f ${plugin_dir}/${1}.hpi -o -f ${plugin_dir}/${1}.jpi ]; then
    if [ "$2" == "1" ]; then
      return 1
    fi
    echo "Skipped: $1 (already installed)"
    return 0
  else
    echo "Installing: $1"
```

```
    curl -L --silent --output ${plugin_dir}/${1}.hpi  https://
     updates.jenkins-ci.org/latest/${1}.hpi
    return 0
  fi
}
```

The .hpi extension stood for *Hudson plugin* (remember, Jenkins was a fork of the Hudson project). With the move away from Hudson to Jenkins, this became *Jenkins plugin* and hence the .jpi format. Since the Jenkins v1.5 release, all .hpi plugin files are renamed automatically to .jpi at boot time.

By now, we have configured and automated all tasks needed to set up a running Jenkins server out of the box. Therefore, there's no need for the setup wizard at Jenkins startup (see figure 4.11). As a result, we will disable it by writing a Groovy init script. Create a skip-jenkins-setup.groovy script with the following content.

Listing 4.11 skip-jenkins-setup.groovy script

```
#!groovy

import jenkins.model.*
import hudson.util.*;
import jenkins.install.*;

def instance = Jenkins.getInstance()
instance.setInstallState(InstallState.INITIAL_SETUP_COMPLETED)
```

Finally, we will update the Packer template file used in the first section to copy the Groovy scripts described previously to the temporary instance by using the file *provisioner* (www.packer.io/docs/provisioners/file/). Next, we use a shell provisioner to move these files to the init.groovy.d folder. The template.json file should look similar to the following listing.

Listing 4.12 Jenkins master template file

```
{
    "variables" : {...},
    "builders" : [
        {
            "type" : "amazon-ebs",
            "profile" : "{{user `aws_profile`}}",
            "region" : "{{user `region`}}",
            "instance_type" : "{{user `instance_type`}}",
            "source_ami" : "{{user `source_ami`}}",
            "ssh_username" : "ec2-user",
            "ami_name" : "jenkins-master-2.204.1",
            "ami_description" : "Amazon Linux Image with Jenkins Server"
        }
    ],
    "provisioners" : [
        {
```

List of variables should be declared here such as: aws_profile, region, instance_type, and source_ami

Name of the baked machine image. The version number (2.204.1) should be replaced based on the current version you have installed.

```
            "type" : "file",
            "source" : "./scripts",
            "destination" : "/tmp/"
        },
        {
            "type" : "file",
            "source" : "./config",
            "destination" : "/tmp/"
        },
        {
            "type" : "file",
            "source" : "{{user `ssh_key`}}",
            "destination" : "/tmp/id_rsa"
        },
        {
            "type" : "shell",
            "script" : "./setup.sh",
            "execute_command" : "sudo -E -S sh '{{ .Path }}'"
        }
    ]
}
```

Copies the Groovy scripts folder from the local machine to /tmp in the host machine

Copies the configuration files from the local machine to /tmp in the host machine

Copies the user private SSH key to the /tmp folder

Executes the setup.sh shell script to copy the files from the /tmp folder to the right folder and installs Jenkins and its dependencies

NOTE The variables section has been omitted for brevity. The full template file can be found on GitHub at chapter4/distributed/master/template.json.

The SSH key can be generated with `ssh-keygen`. The command will provide a series of prompts. Feel free to use the defaults. However, from a security perspective, it's a good idea to enter a passphrase. Table 4.3 provides a complete list of Packer variables.

Table 4.3 Jenkins master Packer variables

Variable	Description
`region`	AWS region where the Jenkins master machine image will be created, such as `eu-central-1` (aka Frankfurt).
`aws_profile`	The profile to use in the shared credentials file for AWS. See Amazon's documentation on specifying profiles for more details: https://docs.aws.amazon.com/sdk-for-go/v1/developer-guide/configuring-sdk.html.
`instance_type`	The EC2 instance type to use while baking the target AMI, such as `t2.micro`, which is Free Tier eligible.
`source_ami`	The source AMI that the temporary instance will be based on. We're using the official Amazon Linux image. The ID should be updated according to the AWS region you're using. Refer to figure 4.6 for an example.
`ssh_key`	Private SSH key location (~/.ssh/id_rsa), the same key you will use to SSH to Jenkins worker instances. A Groovy script will be executed at boot time to add the private key as a credential on the Jenkins master to set up the initial connection with Jenkins workers over SSH.

Once files are uploaded to the temporary instance built by Packer, a setup.sh script will be executed to install the Jenkins LTS version. Next, the script installs the Git client (to clone GitHub repositories in advanced chapters). Then, it copies the workers' private SSH key to the /var/lib/jenkins/.ssh folder and set permissions. Finally, it moves Groovy scripts to the initialization folder, installs essentials plugins by executing the install-plugins.sh script, and starts the Jenkins server.

It's worth mentioning that scripts files were uploaded to the /tmp folder; Packer can upload files only to locations that the provisioning user (`ec2-user`) has permission to access. The following listing contains the content of setup.sh.

Listing 4.13 setup.sh script (install Jenkins)

```
#!/bin/bash
yum remove -y java
yum install -y java-1.8.0-openjdk
wget -O /etc/yum.repos.d/jenkins.repo
http://pkg.jenkins-ci.org/redhat-stable/jenkins.repo
rpm --import https://jenkins-ci.org/redhat-stable/jenkins-ci.org.key
yum install -y jenkins
chkconfig jenkins on

yum install -y git
mkdir /var/lib/jenkins/.ssh
touch /var/lib/jenkins/.ssh/known_hosts
chown -R jenkins:jenkins /var/lib/jenkins/.ssh
chmod 700 /var/lib/jenkins/.ssh
mv /tmp/id_rsa /var/lib/jenkins/.ssh/id_rsa
chmod 600 /var/lib/jenkins/.ssh/id_rsa
chown -R jenkins:jenkins /var/lib/jenkins/.ssh/id_rsa

mkdir -p /var/lib/jenkins/init.groovy.d
mv /tmp/*.groovy /var/lib/jenkins/init.groovy.d/
mv /tmp/jenkins /etc/sysconfig/jenkins
chmod +x /tmp/install-plugins.sh
bash /tmp/install-plugins.sh
service jenkins start
```

Installs JDK (minimum v1.8.0), which is required for Jenkins to be up and running

Installs Git client, which will be needed to clone project GitHub repositories in upcoming chapters

Copies the private SSH key used to deploy Jenkins workers/agents to JENKINS_HOME

Moves the Groovy scripts to init.groovy.d

Installs needed dependencies by running install-plugins.sh

Starts the Jenkins service

The template directory structure should look like the following. The scripts directory holds initial configuration and seeding scripts. The config folder contains the list of essential plugins to install, as well as the shell script to install plugins from the Jenkins Plugin Index:

```
├── config
│   ├── install-plugins.sh
│   ├── jenkins
│   └── plugins.txt
├── scripts
│   ├── basic-security.groovy
│   ├── csrf-protection.groovy
│   └── disable-cli.groovy
```

```
│   ├── disable-jnlp.groovy
│   ├── node-agent.groovy
│   └── skip-jenkins-setup.groovy
├── setup.sh
└── template.json
```

> **NOTE** Jenkins captures launch configuration parameters in the /etc/sysconfig/jenkins file. If you want to add Java arguments, it's the file you're looking for.

Prior to building the AMI, it's a good idea to validate the syntactical correctness of the template file by issuing the `packer validate` command. `Template validated successfully` is the expected output if the template is valid.

Now that the template is validated, we will bake the AMI with the `packer build` command:

```
packer build template.json
```

The process can take several minutes. Output similar to this is expected:

```
==> amazon-ebs: Prevalidating any provided VPC information
==> amazon-ebs: Prevalidating AMI Name: jenkins-master-2.204.1
    amazon-ebs: Found Image ID: ami-010fae13a16763bb4
==> amazon-ebs: Creating temporary keypair: packer_5e10856e-9f7d-06d4-88dc-739f5737cad3
==> amazon-ebs: Creating temporary security group for this instance: packer_5e10856f-2ed6-76c3-17d9-eac59e672186
==> amazon-ebs: Authorizing access to port 22 from [0.0.0.0/0] in the temporary security groups...
==> amazon-ebs: Launching a source AWS instance...
==> amazon-ebs: Adding tags to source instance
    amazon-ebs: Adding tag: "Name": "packer-builder"
    amazon-ebs: Instance ID: i-03b1fb7327193394f
==> amazon-ebs: Waiting for instance (i-03b1fb7327193394f) to become ready...
==> amazon-ebs: Using ssh communicator to connect: 18.184.247.211
==> amazon-ebs: Waiting for SSH to become available...
==> amazon-ebs: Connected to SSH!
==> amazon-ebs: Uploading ./scripts/basic-security.groovy => /tmp/basic-security.groovy
basic-security.groovy 419 B / 419 B [==========================================================================]
==> amazon-ebs: Uploading ./scripts/disable-cli.groovy => /tmp/disable-cli.groovy
disable-cli.groovy 174 B / 174 B [==============================================================================]
==> amazon-ebs: Uploading ./scripts/csrf-protection.groovy => /tmp/csrf-protection.groovy
csrf-protection.groovy 228 B / 228 B [==========================================================================]
==> amazon-ebs: Uploading ./scripts/disable-jnlp.groovy => /tmp/disable-jnlp.groovy
disable-jnlp.groovy 463 B / 463 B [==============================================================================]
==> amazon-ebs: Uploading ./scripts/skip-jenkins-setup.groovy => /tmp/skip-jenkins-setup.groovy
skip-jenkins-setup.groovy 182 B / 182 B [========================================================================]
==> amazon-ebs: Uploading ./config/jenkins => /tmp/jenkins
jenkins 3.09 KiB / 3.09 KiB [====================================================================================]
==> amazon-ebs: Uploading /Users/mlabouardy/keys/komiser.pem => /tmp/id_rsa
komiser.pem 1.66 KiB / 1.66 KiB [================================================================================]
==> amazon-ebs: Uploading ./scripts/node-agent.groovy => /tmp/node-agent.groovy
node-agent.groovy 705 B / 705 B [================================================================================]
==> amazon-ebs: Uploading ./plugins.txt => /tmp/plugins.txt
plugins.txt 1.85 KiB / 1.85 KiB [================================================================================]
==> amazon-ebs: Uploading ./install-plugins.sh => /tmp/install-plugins.sh
install-plugins.sh 1.18 KiB / 1.18 KiB [==========================================================================]
==> amazon-ebs: Provisioning with shell script: ./setup.sh
    amazon-ebs: Install Jenkins stable release
    amazon-ebs: Loaded plugins: priorities, update-motd, upgrade-helper
    amazon-ebs: Resolving Dependencies
```

If the script succeeds, Packer should show a message containing the AMI ID, and the Jenkins master AMI will be available in the EC2 dashboard, as shown in figure 4.14.

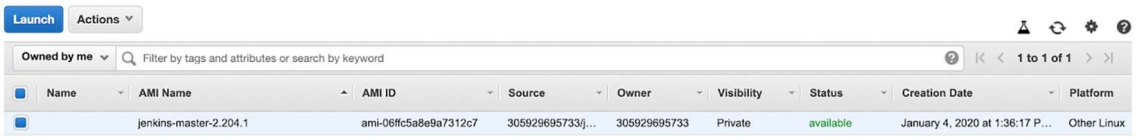

Figure 4.14 Jenkins master AMI

> **NOTE** The AMI name should be unique. Therefore, you might need to delete the existing image from your AWS account if it exists already.

Finally, we can spin up an EC2 instance based on the baked AMI. Once the instance is running, point your browser to the instance's public IP address on port 8080. After a while, you'll see the screen in figure 4.15.

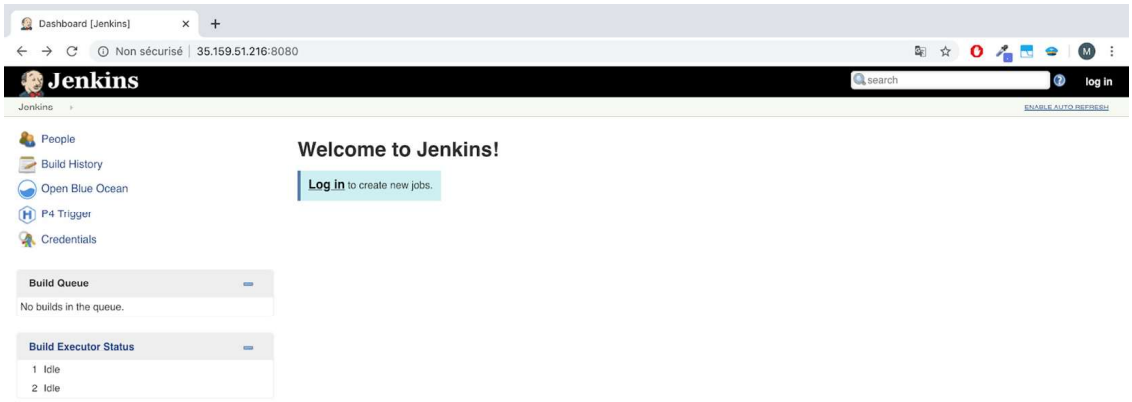

Figure 4.15 Jenkins web dashboard

This time, the setup wizard should disappear and many functionalities should be added. Sign in using the admin credentials defined in the basic-security.groovy script from listing 4.5. After login, you can verify that Jenkins credentials are created by going to the Credentials item on the left; see figure 4.16. So far, only the Jenkins

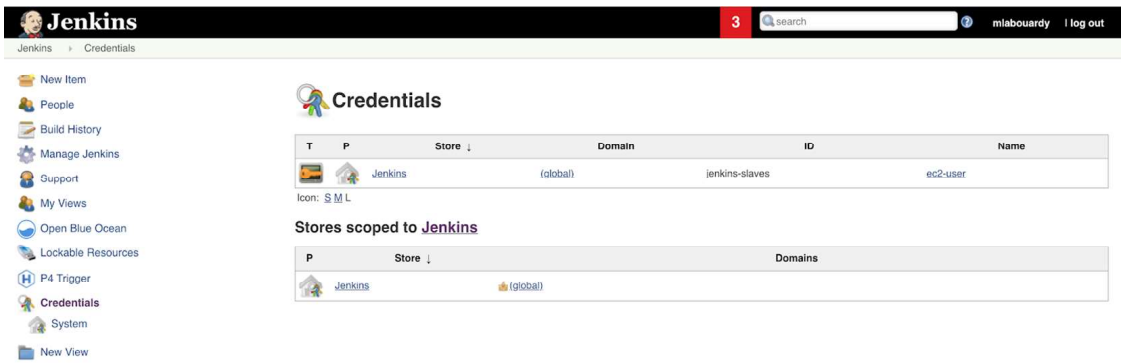

Figure 4.16 Jenkins credentials

worker SSH credential has been created (see listing 4.9), but you can customize the Groovy script to create additional credentials for external services like GitHub, Nexus, or SonarQube.

Moreover, the essential plugins were also installed. Jump to Manage Jenkins from the home page and then navigate to Plugins. You should see a list of plugins installed by default on the Installed tab, as shown in figure 4.17.

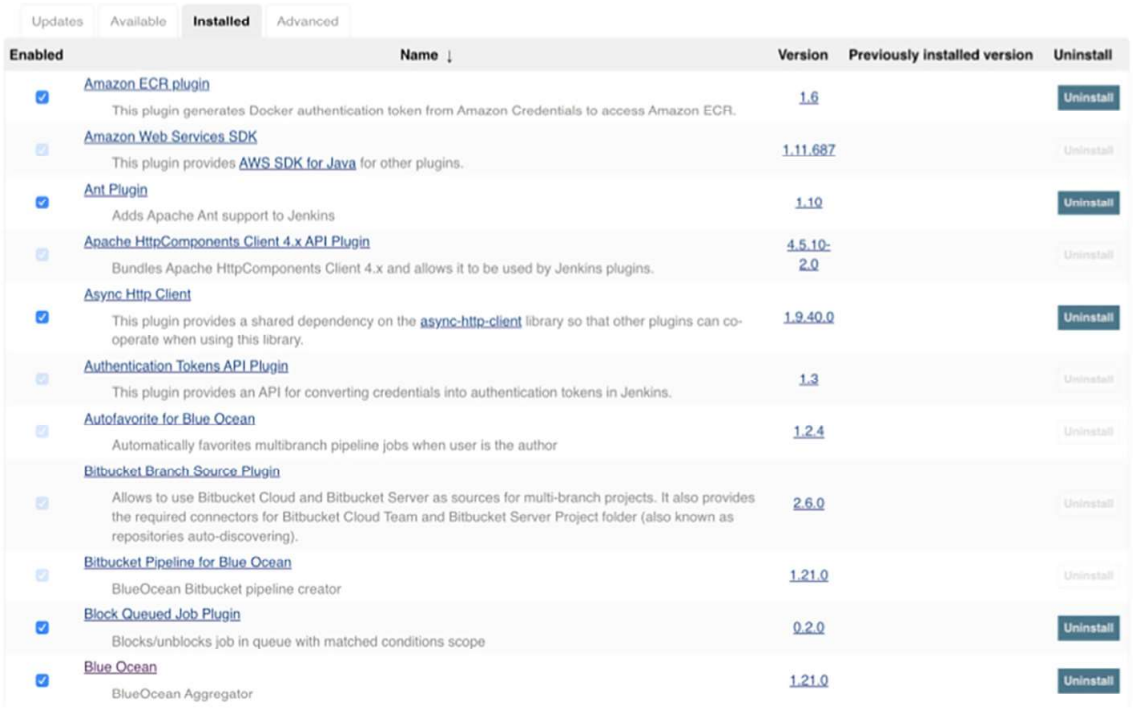

Figure 4.17 Jenkins installed plugins

Now that we have defined a Jenkins configuration as code, we can spawn it as many times as possible, on different machines, with the same result. And we've had no tiresome manual walks through the GUI.

4.4 Baking the Jenkins worker AMI

The Jenkins worker AMI baking process should be straightforward; see the following listing. The only requirement for an instance to be a Jenkins worker or build agent is to have a JDK. Modern Jenkins versions require a Java 8 runtime environment.

Listing 4.14 Jenkins worker template file

```
{
    "variables" : {...},
    "builders" : [
        {
```

```
            "type" : "amazon-ebs",
            "profile" : "{{user `aws_profile`}}",
            "region" : "{{user `region`}}",
            "instance_type" : "{{user `instance_type`}}",
            "source_ami" : "{{user `source_ami`}}",
            "ssh_username" : "ec2-user",
            "ami_name" : "jenkins-worker",
            "ami_description" : "Jenkins worker's AMI",

    ],
    "provisioners" : [
        {
            "type" : "shell",
            "script" : "./setup.sh",
            "execute_command" : "sudo -E -S sh '{{ .Path }}'"
        }
    ]
}
```

The variables in table 4.4 should be provided during build time within the template file or with the `-var` flag.

Table 4.4 Jenkins worker Packer variables

Variable	Description
`region`	AWS region where the Jenkins worker machine image will be created. Similar to the Jenkins master AWS region value.
`aws_profile`	The profile to use in the shared credentials file for AWS. See Amazon's documentation on specifying profiles for more details: http://mng.bz/01Yx.
`instance_type`	The EC2 instance type to use while baking the target AMI, such as `t2.micro`, which is Free Tier eligible.
`source_ami`	The source AMI that the temporary instance will be based on. We're using the official Amazon Linux image. The ID should be updated according to the AWS region you're using.

Packer will use the shell provisioner to install the JDK, as well as any tool that you may require to run your builds (Git or Docker, for example). You can take this script further and create a user called `jenkins` with a home directory to store Jenkins job workspaces, as shown in the following listing.

Listing 4.15 setup.sh script

```
#!/bin/bash
yum remove -y java
yum update -y
yum install -y git docker java-1.8.0-openjdk
usermod -aG docker ec2-user
systemctl enable docker
```

NOTE Docker is necessary, as we are going to define CI/CD pipelines for Dockerized microservices in upcoming chapters.

Issue the `packer build` command to bake the Jenkins worker AMI. Once the image-baking process is finished, the worker's AMI will be available on the EC2 dashboard, as shown in figure 4.18.

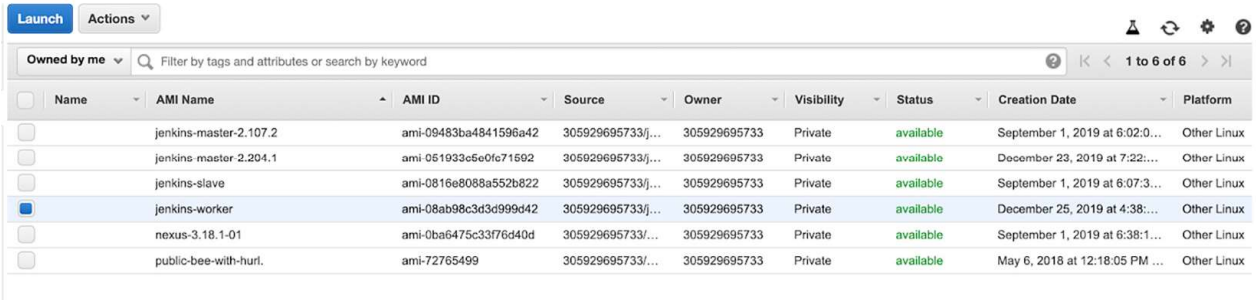

Figure 4.18 Jenkins worker AMI

NOTE After running the preceding examples, your AWS account now has an AMI associated with it. AMIs are stored in S3 by Amazon, so unless you want to be charged about $0.01 per month, you'll probably want to remove these images if they're not needed.

Now that our Jenkins cluster AMIs are ready to use, we will use them in the next chapter to deploy our cluster on AWS with the IaC tool HashiCorp Terraform. Figure 4.19 illustrates how Terraform will be integrated.

If you plan to embrace the immutable infrastructure approach for upgrading Jenkins or installing additional plugins, triggering the provisioning process with Packer can get challenging. That's why you should opt for automation and set up a pipeline with Jenkins to automate the baking workflow for AMI. A basic workflow will use

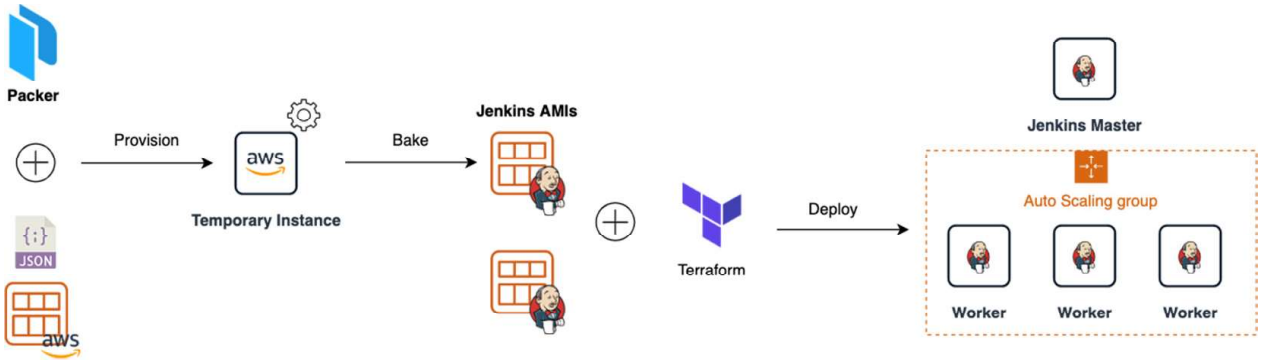

Figure 4.19 Packer will provision a temporary instance from a template file, and provision the instance with all needed configs and dependencies. From there, Terraform will deploy EC2 instances based on the baked image.

GitHub to store Packer template files and trigger a build on Jenkins upon the push event. The job will validate the template changes, start the baking process (1), and create an EC2 instance (2) based on the new baked AMI. Figure 4.20 summarizes the entire workflow.

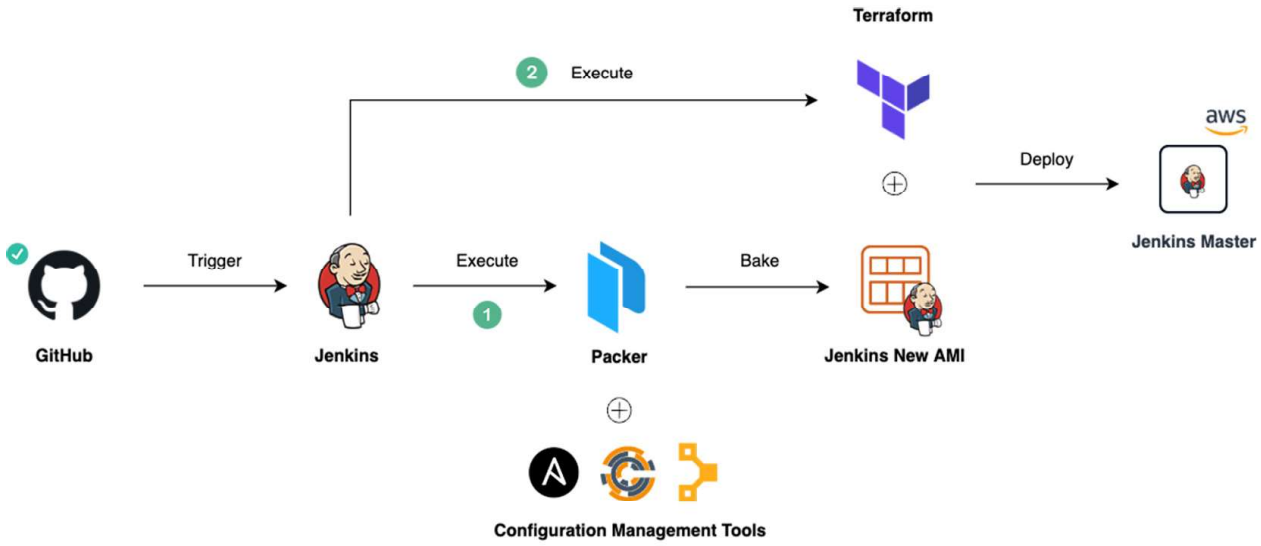

Figure 4.20 Automating the AMIs with Jenkins

NOTE Chapter 7 covers how to set up GitHub webhooks to continuously trigger Jenkins build jobs when a push or merge event occurs.

Summary

- HashiCorp Packer leverages the power of immutable infrastructure to bake custom machine images with all needed dependencies.
- Setting up Jenkins is a complex process, as both Jenkins and its plugins require tuning and configuration, with dozens of parameters to set within the web UI Manage Jenkins section.
- Configuration scripts in the init.groovy directory are executed in alphabetical order during Jenkins boot time. This is ideal for setting up seeding and configuration job interfaces.
- Jenkins provides thousands of plugins to support building, deploying, and automating any project.
- The weekly Jenkins releases deliver bug fixes and new features rapidly to users and plugin developers who need them. However, the Long-Term Support release is preferred for its stability.

5 Discovering Jenkins as code with Terraform

This chapter covers

- Introducing infrastructure as code (IaC)
- Using HashiCorp Terraform, which enables IaC
- Deploying Jenkins in a secure private network
- Scaling Jenkins workers dynamically with AWS Auto Scaling

In the previous chapter, we used HashiCorp Packer to create custom Jenkins machine images; in this chapter, we will use those images (figure 5.1) to deploy the machines. To do that, we will write declarative definitions of the Jenkins infrastructure we want to exist and use an automation tool to deploy the resources on the given infrastructure-as-a service (IaaS) provider.

In the past, managing IT infrastructure was a hard job. System administrators had to manually manage and configure all of the hardware and software that was needed for the applications to run. However, in recent years, things have changed dramatically. Trends like cloud computing revolutionized—and improved—the way organizations design, develop, and maintain their IT infrastructure. One of the critical components of this trend is called infrastructure as code.