cloud provider, I want to make this book useful for others and for those who skipped chapter 5 and jumped right here.

> **NOTE** Using the providers detailed in this chapter carries some benefits and drawbacks. No matter which provider you choose, you'll always encounter issues at some point along the way.

6.1 Google Cloud Platform

We all know that AWS doesn't have the most user-friendly web console. *Google Cloud Platform* (GCP) has managed to outperform AWS by offering a better user experience. GCP consists of a variety of services ranging from computing, to network, to extract-transform-load (ETL) pipelines that are 25% cheaper than its rival (AWS) because of lower-increment billing (10 minutes instead of 1 hour).

Plus, GCP has more expertise when it comes to big data, with services like BigQuery (https://cloud.google.com/bigquery), Cloud Bigtable (https://cloud.google.com/bigtable), and Dataflow (https://cloud.google.com/dataflow). In addition, you can run container workloads on Kubernetes and deploy machine learning (ML) models with TensorFlow; both Kubernetes and TensorFlow originated from Google. However, GCP still lacks features compared to AWS, which is the oldest and most mature cloud vendor on the market.

Why use Jenkins with GCP, then? You can have seamless integration with Kubernetes; with services like Google Kubernetes Engine (GKE), you can run ephemeral Jenkins workers, ensuring that each build runs on a clean environment. Native support for Docker containers is another reason, with services like Container Registry to store and manage Docker images built within CI/CD pipelines. In addition, you can have integrated security and compliance with detailed reports on vulnerability impacts and available fixes of build artifacts. Finally, you pay per usage when you use GCP virtual machines (VMs) to speed up your Jenkins builds.

With that being said, let's head over and deploy a Jenkins cluster with Terraform and Packer on GCP. To get started, sign up for a free account with a Gmail address (https://console.cloud.google.com/). You will automatically get a 12-month free trial with a $300 credit. You need to provide your credit card details, but you won't be charged extra until after your trial period ends or you have exhausted the $300 credit.

> **NOTE** The estimated cost to deploy a Jenkins cluster is $0.00. This cost assumes that you're within the GCP Free Tier limits and that you terminate all resources within 1 hour of deploying the infrastructure.

6.1.1 Building Jenkins VM images

For Packer to build a custom image, it needs to interact with GCP. Therefore, we need to create a dedicated service account for Packer to be authorized to access resources in Google APIs.

Head to the GCP console and navigate to the IAM & Admin dashboard, shown in figure 6.1. In the Service Accounts section, create a new service account with `Packer` as a name, and click the Create button.

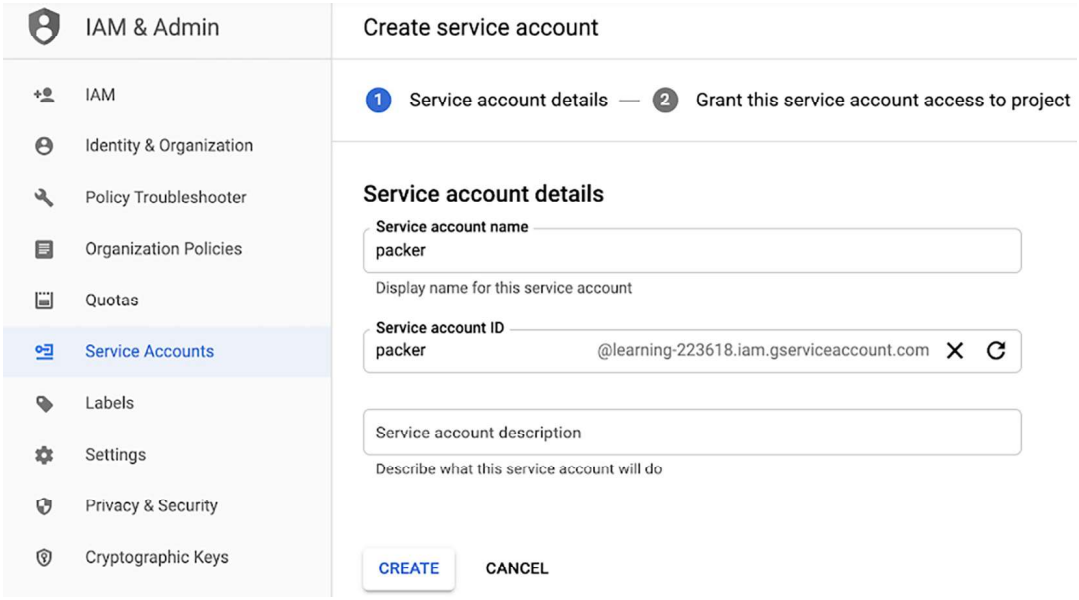

Figure 6.1 Creating a Packer service account

Assign the Project Owner role to the service account (or at least select Compute Engine Instance Admin and Service Account User roles) and click the Continue button, as shown in figure 6.2.

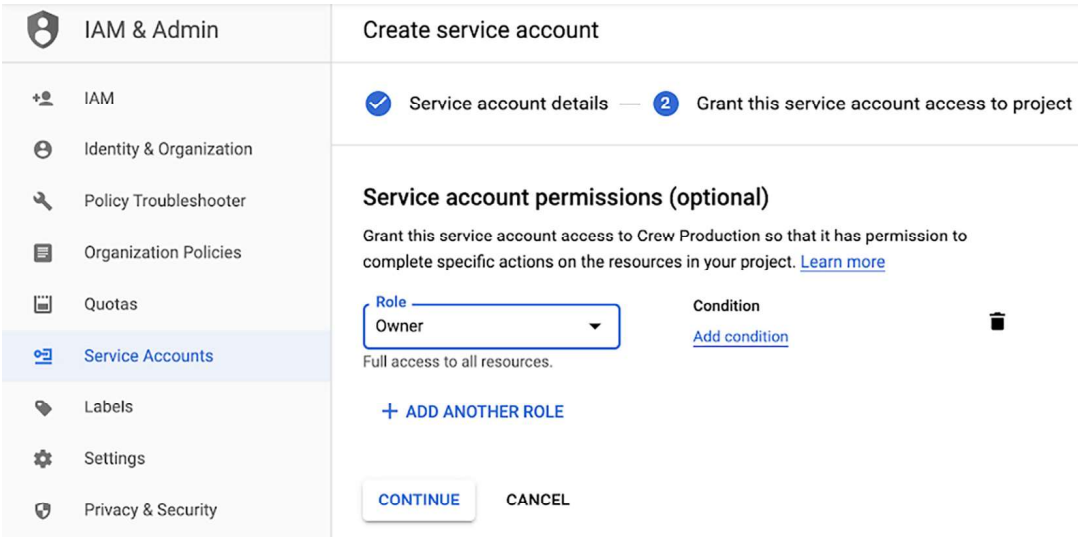

Figure 6.2 Setting Packer service account permissions

Each service account is associated with a key (JSON or P12 format), which is managed by GCP. This key is used for service-to-service authentication. Download the JSON key by clicking the Create Key button. The service account file is created and downloaded on the computer. Copy this JSON file and place it in a secure folder. Ensure that the Google Compute Engine API is enabled on your GCP project.

NOTE If you're unfamiliar with Packer, refer to chapter 4 for a step-by-step guide on installation and configuration.

Next update the Packer template file for the Jenkins worker provided in chapter 4's listing 4.16 with the following content, or copy and paste the content from the GitHub repository at chapter6/gcp/packer/worker/setup.sh.

Listing 6.1 Jenkins worker template file

```
{
    "variables" : {
        "service_account" : "SERVICE ACCOUNT JSON FILE PATH",
        "project": "GCP PROJECT ID",
        "zone": "GCP ZONE ID"
    },
    "builders" : [
        {
            "type": "googlecompute",
            "image_name" : "jenkins-worker",
            "account_file": "{{user `service_account`}}",
            "project_id": "{{user `project`}}",
            "source_image_family": "centos-8",
            "ssh_username": "packer",
            "zone": "{{user `zone`}}"
          }
    ],
    "provisioners" : [
        {
            "type" : "shell",
            "script" : "./setup.sh",
            "execute_command" : "sudo -E -S sh '{{ .Path }}'"
        }
    ]
}
```

Defines variables that will be provided at runtime. The values can be fetched from the GCP dashboard.

Runs the shell script in privileged mode to install the Git client, Docker, and needed dependencies

NOTE The JSON account file is not required if you're running the baking process from a Google Compute Engine (GCE) instance with a properly configured GCE service account. Packer will fetch the credentials from the metadata server.

Listing 6.1 uses the `googlecompute` builder to create a machine image on top of the CentOS base image. Then it uses the shell script provided in chapter 4's listing 4.13 to provision the temporary machine to install all needed dependencies—Git, JDK, and Docker.

The power of Packer comes from leveraging template files to create identical virtual machine images independently of the target platform. Therefore, we can use the same template file to build an identical Jenkins image for AWS, GCP, or Azure.

NOTE The scripted shell is explained in depth in chapter 4. All source code is available on the GitHub repository in the chapter6 folder.

The template file in listing 6.1 uses a set of variables such as the service account key file created earlier, the name of the zone where the builder machine will be provisioned, and the Google Cloud project ID that will own the image. The `service_account` variable can be implicit if you specify the path to the JSON file with the `GOOGLE_APPLICATION_CREDENTIALS` environment variable.

Packer will deploy a temporary instance from CentOS 8. A list of available images can be found on the Images dashboard, as you can see in figure 6.3.

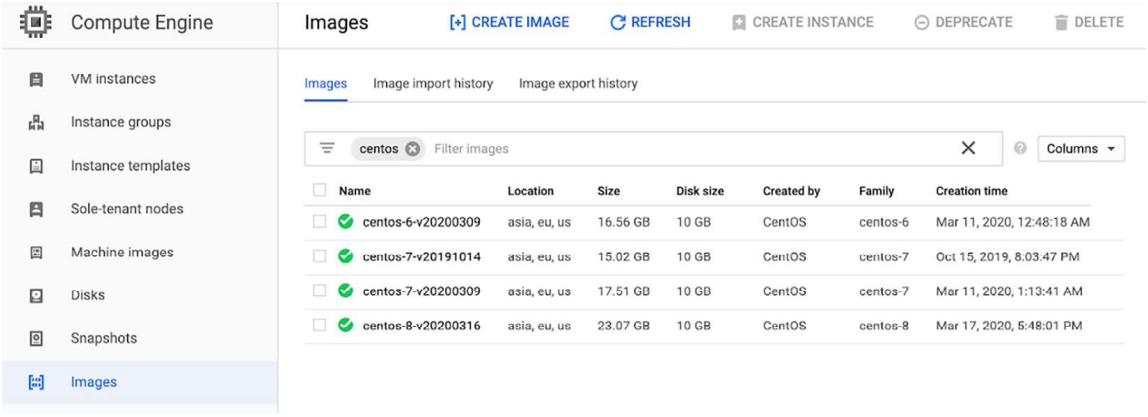

Figure 6.3 CentOS base image from GCE images

NOTE You can also use the `gcloud compute images list` command to list available images in a specific GCP location.

After supplying all the necessary variables, issue a `packer build` command. The output should be similar to the following output, which has been cropped for the sake of brevity:

```
googlecompute: output will be in this color.

==> googlecompute: Checking image does not exist...
==> googlecompute: Creating temporary SSH key for instance...
==> googlecompute: Using image: centos-8-v20200316
==> googlecompute: Creating instance...
    googlecompute: Loading zone: europe-west3-a
    googlecompute: Loading machine type: n1-standard-1
    googlecompute: Requesting instance creation...
    googlecompute: Waiting for creation operation to complete...
    googlecompute: Instance has been created!
==> googlecompute: Waiting for the instance to become running...
    googlecompute: IP: 34.89.251.218
==> googlecompute: Using ssh communicator to connect: 34.89.251.218
==> googlecompute: Waiting for SSH to become available...
```

Once the baking process is done, the Jenkins worker image should be available on the Google Compute Engine (GCE) console, as you can see in figure 6.4.

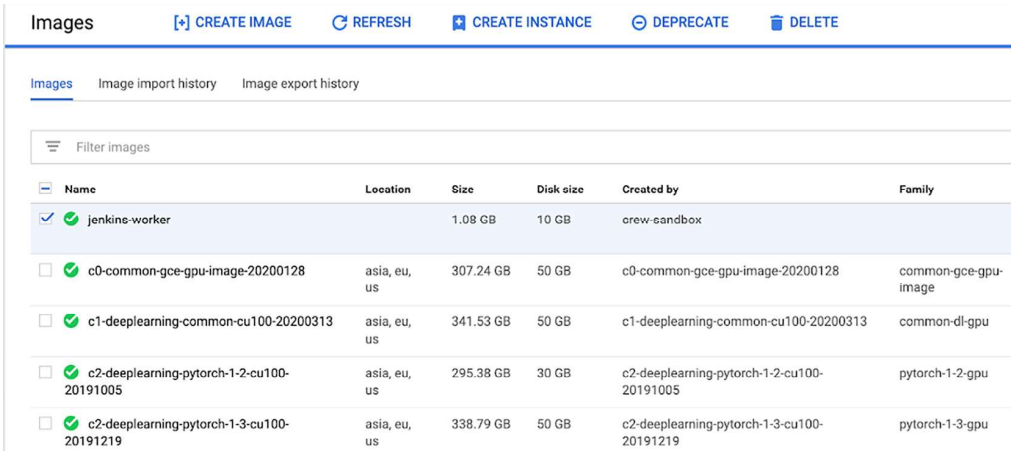

Figure 6.4 Jenkins worker custom image

Next, to build the Jenkins master machine image, we will use the same blueprint provided in chapter 4's listing 4.12. The only difference is the use of `googlecompute` in the `builders` section. The full template file, shown in the following listing, can be downloaded from chapter6/gcp/packer/master/setup.sh.

Listing 6.2 Jenkins master template file

```
{
    "variables" : {
        "service_account" : "SERVICE ACCOUNT JSON PATH",
        "project": "PROJECT ID",
        "zone": "ZONE ID",
        "ssh_key" : "PRIVATE SSH KEY PATH"
    },
    "builders" : [
        {
            "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`}}"
          }
    ],
    "provisioners" : [
       ...
    ]
}
```

NOTE This code listing already exists in the GitHub repository. You do not need to type it. It is shown for illustration purposes only.

Before we take this template and build an image from it, let's validate the template by running the following command:

```
packer validate template.json
```

With a properly validated template, it is time to build the Jenkins images. This is done by calling the `packer build` command with the template file as an argument. The output should look similar to the following. Note that this process typically takes a few minutes:

```
==> googlecompute: Checking image does not exist...
==> googlecompute: Creating temporary SSH key for instance...
==> googlecompute: Using image: centos-8-v20200316
==> googlecompute: Creating instance...
    googlecompute: Loading zone: europe-west3-a
    googlecompute: Loading machine type: n1-standard-1
    googlecompute: Requesting instance creation...
    googlecompute: Waiting for creation operation to complete...
    googlecompute: Instance has been created!
==> googlecompute: Waiting for the instance to become running...
    googlecompute: IP: 34.89.251.218
==> googlecompute: Using ssh communicator to connect: 34.89.251.218
==> googlecompute: Waiting for SSH to become available...
==> googlecompute: Connected to SSH!
==> googlecompute: Uploading ./scripts => /tmp/
==> googlecompute: Uploading ./config => /tmp/
==> googlecompute: Uploading /Users/mlabouardy/.ssh/id_rsa => /tmp/id_rsa
id_rsa 1.81 KiB / 1.81 KiB [==========================================================================
==> googlecompute: Provisioning with shell script: ./setup.sh
    googlecompute: Install Jenkins stable release
==> googlecompute: No packages marked for removal.
    googlecompute: No match for argument: java
    googlecompute: Dependencies resolved.
    googlecompute: Nothing to do.
    googlecompute: Complete!
    googlecompute: CentOS-8 - AppStream                      3.1 MB/s | 6.6 MB     00:02
    googlecompute: CentOS-8 - Base                           2.5 MB/s | 5.0 MB     00:02
    googlecompute: CentOS-8 - Extras                         5.3 kB/s | 4.8 kB     00:00
    googlecompute: CentOS-8 - PowerTools                     1.0 MB/s | 2.0 MB     00:01
    googlecompute: Google Compute Engine                     2.9 kB/s | 6.2 kB     00:02
    googlecompute: Google Cloud SDK                           15 MB/s |  33 MB     00:02
```

When Packer is done building the image, head over to the GCP console, The newly created image will be in the Images section, as shown in figure 6.5.

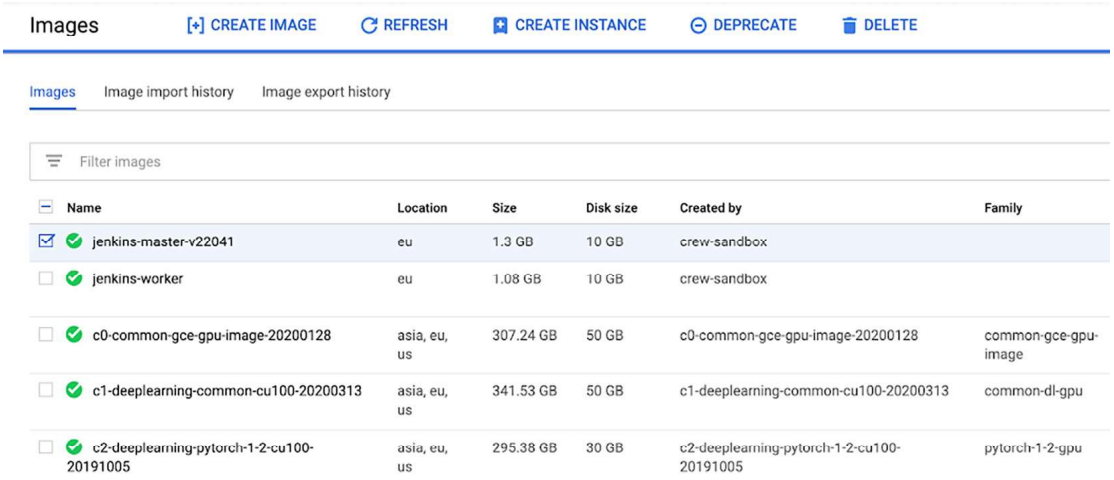

Figure 6.5 Jenkins master custom image

So far, you have learned how to automate the build process for the Jenkins machines images on GCP. In the next section, we will use Terraform to deploy VM instances based on those images. But first, we will deploy a private network on which our Jenkins cluster will be isolated.

6.1.2 Configuring a GCP network with Terraform

At the end of this section, you will have an isolated VPN running in different zones, as shown in figure 6.6.

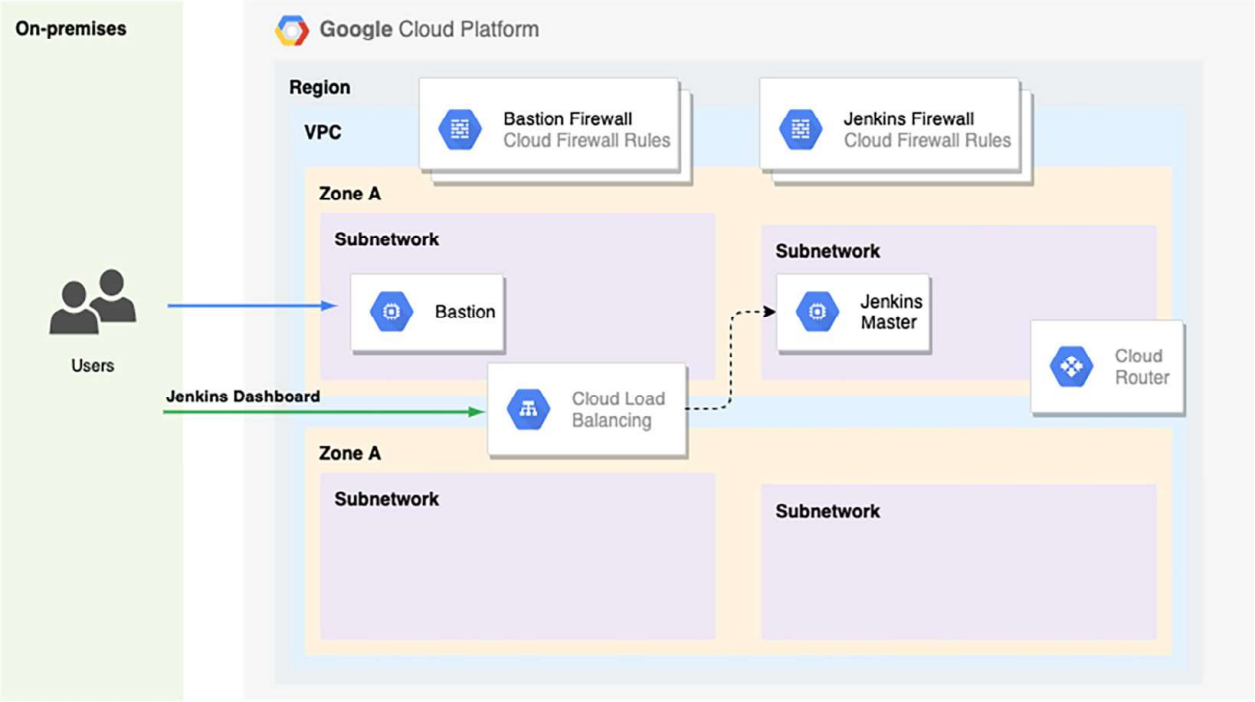

Figure 6.6 The Google VPN architecture consists of multiple subnetworks deployed in different zones. To access private instances, a bastion host can be used.

The VPC will be spun up in a single GCP region. It will be subdivided into subnets, each subnet contained within a single zone. Within a public subnet, a Google compute instance will be deployed with a role of a bastion host to give remote access to instances deployed in private subnets.

On the IAM console, shown in figure 6.7, create a dedicated service account for Terraform with Project Owner permission and download the JSON private key. This file contains credentials that will be needed for Terraform to manage the resources on your GCP project.

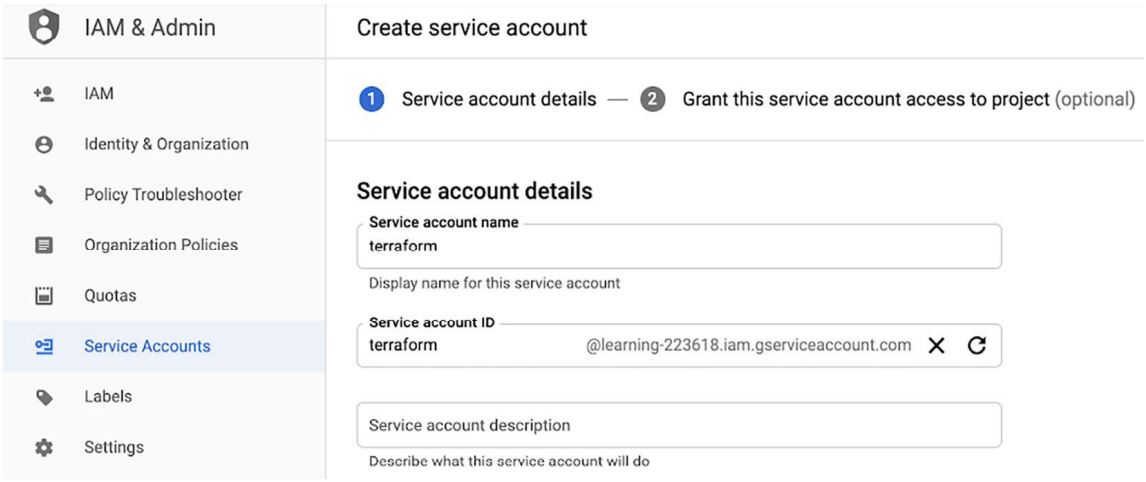

Figure 6.7 Terraform service account

Create a terraform.tf file, declare `google` as a provider, and configure it to use the service account created in the previous step; see the following listing.

Listing 6.3 Declaring Google as a provider

```
provider "google" {
  credentials = file(var.credentials_path)
  project     = var.project
  region      = var.region
}
```

Create a network.tf file and define a regional VPC network, as shown in the following listing. (If you plan to deploy Jenkins instances across multiple GCP regions, you need to change the routing mode to global.)

Listing 6.4 Defining a GCP network named management

```
resource "google_compute_network" "management" {
  name = var.network_name
  auto_create_subnetworks = false
  routing_mode = "REGIONAL"
}
```

Within the same file, declare two public and two private subnets, as shown in the next listing. Each subnet has its own CIDR block that is a subset of the network CIDR block (10.0.0.0/16).

Listing 6.5 Defining public and private subnetworks

```
resource "google_compute_subnetwork" "public_subnets" {
  count = var.public_subnets_count
```

```
  name          = "public-10-0-${count.index * 2 + 1}-0"
  ip_cidr_range = "10.0.${count.index * 2 + 1}.0/24"
  region        = var.region
  network       = google_compute_network.management.self_link
}

resource "google_compute_subnetwork" "private_subnets" {
  count = var.private_subnets_count
  name          = "private-10-0-${count.index * 2}-0"
  ip_cidr_range = "10.0.${count.index * 2}.0/24"
  region        = var.region
  network       = google_compute_network.management.self_link
  private_ip_google_access = true
}
```

Defines a unique CIDR range within the 10.0.0.0/16 block using the count.index variable

Before applying the changes with `terraform apply`, declare variables used to parameterize and customize the deployment in variables.tf. Table 6.1 lists the variables.

Table 6.1 GCP Terraform variables

Name	Type	Value	Description
`credentials_path`	String	None	The path to the service account key file in JSON format. This can be specified using the `GOOGLE_CREDENTIALS` environment variable.
`project`	String	None	The default project to manage resources in. If another project is specified on a resource, it will take precedence. This can also be specified using the `GOOGLE_PROJECT` environment variable.
`region`	String	None	The default region to manage resources in. If another region is specified on a regional resource, it will take precedence. Alternatively, this can be specified using the `GOOGLE_REGION` environment variable.
`network_name`	String	`management`	Name of the virtual network. The name must be 1–63 characters long and match the regular expression `[a-z]([-a-z0-9]*[a-z0-9])?`
`public_subnets_count`	Number	2	The number of public subnetworks. By default, we will create two public subnets in different zones for resiliency.
`private_subnets_count`	Number	2	The number of private subnetworks. By default, we will create two private subnets in different zones for resiliency.

We can now run Terraform to deploy the infrastructure. First, initialize Terraform to download the latest version of the Google Cloud provider plugin:

```
terraform init
```

The command output is given here:

```
Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "google" (hashicorp/google) 3.14.0...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.google: version = "~> 3.14"

Terraform has been successfully initialized!
```

Run a `plan` step to validate the configuration syntax and show a preview of what will be created:

```
terraform plan --var-file=variables.tfvars
```

> **NOTE** To set lots of variables, it is more convenient to specify their values in a variable definitions file (with a filename ending in either .tfvars or .tfvars .json) and then specify that file on the command line with the `-var-file` flag.

Now execute the `terraform apply` command to apply those changes:

```
terraform apply --var-file=variables.tfvars
```

You will see output similar to the following (cropped for brevity):

```
# google_compute_network.management will be created
+ resource "google_compute_network" "management" {
    + auto_create_subnetworks         = false
    + delete_default_routes_on_create = false
    + gateway_ipv4                    = (known after apply)
    + id                              = (known after apply)
    + ipv4_range                      = (known after apply)
    + name                            = "management"
    + project                         = (known after apply)
    + routing_mode                    = (known after apply)
    + self_link                       = (known after apply)
  }
```

It should take only a few moments to provision the private network. When it is finished, you should see something like figure 6.8.

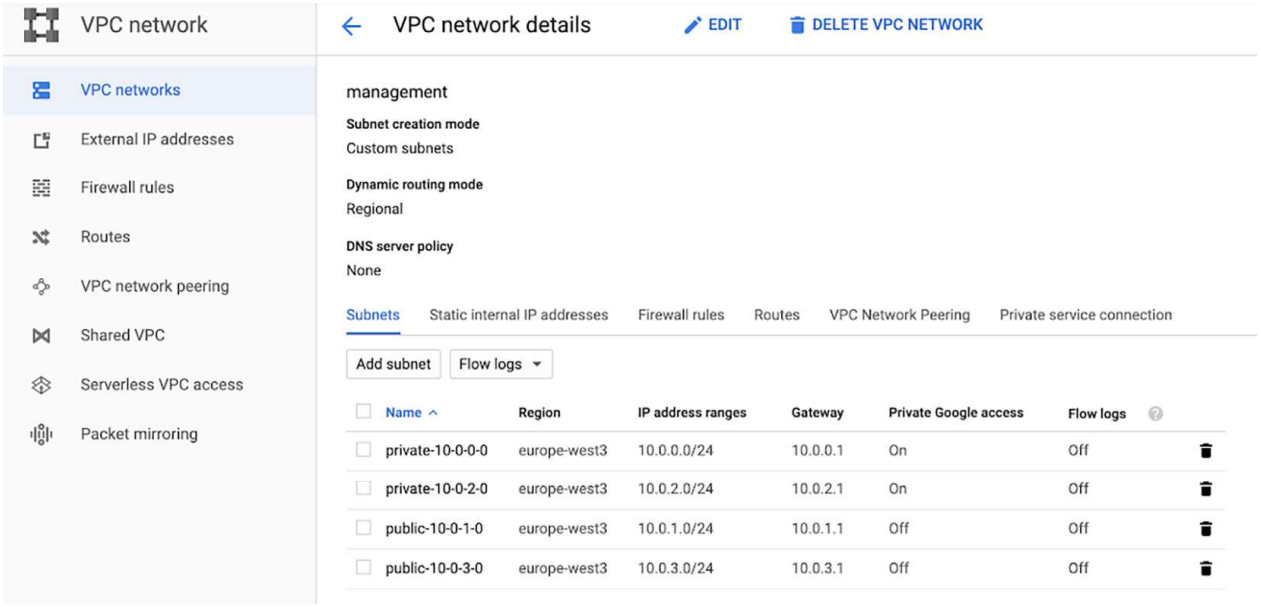

Figure 6.8 VPC network and its public and private subnets

To be able to SSH into private Jenkins instances, we will deploy a bastion host. Create bastion.tf and define a VM instance in a public subnet with a static IPv4 public IP address. To SSH into the bastion instance using Terminal (as opposed to the GCP console), you must generate and upload a public SSH key (located by default under ~/.ssh/id_rsa.pub, or generate a new one with `ssh-keygen`). The `metadata` attribute defined in the following listing references the public SSH key.

Listing 6.6 Bastion host resource

```
resource "google_compute_address" "static" {
  name = "ipv4-address"
}
resource "google_compute_instance" "bastion" {
  project      = var.project
  name         = "bastion"
  machine_type = var.bastion_machine_type
  zone         = var.zone
  tags = ["bastion"]
  boot_disk {
    initialize_params {
      image = var.machine_image
    }
  }
  network_interface {
    subnetwork = google_compute_subnetwork.public_subnets[0].self_link

    access_config {
      nat_ip = google_compute_address.static.address
    }
```

```
  }
  metadata = {
    ssh-keys = "${var.ssh_user}:${file(var.ssh_public_key)}"
  }
}
```

Within the same file, create a firewall rule to allow SSH from anywhere on the bastion host, as shown in the following listing. (It's recommended to enable ingress from only the IP address you wish to allow access from.)

Listing 6.7 Bastion host firewall rules

```
resource "google_compute_firewall" "allow_ssl_to_bastion" {
  project = var.project
  name    = "allow-ssl-to-bastion"
  network = google_compute_network.management.self_link

  allow {
    protocol = "tcp"
    ports    = ["22"]
  }

  source_ranges = ["0.0.0.0/0"]

  source_tags = ["bastion"]
}
```

Allows inbound traffic on port 22 (SSH) from anywhere

Finally, create an outputs.tf file and use the Terraform `output` variable to act as helper to expose the public IP address of the bastion virtual machine:

```
output "bastion" {
    value = "${google_compute_instance.bastion.network_interface
    .0.access_config.0.nat_ip }"
}
```

Outputs the bastion instance's public IP address

After the `terraform apply` command has finished, you should see output similar to this:

```
google_compute_address.static: Refreshing state... [id=projects/learning-223618/regions/europe-west3/addresses/ipv4-address]
google_compute_network.management: Refreshing state... [id=projects/learning-223618/global/networks/management]
google_compute_subnetwork.private_subnets[0]: Refreshing state... [id=projects/learning-223618/regions/europe-west3/subnetworks/private-10-0-0-0]
google_compute_subnetwork.private_subnets[1]: Refreshing state... [id=projects/learning-223618/regions/europe-west3/subnetworks/private-10-0-2-0]
google_compute_firewall.allow_ssl_to_bastion: Refreshing state... [id=projects/learning-223618/global/firewalls/allow-ssl-to-bastion]
google_compute_router.private_router: Refreshing state... [id=projects/learning-223618/regions/europe-west3/routers/private-router-management]
google_compute_subnetwork.public_subnets[0]: Refreshing state... [id=projects/learning-223618/regions/europe-west3/subnetworks/public-10-0-1-0]
google_compute_subnetwork.public_subnets[1]: Refreshing state... [id=projects/learning-223618/regions/europe-west3/subnetworks/public-10-0-3-0]
google_compute_router_nat.nat: Refreshing state... [id=learning-223618/europe-west3/private-router-management/nat-management]
google_compute_instance.bastion: Refreshing state... [id=projects/learning-223618/zones/europe-west3-a/instances/bastion]

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

bastion = 35.246.240.251
```

On the GCE console, a new VM instance should be deployed, as in figure 6.9.

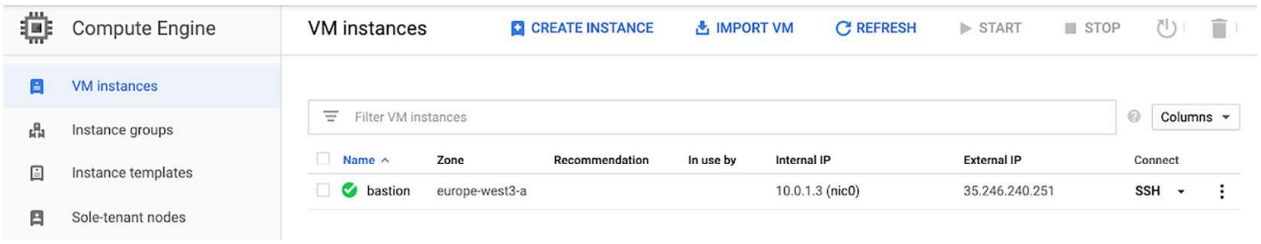

Figure 6.9 Bastion VM instance

With the jump box deployed, we can now access private instances in the VPC network.

6.1.3 Deploying Jenkins on Google Compute Engine

Now that the VPC is created, we will deploy a VM instance based on the Jenkins master image within a private subnet and expose a public load balancer to access the Jenkins web dashboard on port 8080, as described in figure 6.10.

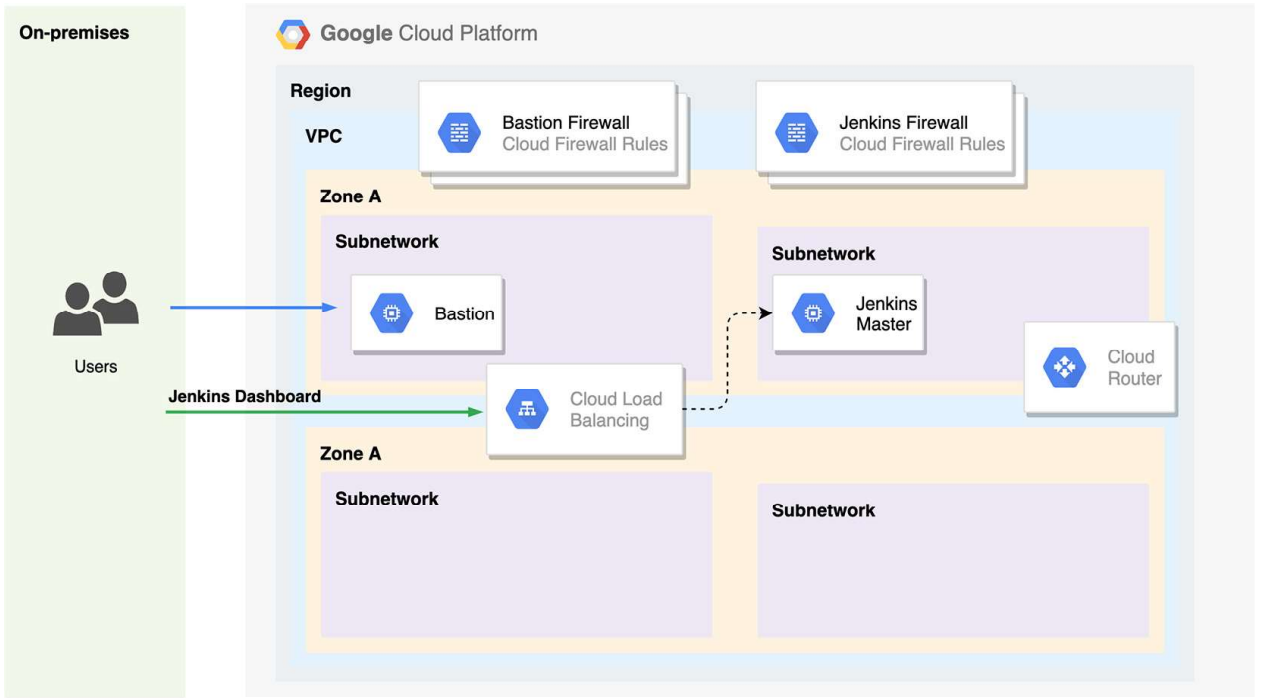

Figure 6.10 Jenkins master VM inside VPC

Create a jenkins_master.tf file and define a private compute instance with the attributes in the following listing.

Listing 6.8 Jenkins master compute instance

```
resource "google_compute_instance" "jenkins_master" {
  project      = var.project
  name         = "jenkins-master"
  machine_type = var.jenkins_master_machine_type
  zone         = var.zone

  tags = ["jenkins-ssh", "jenkins-web"]

  depends_on = [google_compute_instance.bastion]

  boot_disk {
    initialize_params {
      image = var.jenkins_master_machine_image
    }
  }

  network_interface {
    subnetwork = google_compute_subnetwork.private_subnets[0].self_link
  }

  metadata = {
    ssh-keys = "${var.ssh_user}:${file(var.ssh_public_key)}"
  }
}
```

Attaches jenkins-ssh and jenkins-web networks to the VM instance. The groups allow inbound traffic on port 22 and 8080 (Jenkins dashboard), respectively.

The compute instance uses the following firewall, which allows SSH from the bastion host only and inbound traffic on port 8080 from anywhere. (I recommend restricting the traffic to your network CIDR block.)

Listing 6.9 Jenkins master firewall and traffic control

```
resource "google_compute_firewall" "allow_ssh_to_jenkins" {
  project = var.project
  name    = "allow-ssh-to-jenkins"
  network = google_compute_network.management.self_link

  allow {
    protocol = "tcp"
    ports    = ["22"]
  }

  source_tags = ["bastion", "jenkins-ssh"]
}

resource "google_compute_firewall" "allow_access_to_ui" {
  project = var.project
  name    = "allow-access-to-jenkins-web"
  network = google_compute_network.management.self_link
```

Allows inbound traffic on port 22 (SSH)

```
  allow {
    protocol = "tcp"
    ports    = ["8080"]
  }

  source_ranges = ["0.0.0.0/0"]

  source_tags = ["jenkins-web"]
}
```

Allows inbound traffic on port 8080, where the Jenkins dashboard is exposed

Use `terraform apply` to deploy the Jenkins compute instance. Once the deployment is completed, a new VM will be deployed, as you can see in figure 6.11.

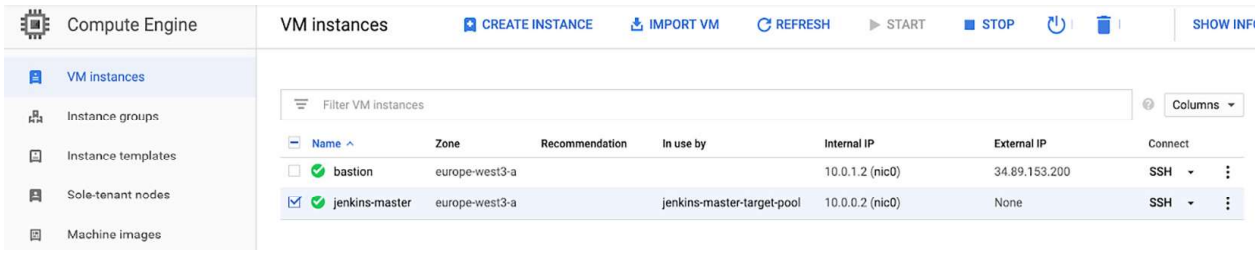

Figure 6.11 Jenkins master VM instance

The instance is deployed inside a private subnetwork. To be able to access the Jenkins web dashboard, we need to deploy a public load balancer in front of the VM instance.

Load balancing on GCP is different than on other cloud providers. The primary difference is that GCP uses forwarding rules instead of routing instances. These forwarding rules are combined with backend services, target pools, and health checks to construct a functional load balancer across an instance group.

First we define a target pool resource that defines the instances that should receive the incoming traffic, as shown in the next listing. In our case, the target pool will consist of the Jenkins master VM instance.

Listing 6.10 Jenkins master target pool

```
resource "google_compute_target_pool" "jenkins-master-target-pool" {
    name             = "jenkins-master-target-pool"
    session_affinity = "NONE"
    region = var.region

    instances = [
        Google_compute_instance.jenkins_master.self_link
    ]

    health_checks = [
        google_compute_http_health_check.jenkins_master_health_check.name
    ]
}
```

Defines Jenkins master VM instance as a target of the network load balancer

The cloud load balancer forwards traffic to the Jenkins master only if it's up and ready to receive the traffic. That's why we define a health-check resource to send health-check requests to the Jenkins master at a specific frequency on port 8080; see the following listing.

Listing 6.11 Jenkins master health check

```
resource "google_compute_http_health_check" "jenkins_master_health_check" {
  name          = "jenkins-master-health-check"
  request_path = "/"
  port = "8080"
  timeout_sec         = 4
  check_interval_sec = 5
}
```

Defines a template for how the Jenkins master should be checked for health, via HTTP

Finally, in the next listing, we define a forwarding rule to direct traffic to the target pool defined earlier.

Listing 6.12 Load balancer forwarding rule

```
resource "google_compute_forwarding_rule" "jenkins_master_forwarding_rule" {
  name    = "jenkins-master-forwarding-rule"
  region = var.region
  load_balancing_scheme = "EXTERNAL"
  target = google_compute_target_pool.jenkins-master-target-pool.self_link
  port_range             = "8080"
  ip_protocol            = "TCP"
}
```

If the incoming packet matches the given IP address, IP protocol, and port range tuple, it will be forwarded to the Jenkins master target pool.

Use `terraform apply` to deploy the public load balancer. On the Network Services dashboard, you should have the configuration shown in figure 6.12.

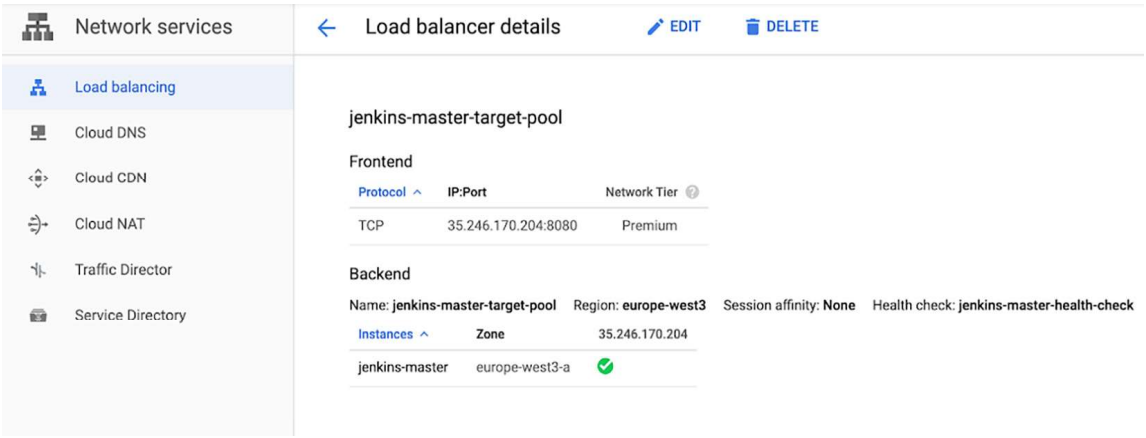

Figure 6.12 Public load balancer with Jenkins VM as a backend

As a backend, the load balancer uses Jenkins master instance and forwards incoming traffic on port 8080 to the backend on the same port. Also, it sets up an HTTP health check on port 8080.

To display the IP address of the load balancer, create an output section in the outputs.tf file:

```
output "jenkins" {
   value = google_compute_forwarding_rule \
.jenkins_master_forwarding_rule.ip_address
}
```

Issue the `terraform output` command on the console, and the Jenkins load balancer IP address should be displayed:

```
Apply complete! Resources: 1 added, 0 changed, 1 destroyed.

Outputs:

bastion = 34.89.153.200
jenkins = 35.246.170.204
```

You can now point your browser to the IP address on port 8080 and see the Jenkins welcome screen. If you see a screen like the one in figure 6.13, you've successfully deployed Jenkins on GCP!

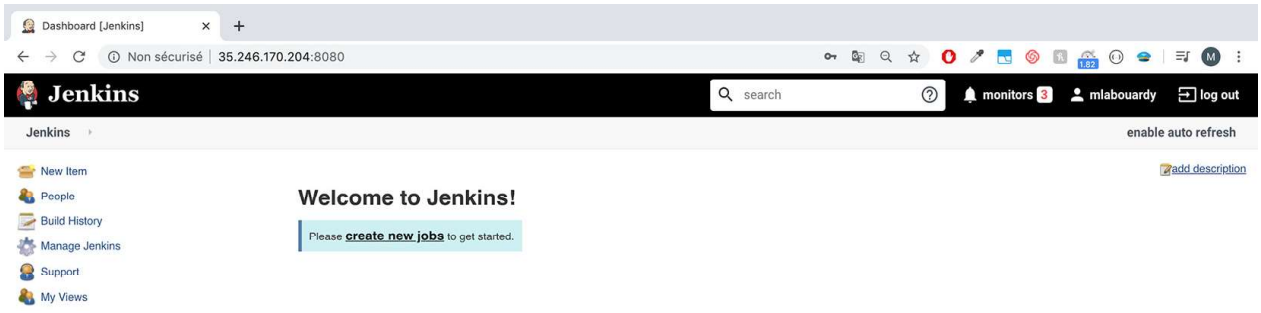

Figure 6.13 Public load balancer IP address to access the Jenkins dashboard

NOTE The forwarding rule may take several minutes to be provisioned. While it's being created, you might see 404 and 500 errors in the browser.

6.1.4 Launching automanaged workers on GCP

Arguably one of the most powerful features of Jenkins is its ability to dispatch build jobs across many workers. It is quite easy to set up a farm of build machines, either to share the load across multiple machines or to run build jobs in different environments. This is an effective strategy that can potentially increase the capacity of your CI infrastructure dramatically.

Demand for Jenkins workers can also fluctuate over time. If you are working with product release cycles, you may need to run a much higher number of workers toward the end of the cycle. Therefore, to avoid paying for extra resources while Jenkins workers are idle, we will deploy Jenkins workers inside an instance group and set up autoscaling policies to trigger scale-out or scale-in events that add or remove Jenkins workers, respectively, based on metrics such as CPU utilization.

NOTE In chapter 13, we will cover how to use an open source solution like Prometheus to export Jenkins custom metrics, including its integration with the scaling process of Jenkins workers.

Figure 6.14 summarizes the architecture we're going to deploy in this section.

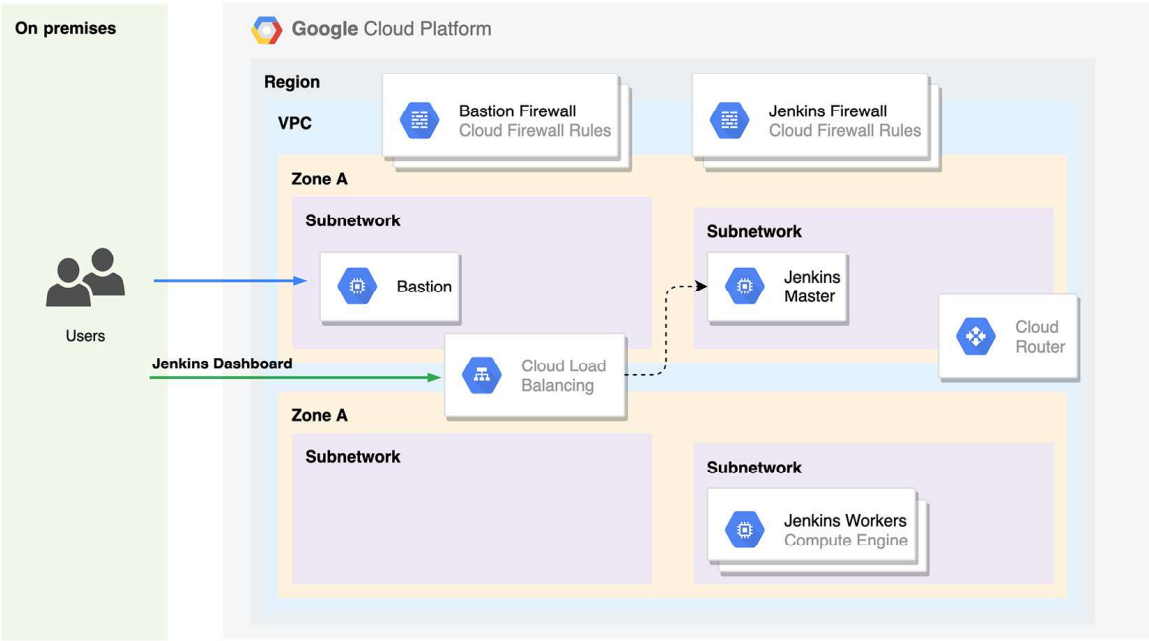

Figure 6.14 Jenkins cluster deployment on Google Cloud

First, create a jenkins_workers.tf file and define the instance template that will be used as a blueprint to define the Jenkins workers configurations; see the following listing.

Listing 6.13 Jenkins worker template configuration

```
resource "google_compute_instance_template" "jenkins-worker-template" {
  name_prefix = "jenkins-worker"
  description = "Jenkins workers instances template"
  region       = var.region

  tags = ["jenkins-worker"]
  machine_type         = var.jenkins_worker_machine_type
```

```
  metadata_startup_script =
     data.template_file.jenkins_worker_startup_script.rendered

  disk {
    source_image = var.jenkins_worker_machine_image
    disk_size_gb = 50
  }

  network_interface {
    network = google_compute_network.management.self_link
    subnetwork = google_compute_subnetwork.private_subnets[0].self_link
  }

  metadata = {
    ssh-keys = "${var.ssh_user}:${file(var.ssh_public_key)}"
  }
}
```

A shell script that will be executed the first time the VM instance is launched. The script will autojoin the instance as a Jenkins agent.

We will deploy the instances inside a private subnetwork and will execute the startup script in the following listing to make the running virtual machine join the cluster. This script is similar to the shell script provided in chapter 5's listing 5.7.

Listing 6.14 Jenkins worker startup script

```
data "template_file" "jenkins_worker_startup_script" {
  template = "${file("scripts/join-cluster.tpl")}"

  vars = {
    jenkins_url            = "http://${google_compute_forwarding_rule.
jenkins_master_forwarding_rule.ip_address}:8080"
    jenkins_username       = var.jenkins_username
    jenkins_password       = var.jenkins_password
    jenkins_credentials_id = var.jenkins_credentials_id
  }
}
```

The join-cluster.tpl template file takes as parameters the Jenkins credentials and URL. The values will be interpolated at runtime.

We will be using the Google Cloud metadata server to fetch the instance name and private IP address. The metadata server request's output is in JSON format, so we'll use the `jq` utility to parse the JSON and grab the target attributes:

```
INSTANCE_NAME=$(curl -s metadata.google.internal/0.1/meta-data/hostname)
INSTANCE_IP=$(curl -s metadata.google.internal/0.1/meta-data/network
| jq -r '.networkInterface[0].ip')
```

Next, we will define a firewall rule to allow SSH on Jenkins workers from the Jenkins master and bastion host, as shown in the following listing.

Listing 6.15 Jenkins master firewall and traffic control

```
resource "google_compute_firewall" "allow_ssh_to_worker" {
  project = var.project
  name    = "allow-ssh-to-worker"
  network = google_compute_network.management.self_link
```

```
  allow {
    protocol = "tcp"
    ports    = ["22"]
  }

  source_tags = ["bastion", "jenkins-ssh", "jenkins-worker"]
}
```

Allows inbound traffic on port 22 (SSH)

Then, we define an instance group based on the template file with a target size of two workers by default; see the next listing.

Listing 6.16 Jenkins worker instance group

```
resource "google_compute_instance_group_manager" "jenkins-workers-group" {
  provider = google-beta
  name = "jenkins-workers"
  base_instance_name = "jenkins-worker"
  zone               = var.zone

  version {
    instance_template  = google_compute_instance_template
.jenkins-worker-template.self_link
  }

  target_pools = [google_compute_target_pool
.jenkins-workers-pool.id]
  target_size = 2
}

resource "google_compute_target_pool" "jenkins-workers-pool" {
  provider = google-beta
  name = "jenkins-workers-pool"
}
```

Creates and manages pools of homogeneous VM instances (two instances) from a common instance template (jenkins-worker-template)

Once the new resources are deployed with `terraform apply`, two worker instances should be running, as shown in figure 6.15.

However, the number of workers is static and fixed, for now. To be able to scale Jenkins workers for heavy build jobs, we will deploy an autoscaler based on CPU

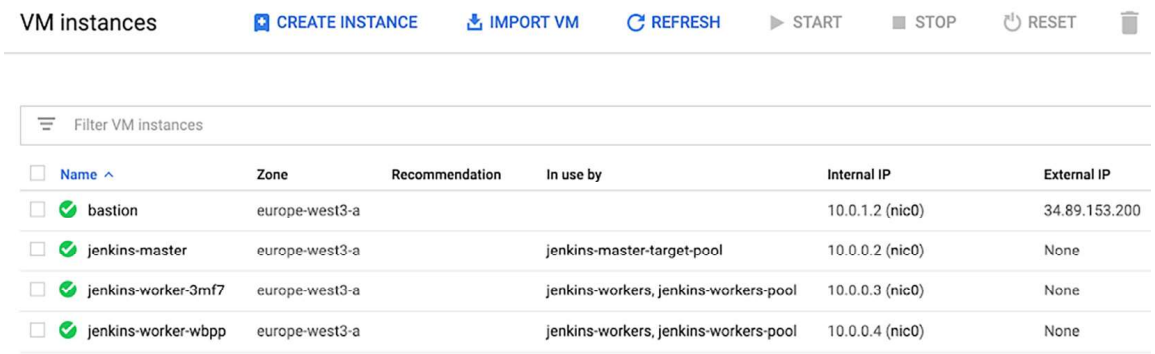

Figure 6.15 Jenkins worker instance groups

utilization. Define the following resource to trigger a scale-out event if the CPU utilization is over 80%. Within jenkins_workers.tf, add the code in the following listing.

Listing 6.17 Jenkins worker autoscaler

```
resource "google_compute_autoscaler" "jenkins-workers-autoscaler" {
  name   = "jenkins-workers-autoscaler"
  zone   = var.zone
  target = google_compute_instance_group_manager.jenkins-workers-group.id

  autoscaling_policy {
    max_replicas    = 6
    min_replicas    = 2
    cooldown_period = 60

    cpu_utilization {
      target = 0.8
    }
  }
}
```

Scales Jenkins worker instances in managed instance groups according to the autoscaling policy. The policy is based on the CPU utilization of the instances.

Once the changes are deployed with Terraform, the autoscaling policy will be configured on the Jenkins worker instance group, as you can see in figure 6.16.

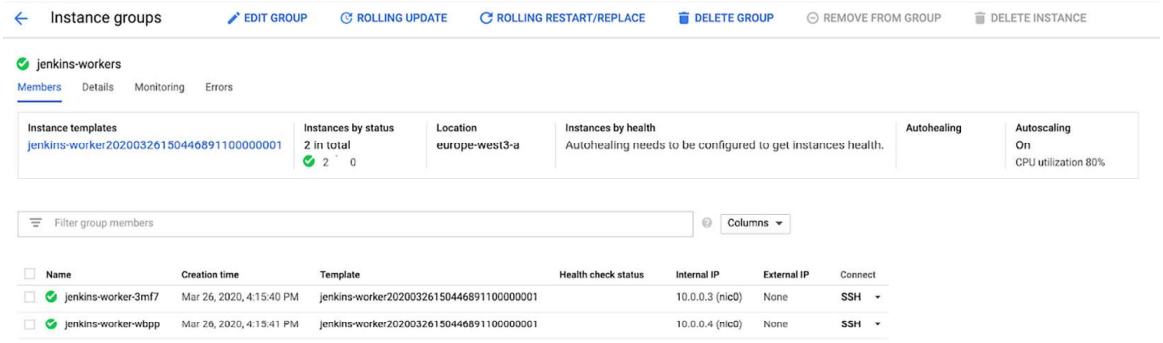

Figure 6.16 Instance group scaling based on CPU utilization

As a result, the workers will automatically join the cluster after the startup script is executed (figure 6.17). Awesome! You are running a Jenkins cluster on GCP.

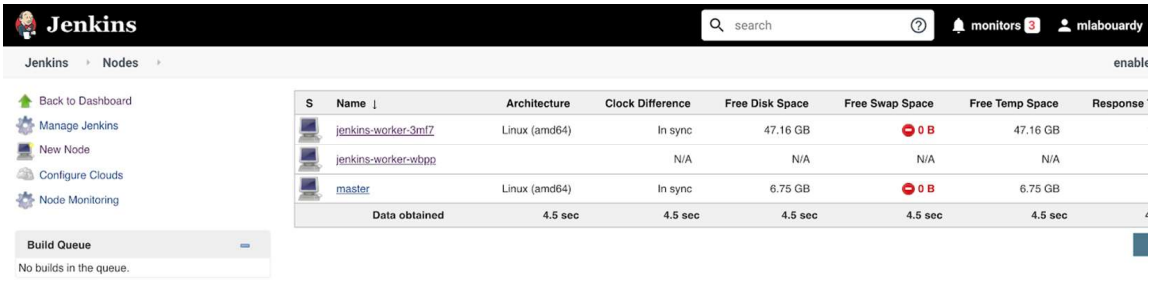

Figure 6.17 Jenkins worker VM instances joined the cluster.

6.2 Microsoft Azure

Both Microsoft Azure and AWS follow a similar approach by offering a variety of cloud-based services under one hood. However, organizations that use Microsoft software typically have an Enterprise Agreement that provides discounts on that software. These organizations can typically obtain significant incentives for using Azure.

If you plan to use Azure, you can deploy the Jenkins solution template from the Azure Marketplace. However, if you're looking to have full control over Jenkins, follow this section to learn how to build a Jenkins cluster from scratch and scale your Jenkins workers on demand based on Azure virtual machines.

> **NOTE** While Azure and Google Cloud have seen a fairly significant amount of growth, AWS is still the leader. This is mainly due to AWS being the first to invest in and shape the cloud computing industry. Google Cloud and Azure have some catching up to do.

Before getting started, if you're new to Azure, you may sign up for an Azure free account (https://portal.azure.com/) to start exploring with a free $200 credit.

6.2.1 Building golden Jenkins VM images in Azure

During the build process, Packer creates temporary Azure resources as it builds the source VM. Therefore, it needs to be authorized to interact with the Azure API.

Create an Azure service principal (SP) with permissions to create and manage resources with the following commands. An SP represents an application accessing your Azure resources. It is identified by a client ID (aka *application ID*) and can use a password or a certificate for authentication.

To create an SP, copy these commands:

```
$sp = New-AzADServicePrincipal -DisplayName "PackerServicePrincipal"
$BSTR = [System.Runtime
.InteropServices.Marshal]::SecureStringToBSTR($sp.Secret)
$plainPassword = [System.Runtime
.InteropServices.Marshal]::PtrToStringAuto($BSTR)
New-AzRoleAssignment -RoleDefinitionName
 Contributor -ServicePrincipalName $sp.ApplicationId
```

You can execute the commands on Azure PowerShell, as shown in figure 6.18.

```
PS /home/mohamed> Get-AzSubscription

Name          Id                                    TenantId                              State
----          --                                    --------                              -----
Pay-As-You-Go 50c09e38-b0cd-40d2-a5d3-02dcd8d85713 37741b5b-842b-4c1b-bfbe-fb97bb74cef8 Enabled

PS /home/mohamed> $sp = New-AzADServicePrincipal -DisplayName "PackerServicePrincipal"
WARNING: Assigning role 'Contributor' over scope '/subscriptions/50c09e38-b0cd-40d2-a5d3-02dcd8d85713' to the new service principal.
PS /home/mohamed> $BSTR = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($sp.Secret)
PS /home/mohamed> $plainPassword = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($BSTR)
PS /home/mohamed> New-AzRoleAssignment -RoleDefinitionName Contributor -ServicePrincipalName $sp.ApplicationId
New-AzRoleAssignment: The role assignment already exists.
```

Figure 6.18 Creating Azure credentials

Then output the password and application ID by executing the following commands:

```
$plainPassword
$sp.ApplicationId
```

Save the application ID and password for later.

To authenticate to Azure, you also need to obtain your Azure tenant and subscription IDs, which can be fetched with `Get-AzSubscription` or from Azure Active Directory (AD). AD, shown in figure 6.19, is an identity management service that controls access and security to Azure resources with the right roles and permissions.

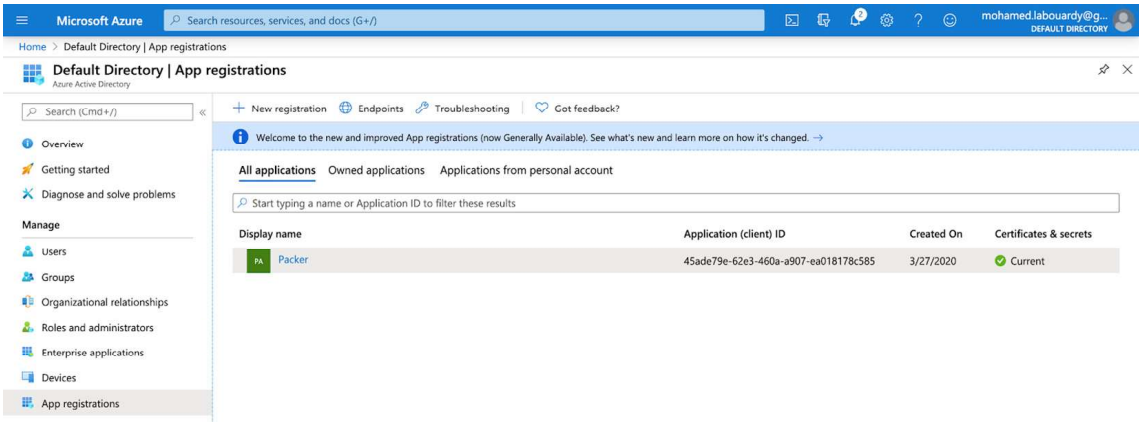

Figure 6.19 Packer registration on Azure Active Directory

Note the client ID and key. This will be used as credentials in Packer to provision resources in Azure.

To build the Jenkins worker image, create a template.json file. In the template, you define builders and provisioners that carry out the actual build process. Packer has a builder for Azure called `azure-arm` that allows you to define Azure images. Add the following content to template.json or download the full template from chapter6/azure/packer/worker/template.json.

Listing 6.18 Jenkins worker template with Azure builder

```
{
    "variables" : {
        "subscription_id" : "YOUR SUBSCRIPTION ID",
        "client_id": "YOUR CLIENT ID",
        "client_secret": "YOUR CLIENT SECRET",
        "tenant_id": "YOUR TENANT ID",
        "resource_group": "RESOURCE GROUP NAME",
        "location": "LOCATION NAME"
    },
    "builders" : [
        {
            "type": "azure-arm",
```

List of runtime variables to make the Packer template portable and reusable

```
            "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-worker",
            "os_type": "Linux",
            "image_publisher": "OpenLogic",
            "image_offer": "CentOS",
            "image_sku": "8.0",
            "location": "{{user `location`}}",
            "vm_size": "Standard_B1s"
        }
    ],
    "provisioners" : [
        {
            "type" : "shell",
            "script" : "./setup.sh",
            "execute_command" : "sudo -E -S sh '{{ .Path }}'"
        }
    ]
}
```

Packer will provision an instance of type Standard_B1s (1 RAM and 1vCPU) based on the CentOS 8.0 machine image.

If you're running Packer in a virtual machine, you can assign a managed identity to the virtual machine. No configuration properties are required to be set.

The template in listing 6.18 deploys a temporary instance based on CentOS 8.0 and provisions the instance with a shell script to install needed dependencies. The choice of CentOS is not arbitrary. Both Amazon Linux Image and CentOS have similarities, especially the support of the Yum package manager. To use the same scripts provided in previous chapters and keep consistent and identical Jenkins images, we'll use CentOS.

Bake the image with the `packer build` command. Here's an example of the output:

```
azure-arm: output will be in this color.

==> azure-arm: Running builder ...
==> azure-arm: Getting tokens using client secret
==> azure-arm: Getting tokens using client secret
    azure-arm: Creating Azure Resource Manager (ARM) client ...
==> azure-arm: Creating resource group ...
==> azure-arm:  -> ResourceGroupName : 'packer-Resource-Group-gr48cazkyb'
==> azure-arm:  -> Location          : 'centralus'
==> azure-arm:  -> Tags              :
==> azure-arm: Validating deployment template ...
==> azure-arm:  -> ResourceGroupName : 'packer-Resource-Group-gr48cazkyb'
==> azure-arm:  -> DeploymentName    : 'pkrdpgr48cazkyb'
==> azure-arm: Deploying deployment template ...
==> azure-arm:  -> ResourceGroupName : 'packer-Resource-Group-gr48cazkyb'
==> azure-arm:  -> DeploymentName    : 'pkrdpgr48cazkyb'
==> azure-arm: Getting the VM's IP address ...
==> azure-arm:  -> ResourceGroupName   : 'packer-Resource-Group-gr48cazkyb'
==> azure-arm:  -> PublicIPAddressName : 'pkripgr48cazkyb'
==> azure-arm:  -> NicName             : 'pkrnigr48cazkyb'
==> azure-arm:  -> Network Connection  : 'PublicEndpoint'
==> azure-arm:  -> IP Address          : '40.122.174.203'
==> azure-arm: Waiting for SSH to become available...
==> azure-arm: Connected to SSH!
==> azure-arm: Provisioning with shell script: ./setup.sh
    azure-arm: Install Java JDK 8
```

It takes a few minutes for Packer to build the VM, run the provisioners, and bake the Jenkins worker image. Once completed, the image is created in the resource group set in the `resource_group` variable, as shown in figure 6.20.

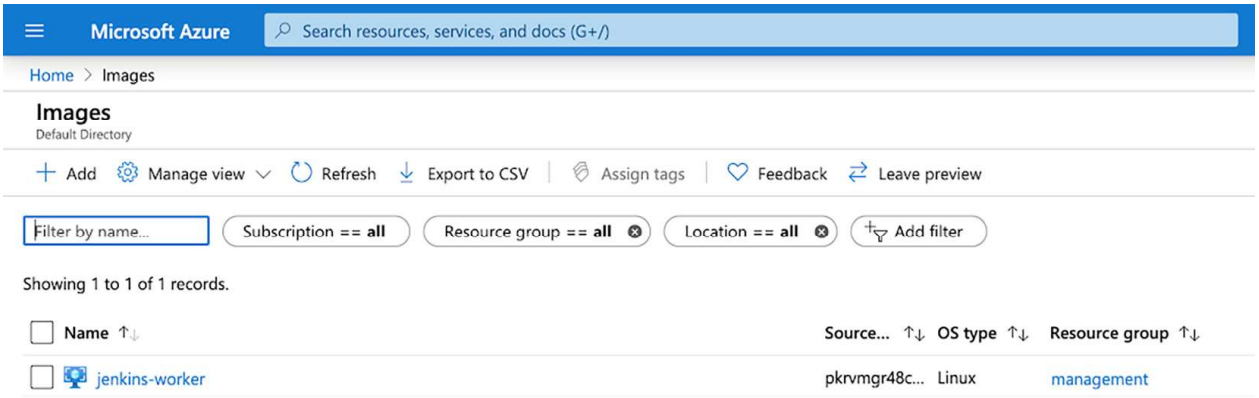

Figure 6.20 Jenkins worker machine image

A similar workflow will be applied to build the Jenkins master image. The following is the template.json file (the complete template is available at chapter6/azure/packer/master/template.json).

Listing 6.19 Jenkins worker template with Azure builder

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

List of variables has been omitted for brevity; the complete list is in listing 6.18.

Once the template is defined, bake the image with Packer. The baking process should take a few minutes to create the image. Once the image has been created, it should be available on the Images dashboard from the Azure portal, as shown in figure 6.21.

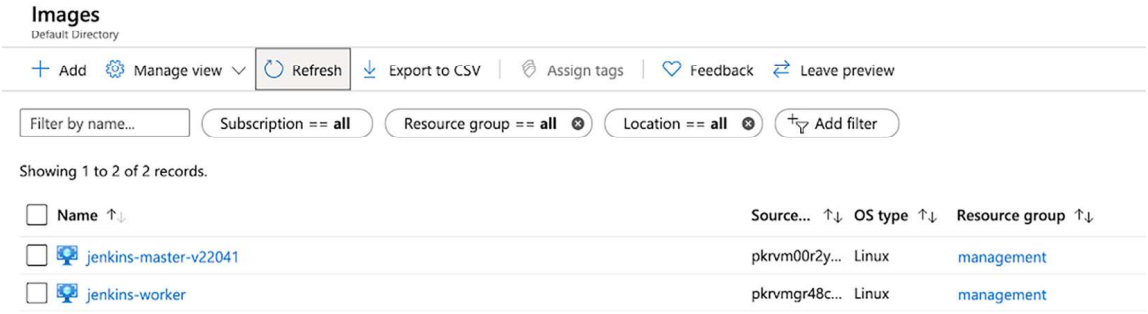

Figure 6.21 Jenkins master machine image

With both Jenkins master and worker images available, you can now create a Jenkins cluster from your custom images with Terraform.

6.2.2 Deploying a private virtual network

Before deploying the Jenkins cluster, we need to set up a private network with the architecture shown in figure 6.22 to secure access to the cluster.

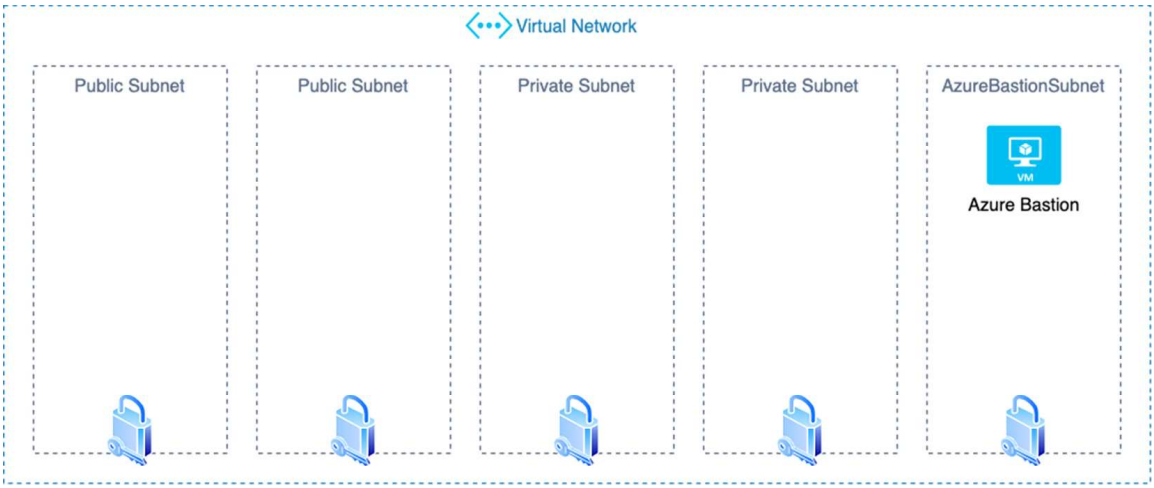

Figure 6.22 VPN on Azure

NOTE To enable Terraform to provision resources into Azure, create an Azure Active Directory service principal by following the same steps described in section 6.2.1.

Create a terraform.tf file and declare `azurerm` as a provider, as shown in the following listing. The `provider` section tells Terraform to use an Azure provider. To get values for `subscription_id`, `client_id`, `client_secret`, and `tenant_id`, see section 6.2.1.

Listing 6.20 Defining an Azure provider

```
provider "azurerm" {
  version = "=1.44.0"

  subscription_id = var.subscription_id
  client_id       = var.client_id
  client_secret   = var.client_secret
  tenant_id       = var.tenant_id
}
```

Run `terraform init` to download the latest version of the Azure plugin and build the .terraform directory:

```
Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "azurerm" (hashicorp/azurerm) 1.44.0...

Terraform has been successfully initialized!
```

Next, create a virtual_network.tf file on which you define a virtual network called `management` in the 10.0.0.0/16 address space with public and private subnets and an additional subnet called `AzureBastionSubnet` reserved for a bastion host, as shown in the following listing.

Listing 6.21 Azure virtual network definition

```
data "azurerm_resource_group" "management" {
  name = var.resource_group
}

resource "azurerm_virtual_network" "management" {
  name                = "management"
  location            = var.location
  resource_group_name = data.azurerm_resource_group.management.name
  address_space       = [var.base_cidr_block]
  dns_servers         = ["10.0.0.4", "10.0.0.5"]

  dynamic "subnet" {
    for_each = [for s in var.subnets: {
      name   = s.name
      prefix = cidrsubnet(var.base_cidr_block, 8, s.number)
    }]
```

List of IP addresses of DNS servers

Defines a list of subnets within the 10.0.0.0/16 space

```
    content {
      name           = subnet.value.name
      address_prefix = subnet.value.prefix
    }
  }

  subnet {
    name           = "AzureBastionSubnet"
    address_prefix = cidrsubnet(var.base_cidr_block, 11, 224)
  }

  tags = {
    environment = "management"
  }
}
```

Defines a list of subnets within the 10.0.0.0/16 space

Defines a dedicated subnet where the Bastion host will be deployed

NOTE We can tag our resources in Azure with a key-value pair. It's useful for cost optimization. So we will add the `environment` tag with value management to all the resources we create.

Before applying the changes, declare the variables used to parameterize and customize the Terraform deployment in variables.tf. Table 6.2 lists the variables.

Table 6.2 Azure Terraform variables

Name	Type	Value	Description
`subscription_id`	String	None	The subscription ID to be used. This can also be sourced from the `ARM_SUBSCRIPTION_ID` environment variable.
`client_id`	String	None	The client ID to be used. This can also be sourced from the `ARM_CLIENT_ID` environment variable.
`client_secret`	String	None	The client secret to be used. This can also be sourced from the `ARM_CLIENT_SECRET` environment variable.
`tenant_id`	String	None	The Tenant/Directory ID to be used. This can also be sourced from the `ARM_TENANT_ID` environment variable
`resource_group`	String	None	The name of the resource group in which to create the virtual network.
`location`	String	None	The location/region where the virtual network is created. Changing this forces a new resource to be created. Refer to Azure Locations documentation for a full list of supported locations.
`base_cidr_block`	String	`10.0.0.0/16`	The address space (CIDR block) that is used for the virtual network.
`subnets`	Map	None	A map holding a list of subnets to create inside the virtual network.

When authenticating as a service principal using a client certificate, the following fields should be set: `client_certificate_password` and `client_certificate_path`.

Now it's time to run the `terraform apply` command. Terraform will call Azure APIs to set up the new virtual network as shown here:

```
# azurerm_virtual_network.management will be created
+ resource "azurerm_virtual_network" "management" {
    + address_space       = [
        + "10.0.0.0/16",
      ]
    + dns_servers         = [
        + "10.0.0.4",
        + "10.0.0.5",
      ]
    + id                  = (known after apply)
    + location            = "centralus"
    + name                = "management"
    + resource_group_name = "management"
    + tags                = {
        + "environment" = "management"
      }

    + subnet {
        + address_prefix = "10.0.0.0/24"
        + id             = (known after apply)
        + name           = "public-10.0.0.0"
      }
    + subnet {
        + address_prefix = "10.0.1.0/24"
        + id             = (known after apply)
        + name           = "public-10.0.1.0"
      }
    + subnet {
        + address_prefix = "10.0.2.0/24"
        + id             = (known after apply)
        + name           = "private-10.0.2.0"
      }
    + subnet {
        + address_prefix = "10.0.28.0/27"
        + id             = (known after apply)
        + name           = "AzureBastionSubnet"
      }
    + subnet {
        + address_prefix = "10.0.3.0/24"
        + id             = (known after apply)
        + name           = "private-10.0.3.0"
      }
  }
```

To verify the results within the Azure portal, browse to the management resource group. The new virtual network is located under this group, as shown in figure 6.23.

To access private Jenkins machines, we need to deploy a gateway or proxy servers, also known as jump boxes or bastion hosts. Fortunately, Azure provides a managed service called Azure Bastion offering Remote Desktop Protocol (RDP) and SSH access to any VM without the need to manage a hardened bastion instance and apply security patches (no operational overhead).

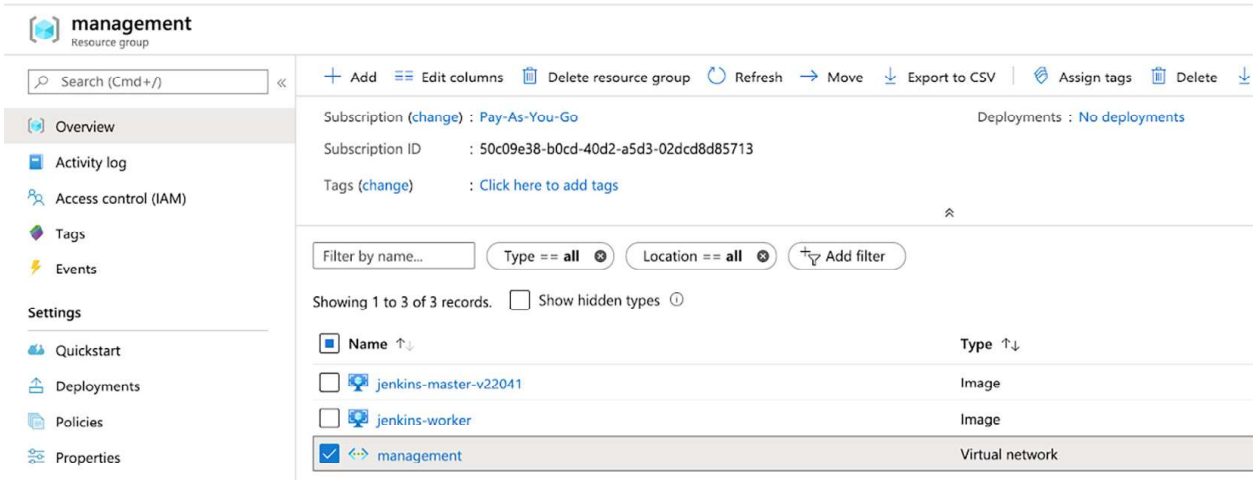

Figure 6.23 Management virtual network

To deploy the Azure Bastion service into the existing Azure virtual network, create a bastion.tf file with the following content. The bastion host service will be deployed into the dedicated `AzureBastionSubnet` subnet:

Listing 6.22 Azure Bastion service deployment

```
resource "azurerm_public_ip" "bastion_public_ip" {
  name                = "bastion-public-ip"
  location            = var.location
  resource_group_name = data.azurerm_resource_group.management.name
  allocation_method   = "Static"
  sku                 = "Standard"
}
data "azurerm_subnet" "bastion_subnet" {
  name                 = "AzureBastionSubnet"
  virtual_network_name = azurerm_virtual_network.management.name
  resource_group_name  = data.azurerm_resource_group.management.name
  depends_on = [azurerm_virtual_network.management]
}
resource "azurerm_bastion_host" "bastion" {
  name                = "bastion"
  location            = var.location
  resource_group_name = data.azurerm_resource_group.management.name
  depends_on = [azurerm_virtual_network.management]

  ip_configuration {
    name                 = "bastion-configuration"
    subnet_id            = data.azurerm_subnet.bastion_subnet.id
    public_ip_address_id = azurerm_public_ip.bastion_public_ip.id
  }
}
```

Requests a static public IP address

Reference to a subnet in which the bastion host will be created. It also associates the provisioned public IP address to the bastion host.

Use a Terraform output variable to act as a helper to expose the bastion IP address by referencing the `azurerm_public_ip` resource.

Listing 6.23 Bastion host public IP address

```
output "bastion" {
    value = azurerm_public_ip.bastion_public_ip.ip_address
}
```

Run `terraform apply` to apply the configuration. A bastion service will be deployed into the `management` resource group, as shown in figure 6.24.

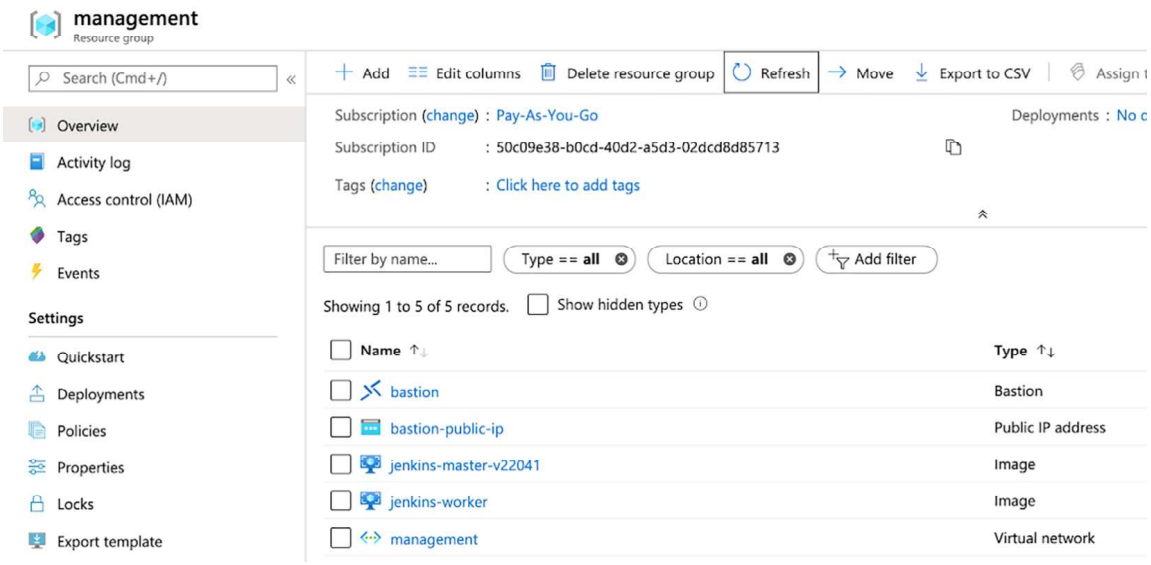

Figure 6.24 Azure bastion host

6.2.3 *Deploying a Jenkins master virtual machine*

With the VPN being deployed, we can deploy our Jenkins cluster. Figure 6.25 summarizes the target architecture.

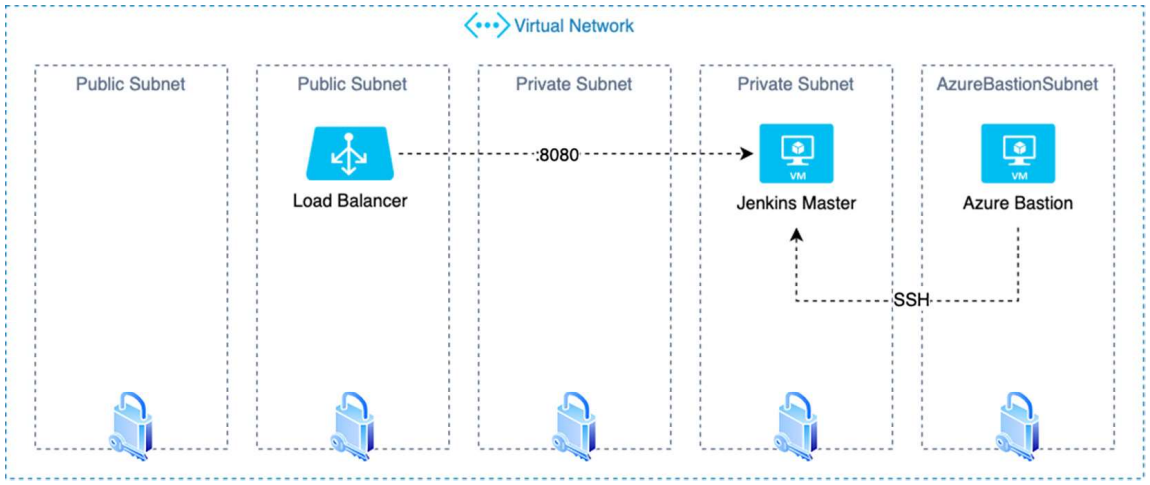

Figure 6.25 Jenkins VM inside a private subnet

Deploy a virtual machine based on the Jenkins master image built with Packer earlier. Define the resource in jenkins_master.tf with the following code.

Listing 6.24 Jenkins master virtual machine

```
data "azurerm_image" "jenkins_master_image" {
  name                = var.jenkins_master_image
  resource_group_name = data.azurerm_resource_group.management.name
}

resource "azurerm_virtual_machine" "jenkins_master" {
  name                = "jenkins-master"
  resource_group_name = data.azurerm_resource_group.management.name
  location            = var.location
  vm_size             = var.jenkins_vm_size

  network_interface_ids = [
    azurerm_network_interface.jenkins_network_interface.id,
  ]

  os_profile {
    computer_name  = var.config["os_name"]
    admin_username = var.config["vm_username"]
  }

  os_profile_linux_config {
    disable_password_authentication = true
    ssh_keys {
      path     = "/home/${var.config["vm_username"]}/.ssh/authorized_keys"
      key_data = file(var.public_ssh_key)
    }
  }

  storage_os_disk {
    name = "main"
    caching           = "ReadWrite"
    managed_disk_type = "Standard_LRS"
    create_option     = "FromImage"
    disk_size_gb      = "30"
  }

  storage_image_reference {
    id = data.azurerm_image.jenkins_master_image.i
  }

  delete_os_disk_on_termination = true
}
```

Disables password authentication and enables SSH as an authentication mechanism

Specifies the type of managed disk that should be created. Possible values are Standard_LRS, StandardSSD_LRS, or Premium_LRS.

Provisions the VM from the baked Jenkins master image

Deletes the OS disk automatically when deleting the VM

NOTE We allowed 30 GB as the disk size for the virtual machine. Jenkins needs some disk space to perform builds and keep archives and build logs.

SSH key data is provided in the `ssh_key` section, and the username is provided in the `os_profile` section with password authentication disabled.

The Jenkins virtual machine uses the B-Series Azure VM family with burstable CPU performances. This VM family provides the right balance between computing and

network bandwidth. I recommend selecting your VM family type based on your project build needs and requirements.

Listing 6.24 created a VM named `jenkins-master`, and now we'll attach the virtual network interface, as shown in the following listing.

Listing 6.25 Jenkins VM network configuration

```
data "azurerm_subnet" "private_subnet" {
  name                 = var.subnets[2].name
  virtual_network_name = azurerm_virtual_network.management.name
  resource_group_name  = data.azurerm_resource_group.management.name
  depends_on = [azurerm_virtual_network.management]
}

resource "azurerm_network_interface" "jenkins_network_interface" {
  name                = "jenkins_network_interface"
  location            = var.location
  resource_group_name = data.azurerm_resource_group.management.name
  depends_on = [azurerm_virtual_network.management]

  ip_configuration {
    name                          = "internal"
    subnet_id                     = data.azurerm_subnet.private_subnet.id
    private_ip_address_allocation = "Dynamic"
  }
}
```

Deploys the Jenkins master instance in a private subnet and assigns a dynamic private IP address

The virtual network interface connects the Jenkins master to the private network subnet.

Once you provide the needed Terraform variables in variables.tfvars, issue `terraform apply`. Creating the Jenkins VM, shown in figure 6.26, from your Packer image and the expected resources takes a few minutes.

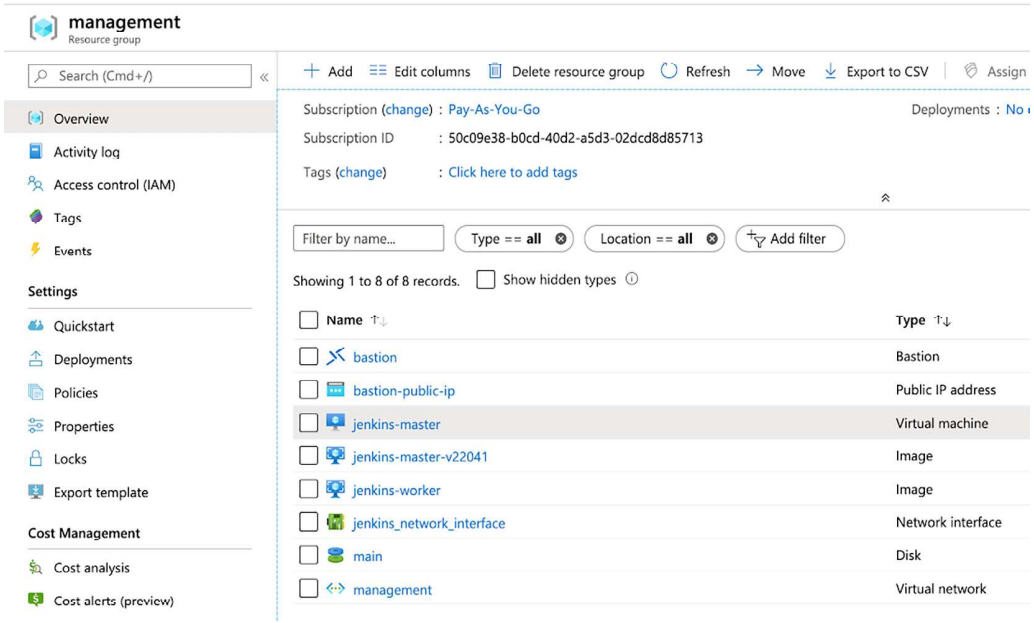

Figure 6.26 Jenkins master virtual machine

The Jenkins virtual machine should be accessible through a Bastion host only. Figure 6.27 confirms that the machine was deployed within a private subnet.

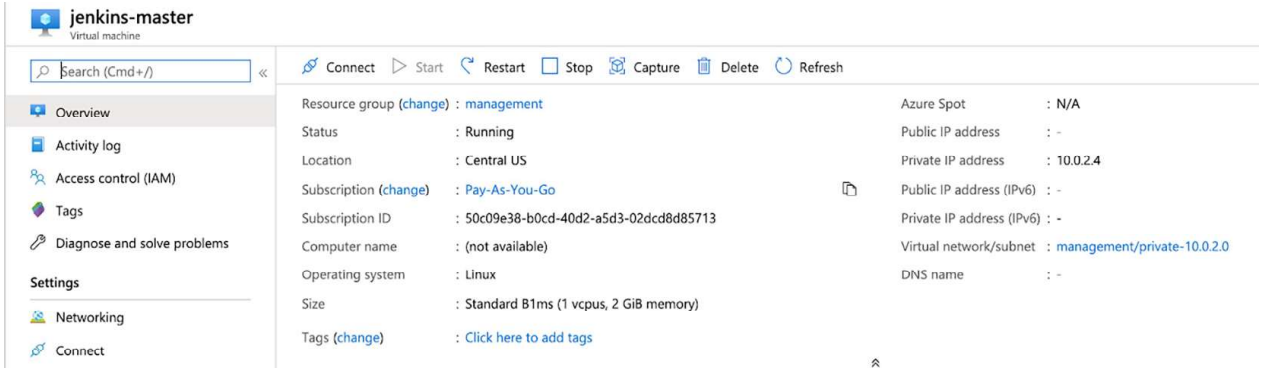

Figure 6.27 Jenkins master deployed in a private subnet

However, to access the Jenkins dashboard, we will deploy a load balancer in front of the VM. Create a loadbalancers.tf file on which you define an Azure load balancer and a security rule to serve the Jenkins dashboard and attach it to a public IP address, as shown in the following listing.

Listing 6.26 Jenkins dashboard load balancer configuration

```
resource "azurerm_public_ip" "jenkins_lb_public_ip" {
 name                         = "jenkins-lb-public-ip"
 location                     = var.location
 resource_group_name          = data.azurerm_resource_group.management.name
 allocation_method            = "Static"
}
resource "azurerm_lb" "jenkins_lb" {
 name                = "jenkins-lb"
 location            = var.location
 resource_group_name = data.azurerm_resource_group.management.name

 frontend_ip_configuration {
   name                 = "publicIPAddress"
   public_ip_address_id = azurerm_public_ip.jenkins_lb_public_ip.id
 }
}
resource "azurerm_lb_rule" "jenkins_lb_rule" {
  name = "jenkins-lb-rule"
  resource_group_name = data.azurerm_resource_group.management.name
  protocol = "tcp"
  enable_floating_ip = false
  probe_id = azurerm_lb_probe.jenkins_lb_probe.id
  loadbalancer_id = azurerm_lb.jenkins_lb.id
  backend_address_pool_id = azurerm_lb_backend_address_pool
```

Associates a public IP address to the load balancer

The load balancer listens on port 80 for incoming requests and communicates with the Jenkins master instance through port 8080.

```
.jenkins_backend.id
  frontend_ip_configuration_name = "publicIPAddress"
  frontend_port = 80
  backend_port = 8080
}
```

The load balancer listens on port 80 for incoming requests and communicates with the Jenkins master instance through port 8080.

Within the same file, define an Azure backend address pool and assign it to the load balancer. Then set a health check on port 8080, as shown in the following listing.

Listing 6.27 Jenkins dashboard health check

```
resource "azurerm_lb_backend_address_pool" "jenkins_backend" {
 resource_group_name = data.azurerm_resource_group.management.name
 loadbalancer_id     = azurerm_lb.jenkins_lb.id
 name                = "jenkins-backend"
}
resource "azurerm_lb_probe" "jenkins_lb_probe" {
  resource_group_name = data.azurerm_resource_group.management.name
  loadbalancer_id     = azurerm_lb.jenkins_lb.id
  name                = "jenkins-lb-probe"
  protocol            = "Http"
  request_path        = "/"
  port                = 8080
}
```

The URI used for requesting health status from the backend endpoint

Port on which the probe queries the backend endpoint

Azure allows for opening ports to traffic via security groups, which can also be managed in the Terraform configuration. Add the following to security_groups.tf and proceed to run `plan/apply` to create the security rule to allow inbound traffic on port 8080 and SSH traffic on TCP port 22.

Listing 6.28 Jenkins master security group

```
resource "azurerm_network_security_group" "jenkins_security_group" {
  name          = "jenkins-sg"
  location      = var.location
  resource_group_name   = data.azurerm_resource_group.management.name

  security_rule {
    name            = "AllowSSH"
    priority        = 100
    direction       = "Inbound"
    access          = "Allow"
    protocol        = "Tcp"
    source_port_range           = "*"
    destination_port_range      = "22"
    source_address_prefix       = "*"
    destination_address_prefix  = "*"
  }
```

Allows inbound traffic on port 22 (SSH) from anywhere

```
  security_rule {
    name                       = "AllowHTTP"
    priority                   = 200
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "8080"
    source_address_prefix      = "Internet"
    destination_address_prefix = "*"
  }
}
```

Allows inbound traffic on port 8080, where the Jenkins web dashboard is served

Finally, assign the security group to the virtual network interface attached to the Jenkins master virtual machine, as shown in the following listing.

Listing 6.29 Jenkins network interface configuration

```
resource "azurerm_network_interface" "jenkins_network_interface" {
  name                = "jenkins_network_interface"
  location            = var.location
  resource_group_name = data.azurerm_resource_group.management.name
  network_security_group_id =
     azurerm_network_security_group.jenkins_security_group.id
  depends_on = [azurerm_virtual_network.management]

  ip_configuration {
    name                          = "internal"
    subnet_id                     = data.azurerm_subnet.private_subnet.id
    private_ip_address_allocation = "Dynamic"
    load_balancer_backend_address_pools_ids =
    [azurerm_lb_backend_address_pool.jenkins_backend.id]
  }
}
```

Assigns the Jenkins security group to the virtual network interface configured in a private subnet

Apply the changes with the `terraform apply` command. Once Terraform completes, your load balancer is ready. Obtain its public IP address from outputs.tf by adding the following code.

Listing 6.30 Jenkins master firewall and traffic control

```
output "jenkins" {
    value = azurerm_public_ip.jenkins_lb_public_ip.ip_address
}
```

Let's verify the resources by using the Azure portal. As you can see in figure 6.28, Terraform created all the expected resources under the `management` resource group.

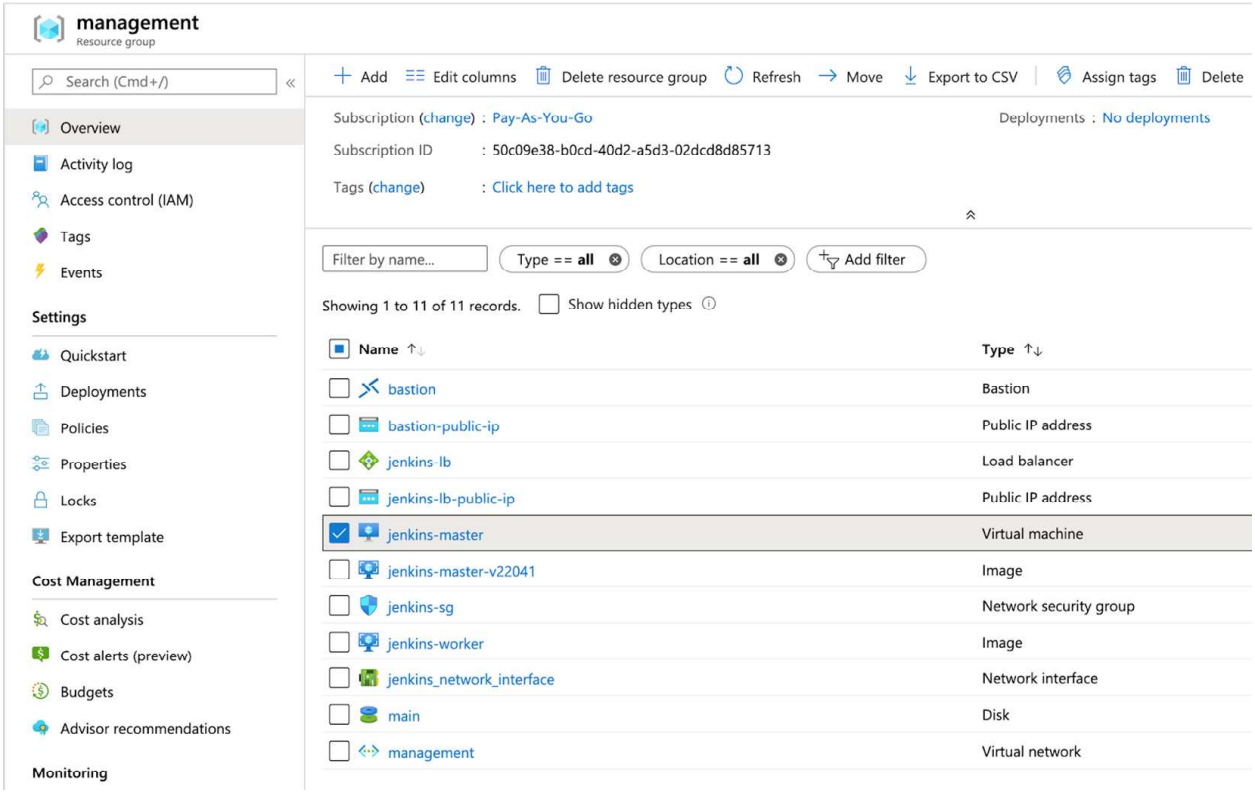

Figure 6.28 Public load balancer pointing to Jenkins master VM

Now point your web browser to the public IP address of the load balancer in the address bar. The default Jenkins home page will be displayed, as shown in figure 6.29.

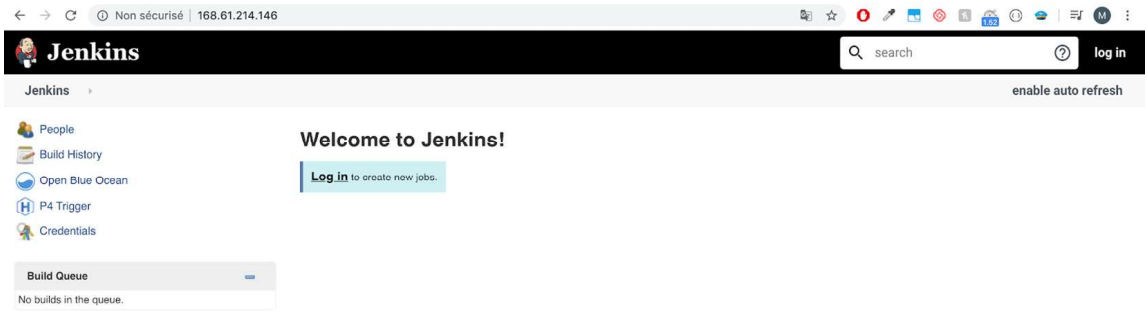

Figure 6.29 Jenkins dashboard accessible from LB public IP address

You can now sign in with admin credentials defined in the Groovy init scripts while baking the Jenkins master machine image.

6.2.4 Applying autoscaling to Jenkins workers

We're ready to deploy Jenkins workers to offload build projects from the master. The workers will be deployed inside an autoscaling set to be provisioned dynamically. Figure 6.30 illustrates the target deployment architecture.

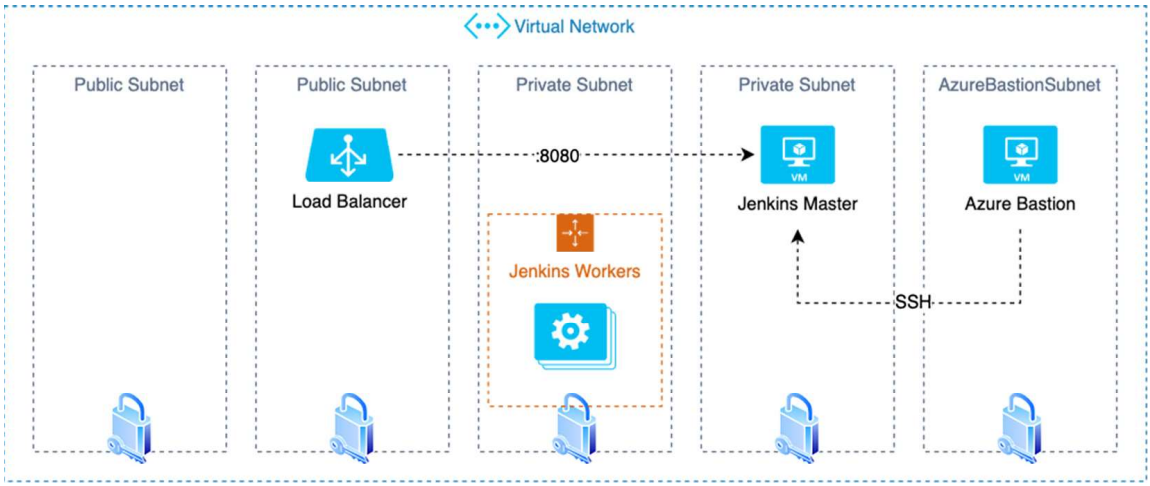

Figure 6.30 Jenkins workers scale set

We need to deploy Jenkins worker machines inside a machine scale set. A Jenkins worker will be based on the Jenkins worker image built earlier with Packer and will be deployed inside a private subnet. Create jenkins_workers.tf with the following content.

Listing 6.31 Jenkins worker machine scale set

References the Jenkins worker machine image ID

```
data "azurerm_image" "jenkins_worker_image" {
  name                = var.jenkins_worker_image
  resource_group_name = data.azurerm_resource_group.management.name
}
resource "azurerm_virtual_machine_scale_set" "jenkins_workers_set" {
  name                = "jenkins-workers-set"
  location            = var.location
  resource_group_name = data.azurerm_resource_group.management.name
  upgrade_policy_mode = "Manual"
  sku {
    name     = var.jenkins_vm_size
    tier     = "Standard"
    capacity = 2
  }
  storage_profile_image_reference {
    id = data.azurerm_image.jenkins_worker_image.id
  }
  storage_profile_os_disk {
    caching           = "ReadWrite"
    create_option     = "FromImage"
```

```
    managed_disk_type = "Standard_LRS"
  }
  os_profile {
    computer_name_prefix = "jenkins-worker"
    admin_username = var.config["vm_username"]
    custom_data = data.template_file.jenkins_worker_startup_script.rendered
  }
  os_profile_linux_config {
    disable_password_authentication = true
    ssh_keys {
      path     = "/home/${var.config["vm_username"]}/.ssh/authorized_keys"
      key_data = file(var.public_ssh_key)
    }
  }
  network_profile {
    name    = "private-network"
    primary = true
    network_security_group_id =
     azurerm_network_security_group.jenkins_worker_security_group.id
    ip_configuration {
      name = "private-ip-configuration"
      primary = true
      subnet_id = data.azurerm_subnet.private_subnet.id
    }
  }
}
```

Disables password authentication and configures the SSH credentials

Assigns a security group to the VM instances and requests private IP addresses

NOTE You should test your projects on multiple Azure VM family types to determine the appropriate machine type for Jenkins workers, as well as the amount of disk space.

Each Jenkins worker machine will execute a custom script (chapter6/azure/terraform/scripts/join-cluster.tpl) at runtime to join the Jenkins cluster; see the following listing.

Listing 6.32 Jenkins workers launch script

```
data "template_file" "jenkins_worker_startup_script" {
  template = "${file("scripts/join-cluster.tpl")}"

  vars = {
    jenkins_url            = "http://
     ${azurerm_public_ip.jenkins_lb_public_ip.ip_address}:8080"
    jenkins_username       = var.jenkins_username
    jenkins_password       = var.jenkins_password
    jenkins_credentials_id = var.jenkins_credentials_id
  }
}
```

Initialization script to autojoin the VM as a Jenkins agent

The script will use Azure Instance Metadata Service (IMDS) to fetch information regarding the machine's private IP address and hostname and will issue a POST HTTP request to the Jenkins RESTful API to establish a bidirectional connection with the machine and join the cluster:

```
INSTANCE_NAME=$(curl -s http://169.254.169.254/metadata/instance/compute/name
?api-version=2019-06-01&format=text)
INSTANCE_IP=$(curl -s http://169.254.169.254/metadata/instance/network/
     interface/0/ipv4/ipAddress/0/privateIpAddress
?api-version=2017-08-01&format=text)
```

A security group will be attached to the virtual network interface attached to the scale set. It allows inbound traffic on port 22 (SSH), as shown in the following listing.

Listing 6.33 Jenkins worker security group

```
resource "azurerm_network_security_group" "jenkins_worker_security_group" {
  name      = "jenkins-worker-sg"
  location    = var.location
  resource_group_name    = data.azurerm_resource_group.management.name
  security_rule {
     name       = "AllowSSH"
     priority   = 100
     direction  = "Inbound"
     access     = "Allow"
     protocol   = "Tcp"
     source_port_range             = "*"
     destination_port_range       = "22"
     source_address_prefix        = "*"
     destination_address_prefix   = "*"
  }
}
```

Allows incoming traffic on port 22 (SSH) from anywhere. It's recommended to restrict the access to your network CIDR block.

Once the deployment has completed, the content of the resource group resembles that shown in figure 6.31.

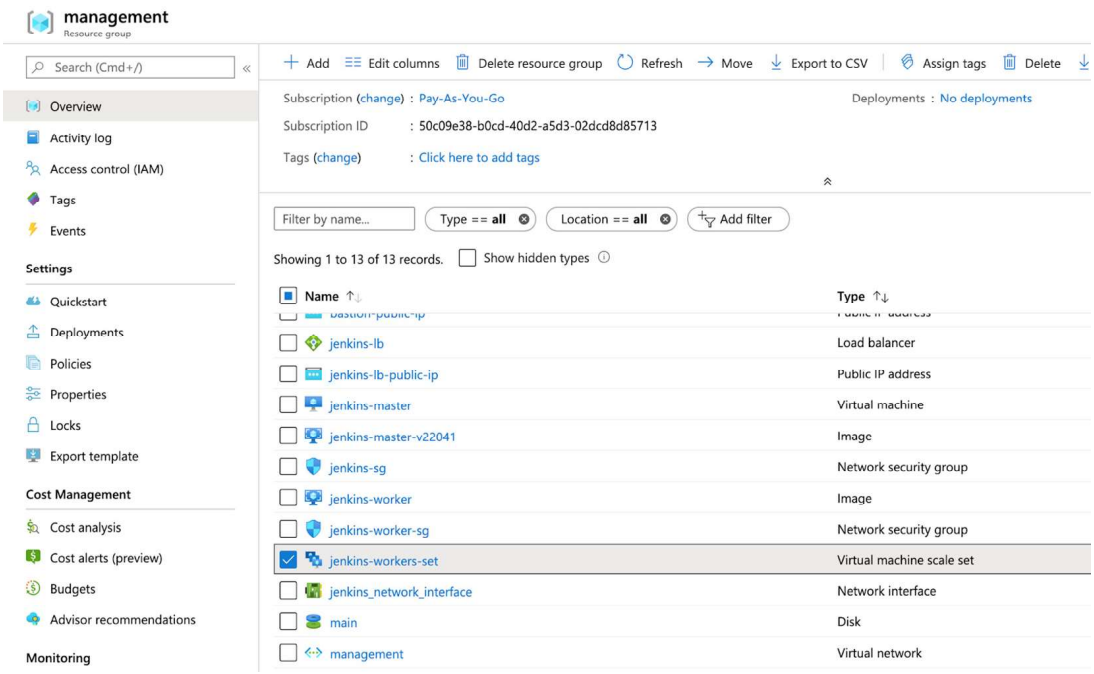

Figure 6.31 Jenkins worker virtual machine scale set