Now, go to GitHub Actions and find the latest build. You will see that the build will error out and give the following output:

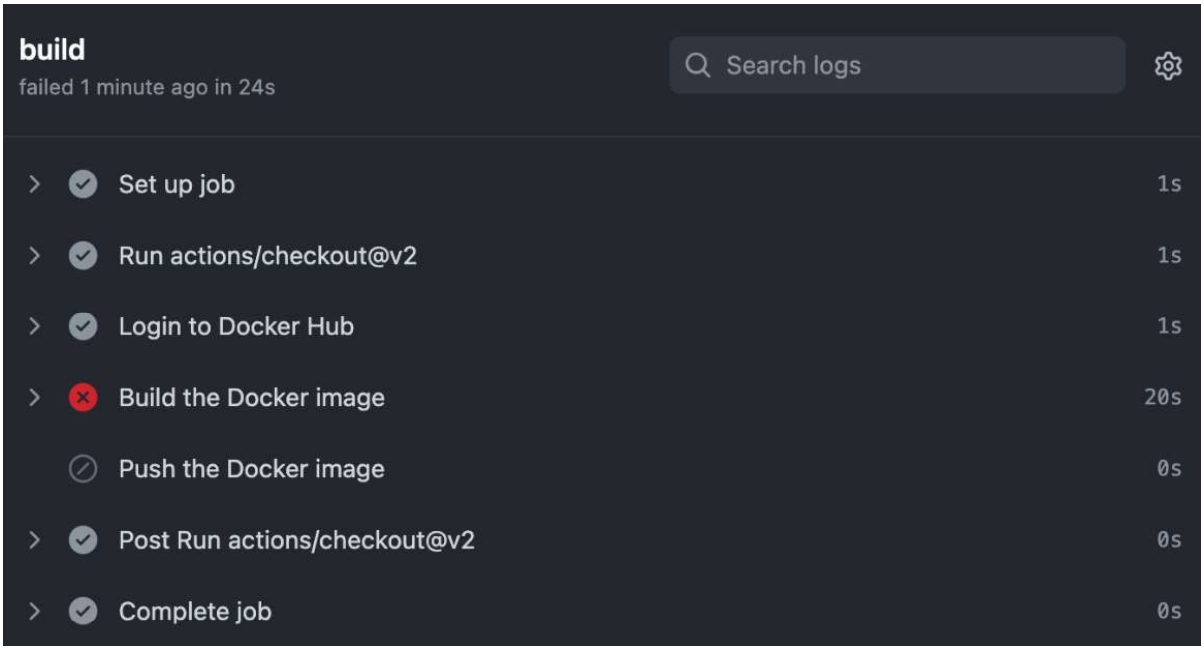

Figure 11.4 – GitHub Actions – build failure

As we can see, the **Build the Docker image** step has failed. If you click on the step and scroll down to see what happened with it, you will find that the `app.test.py` execution failed. This is because of a test case failure with `AssertionError: 'post' not found in {'pos': '60458fb603c395f9a81c9f4a'}`. As the expected `post` key was not found in the output, `{'pos': '60458fb603c395f9a81c9f4a'}`, the test case failed, as shown in the following screenshot:

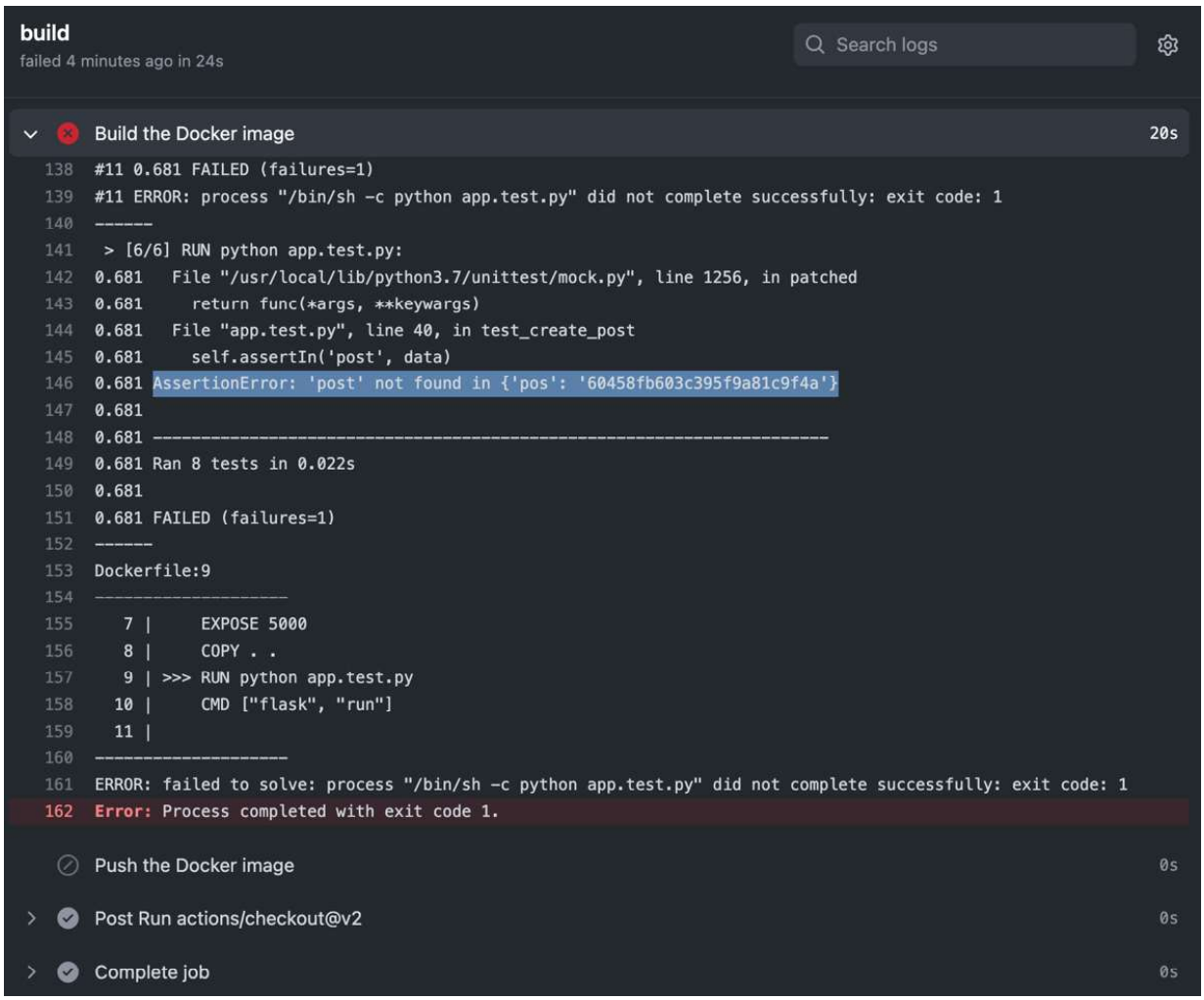

Figure 11.5 – GitHub Actions – test failure

We uncovered the error when someone pushed the buggy code to the Git repository. Are you able to see the benefits of CI already?

Now, let's fix the code and commit the code again.

Modify the `create_post` function of `app.py` so that it looks as follows:

```
@app.route('/posts', methods=['POST'])
def create_post():
    ...
    return jsonify({'post': str(inserted_post.inserted_id)}), 201
```

Then, `commit` and `push` the code to GitHub using the following commands:

```
$ git add --all
$ git commit -m 'Updated create_post'
$ git push
```

This time, the build will be successful:

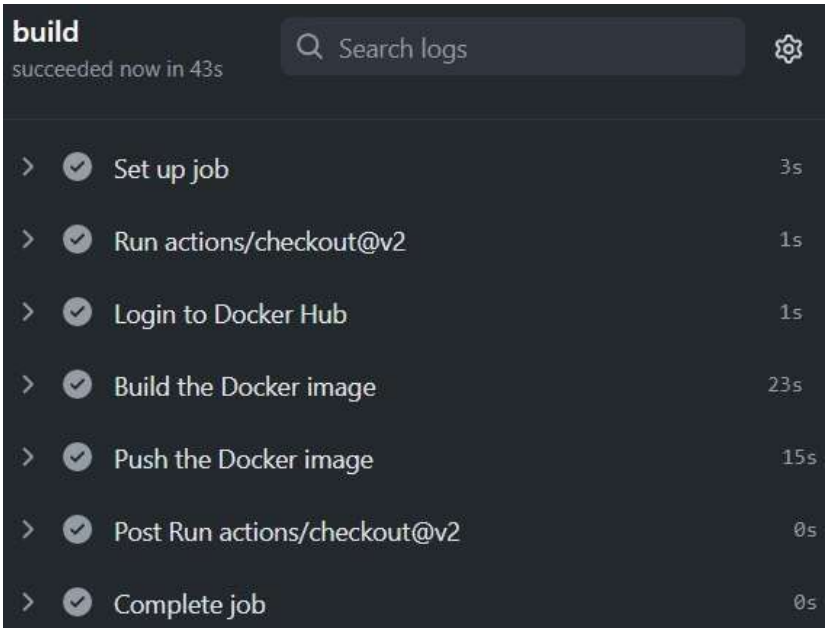

Figure 11.6 – GitHub Actions – build success

Did you see how simple this was? We got started with CI quickly and implemented GitOps behind the scenes since the config file required to build and test the code also resided with the application code.

As an exercise, repeat the same process for the **reviews**, **users**, **ratings**, and **frontend** microservices. You can play around with them to understand how it works.

Not everyone uses GitHub, so the SaaS offering might not be an option for them. Therefore, in the next section, we'll look at the most popular open source CI tool: Jenkins.

Scalable Jenkins on Kubernetes with Kaniko

Imagine you're running a workshop where you build all sorts of machines. In this workshop, you have a magical conveyor belt called Jenkins for assembling these machines. But to make your workshop even more efficient and adaptable, you've got a team of tiny robot workers called Kaniko that assist in constructing the individual parts of each machine. Let's draw parallels between this workshop analogy and the technology world:

- **Scalable Jenkins**: Jenkins is a widely used automation server that helps automate various tasks, particularly those related to building, testing, and deploying software. "Scalable Jenkins" means configuring Jenkins in a way that allows it to efficiently handle a growing workload, much like having a spacious workshop capable of producing numerous machines.

- **Kubernetes**: Think of Kubernetes as the workshop manager. It's an orchestration platform that automates the process of deploying, scaling, and managing containerized applications. Kubernetes ensures that Jenkins and the team of tiny robots (Kaniko) work seamlessly together and can adapt to changing demands.
- **Kaniko**: Kaniko is equivalent to your team of miniature robot workers. In the context of containerization, Kaniko is a tool that aids in building container images, which are akin to the individual parts of your machines. What makes Kaniko special is that it can do this without needing elevated access to the Docker daemon. Unlike traditional container builders, Kaniko doesn't require special privileges, making it a more secure choice for constructing containers, especially within a Kubernetes environment.

Now, let's combine the three tools and see what we can achieve:

- **Building containers at scale**: Your workshop can manufacture multiple machines simultaneously, thanks to Jenkins and the tiny robots. Similarly, with Jenkins on Kubernetes using Kaniko, you can efficiently and concurrently create container images. This ability to scale is crucial in modern application development, where containerization plays a pivotal role.
- **Isolation and security**: Just as Kaniko's tiny robots operate within a controlled environment, Kaniko ensures that container image building takes place in an isolated and secure manner within a Kubernetes cluster. This means that different teams or projects can use Jenkins and Kaniko without interfering with each other's container-building processes.
- **Consistency and automation**: Similar to how the conveyor belt (Jenkins) guarantees consistent machine assembly, Jenkins on Kubernetes with Kaniko ensures uniform container image construction. Automation is at the heart of this setup, simplifying the process of building and managing container images for applications.

To summarize, scalable Jenkins on Kubernetes with Kaniko refers to the practice of setting up Jenkins to efficiently build and manage container images using Kaniko within a Kubernetes environment. It enables consistent, parallel, and secure construction of container images, aligning perfectly with modern software development workflows.

So, the analogy of a workshop with Jenkins, Kubernetes, and Kaniko vividly illustrates how this setup streamlines container image building, making it scalable, efficient, and secure for contemporary software development practices. Now, let's dive deeper into Jenkins.

Jenkins is the most popular CI tool available in the market. It is open source, simple to install, and runs with ease. It is a Java-based tool with a plugin-based architecture designed to support several integrations, such as with a source code management tool such as *Git*, *SVN*, and *Mercurial*, or with popular artifact repositories such as *Nexus* and *Artifactory*. It also integrates well with well-known build tools such as *Ant*, *Maven*, and *Gradle*, aside from the standard shell scripting and Windows batch file executions.

Jenkins follows a *controller-agent* model. Though technically, you can run all your builds on the controller machine itself, it makes sense to offload your CI builds to other servers in your network to have a distributed architecture. This does not overload your controller machine. You can use it to store the build configurations and other management data and manage the entire CI build cluster, something along the lines of what's shown in the following diagram:

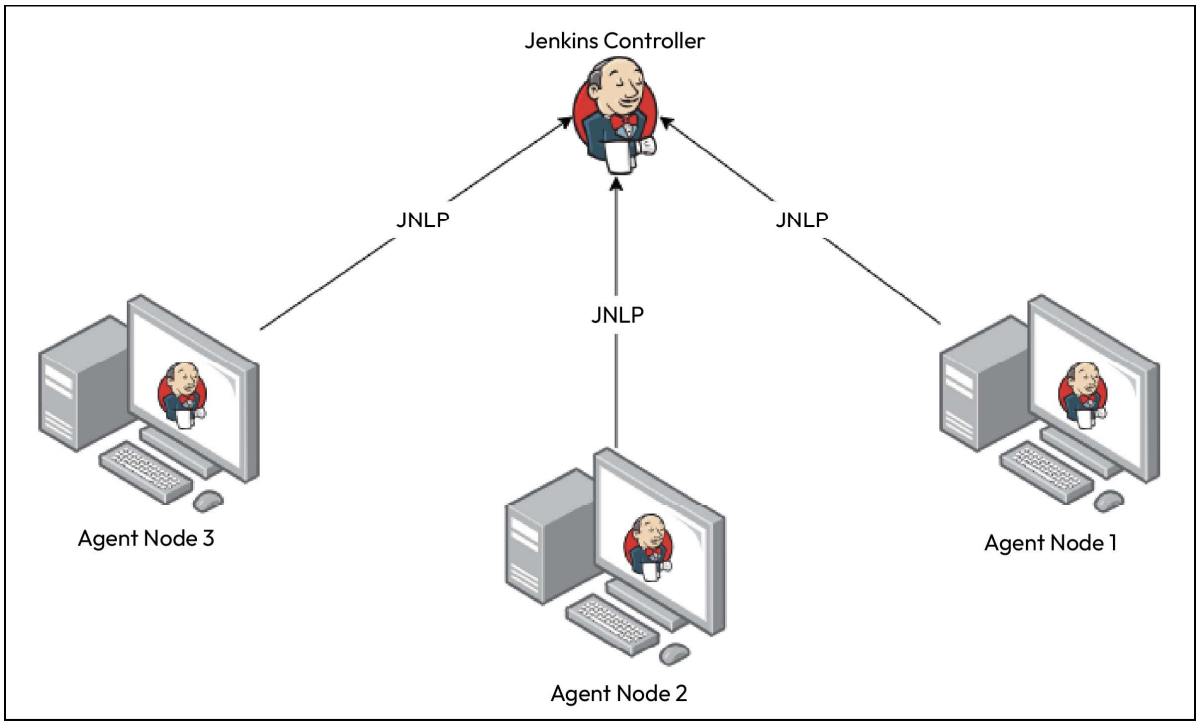

Figure 11.7 – Scalable Jenkins

In the preceding diagram, multiple static Jenkins agents connect to a Jenkins controller. Now, this architecture works well, but it needs to be more scalable. Modern DevOps emphasizes resource utilization, so we only want to roll out an agent machine when we want to build. Therefore, automating your builds to roll out an agent machine when required is a better way to do it. This might be overkill when rolling out new virtual machines, as it takes some minutes to provision a new VM, even when using a prebuilt image with Packer. A better alternative is to use a container.

Jenkins integrates quite well with Kubernetes, allowing you to run your build on a Kubernetes cluster. That way, whenever you trigger a build on Jenkins, Jenkins instructs Kubernetes to create a new agent container that will then connect with the controller machine and run the build within itself. This is *build on-demand* at its best. The following diagram shows this process in detail:

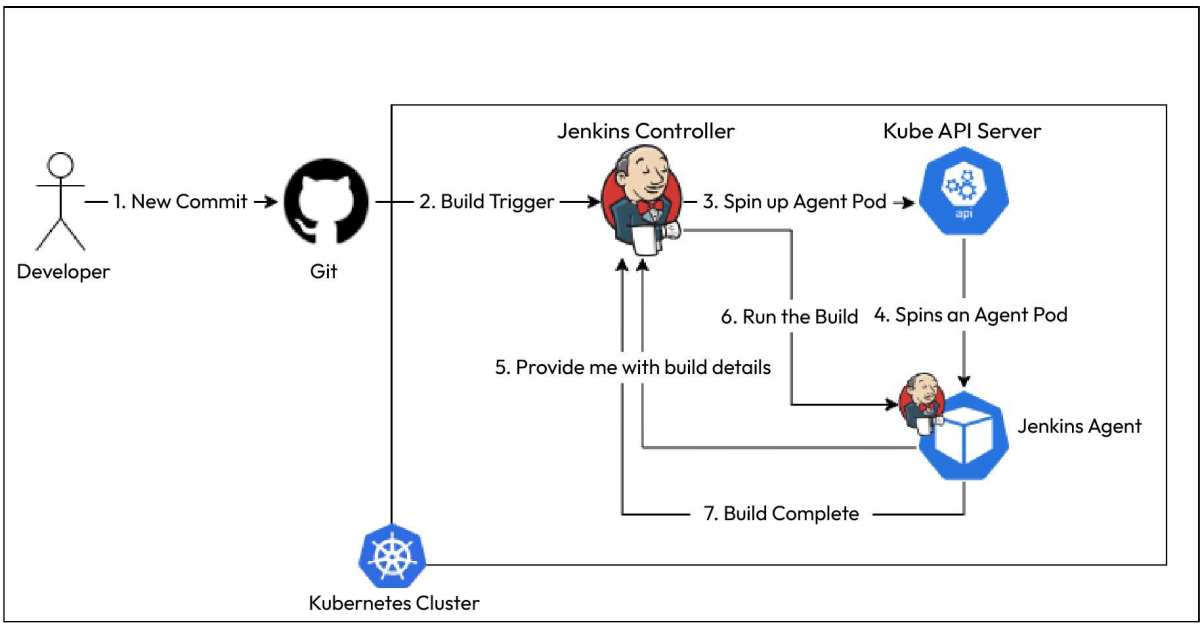

Figure 11.8 – Scalable Jenkins CI workflow

This sounds great, and we can go ahead and run this build, but there are issues with this approach. We must understand that the Jenkins controller and agents run as containers and aren't full-fledged virtual machines. Therefore, if we want to run a Docker build within the container, we must run the container in privileged mode. This isn't a security best practice, and your admin should already have turned that off. This is because running a container in privileged mode exposes your host filesystem to the container. A hacker who can access your container will have full access so that they can do whatever they want in your system.

To solve that problem, you can use a container build tool such as **Kaniko**. Kaniko is a build tool provided by Google that helps you build your containers without access to the Docker daemon, and you do not even need Docker installed in your container. It is a great way to run your builds within a **Kubernetes cluster** and create a scalable CI environment. It is effortless, not hacky, and provides a secure method of building your containers, as we will see in the subsequent sections.

This section will use **Google Kubernetes Engine** (**GKE**). As mentioned previously, Google Cloud provides a free trial worth $300 for 90 days. You can sign up at `https://cloud.google.com/free` if you have not already done so.

Spinning up Google Kubernetes Engine

Once you've signed up and are in your console, open the **Google Cloud Shell** CLI to run the following commands.

You need to enable the Kubernetes Engine API first using the following command:

```
$ gcloud services enable container.googleapis.com
```

To create a two-node autoscaling GKE cluster that scales from *one* to *five* nodes, run the following command:

```
$ gcloud container clusters create cluster-1 --num-nodes 2 \
--enable-autoscaling --min-nodes 1 --max-nodes 5 --zone us-central1-a
```

And that's it! The cluster will be up and running.

You must also clone the following GitHub repository for some of the exercises provided: `https://github.com/PacktPublishing/Modern-DevOps-Practices-2e`.

Run the following command to clone the repository into your home directory and `cd` into the following directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch11/jenkins/jenkins-controller
```

We will use the **Jenkins Configuration as Code** feature to configure Jenkins as it is a declarative way of managing your configuration and is also GitOps-friendly. You need to create a simple YAML file with all the required configurations and then copy the file to the Jenkins controller after setting an environment variable that points to the file. Jenkins will then automatically configure all aspects defined in the YAML file on bootup.

Let's start by creating the `casc.yaml` file to define our configuration.

Creating the Jenkins CaC (JCasC) file

The **Jenkins CaC** (**JCasC**) file is a simple YAML file that helps us define Jenkins configuration declaratively. We will create a single `casc.yaml` file for that purpose, and I will explain parts of it. Let's start by defining **Jenkins Global Security**.

Configuring Jenkins Global Security

By default Jenkins is insecure – that is, if you fire up a vanilla Jenkins from the official Docker image and expose it, anyone can do anything with that Jenkins instance. To ensure that we protect it, we need the following configuration:

```
jenkins:
  remotingSecurity:
   enabled: true
  securityRealm:
```

```
    local:
      allowsSignup: false
      users:
       - id: ${JENKINS_ADMIN_ID}
         password: ${JENKINS_ADMIN_PASSWORD}
  authorizationStrategy:
    globalMatrix:
      permissions:
        - "Overall/Administer:admin"
        - "Overall/Read:authenticated"
```

In the preceding configuration, we've defined the following:

- `remotingSecurity`: We've enabled this feature, which will secure the communication between the Jenkins controller and agents that we will create dynamically using Kubernetes.
- `securityRealm`: We've set the security realm to `local`, which means that the Jenkins controller itself will do all authentication and user management. We could have also offloaded this to an external entity such as LDAP:
  - `allowsSignup`: This is set to `false`. This means you don't see a sign-up link on the Jenkins home page, and the Jenkins admin should manually create users.
  - `users`: We'll create a single user with `id` and `password` sourced from two environment variables called `JENKINS_ADMIN_ID` and `JENKINS_ADMIN_PASSWORD`, respectively.
- `authorizationStrategy`: We've defined a matrix-based authorization strategy where we provide administrator privileges to `admin` and read privileges to `authenticated` non-admin users.

Also, as we want Jenkins to execute all their builds in the agents and not the controller machine, we need to specify the following settings:

```
jenkins:
  systemMessage: "Welcome to Jenkins!"
  numExecutors: 0
```

We've set `numExecutors` to `0` to allow no builds on the controller and also set `systemMessage` on the Jenkins welcome screen.

Now that we've set up the security aspects of the Jenkins controller, we will configure Jenkins to connect with the Kubernetes cluster.

Connecting Jenkins with the cluster

We will install the Kubernetes plugin to connect the Jenkins controller with the cluster. We're doing this because we want Jenkins to dynamically spin up agents for builds as Kubernetes **pods**.

We will start by creating a `kubernetes` configuration under `jenkins.clouds`, as follows:

```
jenkins
  clouds:
  - kubernetes:
      serverUrl: "https://<kubernetes_control_plane_ip>"
      jenkinsUrl: "http://jenkins-service:8080"
      jenkinsTunnel: "jenkins-service:50000"
      skipTlsVerify: false
      useJenkinsProxy: false
      maxRequestsPerHost: 32
      name: "kubernetes"
      readTimeout: 15
      podLabels:
        - key: jenkins
          value: agent
...
```

As we have a placeholder called `<kubernetes_control_plane_ip>` within the configuration, we must replace this with the Kubernetes control plane's IP address. Run the following command to fetch the control plane's IP address:

```
$ kubectl cluster-info | grep "control plane"
Kubernetes control plane is running at https://35.224.6.58
```

Now, replace the `<kubernetes_control_plane_ip>` placeholder with the actual IP address you obtained from the preceding command by using the following command:

```
$ sed -i 's/<kubernetes_control_plane_ip>/actual_ip/g' casc.yaml
```

Let's look at each attribute in the config file:

- `serverUrl`: This denotes the Kubernetes control plane server URL, allowing the Jenkins controller to communicate with the Kubernetes API server.
- `jenkinsUrl`: This denotes the Jenkins controller URL. We've set it to `http://jenkins-service:8080`.
- `jenkinsTunnel`: This describes how the agent pods will connect with the Jenkins controller. As the JNLP port is `50000`, we've set it to `jenkins-service:50000`.
- `podLabels`: We've also set up some pod labels, `key=jenkins` and `value=agent`. These will be set on the agent pods.

Other attributes are also set to their default values.

Every Kubernetes cloud configuration consists of multiple pod `templates` describing how the agent pods will be configured. The configuration looks like this:

```
   - kubernetes:
...
      templates:
      - name: "jenkins-agent"
        label: "jenkins-agent"
        hostNetwork: false
        nodeUsageMode: "NORMAL"
        serviceAccount: "jenkins"
        imagePullSecrets:
          - name: regcred
        yamlMergeStrategy: "override"
        containers:
        ...
```

Here, we've defined the following:

- The template's `name` and `label`. We set both to `jenkins-agent`.
- `hostNetwork`: This is set to `false` as we don't want the container to interact with the host network.
- `seviceAccount`: We've set this to `jenkins` as we want to use this service account to interact with Kubernetes.
- `imagePullSecrets`: We have also provided an image pull secret called `regcred` to authenticate with the container registry to pull the `jnlp` image.

Every pod template also contains a **container template**. We can define that using the following configuration:

```
        ...
        containers:
        - name: jnlp
          image: "<your_dockerhub_user>/jenkins-jnlp-kaniko"
          workingDir: "/home/jenkins/agent"
          command: ""
          args: ""
          livenessProbe:
            failureThreshold: 1
            initialDelaySeconds: 2
            periodSeconds: 3
            successThreshold: 4
            timeoutSeconds: 5
        volumes:
          - secretVolume:
              mountPath: /kaniko/.docker
              secretName: regcred
```

Here, we have specified the following:

- `name`: Set to `jnlp`.
- `image`: Here, we've specified the *Docker agent image* we will build in the next section. Ensure that you replace the `<your_dockerhub_user>` placeholder with your Docker Hub user by using the following command:

```
$ sed -i 's/<your_dockerhub_user>/actual_dockerhub_user/g' casc.yaml
```

- `workingDir`: Set to `/home/jenkins/agent`.
- We've set the `command` and `args` fields to blank as we don't need to pass them.
- `livenessProbe`: We've defined a liveness probe for the agent pod.
- `volumes`: We've mounted the `regcred` secret to the `kaniko/.docker` file as a volume. As `regcred` contains the Docker registry credentials, Kaniko will use this to connect with your container registry.

Now that our configuration file is ready, we'll go ahead and install Jenkins in the next section.

Installing Jenkins

As we're running on a Kubernetes cluster, we only need the latest official Jenkins image from Docker Hub. We will customize the image according to our requirements.

The following `Dockerfile` file will help us create the image with the required plugins and the initial configuration:

```
FROM jenkins/jenkins
ENV CASC_JENKINS_CONFIG /usr/local/casc.yaml
ENV JAVA_OPTS -Djenkins.install.runSetupWizard=false
COPY casc.yaml /usr/local/casc.yaml
COPY plugins.txt /usr/share/jenkins/ref/plugins.txt
RUN jenkins-plugin-cli --plugin-file /usr/share/jenkins/ref/plugins.txt
```

The `Dockerfile` starts from the Jenkins base image. Then, we declare two environment variables – `CASC_JENKINS_CONFIG`, which points to the `casc.yaml` file we defined in the previous section, and `JAVA_OPTS`, which tells Jenkins not to run the setup wizard. Then, we copy the `casc.yaml` and `plugins.txt` files to their respective directories within the Jenkins container. Finally, we run `jenkins-plugins-cli` on the `plugins.txt` file, which installs the required plugins.

The `plugins.txt` file contains a list of all Jenkins plugins that we will need in this setup.

> **Tip**
>
> You can customize and install more plugins for the controller image based on your requirements by updating the `plugins.txt` file.

Let's build the image from the `Dockerfile` file using the following command:

```
$ docker build -t <your_dockerhub_user>/jenkins-controller-kaniko .
```

Now that we've built the image, use the following command to log in and push the image to Docker Hub:

```
$ docker login
$ docker push <your_dockerhub_user>/jenkins-controller-kaniko
```

We must also build the Jenkins agent image to run our builds. Remember that Jenkins agents need all the supporting tools you need to run your builds. You can find the resources for the agents in the following directory:

```
$ cd ~/modern-devops/ch11/jenkins/jenkins-agent
```

We will use the following `Dockerfile` to do that:

```
FROM gcr.io/kaniko-project/executor:v1.13.0 as kaniko
FROM jenkins/inbound-agent
COPY --from=kaniko /kaniko /kaniko
WORKDIR /kaniko
USER root
```

This `Dockerfile` uses a multi-stage build to take the `kaniko` base image and copy the `kaniko` binary from the `kaniko` base image to the `inbound-agent` base image. Let's go ahead and build and push the container using the following commands:

```
$ docker build -t <your_dockerhub_user>/jenkins-jnlp-kaniko .
$ docker push <your_dockerhub_user>/jenkins-jnlp-kaniko
```

To deploy Jenkins on our Kubernetes cluster, we will first create a `jenkins` service account. A Kubernetes **service account** resource helps pods authenticate with the **Kubernetes API server**. We will give the service account permission to interact with the Kubernetes API server as `cluster-admin` using a cluster role binding. A Kubernetes **ClusterRoleBinding** resource helps provide permissions to a service account to perform certain actions in the Kubernetes cluster. The `jenkins-sa-crb.yaml` manifest describes this. To access these resources, run the following command:

```
$ cd ~/modern-devops/ch11/jenkins/jenkins-controller
```

To apply the manifest, run the following command:

```
$ kubectl apply -f jenkins-sa-crb.yaml
```

The next step involves creating a **PersistentVolumeClaim** resource to store Jenkins data to ensure that the Jenkins data persists beyond the pod's life cycle and will exist even when we delete the pod.

To apply the manifest, run the following command:

```
$ kubectl apply -f jenkins-pvc.yaml
```

Then, we will create a Kubernetes **Secret** called `regcred` to help the Jenkins pod authenticate with the Docker registry. Use the following command to do so:

```
$ kubectl create secret docker-registry regcred --docker-username=<username> \
--docker-password=<password> --docker-server=https://index.docker.io/v1/
```

Now, we'll define a **Deployment** resource, `jenkins-deployment.yaml`, that will run the Jenkins container. The pod uses the `jenkins` service account and defines a **PersistentVolume** resource called `jenkins-pv-storage` using the `PersistentVolumeClaim` resource called `jenkins-pv-claim` that we defined. We define the Jenkins container that uses the Jenkins controller image we created. It exposes HTTP port `8080` for the *Web UI*, and port `50000` for *JNLP*, which the agents would use to interact with the Jenkins controller. We will also mount the `jenkins-pv-storage` volume to `/var/jenkins_home` to persist the Jenkins data beyond the pod's life cycle. We specify `regcred` as the `imagePullSecret` attribute in the pod image. We also use `initContainer` to assign ownership to `jenkins` for `/var/jenkins_home`.

As the file contains placeholders, replace `<your_dockerhub_user>` with your Docker Hub user and `<jenkins_admin_pass>` with a Jenkins admin password of your choice using the following commands:

```
$ sed -i 's/<your_dockerhub_user>/actual_dockerhub_user/g' jenkins-deployment.yaml
```

Apply the manifest using the following command:

```
$ kubectl apply -f jenkins-deployment.yaml
```

As we've created the deployment, we can expose the deployment on a **LoadBalancer** Service using the `jenkins-svc.yaml` manifest. This service exposes ports `8080` and `50000` on a load balancer. Use the following command to apply the manifest:

```
$ kubectl apply -f jenkins-svc.yaml
```

Let's get the service to find the external IP to use that to access Jenkins:

```
$ kubectl get svc jenkins-service
NAME             EXTERNAL-IP               PORT(S)
jenkins-service  LOAD_BALANCER_EXTERNAL_IP 8080,50000
```

Now, to access the service, go to `http://<LOAD_BALANCER_EXTERNAL_IP>:8080` in your browser window:

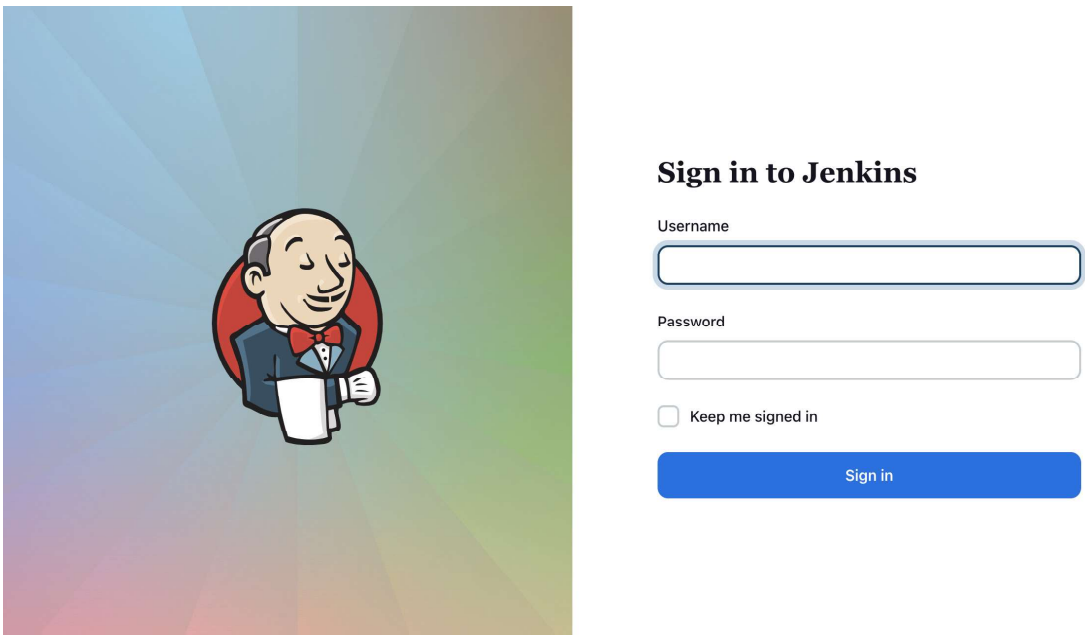

Figure 11.9 – Jenkins login page

As we can see, we're greeted with a login page. This means Global Security is working correctly. Let's log in using the admin username and password we set:

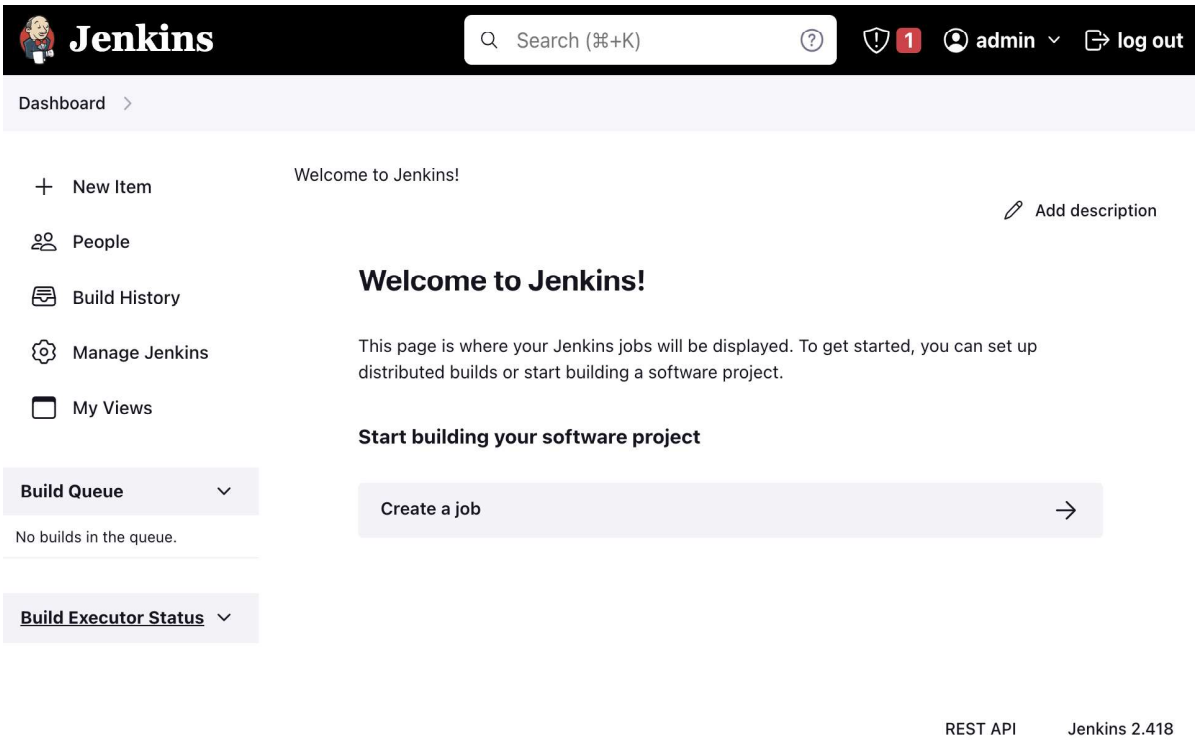

Figure 11.10 – Jenkins home page

As we can see, we've successfully logged in to Jenkins. Now, let's go ahead and create our first Jenkins job.

Running our first Jenkins job

Before we create our first job, we'll have to prepare our repository to run the job. We will reuse the `mdo-posts` repository for this. We will copy a `build.sh` file to the repository, which will build the container image for the **posts** microservice and push it to Docker Hub.

The `build.sh` script takes `IMAGE_ID` and `IMAGE_TAG` as arguments. It passes them to the **Kaniko** executor script, which builds the container image using the `Dockerfile` and pushes it to Docker Hub using the following code:

```
IMAGE_ID=$1 && \
IMAGE_TAG=$2 && \
export DOCKER_CONFIG=/kaniko/.dockerconfig && \
/kaniko/executor \
  --context $(pwd) \
  --dockerfile $(pwd)/Dockerfile \
  --destination $IMAGE_ID:$IMAGE_TAG \
  --force
```

We will need to copy this file to our local repository using the following commands:

```
$ cp ~/modern-devops/ch11/jenkins/jenkins-agent/build.sh ~/mdo-posts/
```

Once you've done this, `cd` into your local repository – that is, `~/mdo-posts` – and commit and push your changes to GitHub. Once you've done this, you'll be ready to create a job in Jenkins.

To create a new job in Jenkins, go to the Jenkins home page and select **New Item | Freestyle Job**. Provide a job name (preferably the same as the Git repository name), then click **Next**.

Click on **Source Code Management**, select **Git**, and add your Git repository URL, as shown in the following example. Specify the branch from where you want to build:

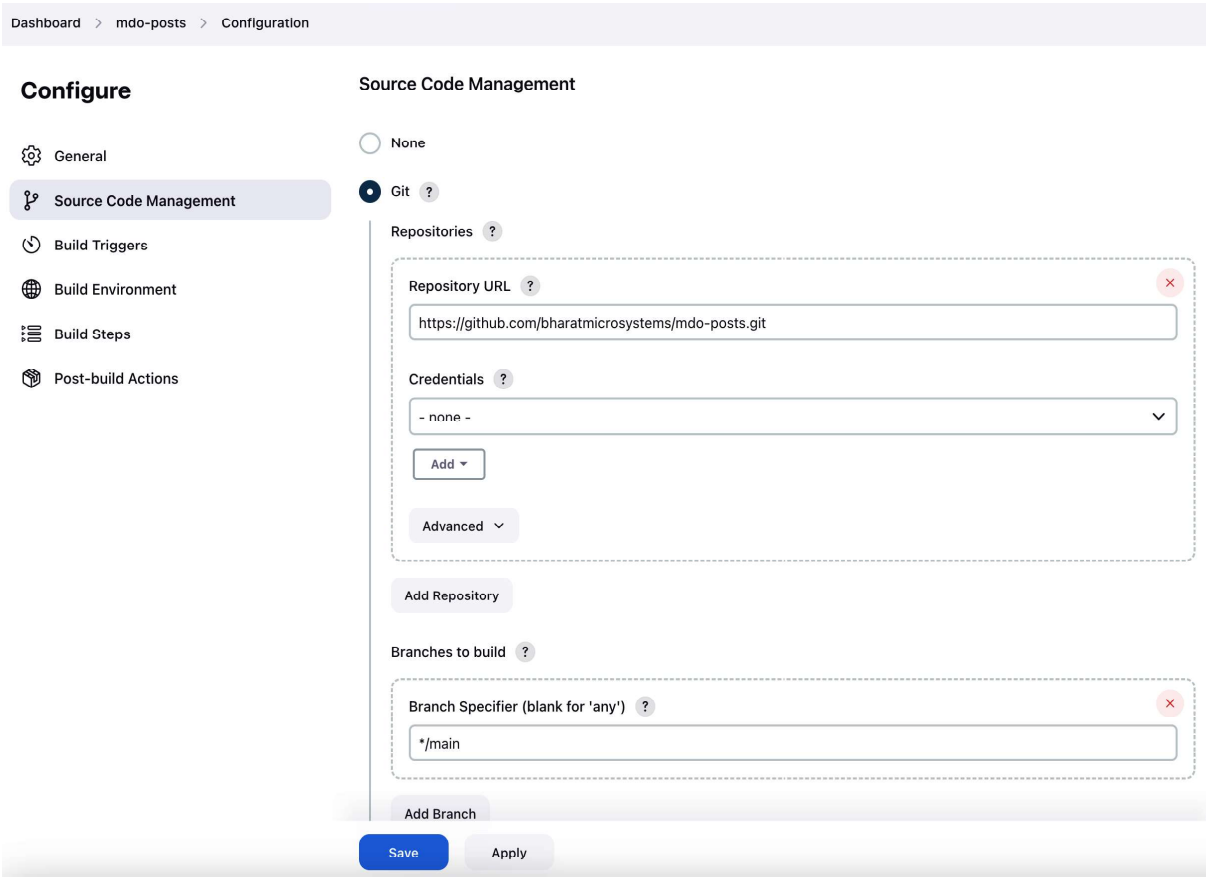

Figure 11.11 – Jenkins Souce Code Management configuration

Go to **Build Triggers**, select **Poll SCM**, and add the following details:

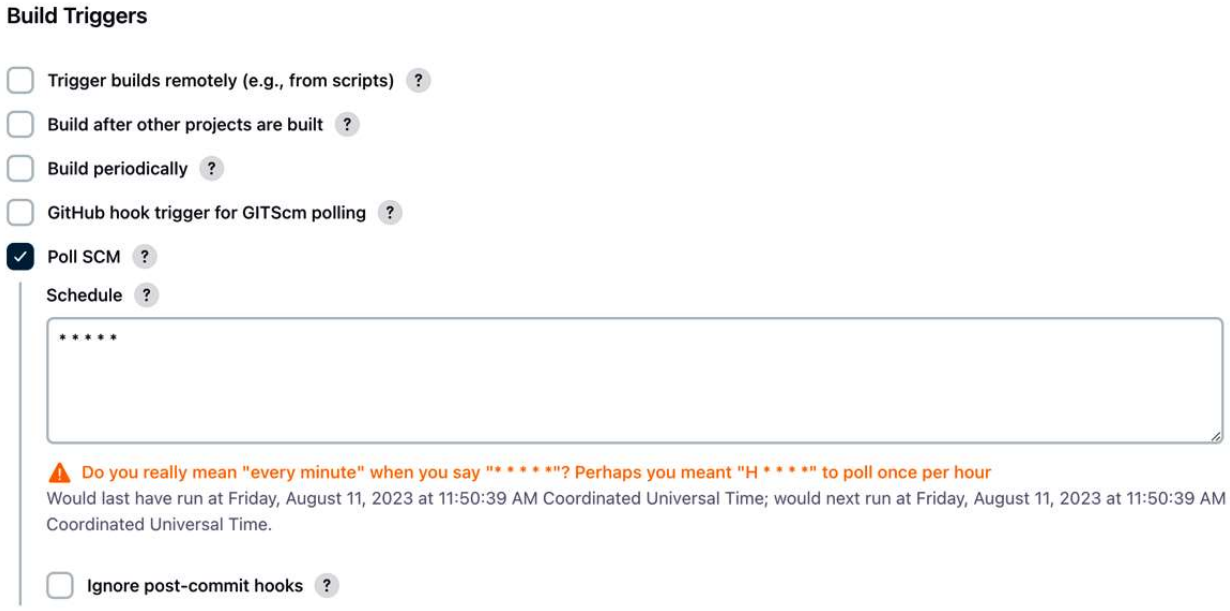

Figure 11.12 – Jenkins – Build Triggers configuration

Then, click on **Build | Add Build Step | Execute shell**. The **Execute shell** build step executes a sequence of shell commands on the Linux CLI. In this example, we're running the `build.sh` script with the `<your_dockerhub_user>/<image>` argument and the image tag. Change the details according to your requirements. Once you've finished, click **Save**:

Figure 11.13 – Jenkins – Execute shell configuration

Now, we're ready to build this job. To do so, you can either go to your job configuration and click **Build Now** or push a change to GitHub. You should see something like the following:

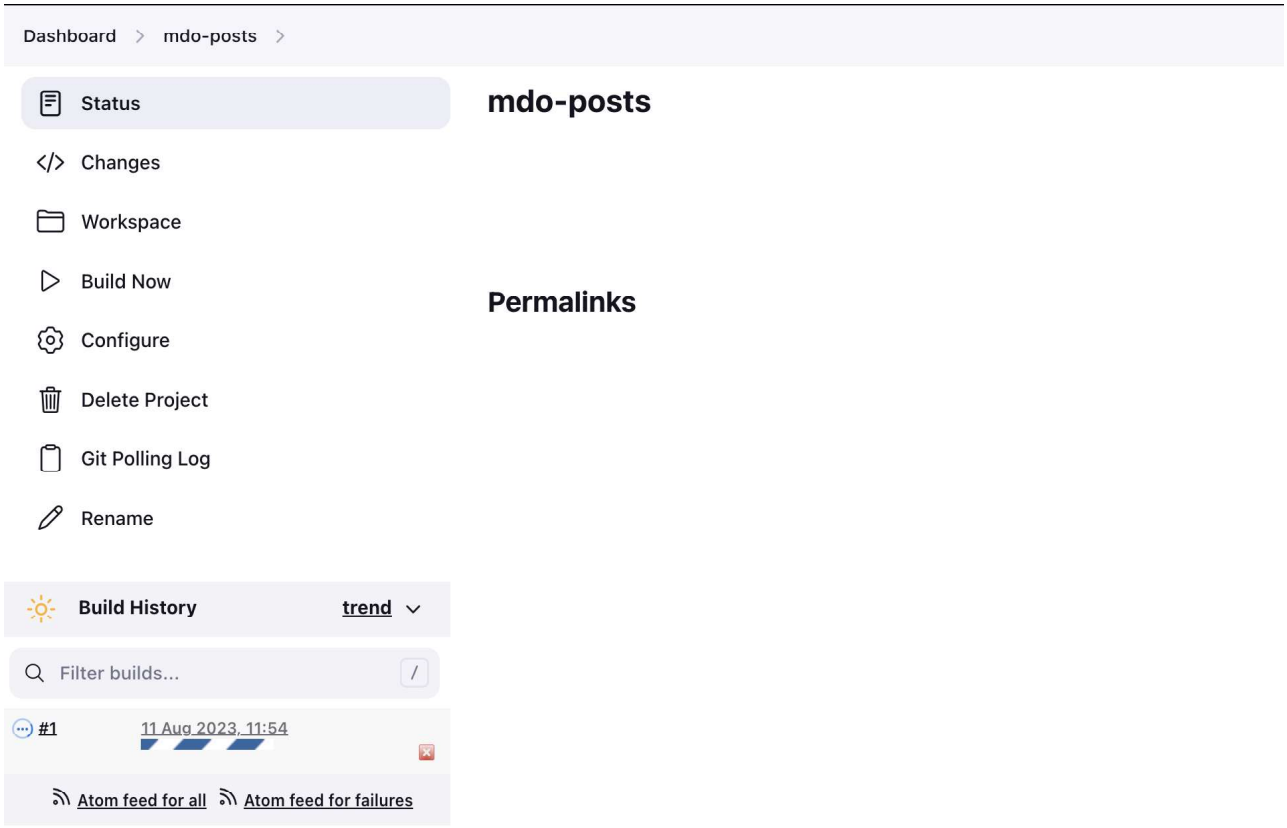

Figure 11.14 – Jenkins job page

Jenkins will successfully create an agent pod in Kubernetes, where it will run this job, and soon, the job will start building. Click **Build | Console Output**. If everything is OK, you'll see that the build was successful and that Jenkins has built the **posts** service and executed a unit test before pushing the Docker image to the registry:

```
[notice] A new release of pip is available: 23.0.1 -> 23.2.1
[notice] To update, run: pip install --upgrade pip
 [36mINFO [0m[0030] Taking snapshot of full filesystem...
 [36mINFO [0m[0035] EXPOSE 5000
 [36mINFO [0m[0035] Cmd: EXPOSE
 [36mINFO [0m[0035] Adding exposed port: 5000/tcp
 [36mINFO [0m[0035] COPY . .
 [36mINFO [0m[0036] Taking snapshot of files...
 [36mINFO [0m[0036] RUN python app.test.py
 [36mINFO [0m[0036] Cmd: /bin/sh
 [36mINFO [0m[0036] Args: [-c python app.test.py]
 [36mINFO [0m[0036] Running: [/bin/sh -c python app.test.py]
........
----------------------------------------------------------------------
Ran 8 tests in 0.059s

OK
 [36mINFO [0m[0037] Taking snapshot of full filesystem...
 [36mINFO [0m[0038] CMD ["flask", "run"]
 [36mINFO [0m[0038] Pushing image to bharamicrosystems/mdo-posts:latest
 [36mINFO [0m[0047] Pushed index.docker.io/bharamicrosystems/mdo-
posts@sha256:588d8a408b365580bec4864690e7a42401ff2a3374d500eefc0e93333c3bfb05
Finished: SUCCESS
```

Figure 11.15 – Jenkins console output

With that, we're able to run a Docker build using a scalable Jenkins server. As we can see, we've set up polling on the SCM settings to look for changes every minute and build the job if we detect any. However, this is resource-intensive and does not help in the long run. Just imagine that you have hundreds of jobs interacting with multiple GitHub repositories, and the Jenkins controller is polling them every minute. A better approach would be if GitHub could trigger a **post-commit webhook** on Jenkins. Here, Jenkins can build the job whenever there are changes in the repository. We'll look at that scenario in the next section.

Automating a build with triggers

The best way to allow your CI build to trigger when you make changes to your code is to use a post-commit webhook. We looked at such an example in the GitHub Actions workflow. Let's try to automate the build with triggers in the case of Jenkins. We'll have to make some changes on both the Jenkins and the GitHub sides to do so. We'll deal with Jenkins first; then, we'll configure GitHub.

Go to **Job configuration** | **Build Triggers** and make the following changes:

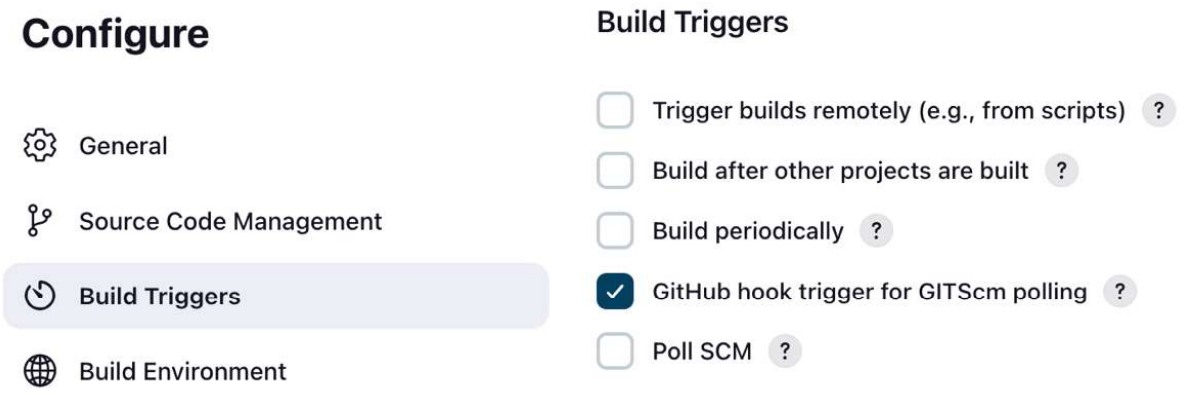

Figure 11.16 – Jenkins GitHub hook trigger

Save the configuration by clicking **Save**. Now, go to your GitHub repository, click **Settings** | **Webhooks** | **Add Webhook**, and add the following details. Then, click **Add Webhook**:

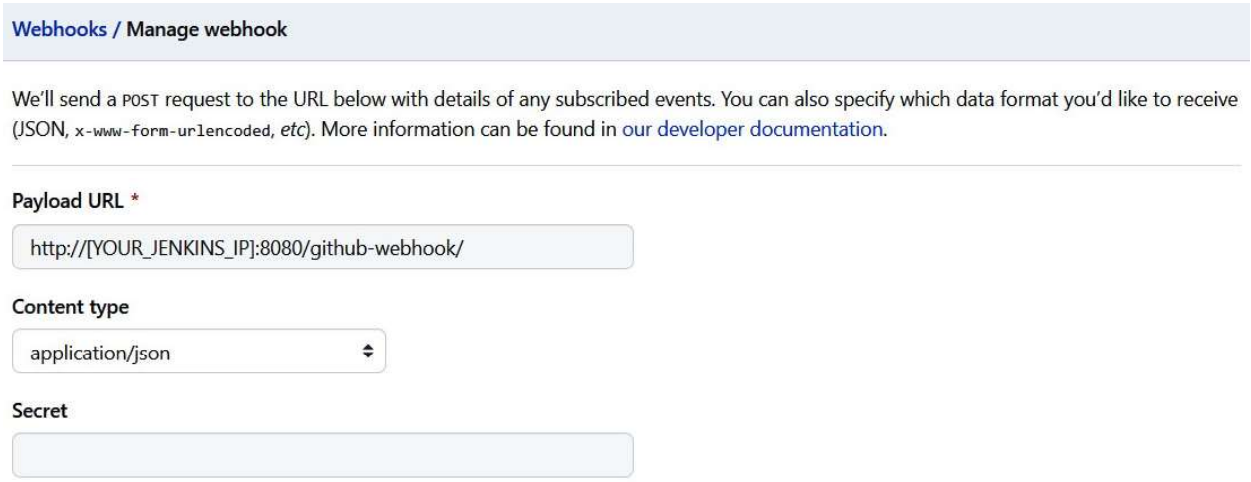

Figure 11.17 – GitHub webhook

Now, push a change to the repository. The job on Jenkins will start building:

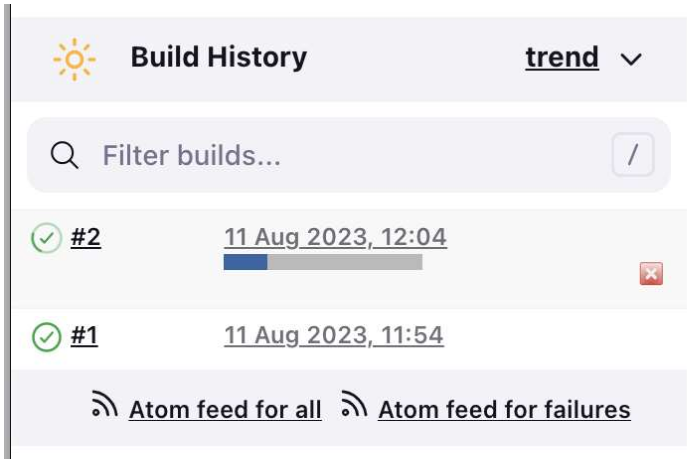

Figure 11.18 – Jenkins GitHub webhook trigger

This is automated build triggers in action. Jenkins is one of the most popular open source CI tools on the market. The most significant advantage of it is that you can pretty much run it anywhere. However, it does come with some management overhead. You may have noticed how simple it was to start with GitHub Actions, but Jenkins is slightly more complicated.

Several other SaaS platforms offer CI and CD as a service. For instance, if you are running on AWS, you'd get their inbuilt CI with **AWS Code Commit** and **Code Build**; Azure provides an entire suite of services for CI and CD in their **Azure DevOps** offering; and GCP provides **Cloud Build** for that job.

CI follows the same principle, regardless of the tooling you choose to implement. It is more of a process and a cultural change within your organization. Now, let's look at some of the best practices regarding CI.

Building performance best practices

CI is an ongoing process, so you will have a lot of parallel builds running within your environment at a given time. In such situations, we can optimize them using several best practices.

Aim for faster builds

The faster you can complete your build, the quicker you will get feedback and run your next iteration. A slow build slows down your development team. Take steps to ensure that builds are faster. For example, in Docker's case, it makes sense to use smaller base images as it will download the code from the image registry every time it does a build. Using a single base image for most builds will also speed up your build time. Using tests will help, but make sure that they aren't long-running. We want to avoid a CI build that runs for hours. Therefore, it would be good to offload long-running tests into another job or use a pipeline. Run activities in parallel if possible.

Always use post-commit triggers

Post-commit triggers help your team significantly. They will not have to log in to the CI server and trigger the build manually. That completely decouples your development team from CI management.

Configure build reporting

You don't want your development team to log in to the CI tool and check how the build runs. Instead, all they want to know is the result of the build and the build logs. Therefore, you can configure build reporting to send your build status via email or, even better, using a **Slack** channel.

Customize the build server size

Not all builds work the same in similar kinds of build machines. You may want to choose machines based on what suits your build environment best. If your builds tend to consume more CPU than memory, it will make sense to choose such machines to run your builds instead of the standard ones.

Ensure that your builds only contain what you need

Builds move across networks. You download base images, build your application image, and push that to the container registry. Bloated images not only take a lot of network bandwidth and time to transmit but also make your build vulnerable to security issues. Therefore, it is always best practice to only include what you require in the build and avoid bloat. You can use Docker's **multi-stage builds** for these kinds of situations.

Parallelize your builds

Run tests and build processes concurrently to reduce overall execution time. Leverage distributed systems or cloud-based CI/CD platforms for scalable parallelization, allowing you to handle larger workloads efficiently.

Make use of caching

Cache dependencies and build artifacts to prevent redundant downloads and builds, saving valuable time. Implement caching mechanisms such as Docker layer caching or use your package manager's built-in caches to minimize data transfer and build steps.

Use incremental building

Configure your CI/CD pipeline to perform incremental builds, rebuilding only what has changed since the last build. Maintain robust version control practices to accurately track and identify changes.

Optimize testing

Prioritize and optimize tests by running quicker unit tests before slower integration or end-to-end tests. Use testing frameworks such as TestNG, JUnit, or PyTest to categorize and parallelize tests effectively.

Use artifact management

Efficiently store and manage build artifacts, preferably in a dedicated artifact repository such as Artifactory or Nexus. Implement artifact versioning and retention policies to maintain a clean artifact repository.

Manage application dependencies

Keep a clean and minimal set of dependencies to reduce build and test times. Regularly update dependencies to benefit from performance improvements and security updates.

Utilize Infrastructure as Code

Utilize **Infrastructure as Code** (**IaC**) to provision and configure build and test environments consistently. Optimize IaC templates to minimize resource utilization, ensuring efficient resource allocation.

Use containerization to manage build and test environments

Containerize applications and utilize container orchestration tools such as Kubernetes to manage test environments efficiently. Leverage container caching to accelerate image builds and enhance resource utilization.

Utilize cloud-based CI/CD

Consider adopting cloud-based CI/CD services such as AWS CodePipeline, Google Cloud Build, Azure DevOps, or Travis CI for enhanced scalability and performance. Harness on-demand cloud resources to expand parallelization capabilities and adapt to varying workloads.

Monitor and profile your CI/CD pipelines

Implement performance monitoring and profiling tools to identify bottlenecks and areas for improvement within your CI/CD pipeline. Regularly analyze build and test logs to gather insights for optimizing performance.

Pipeline optimization

Continuously review and optimize your CI/CD pipeline configuration for efficiency and relevance. Remove unnecessary steps or stages that do not contribute significantly to the process.

Implement automated cleanup

Implement automated cleanup routines to remove stale artifacts, containers, and virtual machines, preventing resource clutter. Regularly purge old build artifacts and unused resources to maintain a tidy environment.

Documentation and training

Document best practices and performance guidelines for your CI/CD processes, ensuring that the entire team follows these standards consistently. Provide training and guidance to team members to empower them to implement and maintain these optimization strategies effectively.

By implementing these strategies, you can significantly enhance the speed, efficiency, and reliability of your CI/CD pipeline, ultimately leading to smoother software development and delivery processes. These are some of the best practices at a high level, and they are not exhaustive, but they are good enough so that you can start optimizing your CI environment.

Summary

This chapter covered CI, and you understood the need for CI and the basic CI workflow for a container application. We then looked at GitHub Actions, which we can use to build an effective CI pipeline. Next, we looked at the Jenkins open source offering and deployed a scalable Jenkins on Kubernetes with Kaniko, setting up a Jenkins controller-agent model. We then understood how to use hooks for automating builds, both in the GitHub Actions-based workflow and the Jenkins-based workflow. Finally, we learned about build performance best practices and dos and don'ts.

By now, you should be familiar with CI and its nuances, along with the various tooling you can use to implement it.

In the next chapter, we will delve into continuous deployment/delivery in the container world.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Which of the following are CI tools? (Choose three)

   A. Jenkins

   B. GitHub Actions

   C. Kubernetes

   D. AWS Code Build

2. It is a best practice to configure post-commit triggers. (True/False)
3. Jenkins is a SaaS-based CI tool. (True/False)
4. Kaniko requires Docker to build your containers. (True/False)
5. Jenkins agents are required for which of the following reasons? (Choose three)

   A. They make builds more scalable

   B. They help offload the management function from the Jenkins controller

   C. They allow for parallel builds

   D. They keep the Jenkins controller less busy

6. Which of the following is required for a scalable Jenkins server, as described in the example in this chapter? (Choose three)

   A. Kubernetes cluster

   B. Jenkins controller node

   C. Jenkins agent node

   D. Credentials to interact with the container registry

Answers

The following are the answers to this chapter's questions:

1. A, B, D
2. True
3. False
4. False
5. A, C, D
6. A, B, D

12
Continuous Deployment/ Delivery with Argo CD

In the previous chapter, we looked at one of the key aspects of modern DevOps – **continuous integration** (**CI**). CI is the first thing most organizations implement when they embrace DevOps, but things don't end with CI, which only delivers a tested build in an artifact repository. Instead, we would also want to deploy the artifact to our environments. In this chapter, we'll implement the next part of the DevOps toolchain – **continuous deployment/delivery** (**CD**).

In this chapter, we're going to cover the following main topics:

- The importance of CD and automation
- CD models and tools
- The Blog App and its deployment configuration
- Continuous declarative IaC using an Environment repository
- Introduction to Argo CD
- Installing and setting up Argo CD
- Managing sensitive configurations and secrets
- Deploying the sample Blog App

Technical requirements

In this chapter, we will spin up a cloud-based Kubernetes cluster, **Google Kubernetes Engine** (**GKE**), for the exercises. At the time of writing, **Google Cloud Platform** (**GCP**) provides a free $300 trial for 90 days, so you can go ahead and sign up for one at `https://console.cloud.google.com/`.

You will also need to clone the following GitHub repository for some exercises: `https://github.com/PacktPublishing/Modern-DevOps-Practices`.

Run the following command to clone the repository into your home directory, and `cd` into the `ch12` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch12
```

So, let's get started!

The importance of CD and automation

CD forms the Ops part of your DevOps toolchain. So, while your developers are continuously building and pushing code and your CI pipeline is building, testing, and publishing the builds to your artifact repository, the Ops team will deploy the build to the test and staging environments. The QA team is the gatekeeper that will ensure that the code meets a certain quality, and only then will the Ops team deploy the code to production.

Now, for organizations implementing only the CI part, the rest of the activities are manual. For example, operators will pull the artifacts and run commands to do the deployments manually. Therefore, your deployment's velocity will depend on the availability of your Ops team to do it. As the deployments are manual, the process is error-prone, and human beings tend to make mistakes in repeatable jobs.

One of the essential principles of modern DevOps is to avoid **toil**. Toil is nothing but repeatable jobs that developers and operators do day in and day out, and all of that toil can be removed by automation. This will help your team focus on the more important things at hand.

With **continuous delivery**, standard tooling can deploy code to higher environments based on certain gate conditions. CD pipelines will trigger when a tested build arrives at the artifact repository or, in the case of GitOps, if any changes are detected in the Environment repository. The pipeline then decides, based on a set configuration, where and how to deploy the code. It also establishes whether manual checks are required, such as raising a change ticket and checking whether it's approved.

While **continuous deployment** and delivery are often confused with being the same thing, there is a slight difference between them. Continuous delivery enables your team to deliver tested code in your environment based on a human trigger. So, while you don't have to do anything more than click a button to do a deployment to production, it would still be initiated by someone at a convenient time (a maintenance window). Continuous deployments go a step further when they integrate with the CI process and will start the deployment process as soon as a new tested build is available for them to consume. There is no need for manual intervention, and continuous deployment will only stop in case of a failed test.

The monitoring tool forms the next part of the DevOps toolchain. The Ops team can learn from managing their production environment and provide developers with feedback regarding what they need to do better. That feedback ends up in the development backlog, and they can deliver it as features

in future releases. That completes the cycle, and now you have your team churning out a technology product continuously.

CD offers several advantages. Some of them are as follows:

- **Faster time to market**: CD and CI reduce the time it takes to deliver new features, enhancements, and bug fixes to end users. This agility can give your organization a competitive edge by allowing you to respond quickly to market demands.
- **Reduced risk**: By automating the deployment process and frequently pushing small code changes, you minimize the risk of large, error-prone deployments. Bugs and issues are more likely to be caught early, and rollbacks can be less complex.
- **Improved code quality**: Frequent automated testing and quality checks are an integral part of CD and CI. This results in higher code quality as developers are encouraged to write cleaner, more maintainable code. Any issues are caught and addressed sooner in the development process.
- **Enhanced collaboration**: CD and CI encourage collaboration between development and operations teams. It breaks down traditional silos and encourages cross-functional teamwork, leading to better communication and understanding.
- **Efficiency and productivity**: Automation of repetitive tasks, such as testing, building, and deployment, frees up developers' time to focus on more valuable tasks, such as creating new features and improvements.
- **Customer feedback**: CD allows you to gather feedback from real users more quickly. By deploying small changes frequently, you can gather user feedback and adjust your development efforts accordingly, ensuring that your product better meets user needs.
- **Continuous improvement**: CD promotes a culture of continuous improvement. By analyzing data on deployments and monitoring, teams can identify areas for enhancement and iterate on their processes.
- **Better security**: Frequent updates mean that security vulnerabilities can be addressed promptly, reducing the window of opportunity for attackers. Security checks can be automated and integrated into the CI/CD pipeline.
- **Reduced manual intervention**: CD minimizes the need for manual intervention in the deployment process. This reduces the potential for human error and streamlines the release process.
- **Scalability**: As your product grows and the number of developers and your code base complexity increases, CD can help maintain a manageable development process. It scales effectively by automating many of the release and testing processes.
- **Cost savings**: Although implementing CI/CD requires an initial investment in tools and processes, it can lead to cost savings in the long run by reducing the need for extensive manual testing, lowering deployment-related errors, and improving resource utilization.

- **Compliance and auditing**: For organizations with regulatory requirements, CD can improve compliance by providing a detailed history of changes and deployments, making it easier to track and audit code changes.

It's important to note that while CD and CI offer many advantages, they also require careful planning, infrastructure, and cultural changes to be effective.

There are several models and tools available to implement CD. We'll have a look at some of them in the next section.

CD models and tools

A typical CI/CD workflow looks as described in the following figure and the subsequent steps:

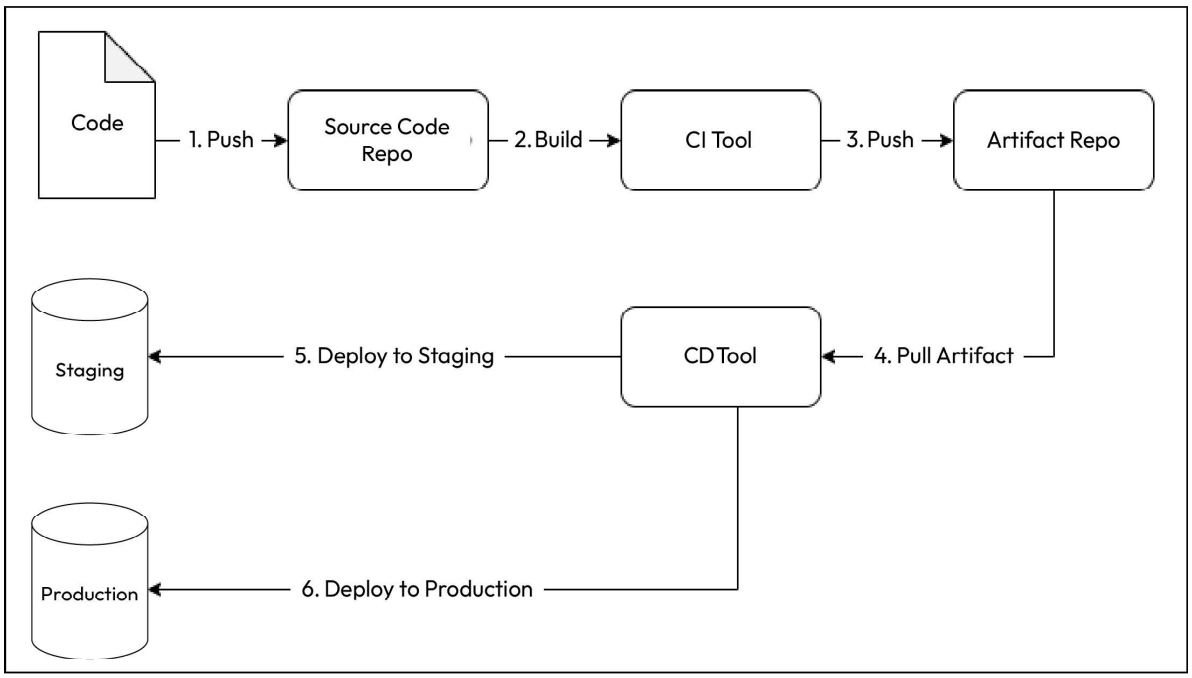

Figure 12.1 – CI/CD workflow

1. Your developers write code and push it to a code repository (typically a Git repository).
2. Your CI tool builds the code, runs a series of tests, and pushes the tested build to an artifact repository. Your CD tool then picks up the artifact and deploys it to your test and staging environments. Based on whether you want to do continuous deployment or delivery, it automatically deploys the artifact to the production environment.

Well, what do you choose for a delivery tool? Let's look at the example we covered in *Chapter 11*, *Continuous Integration*. We picked up the **posts** microservice app and used a CI tool such as GitHub Actions/Jenkins that uses **Docker** to create a container out of it and push it to our **Docker Hub** container registry. Well, we could have used the same tool for deploying to our environment.

For example, if we wanted to deploy to **Kubernetes**, it would have been a simple YAML update and `kubectl apply`. We could easily do this with any of those tools, but we chose not to do it. Why? The answer is simple – CI tools are meant for CI, and if you want to use them for anything else, you'll get stuck at a certain point. That does not mean that you cannot use these tools for CD. It will only suit a few use cases based on the deployment model you follow.

Several deployment models exist based on your application, technology stack, customers, risk appetite, and cost consciousness. Let's look at some of the popular deployment models that are used within the industry.

Simple deployment model

The **simple deployment model** is one of the most straightforward of all: you deploy the required version of your application after removing the old one. It completely replaces the previous version, and rolling back involves redeploying the older version after removing the deployed one:

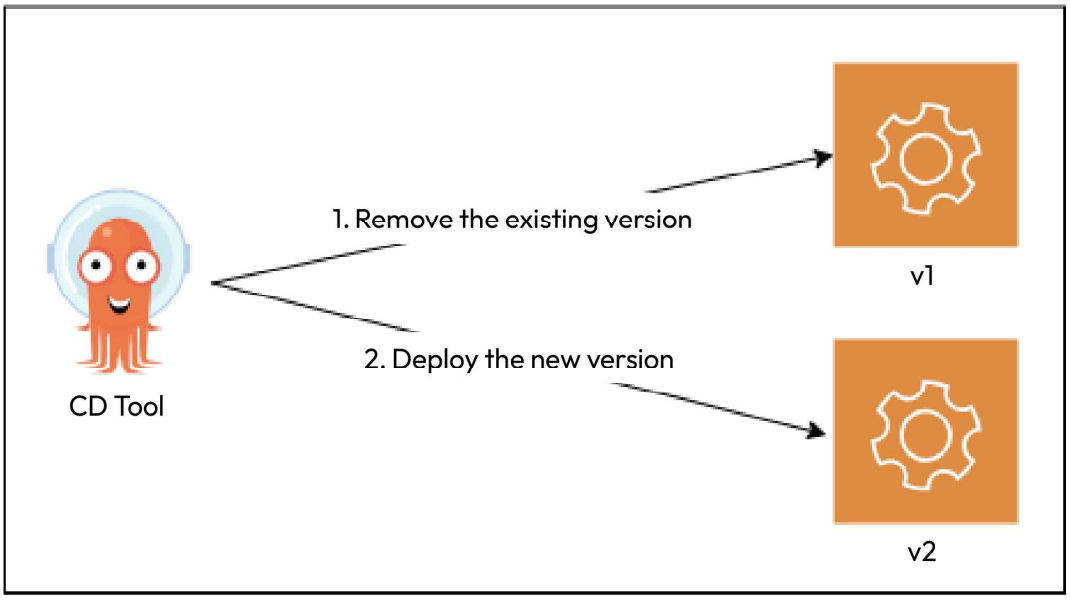

Figure 12.2 – Simple deployment model

As it is a simple way of deploying things, you can manage this using a CI tool such as **Jenkins** or **GitHub Actions**. However, the simple deployment model is not the most desired deployment method because of some inherent risks. This kind of change is disruptive and typically needs downtime. This means your service would remain unavailable to your customers for the upgrade period. It might be OK for organizations that do not have users 24/7, but disruptions eat into the **service-level objectives** (**SLOs**) and **service-level agreements** (**SLAs**) of global organizations. Even if there isn't one, they hamper customer experience and the organization's reputation.

Therefore, to manage such kinds of situations, we have some complex deployment models.

Complex deployment models

Complex deployment models, unlike simple deployment models, try to minimize disruptions and downtimes within the application and make rolling out releases more seamless to the extent that most users don't even notice when the upgrade is being conducted. Two main kinds of complex deployments are prevalent in the industry; let's take a look.

Blue/Green deployments

Blue/Green deployments (also known as **Red/Black deployments**) roll out the new version (*Green*) in addition to the existing version (*Blue*). You can then do sanity checks and other activities with the latest version to ensure that everything is good to go. Then, you can switch traffic from the old to the new version and monitor for any issues. If you encounter problems, you switch back traffic to the old version. Otherwise, you keep the latest version running and remove the old version:

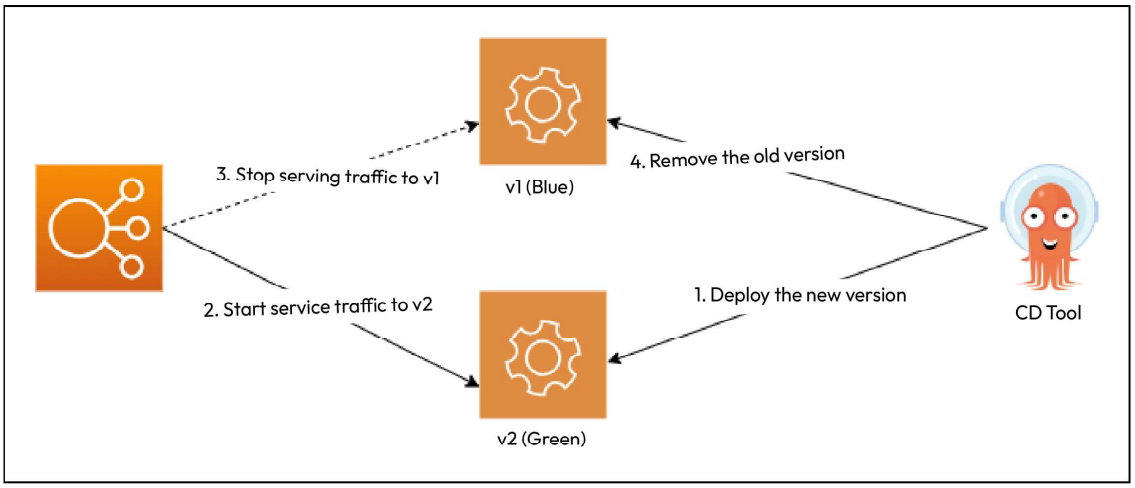

Figure 12.3 – Blue/Green deployments

You can take Blue/Green deployments to the next level using canary deployments.

Canary deployments and A/B testing

Canary deployments are similar to Blue/Green deployments but are generally utilized for risky upgrades. So, like Blue/Green deployments, we deploy the new version alongside the existing one. Instead of switching all traffic to the latest version at once, we only switch traffic to a small subset of users. As we do that, we can understand from our logs and user behaviors whether the switchover is causing any issues. This is called **A/B testing**. When we do A/B testing, we can target a specific group of users based on location, language, age group, or users who have opted to test Beta versions of a product. That will help organizations gather feedback without disrupting general users and make changes to the product once they're satisfied with what they are rolling out. You can make the release generally available by switching over the total traffic to the new version and getting rid of the old version:

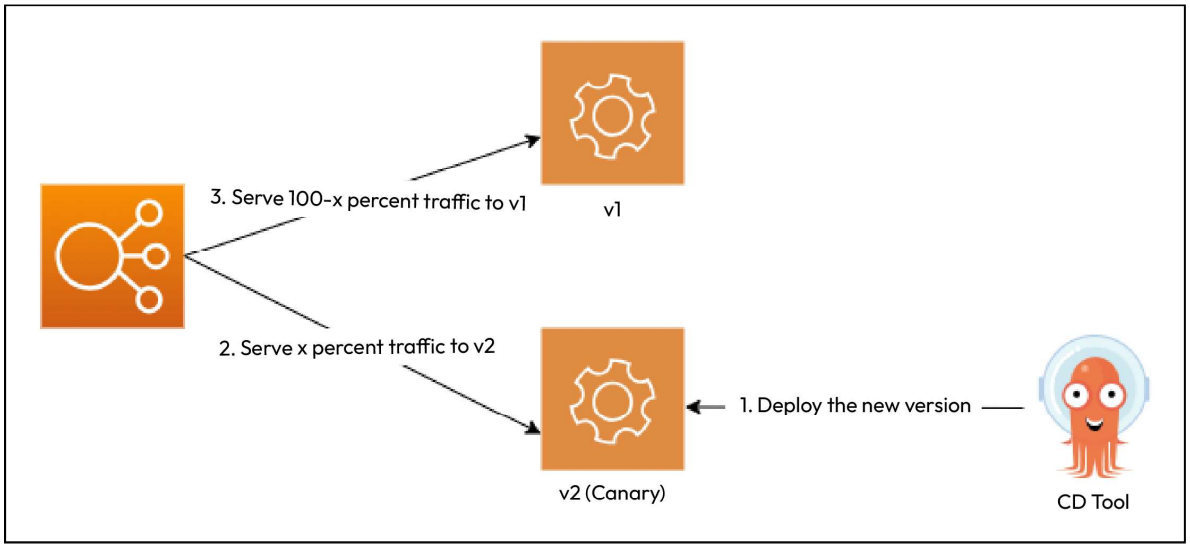

Figure 12.4 – Canary deployments

While complex deployments cause the least disruption to users, they are generally complex to manage using traditional CI tools such as Jenkins. Therefore, we need to get the tooling right on it. Several CD tools are available in the market, including **Argo CD**, **Spinnaker**, **Circle CI**, and **AWS Code Deploy**. As this entire book is focused on GitOps, and Argo CD is a GitOps native tool, for this chapter, we will focus on Argo CD. Before we delve into deploying the application, let's revisit what we want to deploy.

The Blog App and its deployment configuration

Since we discussed the Blog App in the last chapter, let's look at the services and their interactions again:

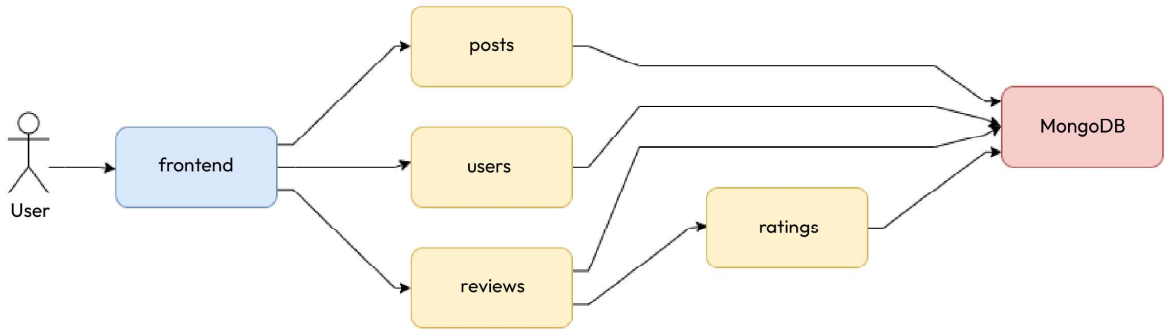

Figure 12.5 – The Blog App and its services and interactions

So far, we've created CI pipelines for building, testing, and pushing our Blog App microservice containers. These microservices need to run somewhere. So, we need an environment for this. We will deploy the application in a **GKE** cluster; for that, we will need a Kubernetes YAML manifest. We built the container for the `posts` microservice as an example in the previous chapter, and I also left building the rest of the services as an exercise for you. Assuming you've built them, we will need the following resources for the application to run seamlessly:

- **MongoDB**: We will deploy an auth-enabled MongoDB database with root credentials. The credentials will be injected via environment variables sourced from a Kubernetes **Secret** resource. We also need to persist our database data, so for that, we need a **PersistentVolume** mounted to the container, which we will provision dynamically using a **PersistentVolumeClaim**. As the container is stateful, we will use a **StatefulSet** to manage it and, therefore, a headless **Service** to expose the database.
- **Posts**, **reviews**, **ratings**, **and users**: The `posts`, `reviews`, `ratings`, `and` `users` microservices will interact with MongoDB through the root credentials injected via environment variables sourced from the same **Secret** as MongoDB. We will deploy them using their respective **Deployment** resources and expose all of them via individual **ClusterIP Services**.
- **Frontend**: The *frontend* microservice does not need to interact with MongoDB, so there will be no interaction with the Secret resource. We will also deploy this service using a **Deployment** resource. As we want to expose the service on the internet, we will create a **LoadBalancer Service** for it.

We can summarize these aspects with the following diagram:

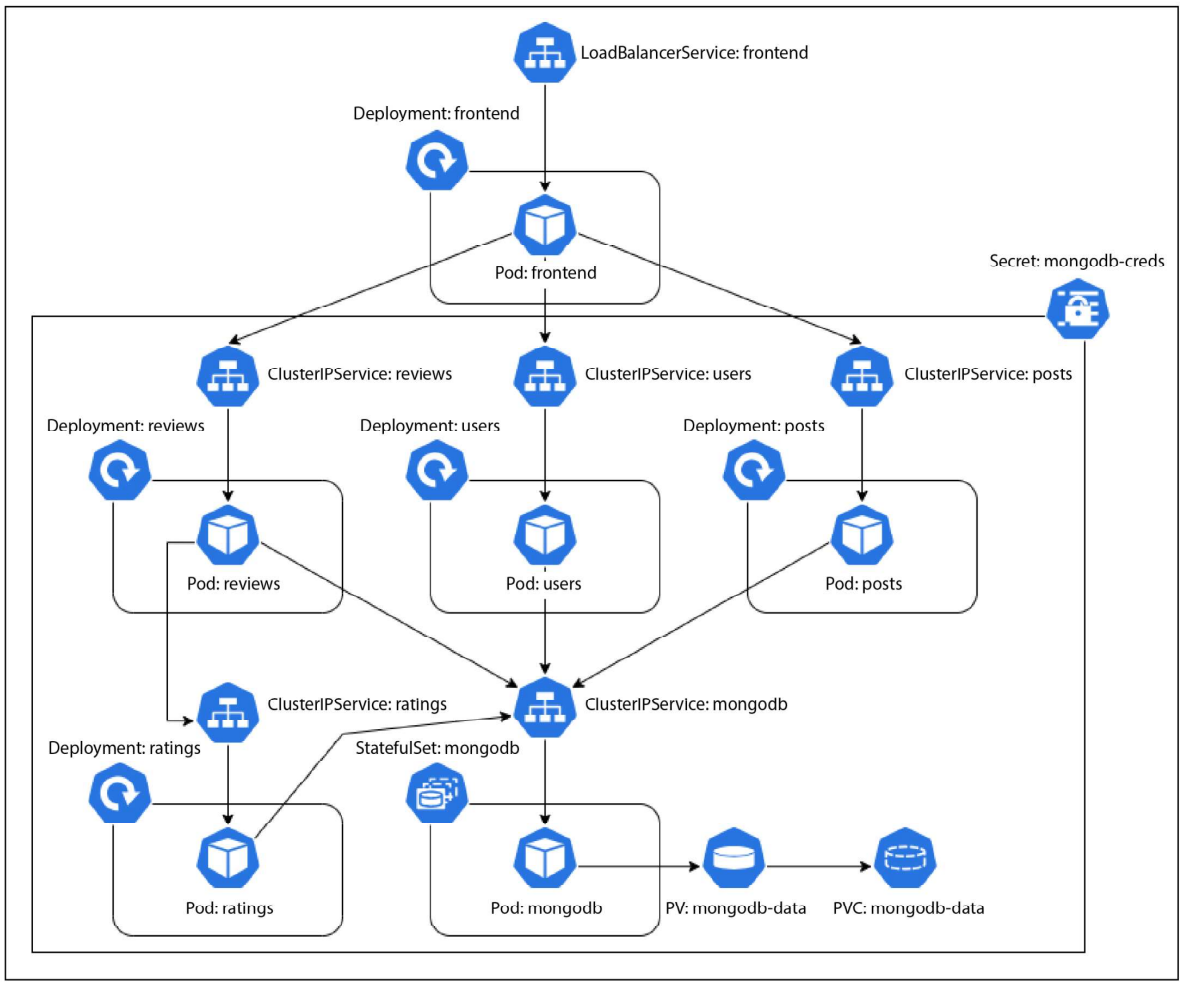

Figure 12.6 – The Blog App – Kubernetes resources and interactions

Now, as we're following the GitOps model, we need to store the manifests of all the resources on Git. However, since Kubernetes Secrets are not inherently secure, we cannot store their manifests directly on Git. Instead, we will use another resource called **SealedSecrets** to manage this securely.

In *Chapter 2*, *Source Code Management with Git and GitOps*, we discussed application and environment repositories forming the fundamental building blocks of GitOps-based CI and CD, respectively. In the previous chapter, we created an application repository on GitHub and used GitHub Actions (and Jenkins) to build, test, and push our application container to Docker Hub. As CD focuses on the Ops part of DevOps, we will need an **Environment repository** to implement it, so let's go ahead and create our Environment repository in the next section.

Continuous declarative IaC using an Environment repository

As we know by now, we must create a GKE cluster to host our microservices. So far, we've been using `gcloud` commands to do this; however, because `gcloud` commands are not declarative, using them is not ideal when implementing GitOps. Instead, we'll use **Terraform** to create the GKE cluster for us. This will ensure we can deploy and manage the cluster declaratively using a Git Environment repository. So, let's go ahead and create one.

Creating and setting up our Environment repository

Navigate to `https://github.com` and create a repository using a name of your choice. For this exercise, we will use `mdo-environments`. Once you have done that, navigate to Google Cloud Shell, generate a `ssh-key` pair using the `ssh-keygen` command, copy the public key to GitHub (refer to *Chapter 2, Source Code Management with Git and GitOps*, for step-by-step instructions), and clone the repository using the following commands:

```
$ cd ~
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ git clone git@github.com:<your_account>/mdo-environments.git
$ cd mdo-environments
```

Let's copy a `.gitignore` file for Terraform to ensure that we do not unexpectedly check in Terraform state, backend, or `.tfvars` files by using the following command:

```
$ cp -r ~/modern-devops/ch12/.gitignore .
```

Now, let's push this code to GitHub using the following commands:

```
$ git add --all
$ git commit -m 'Added gitignore'
$ git push
```

Now that we've pushed our first file and initialized our repository, let's structure our repository according to our environments. We will have two branches within the Environment repository – **dev** and **prod**. All configurations in the **dev** branch will apply to the **development environment**, and those on **prod** will apply to the **production environment**. The following diagram illustrates this approach in detail:

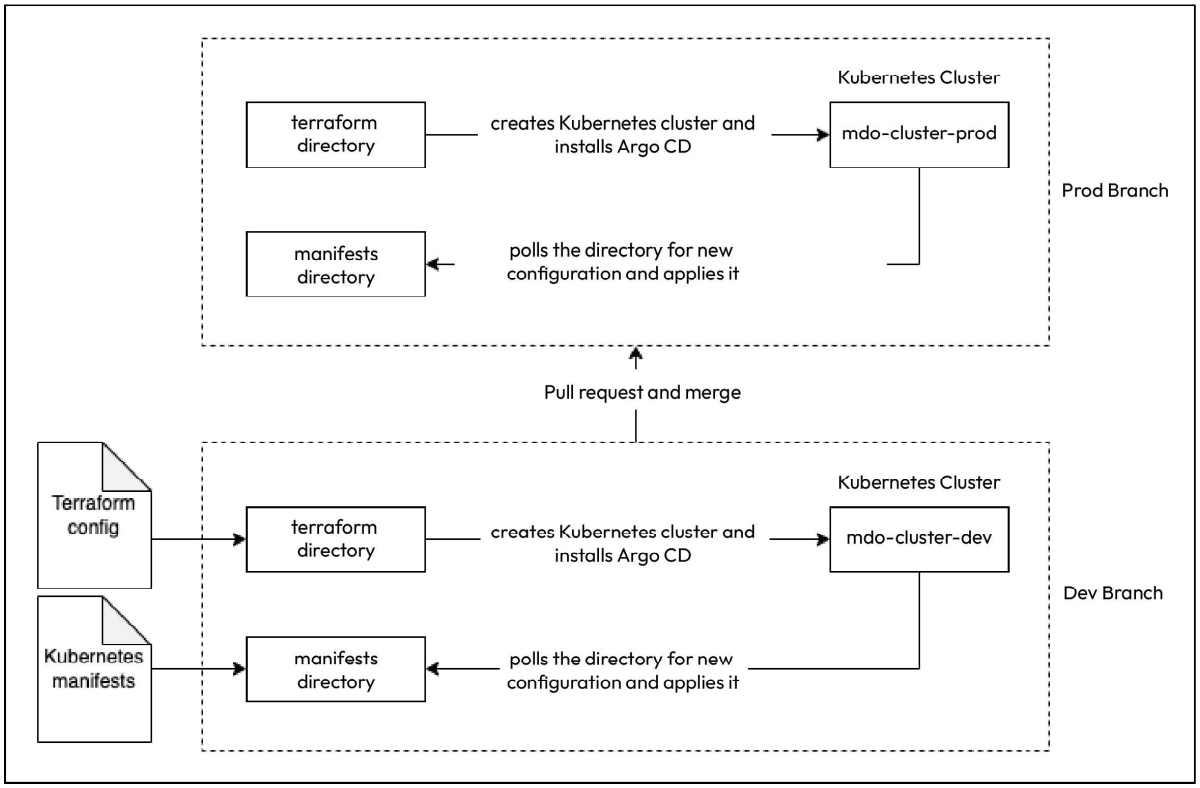

Figure 12.7 – CD process

The existing repository has a single branch called `master`. However, since we will be managing multiple environments in this repository, it would be good to rename the `master` branch to `prod`.

Go to `https://github.com/<your_user>/mdo-environments/branches` and click the pencil icon beside **master**. Type in `prod` and click on **Rename Branch**.

Now that we've renamed the branch, let's remove the existing local repository and clone the repository again using the following commands:

```
$ cd ~ && rm -rf mdo-environments
$ git clone git@github.com:<your_account>/mdo-environments.git
$ cd mdo-environments
```

We want to start with the dev environment, so it will be good to create a branch called `dev` from the `prod` branch. Run the following command to do so:

```
$ git branch dev && git checkout dev
```

Now, we can start writing the Terraform configuration within this directory. The configuration is available in `~/modern-devops/ch12/mdo-environments/environments`. Copy everything from that directory to the current directory using the following commands:

```
$ cp -r ~/modern-devops/ch12/environments/terraform .
$ cp -r ~/modern-devops/ch12/environments/.github .
```

Within the `terraform` directory, there are several Terraform files.

The `cluster.tf` file contains the configuration to create the Kubernetes cluster. It looks like this:

```
resource "google_service_account" "main" {
  account_id   = "gke-${var.cluster_name}-${var.branch}-sa"
  display_name = "GKE Cluster ${var.cluster_name}-${var.branch} Service Account"
}
resource "google_container_cluster" "main" {
  name               = "${var.cluster_name}-${var.branch}"
  location           = var.location
  initial_node_count = 3
  node_config {
    service_account = google_service_account.main.email
    oauth_scopes = [
      "https://www.googleapis.com/auth/cloud-platform"
    ]
  }
  timeouts {
    create = "30m"
    update = "40m"
  }
}
```

It creates two resources – a **service account** and a three-node **GKE instance** that uses the service account with the `cloud platform` **OAuth scope**.

We name the service account with a combination of the `cluster_name` and `branch` variables. This is necessary as we need to distinguish clusters between environments. So, if the cluster name is `mdo-cluster` and the Git branch is `dev`, we will have a service account called `gke-mdo-cluster-dev-sa`. We will use the same naming convention on the GKE cluster. Therefore, the cluster's name would be `mdo-cluster-dev`.

We have a `provider.tf` file that contains the `provider` and `backend` configuration. We're using a remote backend here as we want to persist the Terraform state remotely. In this scenario, we will use a **Google Cloud Storage** (**GCS**) **bucket**. The `provider.tf` file looks like this:

```
provider "google" {
  project     = var.project_id
  region      = "us-central1"
  zone        = "us-central1-c"
}
terraform {
  backend "gcs" {
```

```
    prefix  = "mdo-terraform"
  }
}
```

Here, we've specified our default `region` and `zone` within the `provider` config. Additionally, we've declared the `gcs` backend, which only contains the `prefix` attribute with a value of `mdo-terraform`. We can separate configurations using the prefixes to store multiple Terraform states in a single bucket. We have purposefully not supplied the `bucket` name, which we will do at runtime using `-backend-config` during `terraform init`. The bucket name will be `tf-state-mdo-terraform-<PROJECT_ID>`.

> **Tip**
>
> As GCS buckets should have a globally unique name, it is good to use something such as `tf-state-mdo-terraform-<PROJECT_ID>` as the project ID is globally unique.

We also have the `variables.tf` file, which declares the `project_id`, `branch`, `cluster_name`, and `location` variables, as follows:

```
variable project_id {}
variable branch {...
  default     = "dev"
}
variable cluster_name {...
  default   = "mdo-cluster"
}
variable "location" {...
  default     = "us-central1-a"
}
```

Now that we have the Terraform configuration ready, we need a workflow file that can be applied to our GCP project. For that, we've created the following GitHub Actions workflow file – that is, `.github/workflows/create-cluster.yml`:

```
name: Create Kubernetes Cluster
on: push
jobs:
  deploy-terraform:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./terraform
    steps:
    - uses: actions/checkout@v2
    - name: Install Terraform
      id: install-terraform
      run: wget -O terraform.zip https://releases.hashicorp.com/terraform/1.5.5/
terraform_1.5.5_linux_amd64.zip && unzip terraform.zip && chmod +x terraform && sudo mv
terraform /usr/local/bin
    - name: Apply Terraform
```

```
      id: apply-terraform
      run: terraform init -backend-config="bucket=tf-state-mdo-terraform-${{ secrets.
PROJECT_ID }}" && terraform workspace select ${GITHUB_REF##*/} ||  terraform workspace new
${GITHUB_REF##*/} && terraform apply -auto-approve -var="project_id=${{ secrets.PROJECT_
ID  }}" -var="branch=${GITHUB_REF##*/}"
      env:
        GOOGLE_CREDENTIALS: ${{ secrets.GCP_CREDENTIALS }}
```

This is a two-step build file. The first step installs Terraform, while the second step applies the Terraform configuration. Apart from that, we've specified `./terraform` as the working directory at the global level. Additionally, we're using a few secrets in this file, namely `GCP_CREDENTIALS`, which is the key file of the service account that Terraform uses to authenticate and authorize the GCP API, and the Google Cloud `PROJECT_ID`.

We've also supplied the bucket name as `tf-state-mdo-terraform-${{ secrets.PROJECT_ID }}` to ensure that we have a unique bucket name.

As we've used Terraform workspaces to manage multiple environments, the preceding code selects an existing Terraform workspace with the branch name denoted by `${GITHUB_REF##*/}` or creates a new one. Workspaces are important here as we want to use the same configuration with different variable values for different environments. The Terraform workspaces correspond to environments, and environments correspond to the Git branch. So, as we have the `dev` and `prod` environments, we have the corresponding Terraform workspaces and Git branches.

From the Terraform and workflow configuration, we can deduce that we will need the following:

- A **service account** for Terraform to authenticate and authorize the GCP API and a JSON key file that we need to add as a GitHub secret
- The **project ID** that we'll configure as a GitHub secret
- A **GCS bucket** that we'll use as a backend for Terraform

So, let's go ahead and create a service account within GCP so that Terraform can use it to authenticate and authorize with the Google APIs. Use the following commands to create the service account, provide relevant **Identity and Access Management** (**IAM**) permissions, and download the credentials file:

```
$ PROJECT_ID=<project_id>
$ gcloud iam service-accounts create terraform \
--description="Service Account for terraform" \
--display-name="Terraform"
$ gcloud projects add-iam-policy-binding $PROJECT_ID \
--member="serviceAccount:terraform@$PROJECT_ID.iam.gserviceaccount.com" \
--role="roles/editor"
$ gcloud iam service-accounts keys create key-file \
--iam-account=terraform@$PROJECT_ID.iam.gserviceaccount.com
```

You will see a file called `key-file` within your working directory. Now, navigate to `https://github.com/<your_github_user>/mdo-environments/settings/secrets/actions/new` and create a secret named `GCP_CREDENTIALS`. For the value, print the `key-file` file, copy its contents, and paste it into the **values** field of the GitHub secret.

Next, create another secret, `PROJECT_ID`, and specify your GCP project ID within the **values** field.

The next thing we need to do is create a GCS bucket for Terraform to use as a remote backend. To do this, run the following command:

```
$ gsutil mb gs://tf-state-mdo-terraform-${PROJECT_ID}
```

Additionally, we need to enable the GCP APIs that Terraform will use to create the resources. To do this, run the following command:

```
$ gcloud services enable iam.googleapis.com container.googleapis.com
```

So, now that all the prerequisites have been met, we can push our code to the repository. Run the following commands to do this:

```
$ git add --all
$ git commit -m 'Initial commit'
$ git push --set-upstream origin dev
```

As soon as we push the code, we'll see that the GitHub Actions workflow has been triggered. Soon, the workflow will apply the configuration and create the Kubernetes cluster. This should appear as follows:

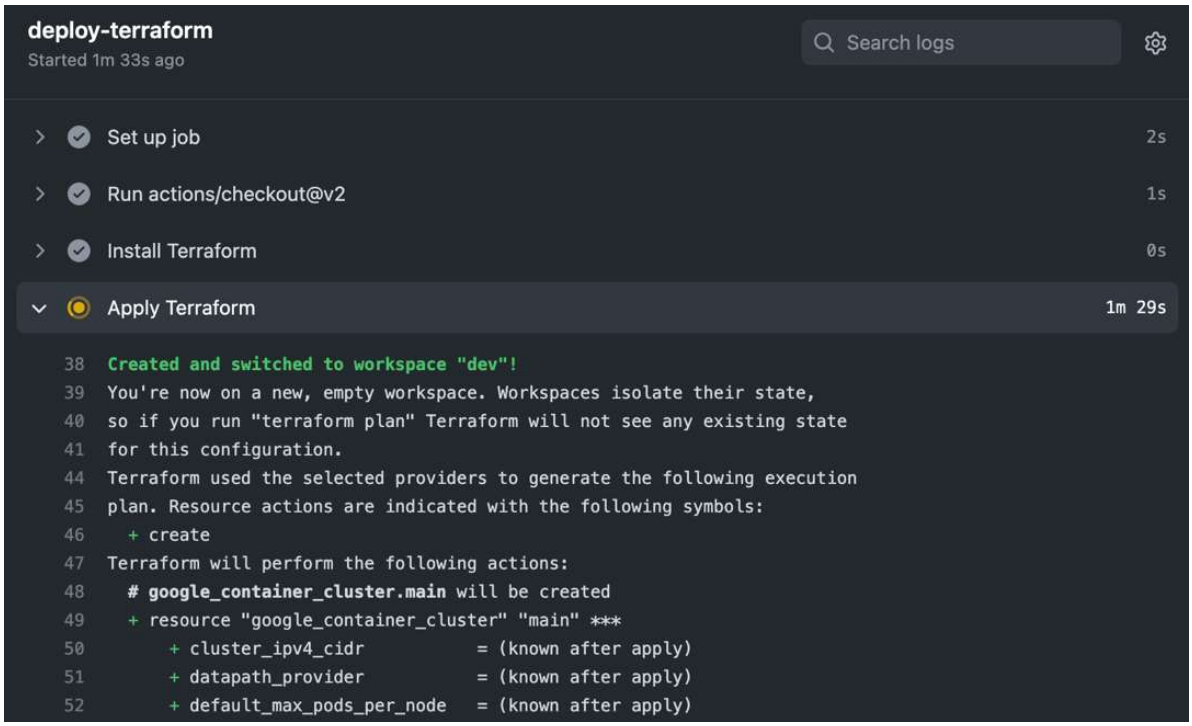

Figure 12.8 – GitOps with GitHub Actions and Terraform

To verify whether the cluster has been created successfully, run the following command:

```
$ gcloud container clusters list
NAME: mdo-cluster-dev
LOCATION: us-central1-a
MASTER_VERSION: 1.27.3-gke.100
MASTER_IP: x.x.x.x
MACHINE_TYPE: e2-medium
NODE_VERSION: 1.27.3-gke.100
NUM_NODES: 3
STATUS: RUNNING
```

As you can see, the `mdo-cluster-dev` cluster is running successfully in the environment. If we make any changes to the Terraform configuration, the changes will automatically be applied. We've successfully created our Environment using an Environment repository. That is the *push model GitOps* in action for you. Now, we need to run our application in the environment; to manage and deploy the application, we will need a dedicated CD tool. As stated previously, we will use Argo CD for this, so let's look at it.

Introduction to Argo CD

Argo CD is an open source, declarative, GitOps-based CD tool designed to automate deploying and managing applications and infrastructure on Kubernetes clusters. Argo CD serves as a robust application controller, efficiently managing and ensuring the smooth and secure operation of your applications. Argo CD works in the *pull-based GitOps model* and, therefore, polls the Environment repository regularly to detect any configuration drift. Suppose it finds any drift between the state in Git and the actual state of applications running in the environment. In that case, it will make corrective changes to reflect the desired configuration declared in the Git repository.

Argo CD is tailored explicitly to Kubernetes environments, making it a popular choice for managing applications on Kubernetes clusters.

In addition to the traditional Kubernetes manifest YAML files, Argo CD offers support for various alternative methods of defining Kubernetes configurations:

- Helm charts
- Kustomize
- Ksonnet
- Jsonnet files
- Plain YAML/JSON manifest files
- Integration with other customized configuration management tools through plugins

Within Argo CD, you can define applications encompassing both a *source* and a *target*. The source specifies details about the associated Git repository, the location of the manifests, helm charts, or kustomize files, and then applies these configurations to designated target environments. This empowers you to monitor changes within a specific branch, tag, or watch particular versions within your Git repository. Diverse tracking strategies are also at your disposal.

You can access a user-friendly web-based UI and a **command-line interface** (**CLI**) to interact with Argo CD. Moreover, Argo CD facilitates reporting on the application's status via sync hooks and app actions. If any modifications are made directly within the cluster that deviate from the GitOps approach, Argo CD can promptly notify your team, perhaps through a Slack channel.

The following diagram provides an overview of the Argo CD architecture:

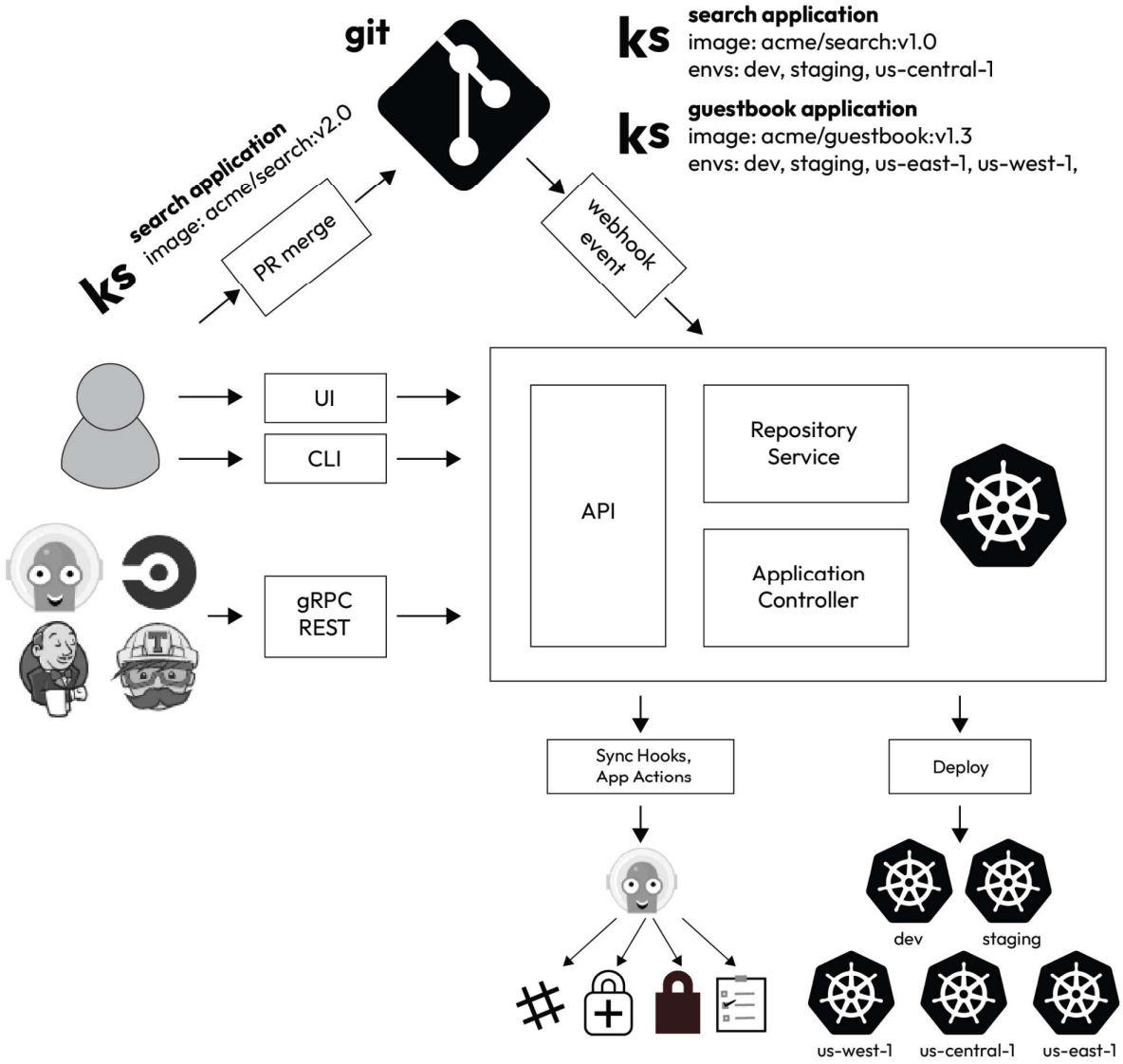

Figure 12.9 – Argo CD architecture

So, without further ado, let's spin up Argo CD.

Installing and setting up Argo CD

Installing Argo CD is simple – we need to apply the `install.yaml` manifest bundle that's available online at `https://github.com/argoproj/argo-cd/blob/master/manifests/install.yaml` on the Kubernetes cluster where we wish to install it. For a more customized installation, you can refer to `https://argo-cd.readthedocs.io/en/stable/operator-manual/installation/`.

As we're using GitOps for this chapter, we will not deploy Argo CD manually. Instead, we will use Terraform to set it up using the Environment repository.

The resources for this section are present in `~/modern-devops/ch12/environments-argocd-app`. We will use the same Environment repository as before for managing this environment.

Therefore, let's `cd` into the `mdo-environments` local repository and run the following commands:

```
$ cd ~/mdo-environments
$ cp -r ~/modern-devops/ch12/environments-argocd-app/terraform .
$ cp -r ~/modern-devops/ch12/environments-argocd-app/manifests .
$ cp -r ~/modern-devops/ch12/environments-argocd-app/.github .
```

Now, let's look at the directory structure to understand what we're doing:

```
.
├── .github
│   └── workflows
│       └── create-cluster.yml
├── manifests
│   └── argocd
│       ├── apps.yaml
│       ├── install.yaml
│       └── namespace.yaml
└── terraform
    ├── app.tf
    ├── argocd.tf
    ├── cluster.tf
    ├── provider.tf
    └── variables.tf
```

As we can see, the structure is similar to before, except for a few changes. Let's look at the Terraform configuration first.