**9**

**DevOps and Deployment Automation**

Automation is a key element of building and managing cloud native applications at scale. Cloud platforms and cloud native architecture patterns have more loosely coupled moving parts than a traditional monolithic application deployed on persistent legacy hardware platforms. Although cloud native architecture provides unprecedented development velocity, resilience, and cost optimization, it also introduces additional operational overhead. Automation is an essential component to overcome this overhead and realize the benefits of a cloud native approach to application development.

Automation systems are not a new concept; many have existed since long before cloud native development was a mainstream idea. Several of these platforms have introduced support for working with cloud platforms and environments such as Kubernetes clusters. The advent of cloud native development has also given rise to a new breed of tools and approaches for continuous integration/continuous deployment (CI/CD) systems that embrace DevOps culture. One example is building infrastructure as code, a type of automation that enables you to use code to express the flows that build, test, deploy, and manage your applications as well as infrastructure. Managing these automation flows as definitions that describe the automation flow in an easy-to-read format is one way to introduce DevOps practices to CI/CD systems. These codified automation flows can be source controlled, making them easy to replicate and to use for tracing changes and making the automation platform itself more resilient to failures. A traditional system might rely on an always active pool of agents that are kept around indefinitely to perform the various tasks of a CI/CD system, such as to compile applications, run tests, and orchestrate deployment jobs. In a traditional environment, this is acceptable because operations teams are working with preprovisioned capacity that is specifically allocated for the CI/CD system. In a cloud-based environment, however, this leads to a massive waste of resources because these resources could have been requested just-in-time, allowing the system to optimize on cost while also being able to scale well beyond the preprovisioned capacity by which a traditional CI/CD system is limited.

Several of the tools that embrace a DevOps approach are themselves cloud native. Some are cloud-based services, such as GitHub actions; others are self-hosted platforms that support cloud native deployment models, such as ArgoCD and Jenkins. Jenkins, one of the most popular CI/CD platforms, has evolved from its on-premises origins to a modern cloud-aware platform. This also puts Jenkins in a unique position to handle non-cloud native pipelines and targets (such as building, testing, and deploying a monolithic application to an on-premises server) just as well as it can handle cloud native pipelines and targets (such as building and pushing container images and deploying microservices to a Kubernetes cluster). Because Jenkins is based on a plug-in model, it can be extended, and new functionality can be added quite easily. A thriving plug-in ecosystem is one of the main advantages of using Jenkins. OCI has plug-ins that help integrate Jenkins with the OCI platform as well. Tools such as ArgoCD and Flux take an opinionated approach to continuous deployment, called GitOps.

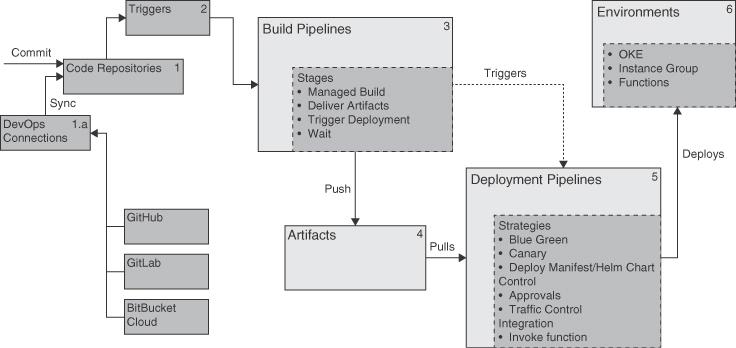
This chapter explores different automation platforms and methods, such as Jenkins, GitOps, ArgoCD, GitHub Actions, and OCI’s native DevOps service, and how they work with OCI.

**OCI DevOps Service**

The OCI DevOps service is a fully managed CI/CD service that helps developers get started quickly and reap the benefits of a DevOps-oriented software development culture using OCI. The service includes everything required for building and delivering software, including managed Git repositories, build pipelines, repositories for build artifacts and Docker images, and sophisticated deployment management tools. As with most other OCI services, the DevOps service provides an open platform. The DevOps service can integrate with a variety of external tools to enable developers to choose the specific features of the service that give them the most benefit, instead of asking them to use the entire platform. For instance, a developer might have an existing Git repository and build tools within the enterprise for compliance reasons, but might use the DevOps service deployment tooling for its capability to integrate with private OKE clusters. This section introduces the DevOps service and examines some of the most common integrations and workflows when building cloud native applications.

The OCI DevOps service relies on several concepts that should be familiar to most developers. Nevertheless, it pays to introduce these concepts and how they relate to each other. This way, you can start to see the service as a set of capabilities to choose from and integrate with your favorite tools and open-source solutions.

The fundamental grouping construct in DevOps is that of a *project*. A project provides a common space for grouping DevOps capabilities that work in concert to build and deliver a related set of applications. The project can group DevOps resources such as code repositories, build pipelines, artifacts, triggers, deployment pipelines, and environments that are related to a set of applications that are managed together, typically by the same team. [Figure 9-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ch09fig01) illustrates the components and overall workflow in the DevOps service:



**Figure 9-1** DevOps Service Components and Workflow

1. Code is committed into a *code repository*. The code repository could be mirrored from an external repository using a DevOps connection.
2. Committing code can *trigger* a *build pipeline*.
3. A *build pipeline* executes a *build specification*.
4. Execution of the build pipeline typically creates *artifacts*.
5. A build can trigger a *deployment pipeline*.
6. The *deployment pipeline* delivers artifacts to a target *environment*.

**Code Repositories**

Code repositories are Git repositories that are hosted and managed by the DevOps service. You can access these repositories using SSH or HTTPS; the access is controlled through OCI Identity and Access Management (IAM). The SSH mode of access uses an SSH key pair that is configured as OCI API keys. Although you can use the same key pair for multiple uses (say, for the CLI as well as Git access), using separate key pairs for groups of resources is typically a good way to isolate and even remove access to specific sets of resources, and it limits the reach of any one key pair.

When using SSH, the username you specify is in the format [federation\_provider/]user\_name@tenancy\_name

The HTTPS mode of access uses Auth Tokens, which are also managed through OCI IAM. When using HTTPS, the username you specify is in the format tenancy\_name/[federation\_provider/]user\_name

In either case, the federation\_provider is optional and used only when using federated identities.

**External Connections and Mirroring Repositories**

In many scenarios, especially when transitioning from an existing source control provider to OCI DevOps, you might want to use the existing repo as the primary repository. In these cases, you can choose to *mirror*, or periodically sync, the commits from the existing repository to the DevOps platform so that you can set up and validate any automation processes without impacting developers’ workflows until you are ready to make a switch.

Mirroring works with both GitLab and GitHub using personal access tokens (PATs). These tokens are generated on the respective platforms and should have the capability to read the repositories; they are stored as secrets in an OCI vault. An *external connection* is a DevOps resource that can connect and access external services such as GitHub or GitLab on behalf of the user.

External connections can be used for build source integration when a build pipeline can use the external repository directly. External connections can be used by repository mirroring as well, which updates your code repository in OCI with commits from an external repository. Repository mirroring uses the external connection resource to query and access external repositories using the personal access token associated with that external connection. The steps to set up mirroring are as follows:

1. Create an access token in GitHub/GitLab.
2. Add it as a secret to an OCI vault.
3. Create an external connection.
4. Create a dynamic group for external connections.
5. Create a policy to grant it read permissions on vault secrets.
6. Choose the external connection for mirroring a repository. The console lists the external repositories available.
7. Choose the repository to mirror, and set the interval for the sync operation.

**Triggers**

As the name suggests, *triggers* in the DevOps service are a way to start build pipelines in response to an event in the source management system, such as pushing new commits to a branch. The most common use for a trigger is to start a build when a new commit is pushed to the repository or when a Pull Request is merged. The type of events that can be used in a trigger depends on the type of the repository. GitHub and GitLab repositories have collaboration flows that use Pull Requests (also known as Merge Requests in GitLab). When using these repositories, triggers can be based on events related to Pull Requests, such as when a Pull Request is opened, updated, merged, or reopened.

**Note**

Deployment triggers are stages in the build pipelines and are similar in naming to triggers within a DevOps project. However, these are stages that the build service supports to trigger external workflows.

**Build Pipelines**

Build pipelines are one of the most central constructs in the DevOps service. At a high level, build pipelines provide a fully managed continuous integration environment within OCI that consists of an orchestration model and a managed environment for building application code into artifacts. Build pipelines also integrate with artifact repositories and deployment tooling.

The build pipeline uses a set of *stages* to describe the build process and to allow the developer to control its flow. A typical build pipeline executes multiple stages, such as running a build tool (which performs the actual build), storing the artifacts (such as application bundles or container images) into an artifact repository, and triggering other processes. Stages in a DevOps pipeline are predefined and represent the various actions, such as compiling source code, running vulnerability checks, and delivering artifacts that form the overall build process. Stages can be executed in sequence or in parallel. For instance, if a single source repository contains the source code for multiple microservices, you can make the build process more efficient by building the microservices in parallel. In such cases, the DevOps pipeline can perform complex orchestrations, such as performing multiple builds in parallel, storing the individual artifacts, and handling multiple deployments.

Build pipelines provide the following stages:

1. Managed Build
2. Deliver Artifacts
3. Trigger Deployment
4. Wait

Of these, the *Managed Build* stage is the heart of the build pipeline. This stage provides a managed build environment, an ephemeral compute instance for running builds—a build runner. The build process for running a build is described as code that uses a build specification that you commit to your source repository. The Managed Build stage is simply pointed to the repository and told where to expect the build specification; the service does the rest. The build specification describes the build environment and the build process in a sequence of steps that is executed by the build runner.

The other stages provide supporting control flow, such as delivering the artifacts generated by the Managed Build stage to appropriate repositories. If the managed build generates a container image, the *Deliver Artifacts* stage can move that image in to OCIR; similarly, if the managed build produces a language-dependent artifact, such as a .jar file or similar generic artifact, it can deliver those to the Artifact Registry.

The *Trigger Deploymen*t stage can trigger a deployment pipeline, typically after a managed build has run and the artifacts it created have been delivered to a repository.

The *Wait* stage pauses the pipeline for a specific duration. This is useful when interacting with external systems that might take a few moments to process an action. For instance, an artifact repository might want to run a vulnerability scan on every new artifact before allowing users to pull that artifact from the repository.

**Understanding the**build\_spec**Structure**

The build specification describes how a build runner should run a build. It is a YAML document that describes how the build runner should be configured and what steps should be run by the build, in what order. The default name for the build specification is either build\_spec.yaml or build\_spec.yml, and the managed build looks for it in the root of the source repository. If the file is located in a nonstandard location, the Managed Build stage configuration can specify a relative path to it. When a Managed Build stage is executed, the service performs the following sequence of actions:

1. Provisions a build runner. An ephemeral instance is provisioned to perform the build.
2. Sets up the build environment. The build runners are configured with tools and runtimes, such as the JDK or Android runtimes, source control utilities, CLIs, and tools such as gradle, Docker, and the OCI CLI.
3. Downloads the source code onto the build runner.
4. Locates, parses, and validates the build\_spec.yaml file.
5. Executes the build\_spec configuration, including handling the environment setup, downloading input artifacts, performing the build steps, and saving the output artifacts.

If multiple artifacts must be built from the same source repository, one strategy to consider is to use multiple build spec files that can be run in parallel stages within the build pipeline. Parallel managed build steps are run on separate build runners. This typically speeds up the overall build process for when multiple artifacts need to be built. The builds can be executed in isolation, without depending on artifacts from each other. [Listing 9-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#list9_1) shows a sample build\_spec.yaml file.

**Listing 9-1** Sample build\_spec.yaml File

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0362-01a)

# Metadata section

version: 0.1

component: build

timeoutInSeconds: 6000

runAs: root

shell: bash

# Environment section

env:

# these are local variables to the build config

variables:

key: "value"

# the value of a vaultVariable is the secret-id (in OCI ID format) stored in the

OCI Vault service

# you can then access the value of that secret in your build\_spec.yaml commands

vaultVariables:

# exportedVariables are made available to use as parameters in successor Build

Pipeline stages

# For this Build to run, the Build Pipeline needs to have a BUILDRUN\_HASH

parameter set

exportedVariables:

- BUILDRUN\_HASH

steps:

- type: Command

name: "Export variables"

timeoutInSeconds: 40

command: |

export BUILDRUN\_HASH='echo ${OCI\_BUILD\_RUN\_ID} | rev | cut -c 1-7'

echo "BUILDRUN\_HASH: " $BUILDRUN\_HASH

uname -a

docker --version

onFailure:

- type: Command

timeoutInSeconds: 40

command: |

echo "Handling Failure"

echo "Failure successfully handled"

timeoutInSeconds: 400

runAs: root

- type: Command

timeoutInSeconds: 1200

name: "Build container image"

command: |

cd ${OCI\_PRIMARY\_SOURCE\_DIR}

docker build -t demo-hugo-site:${BUILDRUN\_HASH} -f Dockerfile .

onFailure:

- type: Command

command: |

echo "Handling Failure"

echo "Failure successfully handled"

timeoutInSeconds: 60

runAs: root

outputArtifacts:

- name: output01

type: DOCKER\_IMAGE

location: <region>.ocir.io/<tenancy\_namespace>/demo-site:${BUILDRUN\_HASH}

The example shows that the build spec is divided into multiple sections. The first section shows metadata about the build spec itself. The version indicates the version of the build spec used; at the time of writing, the only supported value is 0.1. The component indicates the kind of spec file this is; for build spec files, the only applicable value is build, as shown. The default shell used for running builds on a build runner is bash, and this can be overridden to use sh with the shell attribute. Build runners do not support sudo; for scenarios that need superuser privileges, the runAs attribute can be set to root to execute either the whole build or an individual step as the root user. The timeoutInSeconds sets the timeout for steps at the build scope. If the timeout is not specified, the implicit default is 8 hours, which is also the maximum value allowed for timeouts.

The env sections hold environment variables that can be used during the build. env can have three types of variables, named variables, vaultVariables, and exportedVariables. variables are key-value pairs that are declared and whose values can be updated by any build step. vaultVariables are OCI vault secrets that can be used during the build process. The typical purpose for these is to reference a password or an auth token used to log into a private OCIR registry so that images that are built by a build pipeline can be uploaded to the registry. In the build spec, the value of this variable is an OCID that represents the vault secret. When the build runner executes the build, it fetches the actual secret and provides it to all the build steps that reference this variable. This avoids having to create build\_spec.yaml files that embed secrets. exportedVariables are a list of variables that are declared and whose value can be set in any stage of the build. A value set for the variable is available in all subsequent steps of the build spec. In the example, BUILDRUN\_HASH is an exported variable whose value is set in the first step. It is then used in the subsequent step to tag the container image that was built.

The steps section of the build spec specifies the commands that are to be sequentially executed for running the build. The command can be a multiline command or a single-line command, and each command can be accompanied by a timeout value that overrides the timeout for this step from the one set at the build spec scope. The name is a descriptive field that can be used to trace the progress of the build or to troubleshoot the build using logs. Every step can also have an optional onFailure attribute that is also of type Command, which runs the specified command to perform cleanup or otherwise gracefully exit the build step in the event of a failure. The runAs and timeoutInSeconds values can be overridden at each step from the build scoped values set in the metadata section.

The outputArtifacts identifies the artifacts produced by the build. Artifacts can be of either BINARY or DOCKER\_IMAGE type. For a BINARY artifact, the location points to where the artifact can be found. If the artifact cannot be found in the specified location, the build fails. If the artifact is of type DOCKER\_IMAGE, the image needs to have been either built or pulled by one of the build steps, or the build fails. The location attribute for an artifact of type DOCKER\_IMAGE is the image name and tag.

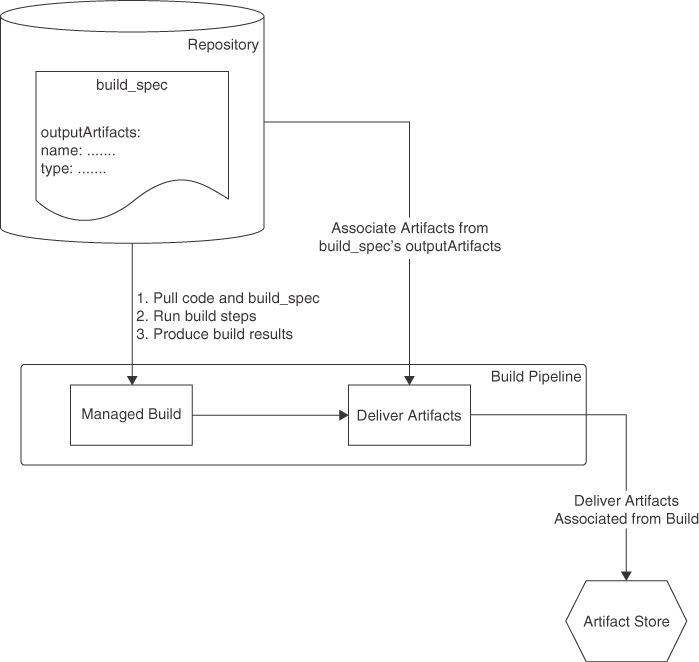
**Artifacts**

Artifacts in DevOps are resources that identify entities that can be deployed. Artifacts are typically created by a build pipeline as its output. However, they can also exist independently and be created or managed outside a build pipeline because the build and deploy aspects of the DevOps service can be used independently. This is usually the case for artifacts that are created by external build tools, which can be represented in the deployment service using the artifact resource. Likewise, the artifacts that are produced by the build pipelines can be represented by the artifact resource, and their deployment can be handled using external tools.

Artifacts can be of various types, including a container image in OCIR, a generic artifact such as a .jar file that is stored in the Artifact Registry, Kubernetes deployment manifests, or Helm charts. For most cloud native applications, developers will be building their applications as container images and deploying to platforms such as OKE using a Kubernetes manifest or a Helm chart.

To use the OCIR container image repository with a build pipeline, the user first creates an *artifact* that represents this image repository. This creates the artifact resource that the DevOps build pipeline can use to interact with the container image repository to push the image that the build generates. The build service includes a stage called *Deliver Artifacts* that can be used after the Managed Build stage to deliver the artifacts created from a managed build to an artifact repository. The Deliver Artifacts stage maps the outputArtifacts identified in the build spec with artifact resources in the DevOps service. This mapping ensures that the artifacts created in the build are propagated and delivered to the appropriate artifact stores.

[Figure 9-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ch09fig02) shows a build pipeline with a Managed Build stage that uses a build spec from a source repository. The Managed Build stage uses the build spec to execute the various steps in the build and produces artifacts such as a container image. The outputArtifacts section of the build spec identifies the artifacts that are created by the build. The subsequent Deliver Artifacts stage associates the artifacts identified in the outputArtifacts section of the build to a DevOps artifact resource. The stage then delivers the artifact (such as the container image) to the artifact store (such as OCI Container Image Registry).



**Figure 9-2** A Build Pipeline with a Deliver Artifacts Stage That Maps the Results of a Build to Artifact Resources and Delivers the Artifacts to the Artifact Store

Artifact references are most used by deployment pipelines to represent and identify the workload that is to be deployed to a target platform. Helm charts and Kubernetes manifests are typical examples of artifacts that are not generated by a build process; instead, they are stored either as an inline artifact (Kubernetes manifest) or in OCIR (Helm chart). These artifacts are referenced by deployment pipelines during deployment.

**Environments**

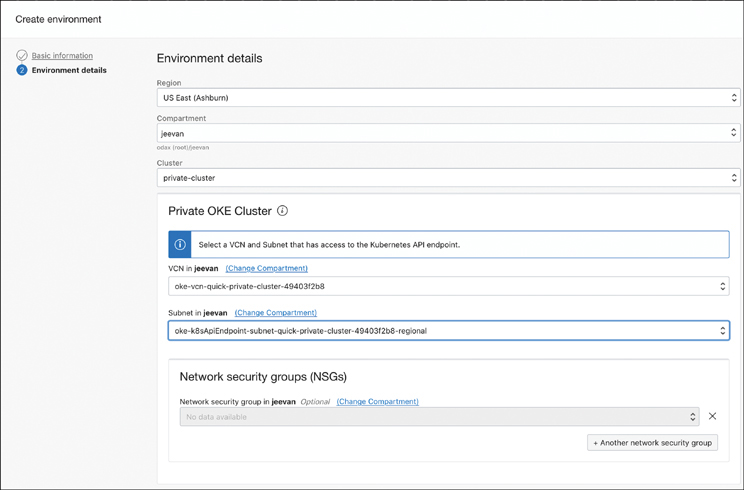
Environments are resources in DevOps that represent the target platforms or execution environments for deploying your artifacts. They identify a target platform, such as an OKE cluster, and the environment reference is used to interact with these target platforms when deploying artifacts to them. The DevOps service supports multiple OCI services, such as OKE, Compute Instance Groups, and OCI Functions, to be represented as environment resources. The kind of artifact is related to the kind of environment in which artifacts have a natural deployment target type. For instance, a Kubernetes manifest or a Helm chart is always deployed to a Kubernetes cluster, whereas an instance group deployment configuration is targeted at a compute instance group.

The environment reference is most used from the deployment pipeline, as a way to identify an execution environment for a deployment. The target environment can be in any region that the tenancy is subscribed to. This enables the DevOps service to roll out changes to global/multiregion applications with ease and precision.

**Deployment Pipelines**

A deployment pipeline is the feature that provides the continuous deployment capability in the DevOps service. Deployment pipelines can be used to construct deployment workflows that push artifacts onto environments. Like the build pipelines, deployment pipelines consist of stages that can be run serially or in parallel. A deployment pipeline’s stages can be categorized as stages that perform workflow control, perform integrations, or do deployments to target environments. Each group offers various strategies that can be used to form a complete workflow. For instance, control stages provide control flow in the deployment process, such as when getting approvals for deployments or performing traffic shift between environments. Deployment pipelines natively support advanced deployment strategies such as blue-green deployments or canary deployments out of the box. This native support for these deployment models makes it easy to implement them in application deployment workflows using deployment pipelines. The build pipeline and deployment pipeline can be used independently or with each other. They are loosely coupled, to allow developers to use their proffered tools for each job. For instance, an enterprise might mandate that its builds be done on existing on-premises environments and tooling. A build pipeline also offers integration between build and deployment pipelines, allowing variable export and triggering deployments on successful builds.

In a cloud native environment, the most common deployment environment is an OKE cluster. For a better security posture, OKE clusters are usually configured with private API endpoints that are accessible only from within the tenancy or from an on-premises network that is peered with the VCN. This limited visibility for the Kubernetes API endpoints can add configuration steps to the deployment tool that needs to access these APIs. Deployment pipelines can also easily deploy to private OKE clusters. To set up deployments to private OKE clusters, at the time of creating the OKE environment, select the VCN and the subnet where the Kubernetes API endpoint has been created (see [Figure 9-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ch09fig03)). Access can further be controlled using the security lists or NSGs on the subnet.



**Figure 9-3** Setting Up Deployments to Private OKE Clusters

**Understanding Deployment Strategies**

Consider an application that is deployed to OKE. The deployment process applies the application definition, which could be a set of Kubernetes manifests, to the OKE cluster. In this process, the build system is not involved, and it is assumed that the container images that are required for the application are already built and available at the repository URL that is referenced in the manifest files. These images could have been built by the DevOps build pipeline, or they could have been built by other tools, perhaps on an on-premises build system. How the application was built and packaged does not matter to the deployment tooling because the deployable artifacts here are either a set of Kubernetes manifests or a Helm chart. These deployable artifacts can reference the container images needed to run the application.

The deployment offers three kinds of deployment strategies for OKE-based environments:

* In-place deployments
* Blue/green deployments
* Canary deployments

*In-place deployments* are the most basic: This type of deployment simply amounts to redeploying an application over the existing one. In-place deployments are optimal, in terms of resource usage, but they also afford lesser control over the rollout (and potential rollback) of the changes because the newly deployed changes are made available to all users at the same time.

A *blue-green deployment model* maintains two identical environments. One is considered the blue (live traffic) environment, and the other is the green (no live traffic) environment. New versions are deployed to the green environment, and regression tests and sanity checks are performed. When everything looks good, the traffic is switched from the blue environment to the green environment; the green environment is then considered to be the new blue environment. Although this strategy requires more resources (a second environment), it affords much better control over the transition. If the sanity checks fail, the traffic switch does not have to happen. If an issue is detected, even after the traffic has switched, a rollback process is simple and just switches the traffic back to the old environment. This strategy is more suited to applications that are highly sensitive to disruptions.

Finally, *canary deployments* offer a strategy to gradually move traffic from one version to another. When deployed to OKE environments, the canary and blue-green strategies use namespaces within the same cluster. They also rely on an ingress resource, for which a NGINX ingress controller needs to be installed in the cluster. In OKE-based environments, the primary difference in the way these two strategies operate is how the traffic is shifted from the old deployment to the new deployment. In the blue-green strategy, the traffic is switched from one environment to another, based on validation and approval, and all the traffic is switched. This means that users experience either the old application or the new one.

The canary strategy, by contrast, temporarily shifts a specific percentage of traffic from the production version to the new version until approval to deploy to production is received. Once the approval to deploy to production is received, the production deployment is replaced and all traffic then is restored to the production deployment. While the canary version is deployed and waiting to be approved for production deployment, users can experience both versions of the application.

**Setting Up a Canary Deployment**

Performing a canary deployment to an OKE cluster is one of the most common tasks for developers to carry out when working with deployment pipelines. To get started, first deploy the NGINX ingress controller on to the OKE cluster using the Kubernetes manifests that the project publishes:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0369-01a)

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/

controller-v1.2.0/deploy/static/provider/cloud/deploy.yaml

**Note**

The installation instructions for the latest version can be found at <https://kubernetes.github.io/ingress-nginx>.

This creates the ingress-nginx namespace and installs the NGINX ingress controller. The ingress controller uses a LoadBalancer to expose the ingress resources externally. Its IP address can be retrieved from the service object in the ingress-nginx namespace using the following command:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0369-02a)

kubectl get svc/ingress-nginx-controller -n ingress-nginx

Next, create an artifact. The artifact in [Listing 9-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#list9_2) deploys an Apache Tomcat container with six replicas. This Kubernetes deployment can be exposed to other resources using a ClusterIP service. Finally, an ingress resource exposes the service externally under the path /tomcat using the NGINX ingress controller. This can be created as an inline artifact.

**Listing 9-2** An Example Apache Tomcat Deployment

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0369-03a)

apiVersion: apps/v1

kind: Deployment

metadata:

name: tomcat

labels:

app: tomcat

spec:

replicas: 6

selector:

matchLabels:

app: tomcat

template:

metadata:

labels:

app: tomcat

spec:

containers:

- name: tomcat

image: tomcat:9

ports:

- containerPort: 8080

---

apiVersion: v1

kind: Service

metadata:

name: tomcat-service

labels:

app: tomcat

spec:

ports:

- port: 80

name: http

targetPort: 8080

selector:

app: tomcat

type: ClusterIP

---

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: ingress-tomcat

spec:

rules:

- http:

paths:

- path: /tomcat

pathType: Prefix

backend:

service:

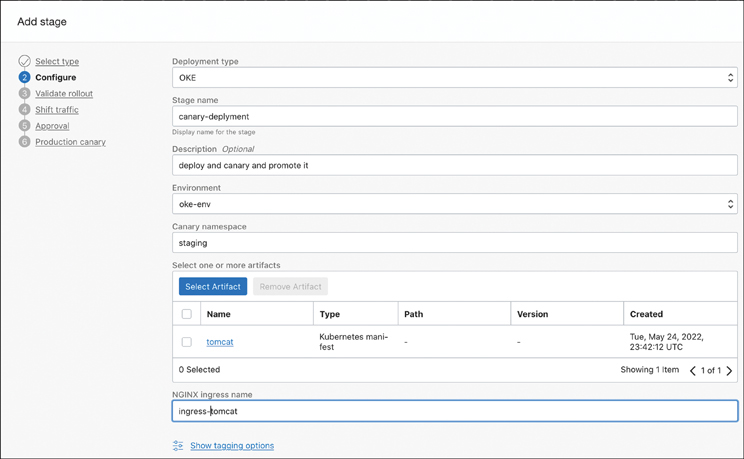
name: tomcat-service

port:

number: 80

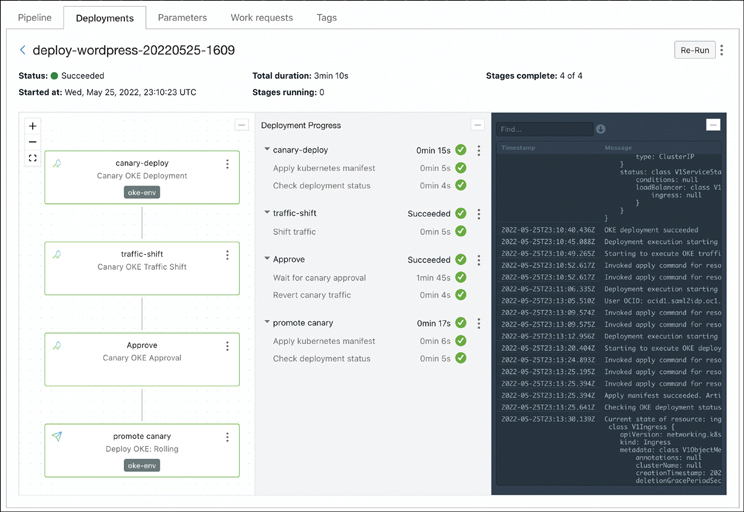
ingressClassName: nginx

Next, add a deployment pipeline and add a canary deployment stage to the pipeline. The canary deployment stage prompts for the environment reference, the artifact reference, and the canary namespace, as shown in [Figure 9-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ch09fig04).



**Figure 9-4** Adding a Canary Deployment Stage to the Pipeline

Optionally, you can validate the deployment using a function that can run tests against the new version and return a true or false response to the deployment pipeline. You can also see the traffic limits for the canary. This is the percentage of traffic the canary deployment will receive; it can range from 1% to 25% of the traffic. You can also set the number of approvals required to perform the production deployment. When the required approvals have been received, the deployment is made to the production namespace of your choice in the cluster. A basic canary deployment with all its steps looks like [Figure 9-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ch09fig05).



**Figure 9-5** Steps of a Basic Canary Deployment

When the deployment pipeline runs, it deploys the artifact (the manifest) to the canary namespace. It then performs the traffic shift step, which annotates the NGINX ingress with canary annotations that direct a percentage of traffic to the canary deployment. The ingress resource for the canary deployment would be annotated as shown in the following example. These annotations cause the ingress controller to ensure that only the predefined percentage of traffic is sent to the canary deployment:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0372-01a)

nginx.ingress.kubernetes.io/canary: true

nginx.ingress.kubernetes.io/canary-by-header: redirect-to-canary

nginx.ingress.kubernetes.io/canary-weight: 25

The deployment process blocks and waits on the approval stage. When the approval has been received, the deployment pipeline resumes by reverting the traffic shift and promoting the canary deployment to production. The blue-green strategy also works in a similar fashion: The artifacts described in the previous listings can be used to replace the canary build stage with a blue-green deployment that deploys the new version of the application to a separate namespace in Kubernetes and switches traffic after an optional verification step.

**Elastically Scaling Jenkins on Kubernetes**

Jenkins[**1**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ref9_1a) is probably the most well-known CI/CD server today. The long history of Jenkins as an active open-source project has led it to continuously evolve and integrate new technologies such as Kubernetes. Jenkins uses a plug-in model for extensions, and its plug-in ecosystem boasts a vibrant community that is continuously adding and extending capabilities. Some Jenkins plug-ins, such as BlueOcean, offer an opinionated CI/CD workflow with a simplified user interface; others, such as the Kubernetes plug-in, fundamentally change how Jenkins operates. Jenkins itself can be deployed on to a Kubernetes cluster, for better management and elasticity.

At a high level, Jenkins operates on the concept of having a controller and having multiple agents. This is similar to Kubernetes itself, in some respects. The controller in Jenkins manages the various agents, monitors them, and schedules jobs on the agents under its management. The agents in Jenkins are the worker nodes that perform jobs. Continuous delivery in Jenkins is typically done using Jenkins pipelines. The pipeline is the definition of the steps that are needed for code in a code repository to be built into software packages and for those packages to be delivered to your end users. Jenkins uses a text file named Jenkinsfile to express these steps and is typically source controlled alongside the application source. When a build job is started, the steps in the Jenkinsfile are executed on an agent to which the controller assigns the job.

In traditional on-premises environments, Jenkins uses a system in which a set of controllers is configured with a predefined set of Jenkins agents. The agents are either bare metal machines or VMs that are preconfigured for certain use. They can be preconfigured in a homogeneous way, in which an enterprise has a baseline configuration for all build jobs; they also can have specific configurations or software packages, such as compilers or other build tools, preinstalled on them. These static installations usually do not get a lot of utilization because an agent that has been configured for a specific project with specialized tooling will not typically be used 24×7 unless it is a very high velocity project. Keeping track of the configuration on the agents can also be cumbersome over time.

Jenkins does not require agents to be static. An agent can be a bare metal machine, a virtual machine, or a container. This can be a machine on-premises, or it can be in the cloud. The only real requirement for an agent is that it must be any type of compute that can run Java. This is because the component that runs on the agent and communicates with the controller requires Java. Applying the cloud native best practices and principles to the Jenkins model and leveraging the plug-in model in Jenkins, the Kubernetes plug-in allows Jenkins to operate using Kubernetes primitives.

**Note**

Oracle has made an OCI plug-in available that can provision OCI compute instances of any shape and dynamically connect them as agents to a Jenkins controller.

**Setting Up Jenkins on OKE**

Setting up a scalable Jenkins environment on OKE is as easy as installing the Jenkins Helm chart. The command in [Listing 9-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#list9_3) adds the Jenkins Helm chart repository and installs Jenkins in a namespace called jenkins; then it overrides the serviceType and servicePort to use a load balancer listening on the default HTTP port. Running the commands displays further commands to be run that show the default generated password for the default user admin, how to identify the load balancer IP address, and how to access the Jenkins login page.

**Listing 9-3** Adding the Jenkins Helm Repo and Configuring the Namespace serviceType and servicePort

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0373-01a)

helm repo add jenkins https://charts.jenkins.io

helm repo update

helm install jenkins jenkins/jenkins -n jenkins --create-namespace \

--set controller.serviceType=LoadBalancer \

--set controller.servicePort=80

**Note**

This installation is a good starting point for exploring how to run Jenkins on OKE. The Jenkins Helm chart gives the developer a lot of control. Developers should refer to the official documentation to implement security and configuration practices that are suited for production use.

The Helm chart installs the Kubernetes plug-in and configures it for using the same Kubernetes cluster where you are installing Jenkins. The Kubernetes plug-in enables a Jenkins server to create agents that are pods. As pods, these “agents” are created when they are needed and discarded when the job is completed. The Kubernetes plug-in has a configuration that points it to a target Kubernetes cluster where new pods will be created to serve as Jenkins agents. Each project defines the agent configuration as a pod template in the Jenkinsfile that is associated with that project. When the Jenkins pipeline executes, a new pod is created in the Kubernetes cluster using the configuration in the Jenkinsfile. This configuration can include the specialized tools required for each project. For instance, one project could be using Java 11 and another could be using Java 15, and you no longer must have agents that are preconfigured with these tool chains always on standby. Instead, the agent configuration is expressed as a pod that can be based on container images that have the required tooling. [Listing 9-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#list9_4) shows an example of this configuration in a Jenkinsfile.

**Listing 9-4** Example Pod Template in a Jenkinsfile

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0374-01a)

podTemplate(yaml: '''

apiVersion: v1

kind: Pod

spec:

containers:

- name: gradle

image: gradle:7-jdk11

command:

- sleep

args:

- 99d

''') {

node(POD\_LABEL) {

stage('Clone repo') {

git 'https://github.com/oracle-quickstart/oci-cloudnative.git'

container('gradle') {

stage('Clean'){

sh 'cd src/orders && gradle clean '

}

stage('Build app') {

sh 'cd src/orders && gradle compileJava '

}

stage('Run Tests') {

sh 'cd src/orders && gradle test '

}

stage('package app') {

sh 'cd src/orders && gradle bootJar'

}

}

}

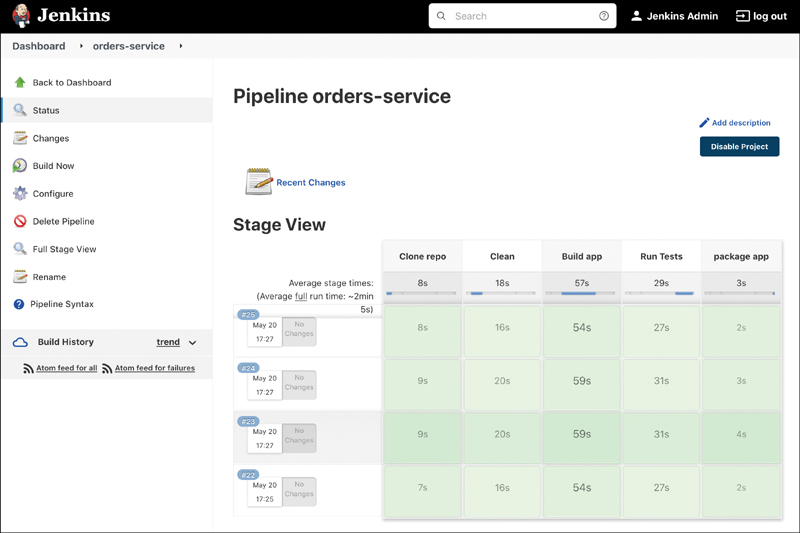
}

}

**Note**

The build stages are split into separate steps, to demonstrate nested stages. In most builds, this can be a single nested step.

The podTemplate defines a pod for running as an agent. The example in [Listing 9-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#list9_4) shows a single container that uses the gradle:7-jdk11 image. The pod definition can have multiple containers as well. The Kubernetes plug-in always has one container named jnlp in the pod; that container runs the Jenkins JNLP agent service that connects to the Jenkins controller and registers the pod as an agent. The first stage clones the Git repo for the application source code. The container('gradle') construct selects the container named gradle from the podTemplate to run its nested stages. The next few stages run inside the gradle container; they run the build and unit tests before packaging the application as a JAR file. [Figure 9-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ch09fig06) shows the pipeline execution in Jenkins.



**Figure 9-6** Example Jenkins Pipeline Execution

This means that each project can define its build environment and use tools and configurations that are unique to it without having to maintain a large fleet of agents. Because these dependencies are also described as code in the Jenkinsfile that is source controlled, developers can easily implement changes to the build environment and tools with the complete traceability that the source control system provides. As the job is queued in Jenkins, perhaps because of a new commit in the source control, the Kubernetes plug-in spawns a pod as an agent and executes the pipeline on that pod. When the job is complete, the pod or that agent is terminated. In this way, Jenkins can truly become elastic, running on a smaller fleet with better utilization rates and using the underlying capabilities of Kubernetes. It also opens the doors to using Kubernetes-based scaling (such as cluster autoscaling) to grow and shrink the Jenkins fleet when required.

**GitOps with ArgoCD**

Developer workflows are centered on source control systems (usually Git) for application code and infrastructure expressed as code. GitOps is a set of practices that expand on developers’ Git-based workflows to provide automated workflows for applications as well as infrastructure. This enables every change in the system, application, or infrastructure to be traced back to a Git commit. Developers can keep track of changes and known good configurations while simplifying and standardizing the workflows for both infrastructure and application changes. Ops and support teams get better visibility into changes and can quickly re-create configurations to help troubleshoot issues. This makes deployments and rollbacks clear and predictable.

With GitOps, a desired configuration is described as code in the Git repository. This could be an application expressed as a Kubernetes manifest or infrastructure expressed as Terraform code. Changes to this configuration are detected by a tool such as ArgoCD[**2**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ref9_2a) or Flux,[**3**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ref9_3a) which can trigger a synchronization or reconciliation of the target environment to the new definition. This enables you to build a completely Git-based workflow using existing practices that developers are accustomed to. For instance, opening a pull request can be detected, and the result could be deployed to a test environment to have automated tests run on it, to ensure that the merge of the pull request will not cause regressions. A merge of that pull request then could deploy the now-tested code to the production environment.

Kubernetes can express applications and infrastructure through Kubernetes manifest files. This makes it an ideal candidate to be operated using the principles of GitOps. Kubernetes is not a hard requirement for implementing a GitOps model, but it makes for a natural fit. Other tools can implement GitOps as well. For instance, automation frameworks such as Ansible and Terraform can also be used to achieve GitOps principles. Regardless of the tools used or the platforms targeted by the deployment process, the essential concept in GitOps is that the Git repository acts as the single source of truth for configurations. Changes to these configurations are propagated to target environments by tools that interface with the Git repository and leverage Git-based workflows.

We chose to show how to deploy ArgoCD onto an OKE cluster and create a GitOps pipeline to describe how you can achieve GitOps practices in OCI. The choice to use ArgoCD was arbitrary, and other tools (such as FluxCD) are equally applicable. Tools such as ArgoCD have one fundamental difference in how they operate, when compared to other tools in the space, such as Jenkins or GitLab: ArgoCD is Kubernetes focused, has Kubernetes-specific workflows, and understands Kubernetes objects natively. Tools such as ArgoCD are typically deployed on the Kubernetes cluster and work on the principle of pulling changes into a cluster. This is contrary to general-purpose tools such as Jenkins or GitLab that can exist outside the cluster as an independent system and push changes onto one or more target clusters when changes are detected in the Git repository.

ArgoCD itself runs on the Kubernetes cluster. The application controller in ArgoCD continuously monitors running applications and compares the live application state against the desired application state that is defined in the Git repository. This is what lets ArgoCD act when the configuration on the cluster diverges from what is in the Git repository. If a new configuration change has been pushed to the Git repository, ArgoCD detects that there is a difference between the desired state in Git and the current state in the cluster and then acts. It works in the reverse direction as well. Because the configuration contained in the Git repository is the source of truth, any changes on the cluster that Argo CD detects are a deviation from the desired configuration in the Git repository and cause ArgoCD to act as well. Running within the cluster also lets ArgoCD visualize the various Kubernetes objects that make up entire application deployments and their relationships.

**Note**

Although it is not mandated by GitOps principles, a best practice when using GitOps is to separate the code repositories from the configuration repositories. This allows the typical application workflow to be undisturbed, and perhaps even use existing tooling to build the code, test it, and archive the artifacts. However, once this has been done, a separate workflow can be kicked off to update the configuration that is updating the Kubernetes manifests or the Helm charts to reference the newly built images or configuration values. This allows the CI and CD portions of the workflow to operate completely independently from each other. Developers thus have more freedom when it comes to modifying the tools or workflows, without having to worry about cascading changes to other workflows. This separation of the application configuration from the application code also lets multiple teams collaborate effectively without stepping on each other’s toes.

**Setting Up Argo CD on OKE**

Setting up Argo CD on OKE is as simple as deploying the ArgoCD manifests. The official documentation for deploying ArgoCD covers the steps in detail and also provides additional and optional tools for managing your deployment flow. To get started, create a namespace for ArgoCD and deploy the stable manifests from the ArgoCD GitHub project:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0377-01a)

kubectl create namespace argocd

kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/

stable/manifests/install.yaml

This sets up ArgoCD on the cluster. By default, the ArgoCD server is exposed as a service of type ClusterIP. To access the server, developers can either create a port-forward for temporary access or change the service to use a LoadBalancer or ingress to expose the server more permanently.

The following command sets up a port-forward that listens on port 8080 on localhost and forwards to port 443 on the service:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0378-01a)

kubectl port-forward service/argocd-server 8080:443 -n argocd

While the port-forward is running, developers can access locahost:8080 to access ArgoCD. Using a port-forward can potentially result in a warning from the browser about the service using a self-signed certificate. The port-forward is temporary and is available only on the environment where kubectl is running. This can be useful for testing but not suitable for long-term or multiuser access.

To use a public LoadBalancer resource for the service, and to expose it externally and more permanently, the service object can be patched as follows:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0378-02a)

kubectl patch svc argocd-server -n argocd -p '{"spec": {"type":

"LoadBalancer"}}'

Alternatively, if the developer is using a private load balancer (in a private subnet) to limit exposure of services, as is popular in many enterprise environments, the annotations for a private load balancer can be added to the patch command as well. The additional annotation that follows creates an internal or private Load Balancer:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0378-03a)

kubectl patch svc argocd-server -n argocd -p '{

"metadata":{

"annotations":{

"service.beta.kubernetes.io/oci-load-balancerinternal":"true"

}

},

"spec":{

"type":"LoadBalancer"

}

}'

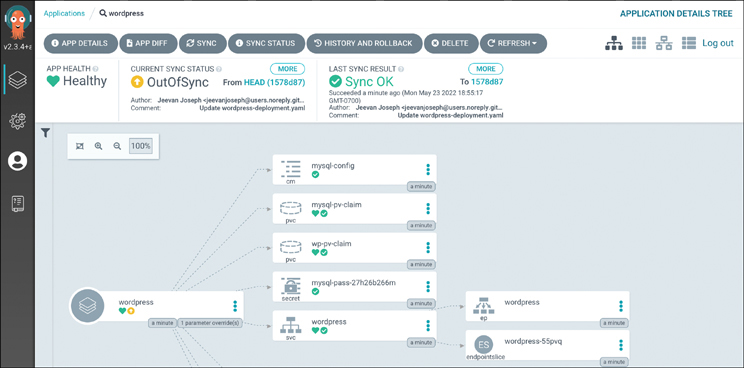
When the ArgoCD service is exposed, developers can log into the ArgoCD UI. Visit https://localhost:8080 if you are using port-forwarding, or use the IP address of the ArgoCD server if you are using a LoadBalancer. To log into the UI, you can retrieve the password for the default user. During installation, this password was generated by ArgoCD and stored as a Kubernetes secret. To retrieve the value, use the following code snippet:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09_images.xhtml#f0378-04a)

kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.

password}" | base64 -d; echo

ArgoCD extends Kubernetes with custom resources such as *Application*. After logging into the ArgoCD UI, developers can create an Application in ArgoCD. These custom resources follow the GitOps model and define an application and its expected configuration using the deployable resource definitions stored in a Git repository. The *Application* resource in ArgoCD identifies the Git repository URL, the branch in the git repository to track changes from, and the path to the resource definition files, among other metadata. These resource definitions can be plain manifest files, Helm charts, or kustomize[**4**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ref9_4a) overlays. [Figure 9-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ch09fig07) shows an application that is managed by ArgoCD.



**Figure 9-7** Managing an Application Using ArgoCD

ArgoCD runs a controller, named the Application Controller, in the OKE cluster and constantly monitors for changes between the live state and the target state. The live state is the state of the application deployed on the cluster. It can include the number of replicas or whether a service is of type LoadBalancer or ClusterIP. The target state is the desired state of the application, and it is defined by the resource definitions and values that are stored in the Git repository in the branch that the ArgoCD *Application* resource is tracking. The live and target state can diverge if someone pushes a new commit to the branch that the *Application* is tracking or if someone updates the live state using kubectl commands such as patch, scale, create, delete, or apply. If the live state and target state have diverged, regardless of where the change originated, ArgoCD considers this Application OutOfSync. Applications that are OutOfSync can be Synced, which essentially applies the definitions and values in the Git repository to converge the live state to the target state. This is because, when using GitOps practices, the Git repository is considered the source of truth.

Because ArgoCD constantly compares the live state with the desired state, it can also visualize the live state with several views. This can be a useful tool for visualizing applications and their topology. ArgoCD tracks the desired state in Git, so it can also create *diffs* between the live state and the desired state. This can help developers and ops teams trace application changes, troubleshoot issues, or roll back deployments.

Apart from the CD tooling, the Argo project offers tools such as Argo Rollouts and Argo Workflows. Argo Rollouts support multiple deployment strategies, such as canary deployments and blue-green deployments for Kubernetes. Argo Workflow provides a workflow engine that can be used to manage and scale data processing or similar workflows.

**Summary**

This chapter looked at a set of automation platforms and approaches for managing cloud native applications at scale. As cloud native applications get more distributed and loosely coupled, these options make it easy to build a loosely coupled fleet of applications that can move at various velocities and use the best-of-breed technologies for what they need to do. However, this also makes the task of building, deploying, patching, and upgrading this growing fleet of applications a challenge in itself. Automation tools and processes described in this chapter focus on this problem; they range from native platforms that are offered by OCI (such as the DevOps platform) to open-source platforms (such as Jenkins) and systems such as ArgoCD that embrace newer methodologies (such as GitOps). With its built-in feature set that supports OCI native services and integration with external tools and systems, the OCI DevOps platform offers a way to build and manage applications within OCI. This helps you consolidate tools and processes onto OCI, making management simple and effective. On the other hand, for users who already have a well-established process of managing cloud native applications, cloud native platforms in OCI (such as OKE) can seamlessly integrate with open-source tools such as Jenkins or ArgoCD so that you can bring your tools of choice and operate without altering your workflows.

**References**

[1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ref9_1) Jenkins: <https://www.jenkins.io/>

[2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ref9_2) Argo: <https://argoproj.github.io/>

[3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ref9_3) Flux: <https://fluxcd.io/>

[4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch09.xhtml#ref9_4) [https://kustomize.io](https://kustomize.io/)