



3RD EDITION

Learning Continuous Integration with Jenkins

An end-to-end guide to creating operational, secure,
resilient, and cost-effective CI/CD processes

NIKHIL PATHANIA



Learning Continuous Integration with Jenkins

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Preet Ahuja

Publishing Product Manager: Suwarna Patil

Book Project Manager: Ashwini C

Senior Content Development Editor: Adrija Mitra

Technical Editor: Arjun Varma

Copy Editor: Safis Editing

Proofreader: Safis Editing

Indexer: Hemangini Bari

Production Designer: Prafulla Nikalje

DevRel Marketing Coordinator: Rohan Dobhal

First published: May 2016

Second edition: December 2017

Third edition: February 2024

Production reference: 1090124

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK

ISBN 978-1-83508-773-2

www.packtpub.com

This book is dedicated to the spirited debaters of social media – your passionate, sometimes eyebrow-raising discussions on CI, CD, and DevOps have not only provided entertainment but also ignited the spark to pen down these pages. Here's to dispelling myths and expanding knowledge, one Jenkins pipeline stage at a time!

– *Nikhil Pathania*

Contributors

About the author

Nikhil Pathania is a tech expert with deep knowledge in the **Software Development Lifecycle (SDLC)** domain. His professional identity is shaped by his specialization in Agile methodologies, DevOps practices, cloud technologies, and container solutions. His significant contributions, particularly in implementing CI/CD frameworks at multinational corporations, underline his expertise in optimizing complex software development processes.

In his current role as a solutions architect at Arla Foods, Nikhil spearheads innovative projects in software development and data analytics, reflecting his keen insight into technological advancements and their practical applications.

I'm deeply grateful to Suwarna Rajput (Publishing Product Manager) for her vital role in reigniting the spark in this book project. Immense thanks to Ashwini Gowda (Book Project Manager) and Adrija Mitra (Content Development Editing) for their continuous support throughout the writing process; their constant motivation and belief to make this book better were invaluable.

Immense gratitude to Arjun Varma (Technical Editor), Werner Dijkerman, and Aditya Soni (Technical reviewers) their meticulous review and insightful feedback on the book. Their expertise and constructive suggestions have been invaluable in enhancing the quality and depth of this work.

Special thanks to my wife, Karishma, whose support made the time for writing this book possible. Her belief in me was both my inspiration and foundation.

A lighthearted shoutout to the Danish 37-hour work week for its “unintentional” role in this book’s creation. It’s amazing how a few extra hours each week can lead to chapters of inspiration!

About the reviewers

Werner Dijkerman is a freelance platform, Kubernetes (certified), and Dev(Sec)Ops engineer. He currently focuses on and works with cloud-native solutions and tools, including AWS, Ansible, Kubernetes, and Terraform. He focuses on infrastructure as code and monitoring the correct “thing,” with tools such as Zabbix, Prometheus, and the ELK stack. He has a passion for automating everything and avoiding doing anything that resembles manual work. He is an active reader of comics, self-care/psychology, and IT-related books. He is also a technical reviewer of various books about DevOps, CI/CD, and Kubernetes.

A big shoutout to the best software architect in the Netherlands: Ernst Vorsteveld!

Aditya Soni is a DevOps/**Site Reliability Engineering (SRE)** tech professional who has taken an inspiring journey with technology and achieved a lot in a short period of time. He has worked with product- and service-based companies including Red Hat and Searce, and is currently positioned at Forrester as a DevOps Engineer II. He holds AWS, GCP, Azure, RedHat, and Kubernetes certifications. He mentors and contributes to the open source community. He has been a CNCF ambassador and AWS Community Builder for four years. He leads AWS, CNCF, and HashiCorp user groups for Rajasthan State in India. He has spoken at many conferences, both in-person and virtual. He is a hardcore traveler who loves food, exploring, and self-care, and shares stories on social media as `@adityasonittyl`.

I would like to thank my parents and family who support me in various ways in my busy schedule. Taking the time to review the book wouldn't have been easy without them. Thanks to my friends, who always cheer me up and give me confidence whenever I feel lost. Last but not least, thanks to the open source communities, mentors, managers, and co-workers from the initial days to now, who have helped me along my path.

Table of Contents

[Preface](#)

Part 1: The Concepts

1

The What, How, and Why of Continuous Integration

What is continuous integration?

Defining continuous integration

Key principles of continuous integration

How to practice continuous integration

Using a version control tool

Verifying every code commit for build errors

Performing unit testing.

Static code analysis

Employing a binary repository manager

Orchestrating using a CI/CD tool

Parallelizing through distributed builds

Why is continuous integration crucial?

A faster time to market (TTM).

Improved software quality

Highly secure applications

Cost-effective system

Summary.

Questions

Answers

Further reading.

Part 2: Engineering the CI Ecosystem

2

Planning, Deploying, and Maintaining Jenkins

Technical requirements

Planning your Jenkins setup

Understanding the Jenkins server architecture

Jenkins system requirements

Evaluating Jenkins deployment scenarios

Understanding JCasC

Deploying your Jenkins setup

Deploying Jenkins on a managed Kubernetes cluster

Deploying Jenkins to a cloud instance

Maintaining your Jenkins setup

Following a structured approach to upgrades

Implementing backup strategies for Jenkins build data

Summary

Questions

Answers

Further reading

3

Securing Jenkins

Technical requirements

Configuring user authentication and permissions

Registering Jenkins on Azure AD

[Installing the Jenkins plugin for Azure AD](#)
[Configuring Azure AD settings in Jenkins](#)
[Configuring user permissions in Jenkins](#)
[Understanding CSRF protection in Jenkins](#)
[Using Jenkins Credentials](#)
[Types of credentials](#)
[Scope of credentials](#)
[Securing secrets in JCasC](#)
[Summary](#)
[Questions](#)
[Answers](#)
[Further reading](#)

4

[Extending Jenkins](#)

[Technical requirements](#)
[Connecting Jenkins to a version control system](#)
[Installing the Jenkins plugin for GitHub](#)
[Creating a personal access token on GitHub](#)
[Creating a secret text credential inside Jenkins](#)
[Creating a username and password credential inside Jenkins](#)
[Configuring the GitHub plugin](#)
[Connecting Jenkins to SonarQube](#)
[Deploying an Nginx Ingress controller for SonarQube](#)
[Deploying a SonarQube server using a Helm chart](#)
[Deploying an Ingress file for SonarQube](#)
[Installing the Jenkins plugin for SonarQube](#)
[Creating a global analysis token on SonarQube](#)

[Creating a secret text credential inside Jenkins](#)
[Configuring the SonarQube plugin](#)
[Connecting Jenkins to Artifactory](#)
[Deploying an Nginx Ingress controller for Artifactory](#)
[Deploying an Artifactory server using a Helm chart](#)
[Deploying an Ingress file for Artifactory](#)
[Installing the Jenkins plugin for Artifactory](#)
[Creating a user in Artifactory](#)
[Creating a username and password credential inside Jenkins](#)
[Configuring the Artifactory plugin](#)
[Summary](#)
[Questions](#)
[Answers](#)
[Further reading](#)

5

Scaling Jenkins

[Technical requirements](#)
[Understanding Jenkins' distributed build architecture](#)
[An overview of the master-agent model](#)
[Methods for building a distributed farm](#)
[Creating on-demand build agents on Kubernetes](#)
[Understanding the Kubernetes plugin configuration](#)
[Creating on-demand build agents on Azure](#)
[Installing the Jenkins Azure VM Agents plugin](#)
[Creating an Azure service principal in Azure](#)
[Creating an Azure service principal credential inside Jenkins](#)
[Configuring the Azure VM Agents plugin](#)

[Summary](#)

[Questions](#)

[Answers](#)

[Further reading](#)

Part 3: Crafting the CI Pipeline

6

Enhancing Jenkins Pipeline Vocabulary

Technical requirements

Building vocabulary for pipeline code

Introducing Jenkins pipeline code

Understanding the components of Declarative Pipelines

Managing complexity using nested stages

Using built-in tools to construct pipeline code

Introducing the Declarative Directive Generator

Introducing Jenkins Snippet Generator

Summary

Questions

Answers

Further reading

7

Crafting AI-Powered Pipeline Code

Technical requirements

Introducing AI, language models, and ChatGPT

Guidelines for effective interaction with ChatGPT

Describing your pipeline requirements

Running your pipeline code

Troubleshooting your pipeline code

Refining your code

[Constructing a pipeline code using ChatGPT](#)

[Understanding the limitations of ChatGPT](#)

[Summary](#)

[Questions](#)

[Answers](#)

8

[Setting the Stage for Writing Your First CI Pipeline](#)

[Technical requirements](#)

[Understanding the software project for CI](#)

[Understanding the architecture](#)

[Understanding the source code](#)

[Understanding the backend code](#)

[Understanding the frontend code](#)

[Understanding the database code](#)

[Creating a repository on GitHub for your project](#)

[High-level CI design](#)

[Source code considerations](#)

[High-level CI pipeline design](#)

[Creating a webhook on the repository](#)

[Setting up a project in SonarQube](#)

[Using Artifactory as a Docker registry](#)

[Creating a Kubernetes Secret on AKS](#)

[Summary](#)

[Questions](#)

[Answers](#)

9

[Writing Your First CI Pipeline](#)

[Technical requirements](#)

[Writing CI pipeline code](#)

[Configuring Jenkins agent settings](#)

[Performing unit testing](#)

[Performing static code analysis](#)

[Performing a SonarQube quality gate check](#)

[Building and publishing images to Artifactory](#)

[Publishing build info to Artifactory](#)

[Publishing testing results to Jenkins](#)

[CI in action](#)

[Creating a Multibranch Pipeline Job in Jenkins](#)

[Visualizing the CI pipeline run and test results](#)

[Visualizing the SCA report and code coverage](#)

[Visualizing build artifacts](#)

[Summary](#)

[Questions](#)

[Answers](#)

[Further reading](#)

Part 4: Crafting the CD Pipeline

10

Planning for Continuous Deployment

Technical requirements

Understanding Continuous Deployment

Universal CI with master-only CD

Universal CI and CD

Understanding GitOps

Planning Continuous Deployment

High-level Continuous Deployment design

Exploring the Helm chart of our web application

Setting up staging and production environments

Installing Argo CD

Creating applications in Argo CD

Integrating Jenkins with Argo CD

Summary

Questions

Answers

Further reading

11

Writing Your First CD Pipeline

Technical requirements

Writing CD pipeline code

Updating Jenkins agent settings

[Configuring pipeline options and the environment](#)

[Update Staging Helm Chart Configuration](#)

[Verify Staging Deployment Health](#)

[Run Performance Testing](#)

[Update Production Helm Chart Configuration](#)

[Verify Production Deployment Health](#)

[CD in action](#)

[Verifying changes on the Helm chart](#)

[Verifying the deployment in staging and production](#)

[Visualizing performance testing results](#)

[Summary](#)

[Questions](#)

[Answers](#)

[Further reading](#)

12

Enhancing Your CI/CD Pipelines

[Technical requirements](#)

[Using GitHub Copilot to write pipeline code](#)

[Enabling GitHub Copilot](#)

[Using Copilot to write pipeline code](#)

[Choosing between Copilot and ChatGPT](#)

[Using Jenkins Shared Libraries](#)

[Writing a Jenkins Shared Library](#)

[Configuring a Shared Library inside Jenkins](#)

[Using the Shared Library inside your pipeline](#)

[Discarding old builds in Jenkins](#)

[Catching security vulnerabilities early in code](#)

[Installing JFrog Xray](#)

[Scanning for vulnerabilities through a CI pipeline](#)

[Adding the Jenkins build to the scans list](#)

[Visualizing the vulnerability report using Xray](#)

[Summary](#)

[Questions](#)

[Answers](#)

[Further reading](#)

[Index](#)

[Other Books You May Enjoy](#)

Preface

Dear readers, welcome to the newly updated *Learning Continuous Integration with Jenkins – Third Edition*! Before diving into the book, I invite you to spend a few moments with this *Preface*. It's not just a formality; it's like a friendly chat before embarking on an exciting exploration. Here, you will get a sneak peek into the book's purpose, the author's perspective, and the overall tone and style. It's like getting to know your guide before setting off on an expedition.

The motivation behind writing this book

When the second edition of *Learning Continuous Integration with Jenkins* was published in 2017, the landscape of **Continuous Integration/Continuous Deployment (CI/CD)** was noticeably different from today's. As 2024 approaches, it's become clear that the practices and tools I discussed have evolved substantially, making many sections of the previous edition less relevant. This shift in the CI/CD realm inspired me to write this updated edition, bridging the gap between then and now, especially in terms of Jenkins' application.

Jenkins itself has undergone significant evolution. The advent of **Jenkins Configuration as Code (JCasC)** and the trend toward deploying Jenkins using Helm charts on Kubernetes exemplify these major shifts. While the core syntax of Jenkins pipeline code has remained mostly stable, the ecosystem surrounding CI/CD has been transformed. The emergence of GitOps, the heightened focus on software composition analysis, and the trend toward container-based applications underscore a broader shift, from monolithic to more modular and scalable architectures.

Another catalyst for this edition has been the confusion and misinformation I've witnessed on social media about Agile, CI/CD, GitOps, **Infrastructure as Code (IaC)**, and DevOps. Misinterpretations and partial truths, often spread by less experienced professionals, have clouded the understanding of those new to these practices. This edition aims to dispel these myths, offering a lucid and comprehensive guide to CI/CD methodologies. It focuses not only on tools but also on the principles and practices essential for successful implementation.

The goal of this book

The primary aim of this book is to instill a foundational understanding of CI/CD. The book goes beyond explaining CI and CD as mere technical processes, delving into the concepts and circumstances that gave rise to CI/CD practices. It emphasizes a comprehensive grasp of the

underlying principles before exploring the tools, focusing on what CI/CD is, why it is crucial, and how it is implemented, with a particular emphasis on its core elements.

As the book progresses, the content becomes more technical, yet it maintains a straightforward approach. You will learn about the efficient and secure deployment of a container-based modular application through the CI/CD process, from development to production, with an emphasis on testing. They will also explore how Jenkins integrates within the broader DevOps ecosystem, working in tandem with various tools.

A critical lesson the book imparts is the “fail fast and shift left” approach. It underscores the importance of encountering and addressing failures in the development or testing stages, rather than in production. This mindset shift is vital – it’s not about preventing failures in production but about ensuring rapid recovery when they occur.

Finally, the book’s purpose extends beyond merely enhancing your knowledge. It is also designed to train your mind, equipping you to think critically and adaptively in the fast-evolving world of DevOps and CI/CD. This dual focus ensures that you are not only well informed but also strategically adept in applying CI/CD principles effectively in real-world scenarios.

Why this book is important

In today's rapidly evolving technological realm, the concept of "continuous" – originally derived from Agile methodologies – has significantly broadened its scope. This concept now underpins a variety of practices, such as continuous testing, continuous improvement, and continuous thinking. Moreover, the principles of CI and CD have expanded their influence beyond traditional software development, finding relevance in fields such as **Machine Learning Operations (MLOps)**.

This book is essential reading in this broadened context. It offers a comprehensive demystification of the core principles of CI and CD, principles that are increasingly becoming across various disciplines. The book delves deeply into the concepts of failing fast, shifting left, and delivering small, frequent, and faster iterations. These are not just software development strategies but also universal principles that are now being applied in areas such as ML and financial operations.

As these principles continue to be adopted in diverse sectors, there is a pressing need for a clear, comprehensive guide that elucidates not just the "how" but also the "why" behind these practices. This book meets that need, providing you with an in-depth understanding of the foundational principles underpinning modern CI/CD practices and their broader applications.

In this updated edition, you'll discover revised content reflecting the latest developments in Jenkins configurations and deployment strategies. It offers practical insights into scaling Jenkins both horizontally and vertically, delving deeper into the evolving CI/CD ecosystem. Whether you're new to Jenkins or seeking to refresh your knowledge, this book aims to guide you through the dynamic landscape of CI/CD with clarity and expertise.

The research behind the book

The creation of this book is the culmination of extensive research and deep personal experience in software configuration management and DevOps, dating back to 2008. My journey in writing this book began with a thorough exploration of the world before Agile, delving into methodologies such as Waterfall and extreme programming. Understanding these historical contexts was crucial in comprehensively addressing the evolution of software development practices.

A significant focus of my research was on the transition from GUI-based configurations to the concept of configuration as code and pipeline as code, as well as the evolution of **Version Control Systems (VCSs)**. These developments represent fundamental shifts in our approach to software configuration and management, and understanding them is key to grasping the principles of CI and CD.

Every technical aspect discussed in this book has been rigorously tested in practice before being committed to paper. This hands-on approach ensures that the content is not only theoretically sound but also practically applicable. The book places a particular emphasis on cloud and container technologies, acknowledging their growing prominence and critical role in modern software development.

Additionally, the book explores the emerging realm of AI tools in DevOps, illustrating how they can be utilized to write pipeline code. This inclusion reflects my commitment to staying abreast of the latest technological advancements and ensuring that the book's content remains relevant and forward-thinking.

To enhance understanding and clarity, this book is enriched with a wealth of illustrations and screenshots. These visual aids are designed to make complex concepts more accessible and to provide you with a clearer picture of practical applications. May your experience with *Learning Continuous Integration with Jenkins* be both enlightening and rewarding, and I trust this book will prove to be an invaluable resource in your professional journey.

Who this book is for

This book is designed for a diverse audience, from university students studying Agile software development to seasoned developers, testers, release engineers, and project managers. It offers a comprehensive guide to mastering CI and CD using Jenkins. If you're already using Jenkins for CI, you can take your project to the next level – CD. Whether you're a novice to the concepts of Agile and CI/CD or a DevOps engineer seeking advanced insights into JCasC, IaC, or Azure, this resource equips you with the tools to harness Jenkins for improved productivity and streamlined deliveries in the cloud.

You are expected to possess a fundamental understanding of software development processes, although in-depth knowledge is not a prerequisite. You should be familiar with basic concepts, such as writing code and the importance of testing, although a comprehensive grasp of more complex software engineering aspects is not necessary. A basic familiarity with version control, especially Git, would be beneficial, since CI/CD processes are intimately linked with source code management, including tasks such as committing changes, creating branches, and merging code.

Having some grounding in programming, even at a rudimentary level, would be advantageous. You don't need to be an expert coder, but being able to write and understand simple code in at least one programming language is helpful. While you might understand why testing is crucial in software development, detailed knowledge of automated testing methods or tools is not required.

What this book covers

[*Chapter 1, The What, How, and Why of Continuous Integration*](#), delves into a comprehensive introduction to CI, guided by the Golden Circle theory. This approach helps us to unravel the "what," "how," and "why" of CI. Our primary focus is to define the practice of CI, understand its key principles, and learn the essential elements required to achieve it. We will also explore the reasons behind the practice of CI.

[*Chapter 2, Planning, Deploying, and Maintaining Jenkins*](#), guides you through planning, deploying, and maintaining a Jenkins server. The aim is to design and deploy a Jenkins setup that is resilient, cost-effective, secure, high-performing, and operational. The chapter starts by examining the Jenkins server architecture, and then it evaluates various deployment scenarios against the Well-Architected Framework. It focuses on the two most popular deployment methods for Jenkins, guiding you through their implementation step by step. This process will integrate crucial DevOps practices, including IaC and JCasC. The chapter also covers the essential aspects of Jenkins server maintenance.

[*Chapter 3, Securing Jenkins*](#), examines the key aspects of securing Jenkins. Here, the vital measures to enhance the security around who gets to do what on your Jenkins instance are explored. Firstly, the chapter delves into user authentication and permissions by integrating Jenkins with Azure **Active Directory (AD)**. After that, it goes through **Cross-Site Request Forgery (CSRF)** protection settings inside Jenkins. Lastly, it explores the powerful Jenkins Credentials feature, which allows for secure storage and usage of sensitive information, such as passwords, API keys, and certificates.

[*Chapter 4, Extending Jenkins*](#), explores the expansive world of Jenkins enhancements, enabling you to tailor its functionalities for specific needs such as CI. Enhancing Jenkins for CI demands the integration of additional tools and services, such as SonarQube, Artifactory, and a VCS, which is what the chapter is all about.

[*Chapter 5, Scaling Jenkins*](#), teaches you how to scale Jenkins horizontally on the cloud with dynamically produced build agents, using both **Virtual Machines (VMs)** and containers on an AKS cluster. Both solutions allow organizations to leverage the strengths of each approach. VMs provide flexibility and compatibility with existing infrastructure, while containers offer efficient resource utilization and faster deployment times.

[*Chapter 6, Enhancing Jenkins Pipeline Vocabulary*](#), explores learning Jenkins pipeline code syntax. The chapter's aim is to prepare you for the use of AI to write a Jenkins pipeline. To achieve this, the chapter begins with an introduction to the Jenkins pipeline syntax. Next, you will learn about the core

and add-on building blocks of pipeline code. The focus is mainly on the structure and skeleton of a pipeline. It also teaches you some internal Jenkins tools to construct pipeline code.

[*Chapter 7, Crafting AI-Powered Pipeline Code*](#), delves into using ChatGPT to write pipeline code. In this chapter, we embark on an enlightening journey into the world of **Artificial Intelligence (AI)**, with a special focus on ChatGPT, a renowned AI model. As we navigate the evolving landscape of AI, you'll be equipped with the knowledge to harness ChatGPT to aid in the construction of pipeline code.

[*Chapter 8, Setting the Stage for Writing Your First CI Pipeline*](#), focuses on planning for CI and understanding the high-level CI design. The chapter begins by explaining the software projects that will be used for CI. Next, we will learn to configure tools such as SonarQube for code quality and Artifactory for Docker image storage. These platforms are vital in our CI pipeline. By the end of the chapter, you'll have grasped how to analyze a software project for CI, comprehend its architecture, and develop a CI design. We'll also touch on setting real-time CI triggers through webhooks.

[*Chapter 9, Writing Your First CI Pipeline*](#), logically rounds up everything you learned in the previous chapters. It's a step-by-step, hands-on guide that will teach you to create a CI pipeline in Jenkins. You will start by writing CI pipeline code stage by stage. At the end of the chapter, we will take a walkthrough of the CI pipeline run, using the Jenkins Blue Ocean interface.

[*Chapter 10, Planning for Continuous Deployment*](#), delves into understanding CD. You will be introduced to the concepts of CD and its elements, including GitOps. Subsequently, you will be acquainted with a high-level CD design. This is followed by setting up the Argo CD tool, establishing staging and production environments on **Azure Kubernetes Service (AKS)**, and undertaking other essential steps to run a CD pipeline with Jenkins.

[*Chapter 11, Writing Your First CD Pipeline*](#), methodically steers through the process of developing a full-fledged CD pipeline. Through the automated CD pipeline, you will master the act of updating an application's Helm chart on GitHub, consequently triggering a deployment in the staging environment. Post-deployment, the pipeline autonomously monitors the application's status in staging, runs performance tests, and, upon validation, facilitates further Helm chart modifications on GitHub to initiate deployment in the production environment.

[*Chapter 12, Enhancing Your CI/CD Pipelines*](#), explores techniques to enhance your CI and CD pipelines. It introduces GitHub Copilot, an AI tool that refines Jenkins pipeline code development, aiming for smarter coding, fewer errors, and faster development. It also discusses Jenkins Shared Libraries, which centralize common code patterns, simplifying the management of multiple pipelines. Additionally, strategies are provided to handle and remove old Jenkins builds, ensuring system optimization. Furthermore, using JFrog Xray, the chapter demonstrates how you can integrate

automated security scans in your Jenkins pipeline, guaranteeing not just functional but also secure code deployment.

To get the most out of this book

This book thoroughly covers concepts and procedures executed on the Azure cloud, eliminating the need for extensive setups on your computer. There's no requirement to install Docker Desktop, any VMs, or a Kubernetes cluster on your local machine. All you need is an Azure subscription and certain CLI tools on your laptop. Here is a concise list:

Tool/subscriptions	OS
An Azure subscription	NA
Visual Studio Code (minimum 1.84.2 or the latest)	Windows, macOS, or Linux
The Azure CLI (minimum 2.49.0 or the latest)	Windows, macOS, or Linux
Helm (minimum v3.12.1 or the latest)	Windows, macOS, or Linux
Git (minimum 2.41.0 or the latest)	Windows, macOS, or Linux
Terraform (minimum v1.5.2 or the latest)	Windows, macOS, or Linux
kubectl (minimum v1.27.3 or the latest)	Windows, macOS, or Linux

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

This book guides you through the process of creating CI and CD pipelines for a container-based, modular, three-tier “Hello World” web application. Once you've mastered the concepts and techniques presented, I encourage you to apply your knowledge to a more complex software project. Delve deeper into the intricacies of each DevOps tool's configuration within the CI and CD platform. Explore the possibilities of configuring Jenkins entirely using JCasC. On the testing front, challenge yourself by integrating more advanced testing methods into your workflow.

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/PacktPublishing/Learning-Continuous-Integration-with-Jenkins_Third-Edition. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “Now, create a Kubernetes cluster using the `az aks create` command with the `-g` option, placing it inside the resource group we created in the previous step. Use the `-n` option to give your Kubernetes cluster a name.”

A block of code is set as follows:

```
configuration-as-code
git
sonar
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
jenkins:
  systemMessage: "Jenkins configured automatically by Jenkins Configuration as Code
plugin\n\n"
  securityRealm:
    local:
      users:
        - id: jenkins-admin
          password: password
```

Any command-line input or output is written as follows:

```
az aks create -g rg-nikhil-sbox -l westeurope \
-n kubernetes-dev --tier free --node-count 1 \
--enable-cluster-autoscaler --min-count 1 --max-count 3 \
--network-plugin kubenet --generate-ssh-keys
```

The commands presented in this book are tested for execution within a PowerShell command-line shell. While alternative command-line shells may be utilized, some modifications to the commands may be necessary.

Bold: Indicates a new term, an important word, or words that you see on screen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “From the dashboard, click on **Azure**

Active Directory, and then select **App registrations** from the menu.”

TIPS OR IMPORTANT NOTES

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com

Share Your Thoughts

Once you’ve read *Learning Continuous Integration with Jenkins*, we’d love to hear your thoughts! Please [click here to go straight to the Amazon review page](#) for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we’re delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835087732>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

Part 1: The Concepts

Consider a factory with a huge production line that snakes across the floor like a metallic river. Each portion of this assembly line flows flawlessly into the next, but there are also moments when components falter and processes stutter here and there. The actual marvel of modern production, however, is not flawless execution but intelligent contingency plans, the ability to self-correct, and the relentless quest for efficiency. Mistakes are not only problems; they are also chances for improvement and growth. **Continuous Integration (CI)** represents this same resilient attitude in the digital art of software development.

The blueprint for our own sophisticated CI manufacturing line is the first step in our journey. We use the Golden Circle theory to unravel the “what,” streamline the “how,” and distill the “why” of CI. Before we roll up our sleeves and begin tinkering with the tools and approaches in the upcoming parts, let us first become acquainted with the CI blueprint. In this part, we describe the CI process, become acquainted with its essential concepts, and prepare to answer why we do CI at all. Just as engineers on the assembly line understand that each turn of the screw is critical, you will learn how each facet of CI is critical for the smooth flow of the development life cycle.

This part has the following chapter:

- [Chapter 1](#), *The What, How, and Why of Continuous Integration*

The What, How, and Why of Continuous Integration

In this chapter, a comprehensive introduction to **continuous integration (CI)** is presented using the **Golden Circle Theory** (see [1] in the *Further reading* section at the end of the chapter) that will help us in answering the *what*, *how*, and *why* of CI.

Our focus here is primarily on defining the practice of CI, understanding its key principles, learning the elements required to achieve it, and lastly, answering why we practice CI.

The practical implementation of CI is, however, presented in detail throughout the remaining chapters of the book.

After completing this chapter, you should be able to do the following:

- Explain what continuous integration is
- Learn how to practice continuous integration
- Understand why continuous integration is crucial

What is continuous integration?

In this section, we will try to answer what CI is in two basic steps. We will first attempt to define CI, and then examine its key principles.

Defining continuous integration

CI is a software development practice in which developers regularly integrate their code changes into the project's shared work area and inspect their changes for build issues and other quality-related problems.

Integration is the act of submitting your modified code to the project's main branch (the potential software solution). This is technically done by merging your feature branch to the main branch in a **trunk-based development** (a Git-based branching strategy) and, additionally, by merging the feature branch into the development branch in a **Gitflow Workflow-based development** (another Git-based branching strategy), as shown in the following figure:

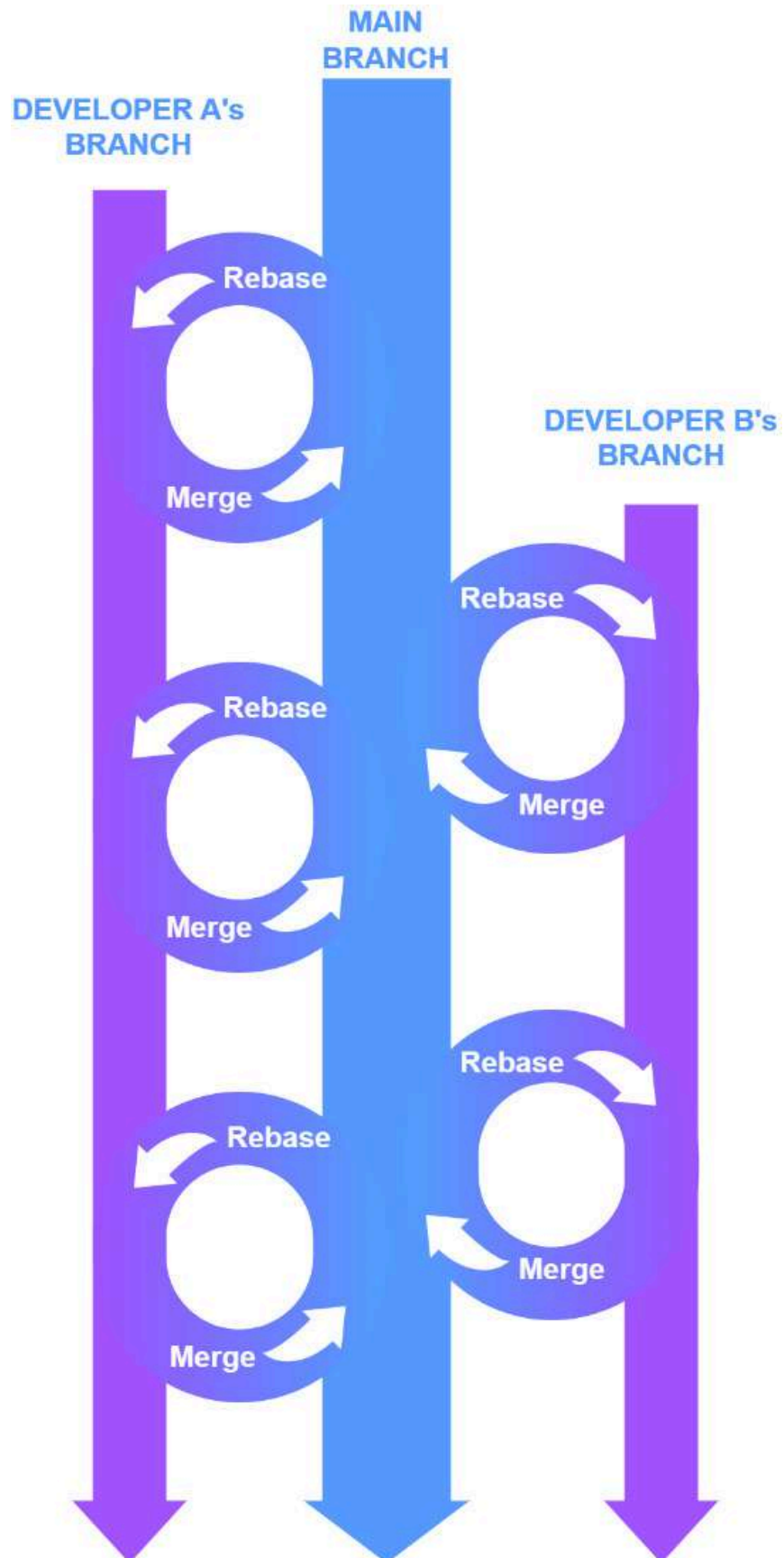


Figure 1.1: The process of frequently integrating your changes with the main code

CI is necessary to detect and resolve issues encountered during and after integration as early in the cycle as possible. This can be understood from *Figure 1.2*, which depicts various issues encountered during a single CI cycle:

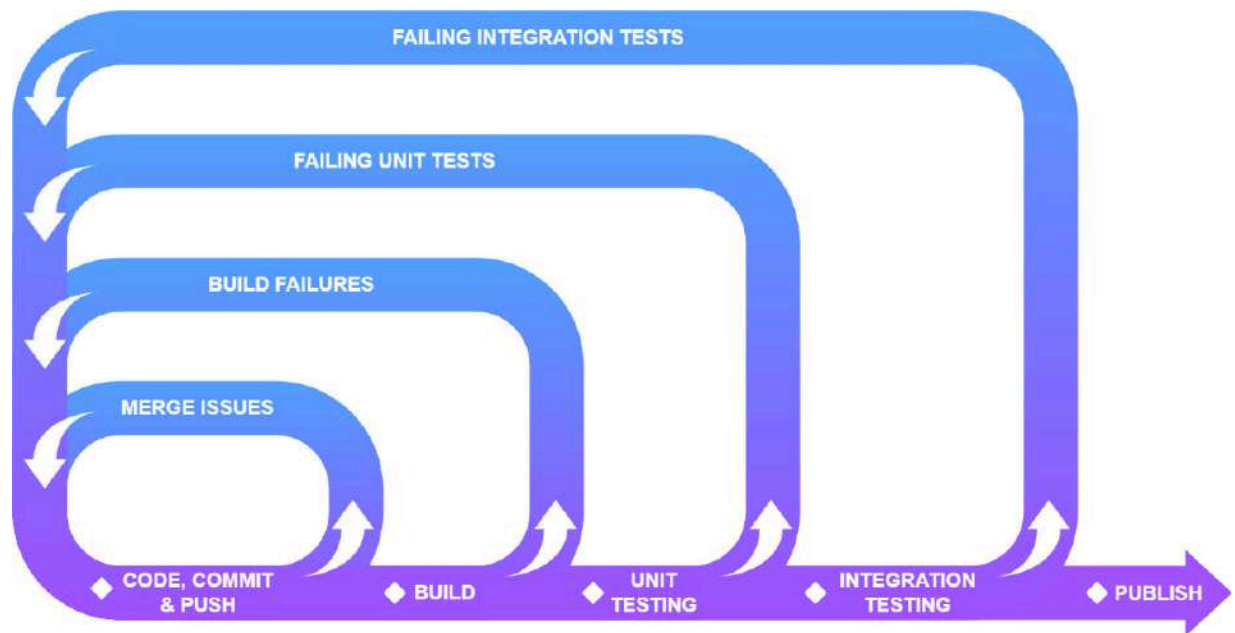


Figure 1.2: The CI process

Figure 1.2: The CI process

When a developer commits and pushes (merges) their code changes to the remote branch, they may experience merging difficulties, if not build failures, failed unit tests, or failed integration tests.

A merging problem can emerge if developers do not routinely pull (rebase) the code from the remote trunk branch on their local branch. Similarly, the code may fail to compile owing to compilation issues, and failing tests indicates a defect, which is a positive thing. If any of these issues occur, the developer must adjust the code to correct it.

Key principles of continuous integration

Now that we've defined CI, let's look at the key ideas that underpin it. Some of it, I suppose, you've already guessed. Let's dive in

Pulling frequently

In the context of CI, a **pull** simply updates your local branch with the most recent code changes from the remote branch, while a **push** publishes your code changes to the remote branch.

Git, the popular distributed version control system, inspired the notion of pull/push. However, prior to the popularity of Git, the terms **rebase** and **merge** were more prominent. Nevertheless, let's understand why frequent pulls promote CI.

Working on your local branch for an extended period without often pulling changes is handy, but it is also quite likely to result in a flurry of merging issues. This occurs when, in big development teams, hundreds of code changes are submitted to the remote repository each day, considerably altering it and raising the likelihood of encountering merge conflicts from a developer who seldom clicks the pull button.

On the other hand, such situations are rare in teams that frequently pull and push code changes. This is the most important principle of CI as it helps avoid merge hell.

Validating every code change

You validate your code change for compilation issues, failing unit tests, failing integration tests, and other quality checks. You first do this locally on your local workspace (laptop), and next on the build server when you push changes to the remote repository.

This practice makes sure that every commit is validated to see its impact on the system. Afterward, make sure to share the results with the team. The idea is to get instant feedback on the changes that have been made.

Fail fast

We think the following idea is based on Murphy's law, which states that *"anything that can go wrong will go wrong."* Building on that, in the context of the software development life cycle, we may argue that if failure is unavoidable, it is preferable to fail sooner in production rather than later. Pulling code changes often and verifying each code change enhances the likelihood of identifying merge problems and functionality and quality concerns, hence failing quicker.

Let us understand the idea of **fail fast** a bit differently. The concept of fail fast advocates that all the pipeline stages, such as validating merge issues, checking for compilation issues, unit testing, integration testing, static code analysis, and so on, must be performed sequentially for a particular code change rather than simultaneously. This is because building the code is pointless if there are merge issues; similarly, running unit testing is pointless if the build fails, and so on and forth. In other words, stop as soon as you find there is an issue and try something else.

Automation

Automation acts as a catalyst. From finding failures to successfully publishing your application, automation accelerates everything. Automate anything you can, whether it's code compilation, unit testing, integration testing, static code analysis, packaging the finished app or service, and so on.

Testing is the most time-consuming and repetitive operation among the others; thus, automating the testing process may considerably boost the speed of software delivery. The testing process, on the other hand, is more difficult to automate than the build, release, and packaging phases.

It usually takes a lot of effort to automate nearly all the test cases used in a project. It is an activity that matures over time. Hence, when beginning to automate the testing, we need to take a few factors into consideration. Test cases that are of great value and easy to automate must be considered first. For example, automate the testing where the steps are the same but they run every time with different data. Also, automate the testing where a software functionality is tested on various platforms. Additionally, automate the testing that involves a software application running with different configurations.

Later sections will discuss the significance of a CI tool in automating and orchestrating the CI process.

Continuous feedback

The software development process is analogous to a factory assembly line. The delivery of new features and upgrades flows from left to right. The far left of the pipeline is where development takes place, and the far right is where your application is operating in production. As depicted in the following diagram, consider such an assembly line with a single feedback loop flowing from the operations team to the development team:

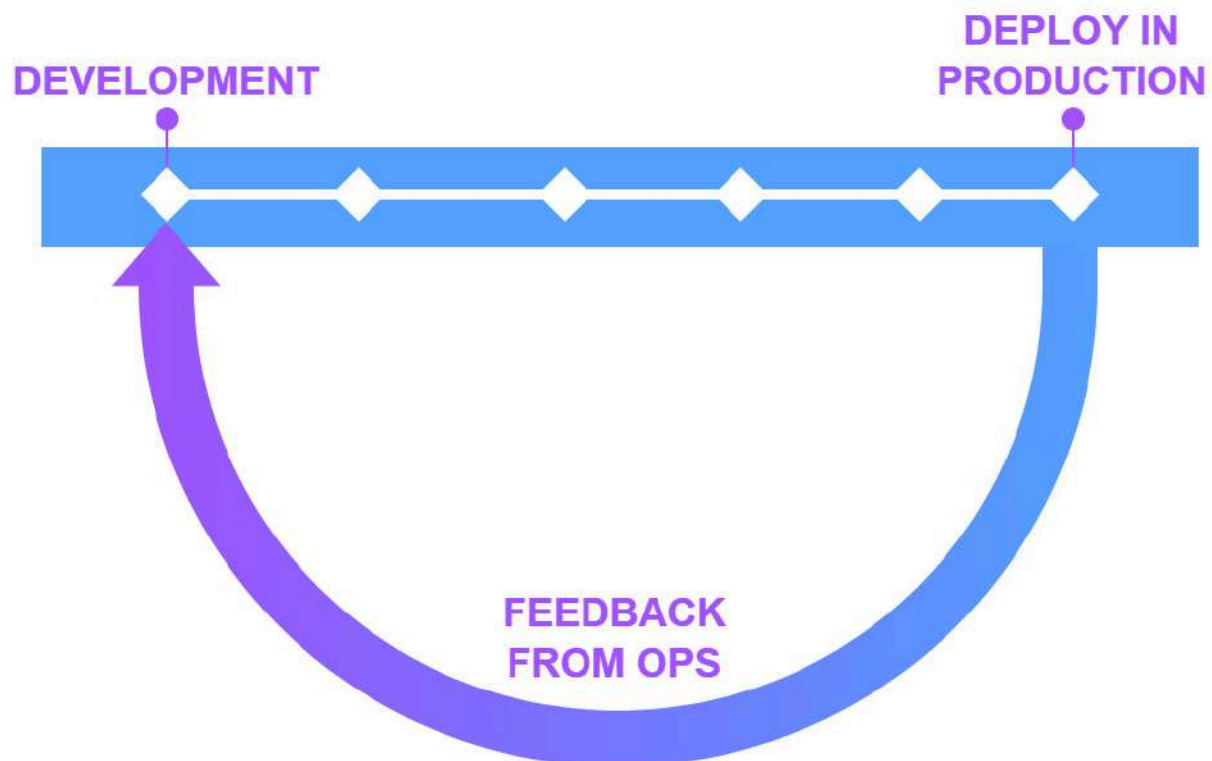


Figure 1.3: Example of a late and slow feedback loop

Because there is a single information flow moving backward in such a case, the developers will never know whether their modifications have produced a well-functioning application until the application is launched. Another thing to keep in mind is that the time it takes for the feedback to reach developers equals the time it takes to bring a feature from development to deployment.

One such method with limited feedback loops is the waterfall model of software development. In this model, the first feedback occurs when all the code is integrated, the second when the code is tested, and the final when it is put into production. Nothing tangible can be done quickly when the operations team complains about a failure in production whose code was created a year earlier. We require more immediate feedback.

And that is the aim of continuous feedback: to have more and more feedback loops in our software development life cycle that are closer together and happen faster. One such example is the practice of peer review, which often occurs throughout the development process. During peer review, the proposed modifications are assessed by a senior developer to assert whether the changes will function as intended, as shown in the following figure:

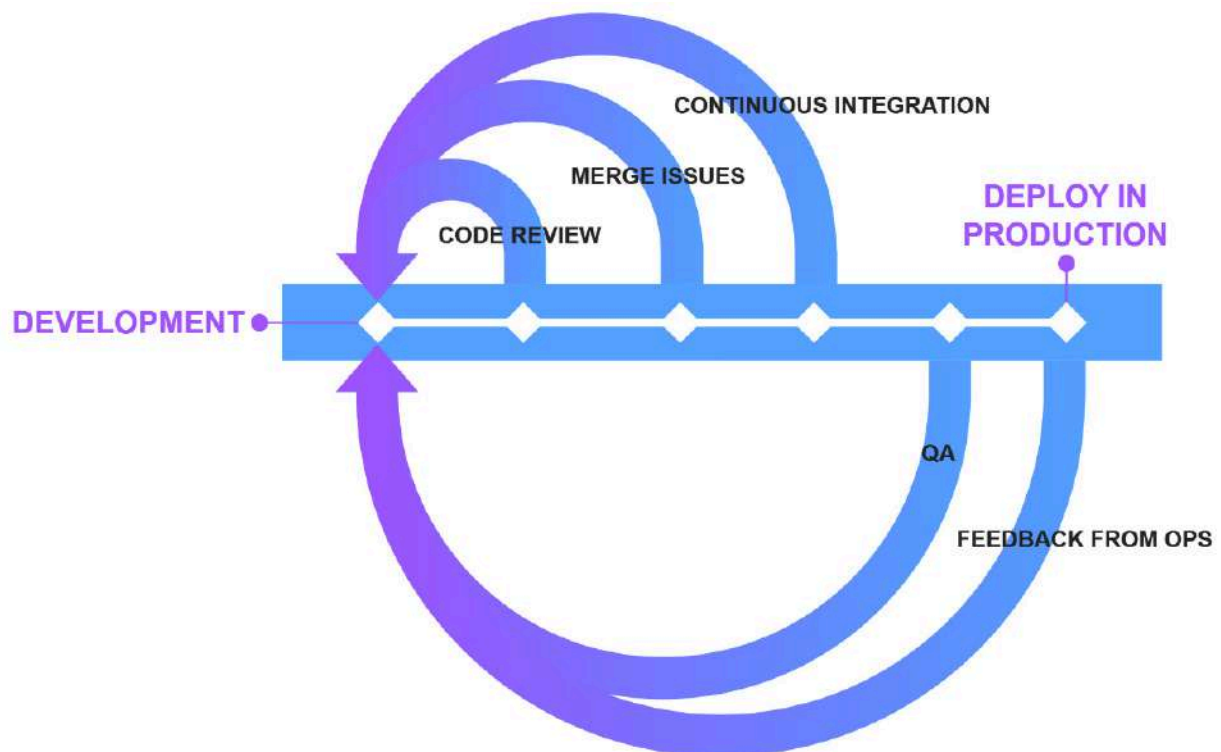


Figure 1.4: Example of a continuous feedback loop

Continuous input from all phases of the software development life cycle should be provided with as much clarity as feasible. Besides, if your software development process is heavily automated, then it

further helps in achieving continuous feedback.

This brings us to the end of this section. We've learned the definition of CI and its key principles; now, let's look at how to put these principles into action in the next section.

How to practice continuous integration

To understand how CI is practiced, you must first realize that it is a constantly evolving ecosystem of tools and technologies, and this section will go through most of them. At the end of this part, it will be obvious how these separate components and their associated tools contribute to the realization of the core concepts of CI outlined previously. Besides, the subsequent chapters of the book are a detailed, thorough elaboration on the practices presented here. As a result, the current section is crucial.

The topics discussed in the following sections are structured sequentially and in accordance with the various stages of an automated CI pipeline:



Figure 1.5: Various stages of a CI pipeline

So, let's start with the first and foremost aspect of CI: the version control system.

Using a version control tool

A **version control system** helps the team in controlling changes to the project's source code. It does this by monitoring every code change made to the source code, resulting in a detailed history of who modified what at any given time. It enables developers to work in their own workspaces, providing isolation and, at the same time, offering a means for collaboration and dispute resolution. Most significantly, it helps you to maintain a single source of truth for your project.

Everything that is necessary for delivering the software application should ideally be version-controlled and tracked, including software dependencies that are tracked in a slightly different way using a binary repository manager (a DevOps tool for managing all the artifacts, binaries, files, and container images throughout the software development life cycle of your application), as we shall see

later in this chapter. You can even expand on this idea by versioning code to manage and synchronize infrastructure configurations and application deployments. This is known as **GitOps**.

A version control tool offers features such as tags to mark important milestones in the software development life cycle, and branches to enable independent development that will ultimately be merged back to where it came from.

Parallel development

Branching allows **parallel development**. It allows developers in a team to work on individual versions of the code at the same time. To understand this, consider a software development project that has all its source code stored in a Git repository with a single main/master branch. Assume there is a requirement to deliver two new features and two different developers from the team are picked to work on them independently, without interfering with each other's work. The solution is that they would use branches. Each developer would create their own branch and work on it in solitude. When they have completed their work and the CI pipeline running on their separate branches is green, the changes are merged back into the main branch.

Though there are no restrictions on how branches can be utilized, it is recommended that certain industry-standard branching methods be employed. Gitflow Workflow and, more recently, the trunk-based workflow, are two such branching strategies.

Gitflow is a more restrictive development paradigm in which not everyone has the privilege to make modifications to the main source. This keeps bugs to a minimum and code quality to the highest level.

On the other hand, because all developers have access to the primary code, trunk-based development is a more vulnerable branching strategy. Nevertheless, it allows teams to iterate fast.

GitFlow Workflow

Gitflow Workflow is a branching methodology that utilizes multiple branches to organize source code. In the workflow shown in the following diagram, the main/master branch is kept clean and only contains releasable, ready-to-ship code. All development takes place on the feature branches, with the development branch acting as a central location for integrating all features.

Figure 1.6 depicts Gitflow Workflow. As you can see, we have release branches that are taken out from the development branch when there is a stable release. All bug fixes for a release are performed on the release branch. There is also a hotfix branch that is created from the main/master branch when a hotfix is required:

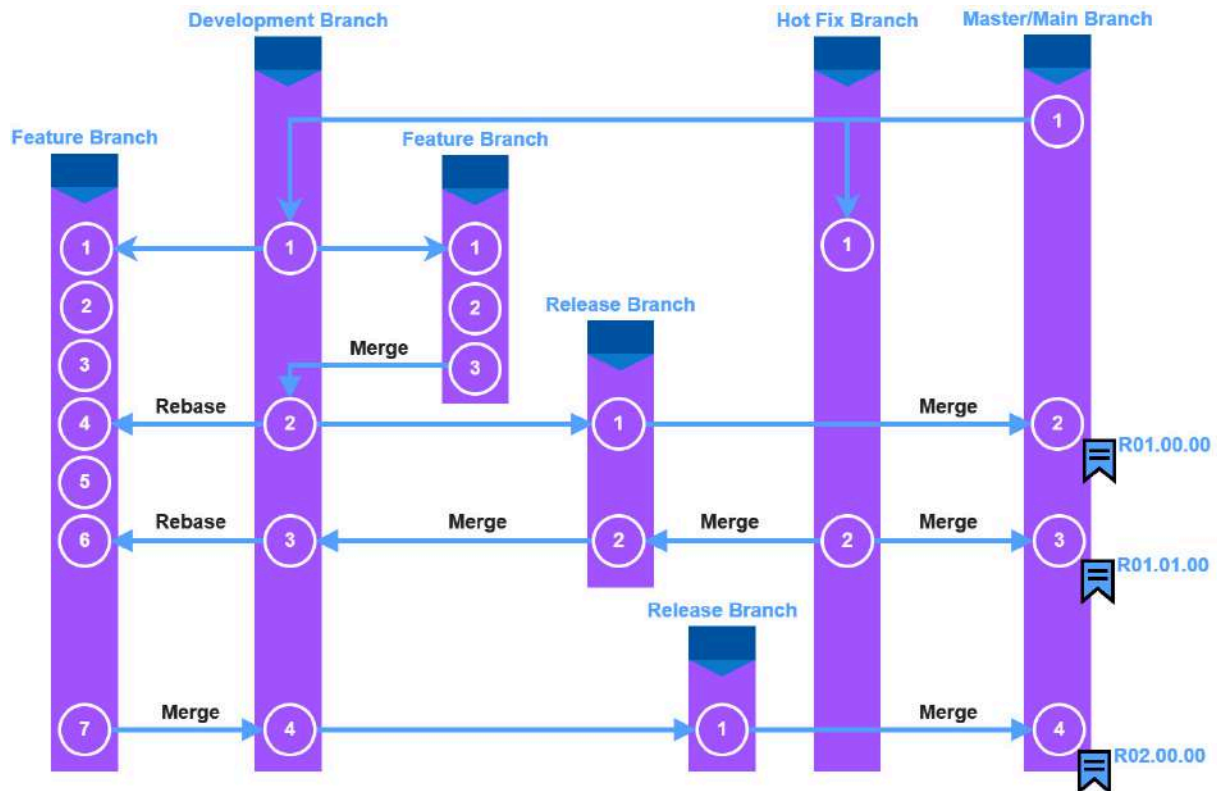


Figure 1.6: Gitflow Workflow

Trunk-based workflow

The **trunk-based branching** technique encourages the use of the main/master branch as the single location to track all significant milestones of a software development life cycle. You may still create tiny, temporary feature branches from the main/master branch, but you must merge them into the main/master branch as soon as possible.

Someone who has been in the IT industry for more than a decade will find it amusing to see that the trunk-based workflow is popular again. When version control and CI were new concepts, the trunk-based workflow was what most teams used; then, Gitflow Workflow became popular for good reasons and remained so for a very long time, only to be overthrown by the trunk-based workflow again.

In the trunk-based branching model, just one CI/CD pipeline that tracks the main/master branch is required. When a commit occurs, the CI/CD pipeline is activated, which results in a releasable candidate. Any commit to the main/master branch might be a potential release candidate, a hotfix, or a new feature.

Because everyone is permitted to merge changes to the main/master branch, a strict code review procedure is required. This methodology is better suited to high-performance contemporary software projects where the entire CI/CD process is automated from start to finish. Microservice-based

software development projects, in particular, can benefit greatly from the trunk-based workflow since they have a faster CI/CD process. Monolithic-based projects, on the other hand, would benefit more from Gitflow Workflow.

Because this branching model lacks feature branches, **feature flags** can be used instead. Feature flags, also known as feature toggles, are a software development approach that allows you to publish software programs that have dormant features that may be toggled on and off during runtime.

Verifying every code commit for build errors

The first and foremost step of any CI pipeline is to check whether the code changes are compilable. Basically, we try to see whether we can generate a working binary that can be tested and deployed. If the code base happens to be in an interpreted language, then we check whether all the dependencies are present and packed together. There is no point moving any further in the CI process if the code changes fail to compile.

The process of CI advocates that you compile every code change, trigger a notification (feedback loop), and fail the whole pipeline run in the event of a failure. As such, the duration of the build run (build time) plays a critical factor in determining the effectiveness of the CI process. If the build time is long (say, several hours), it becomes impractical to build every code change on an infrastructure limited by cost and availability. Automation and CI have no value if pipelines take hours to complete.

Modularizing your application is an effective way to counter software builds that take a longer time to build.

Performing unit testing

Remember the principle of continuous feedback: “to have more and more feedback loops in our software development life cycle that are closer and faster.” Unit testing is one such feedback that is closer and faster to the development stage.

Without unit testing, the next level of testing is the long-running integration testing stage, followed by the QA stage from the continuous delivery/deployment pipeline. It will normally take some time for your build to reach the QA engineering team, who will usually raise a defect if they identify an issue. In summary, if unit testing is used, a lengthy feedback cycle can be avoided.

Unit tests enable you to test the smallest testable portion of your program faster. If your source code is well written, unit tests are easy to write and easy to resolve when they fail.

Code coverage

Code coverage is an important CI metric and it's defined as the proportion of code covered by unit tests. Having a high code coverage score is essential since detecting and resolving defects in QA or production is both time-consuming and expensive.

Any piece of code that lacks a matching unit test is a possible risk; therefore, having a greater code coverage score is crucial. It's also worth noting that code coverage percentage does not ensure the functionality of the code covered by unit tests.

The percentage of code coverage is calculated during unit testing. Almost every programming language has a coverage tool. However, the findings are mostly uploaded to a static code analysis tool such as SonarQube, where they are shown on a lovely dashboard among other data concerning code quality.

Static code analysis

Static code analysis, sometimes known as **white-box testing**, is a type of software testing that searches for structural features in the code. It determines how resilient or maintainable the code is. Static code analysis is conducted without actually running programs. It differs from functional testing, which explores the functional features of the software and is dynamic.

Static code analysis is the evaluation of software's inner structures. It doesn't question the code's functionality. A static code analysis tool such as **SonarQube** comes with a dashboard to show various metrics and statistics of each CI pipeline run. Static code analysis is triggered every time a build runs after the unit testing. SonarQube supports many languages, such as Java, C/C++, Objective-C, C#, PHP, Flex, Groovy, JavaScript, Python, PL/SQL, COBOL, and so on.

Quality metrics

When you run static code analysis on your source code, the SonarQube tool generates a report containing important metrics concerning the quality of your code. These metrics are grouped into explicit sections: complexity, duplications, issues, maintainability, reliability, security, size, and tests [2]. Every static code analysis report is published to its own project-specific dashboard on SonarQube, branch by branch. However, the measurements are shown collectively for the project as a score or a number, or by using a time series graph.

Talking about metrics, let's look at one such section: **tests**. This covers a variety of metrics relating to code coverage and unit testing, such as the following:

- **Uncovered lines**: The number of source code lines that are not covered by the unit tests
- **Unit tests**: The number of unit tests
- **Unit test duration**: The collective duration of all unit tests

- **Unit test errors:** The number of failed unit tests

And the list goes on.

Security is another key quality metric section. It includes metrics such as vulnerabilities (the total number of security problems discovered). Remember the Apache Log4j security flaws? This is where you'd detect it and stop it before it goes any further.

Quality profiles and rules

Quality profiles are collections of rules that you choose to apply during static code analysis. This can be done from the SonarQube dashboard. Quality profiles are language-specific, hence you would usually see more than one quality profile assigned to a project in SonarQube.

The issues you see for a particular static code analysis run are generated based on the SonarQube rules they broke. The SonarQube rules [3] are categorized as follows:

Rule Type	Quality Metric Section
Code smell	Maintainability
Bug	Reliability
Vulnerability	Security
Security hotspot	Security

Table 1.1: The SonarQube rule types mapped against their corresponding quality metrics

Quality profiles can have rules added or removed. However, this is normally done by a specific governance team from an organization's cyber security department, which oversees the evaluation of new and old rules on a regular basis and builds custom quality profiles to be utilized by development teams.

Quality gates

So far, we've looked into quality metrics, profiles, and rules. But how can we turn this insight into action? The answer is **quality gates**. A quality gate serves as a checkpoint through which a pipeline flow must pass. Quality gates are, in practice, combinations of specific conditions. An example of a quality gate condition can go as follows: the number of bugs is less than 10, and the percentage of code coverage is greater than 95%.

CI pipelines can be configured to pass or fail based on the results of the quality gate check. In the subsequent chapters of the book, you will learn how to achieve this in practice. Quality gates are

normally decided and applied by development teams. The following diagram depicts the various stages of a CI/CD pipeline, emphasizing the crucial function of the Quality Gate as a pivotal checkpoint where code must match set quality standards before it can proceed to deployment.

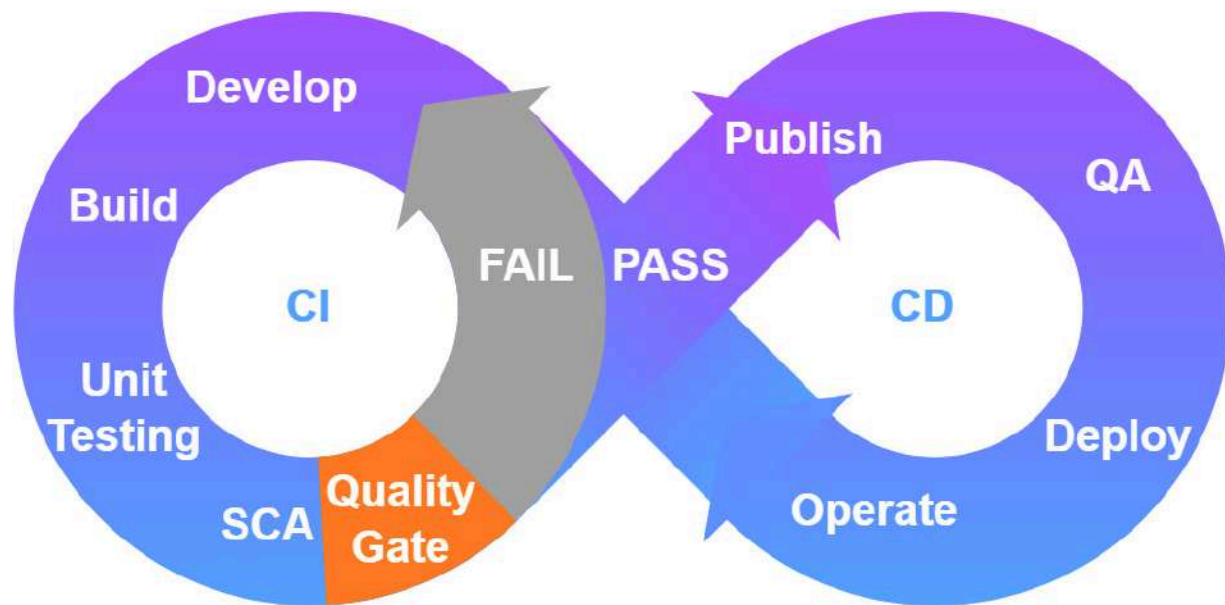


Figure 1.7: SonarQube Quality Gate

If code fails to meet these standards at the Quality Gate, it is returned for additional development and improvement, guaranteeing that only high-quality, thoroughly tested software advances to the Continuous Deployment phase for release and operation.

Employing a binary repository manager

CI pipelines generate binary artifacts that are deployable apps and services. When you choose to run the pipeline for every commit made to the trunk branch, this happens much more quickly. As a result, these built packages must be saved and managed somewhere for subsequent usage, and this is usually done using a **binary repository manager**. But what exactly is a binary repository manager?

A binary repository manager, such as **JFrog Artifactory**, is a system for storing, managing, and promoting built artifacts by incorporating metadata. The metadata attached to every artifact is a group of properties.

In addition to storing built packages, the binary repository manager is also responsible for keeping track of dependencies that go into building your project. Let's see some of the features of a binary repository manager.

Accounting builds

A binary repository manager, such as JFrog Artifactory, is essential for not just storing the built artifacts but also accounting for them.

For example, every artifact hosted on the Artifactory server has a unique URL. This enables you to embed a specifically built artifact into configuration management scripts (for example, Chef recipes) and utilize them afterward to distribute packages on multiple machines.

A **property** (singular) or **property set** (plural) is another essential feature included with Artifactory. A property is an entity with a key and a value pair, while a property set is a collection of several key and value pairs. The property/property set is attached to the artifacts as XML metadata. This makes it possible to query artifacts using the **GUI** (short for **graphical user interface**) or **APIs** (short for **application programming interface**), opening the path to automation. Similarly, properties may be assigned, unassigned, or modified using the GUI or API instructions.

Artifact housekeeping is one of the many use cases of the property feature. For example, in a trunk-based CI system, every commit potentially results in a build artifact. This causes many build artifacts to accumulate on the Artifactory server in a very short amount of time. However, not all these artifacts will get published; some may fail QA testing, while others will simply be ignored for a newer version.

This calls for an automation script to identify and clean artifacts that are older than a particular period. Since *date of creation* is a default auto-assigned property of an artifact, you can use the **AQL** (short for **Artifactory Query Language**), another feature of Artifactory, to query artifacts using the *date of creation* property and then delete them.

Build promotion

A binary repository manager such as JFrog Artifactory helps you in architecting a fully automated CI/CD pipeline workflow through another built-in feature called Build Promotion.

To understand how it works, imagine a fully automated continuous deployment pipeline with stages such as downloading the built package from Artifactory, deploying the package in the QA environment, running automated tests, and lastly, deploying the solution in production. This Build Promotion feature in Artifactory takes care of promoting the status of the artifact by updating the Build Info data. Build Promotion is configured to trigger from the various stages of the CD pipeline using APIs. See *Figure 1.8* to get an overview of the Build Promotion process.

Build Info data is metadata published along with the artifact by the continuous integration pipeline to Artifactory. It includes the commit ID, CI pipeline URL, build number, and a lot more information:

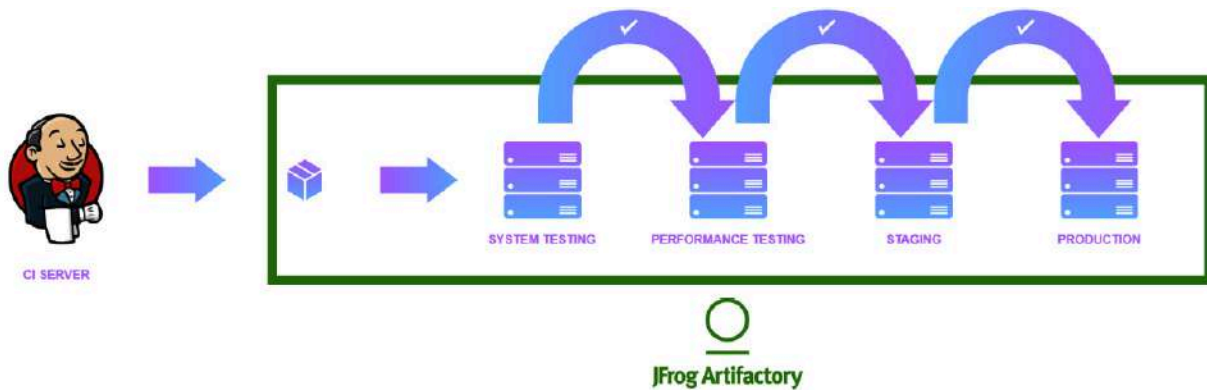


Figure 1.8: The process of Build Promotion

Software dependency management

Pre-built software dependencies or libraries, such as npm, Maven, and Conda, to mention a few, enable developers to build applications more quickly. Using these open source frameworks, however, presents two major challenges: administration and security.

The Artifactory tool helps you to manage dependencies as well as check them for vulnerabilities. It serves as a one-stop shop for accessing all the different dependent packages needed to build your project.

This is often done by creating as many remote repositories as needed within Artifactory, each of which is linked to its appropriate package source on the internet. With an internal Artifactory URL, a remote package repository in Artifactory allows you to list all accessible packages from the internet. All you must do is reference these libraries using the correct Artifactory URLs when writing your code. Since these Artifactory URLs get tracked by your version control tool (Git), you can easily generate a **bill of materials (BOM)**, used to build a particular version of your software application. A remote repository is only a reference to the actual source; nothing else is ever downloaded to the Artifactory server but the metadata.

Software composition analysis (SCA)

Another advantage of utilizing a binary repository manager such as Artifactory is the built-in security tool that scans open source package libraries for vulnerabilities and checks open source libraries for license violations.

Tools such as JFrog Xray, part of the JFrog suite, scan your package libraries for OSS components and security vulnerabilities and create a report before they enter your source code. SCA can also be incorporated as a separate stage in your CI pipeline using other SCA tools such as Black Duck.

Orchestrating using a CI/CD tool

At the heart of the CI system sits a CI/CD tool. Its role is to seamlessly orchestrate other DevOps tools to achieve a fully automated CI/CD workflow. Thus, a good CI/CD tool must provide, at the very least, the following features:

- Seamless integration with various software tools and technologies
- Powerful pipeline syntax to create robust automation
- Intuitive pipeline visualization and notification capabilities

Seamless and broader integration

The various stages of a CI pipeline are scripted using their corresponding tools and technologies. For example, to build code written in Java, you require Maven libraries and the Maven build tool. Likewise, to build a code written in C++ or C#, you require NuGet packages, the MS Build tool, and so on and so forth.

Similarly, to deploy a built application on a given infrastructure, the team might employ a configuration management tool such as Ansible or Chef, or Helm if it's a containerized application. And the list goes on if we continue to include all the other stages of a software development life cycle.

Because there is a wide variety of tools and technologies, a good CI/CD tool is expected to integrate well with as many of them as it can.

Jenkins is one such versatile tool that has stood the test of time. Its ecosystem of plugins allows it to integrate with almost all the tools listed on the DevSecOps periodic table [4]:

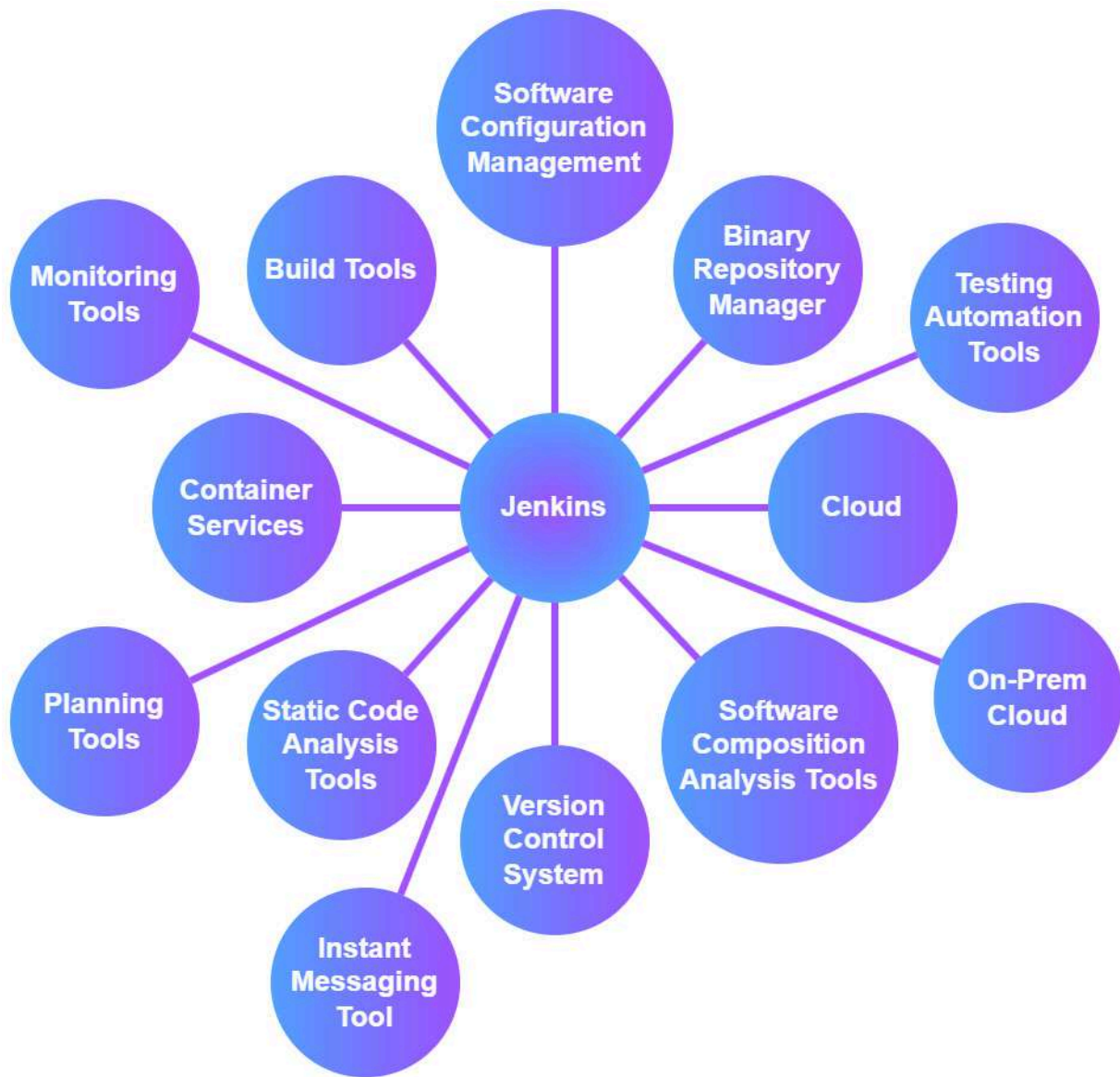


Figure 1.9: Jenkins, a CI/CD tool

Automation through pipelines

Earlier, automation pipelines were first built manually on the CI/CD server using GUI-based parameters. But, as development teams grew and automation pipelines proliferated, engineers found it increasingly challenging to maintain them, because the only method to back up and version the pipelines was to create a backup of the whole CI/CD tool. *Jenkins' Freestyle Job* and *TeamCity's Build* are two such examples.

But thanks to the **pipeline-as-code** movement, automation pipelines can now be defined as code, making it possible to version-control them alongside source code. And not just CI/CD pipelines; describing your complete infrastructure as code is also a reality.

Pipeline as code allows you to share, examine, and create longer automation pipelines. Furthermore, to reduce redundant pipeline code, it is even possible to compose reusable pipeline components as libraries using the **Shared Libraries** feature in Jenkins.

Pipeline code, particularly Jenkins pipeline code, is a **DSL** (short for **domain-specific language**) built on top of the Groovy programming language. The pipeline code syntax could be scripted or declarative, and always follows a structure. Pipelines with scripted syntax are more flexible and extensible than their declarative counterparts, but the latter is more straightforward to write.

However, for CI/CD purposes, it is always recommended to use the declarative pipeline syntax. We will learn more about pipelines in the upcoming chapters.

Pipeline visualization and notification

If your pipeline run fails for some reason, then you might want to know at what stage or step it failed. When you find where it failed, you would then want to know why it failed. And that's exactly what the pipeline visualization feature in any CI/CD tool is meant for.

A good pipeline visualization system helps you pinpoint the actual failed stage or step of a pipeline in an intuitive manner. It also segregates pipeline logs by stages and steps, so you don't have to scroll through the entire pipeline log.

A good notification system, on the other hand, allows you to get quick feedback about the status of your pipeline run. This is done by adding notification triggers inside your pipeline code as a post-step after every stage. Therefore, if the pipeline starts to fail, you immediately get either an email, a message on the chat app, or even a ticket on the Jira dashboard.

Parallelizing through distributed builds

We already know that a pipeline is separated into stages – for example, the build, unit testing, and SCA stages, followed by publishing to a binary repository.

These stages, particularly in Jenkins, can run in either sequence, parallel, or a mix of the two.

Furthermore, each stage of a pipeline requires a machine to execute on, which may be a VM on VMware or the cloud, or a container operating on a Docker host or a Kubernetes cluster.

When you put CI into practice, every commit to your source code repository activates a pipeline on Jenkins. As a result, at any given time, there are several stages of various pipeline runs vying for a machine to operate on. As you can expect, a farm of machines is required to complete all the stages.

In Jenkins, pipeline stages are executed on a cluster of build agents called a **build farm** that is connected to the Jenkins server rather than on the Jenkins server itself. The build farm might be a

cloud provider, on-premises VMware equipment, a Docker host, or a Kubernetes cluster. This is called the Jenkins Distributed Build architecture (it is covered in detail in the present book):

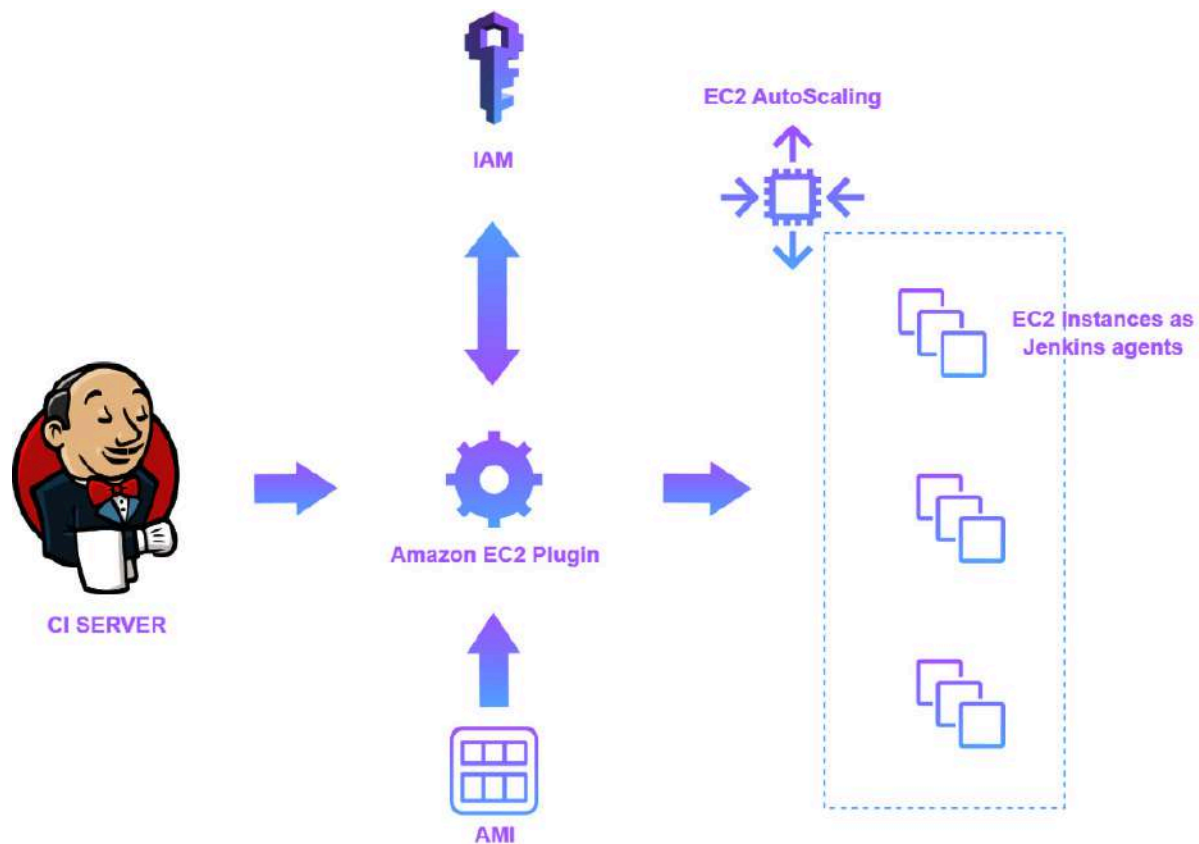


Figure 1.10: AWS EC2 instances as on-demand Jenkins agents

The preceding diagram is an example of a Jenkins build farm built using AWS EC2 instances that auto-scale on demand. Similarly, the following is an example of a Jenkins build farm built using a Kubernetes cluster.

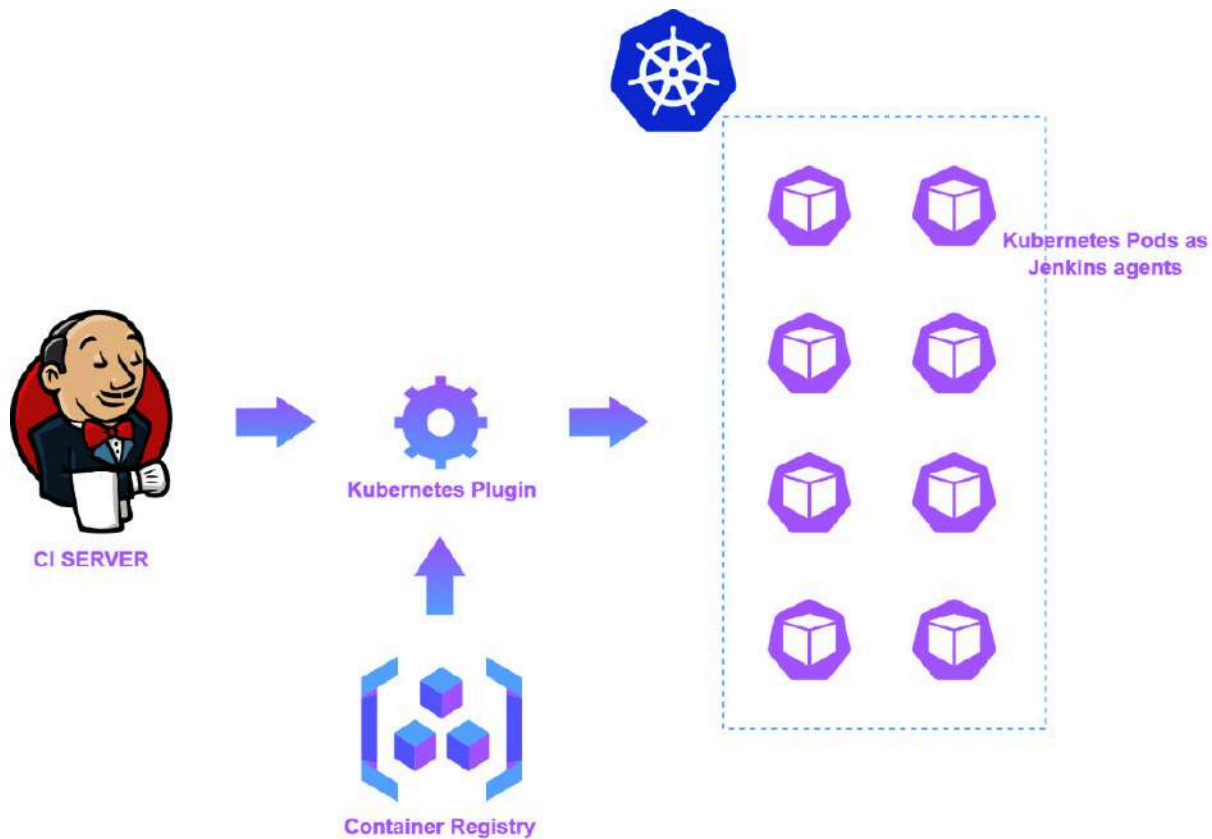


Figure 1.11: Kubernetes pods as on-demand Jenkins agents

This brings us to the end of this part. We've discussed the practice of using a version control system, building every code commit, emphasizing using unit testing and static code analysis to check the quality of the code, using a binary repository manager to track builds, executing builds across many computers in parallel, and utilizing a CI/CD tool to save time and effort. Everything covered in this section constitutes the various steps of a CI pipeline that fulfills the key principles of CI.

Next, let's look at why we should utilize CI.

Why is continuous integration crucial?

The answer is straightforward: CI practices are critical because they enable teams to fulfill modern software development demands. But what are these demands, what benefits of CI help fulfill these demands, and how do they do so? This is what we'll look at in this section.

A faster time to market (TTM)

Also known as lead time or time to delivery, TTM is the time it takes a feature to get from its origin (typically a code change) to deployment in production. A higher TTM number indicates that features are being developed, tested, and delivered more slowly and infrequently. Most current software programs and services need more frequent software upgrades. So, let's see some advantages of CI/CD that help achieve a lower TTM value:

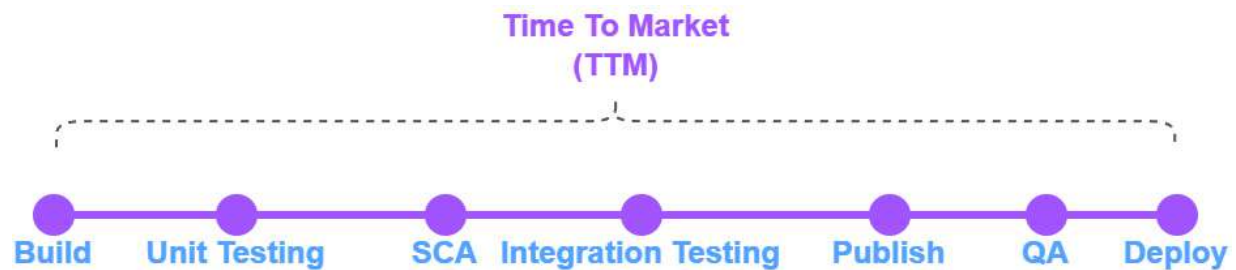


Figure 1.12: Time to market (TTM)

Freedom from long integration – avoiding tolls

CI advocates for regular integration of your work with the main code. This method, as we previously observed, helps engineers to avoid merging issues that might otherwise take weeks to fix. With a very simple practice, we save weeks of time.

Rapid features development – velocity

CI, from a technological standpoint, helps teams operate more effectively. Teams following a CI practice employ an automated and continuous approach for developing, testing, and integrating code changes, resulting in speedier development. This allows developers to spend more time focusing on their code while the CI automation does the rest.

Furthermore, the mindset of failing fast and attempting different approaches to develop a feature, along with constant feedback loops from throughout the CI and CD pipeline, enables developers to be agile.

Developing more features – volume and variety

In using a trunk-based development model, every commit is peer-reviewed, built, tested, analyzed, and deployed, assuming it passes all stages of the CI and deployment pipeline. Also, having a distributed build architecture in place allows pipelines to run in parallel. As a result, every commit is a potential feature release candidate.

Furthermore, employing a globally available distributed version control system in conjunction with a globally accessible CI server allows geographically distant teams to collaborate on a bigger project and deliver more features.

Improved software quality

The **change failure rate (CFR)** metric is a nice way to describe the quality of your code. It refers to the percentage of features deployed in production that cause downtime or a bug resulting in a rollback or hotfix. A lower CFR percentage points toward a robust and effective CI process. In other words, the pipeline is effective in detecting quality issues as early as possible. So, let's see some advantages of CI/CD that help achieve a lower percentage of CFR.

Built-in quality

A rigorous code review process enables developers to identify possible code quality concerns before changes are merged into the main code base. Code review, often known as peer review, is the process of having your code changes reviewed by a more senior developer before merging them into the main branch.

Furthermore, by performing static code analysis on the developer's local branch, code changes may be validated against quality criteria such as readability, maintainability, complexity, duplication, size, and reliability before the code changes are integrated into the main branch. Of course, SCA occurs once more during the CI pipeline run on the main branch. These CI practices allow quality to be built in.

Early defect detection

As we saw previously in the chapter, automated testing or unit testing allows you to test the smallest testable component of your software as quickly as possible. It also enables the detection of errors as early as feasible, before they are identified in the QA environment, staging, or production. Unit tests are simple to develop and pay a higher dividend.

Thus, by considering quality as a built-in capability of the code and spotting quality concerns as near to development as feasible, using affordable automated unit testing, software development teams may obtain a lower CFR percentage.

Highly secure applications

“If you cannot measure it, you cannot improve it” – Lord Kelvin. Software application security is something very similar. It is an imperceptible and tough-to-measure entity.

Security risks can only be anticipated and speculated until a breach occurs, and all existing security measures are built against such anticipations and inferences. Therefore, CI practices – or any other practices, for that matter – cannot guarantee that an application is completely secure, but they do help you prepare well. Let's find out how.

Continuous security – DevSecOps

The culture of thinking about security throughout the software development life cycle as a collaborative effort, as opposed to something that is done by the security team in the end, is known as **continuous security**, also popularly known as **DevSecOps** (short for **development, security, and operations**). The term continuous security is much better, as it sounds more actionable.

The continuous security methodology encourages adding security tests not only when developing the application, but also while performing QA checks on your source code during the CI phase.

It's reasonable for us to suggest that if we extend this concept, we also need to subject the infrastructure and pipeline code to security checks:

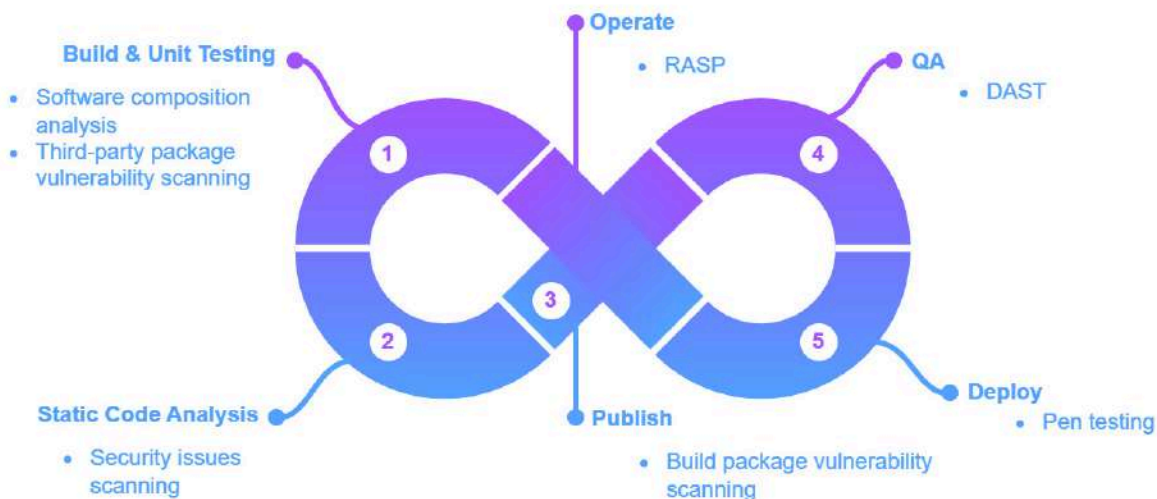


Figure 1.13: Stages of security testing in a CI/CD pipeline

Scanning third-party dependencies

Whatever you choose to call them – software dependencies, software libraries, or third-party binary packages – developers frequently use them to speed up the development process of their applications.

These third-party libraries are one such point of entry for malicious programs to enter your system. However, the CI practice of using a binary repository manager, such as Artifactory, enables you to avoid downloading questionable third-party packages by identifying and tagging them directly in the package repository through using an additional tool, JFrog Xray, that is a part of the same suite as Artifactory.

Static code analysis

While scanning third-party packages is an excellent thing to do, what about identifying security flaws that get carelessly introduced during the development process by the developers themselves? This includes things such as stored passwords, API authentication tokens, and so on. This is precisely what the CI practice of checking your code for quality (SCA) accomplishes; it searches for security-related vulnerabilities in addition to other quality tests.

Software composition analysis

SCA is the practice of automatically identifying the open source components inside your source code. This analysis mainly checks for license violations and security evaluation. A tool such as Black Duck is one such example that can be integrated with Jenkins to perform SCA on your source code as part of the CI pipeline.

In this way, you can check open source components in your source code early, as they are one of the ways your application could be compromised.

Cost-effective system

Technology will only continue to become more integral to business in the future, especially as consumer demands rapidly shift to digital models. IT can serve as a direct line from innovation to greater profitability and market share in this transition. (Vation Ventures [5])

Companies' spending tends to rise as software becomes more and more essential to their core operations, and ultimately, they start considering the cost factors associated with software development.

Almost every facet of the CI and CD methodology helps to reduce the cost of software development without sacrificing quality, security, or speed. Let's examine how.

Automation using pipelines

Automation using pipelines, notably pipeline as code and infrastructure as code, enables development teams to spend less time maintaining the CI/CD platform and its underlying infrastructure and more time building features.

The developers just need to ensure that their pipeline and infrastructure code are error-free. This significantly reduces maintenance overhead, saving money and, more crucially, time.

Pinpointing failures with ease

CI practice advocates using a CI/CD tool to orchestrate CI pipelines. As previously said, a good CI/CD tool has an informative dashboard and an intuitive pipeline visualization system that assists the developer in determining the exact failed stage or step of a pipeline. It also divides pipeline logs into stages and phases, saving the developer from having to read through the full pipeline log. This allows faster debugging of issues, eventually saving time and cost.

Configuration as code

Configuration as code can refer to either an application's or a machine's configuration being written as code. Configuration as code is the practice of defining the entire configuration of an application or a machine as code through a configuration file, usually a JSON or YAML file.

For example, let's consider Jenkins as our application. We can express the Jenkins server's complete configuration as code by utilizing a feature in Jenkins known as JCasC. This functionality enables engineers to swiftly start up a backup Jenkins server in the event that the original server fails due to a problem. This allows pipelines to restart without additional delay, saving time and resources and ultimately saving money.

Machine configuration as code is popularly known as software configuration management. Some of the most prominent software configuration management technologies include Ansible, Chef, and Puppet. They enable configuration management across hundreds of computers using a single recipe file and are also used to deploy programs in production.

Though container technologies have mostly eclipsed them in recent years, they remain relevant to projects that run their programs on Windows- or Linux-based compute servers. This method significantly reduces operational overhead by paving the way for the automation of deployment and rollback processes on hundreds of production computers, saving time and resources and ultimately saving money:

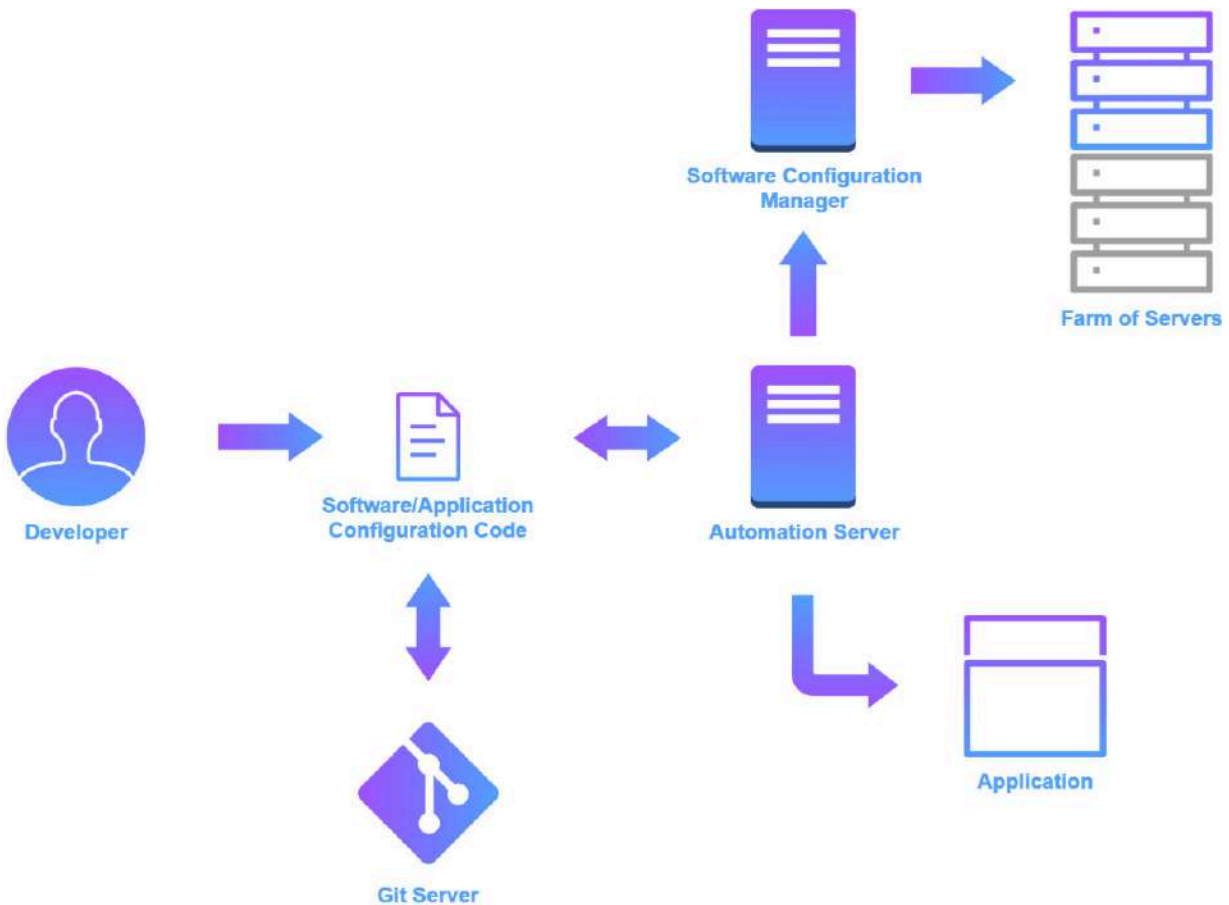


Figure 1.14: Software and application configuration as code

This brings us to the end of this section. We’ve learned how the CI approach can help teams achieve a faster time to market and higher software quality, and build highly secure software applications in a cost-effective manner.

Summary

We began this chapter by studying the basics of CI and understanding its main principles. This was followed by a discussion of the numerous practices promoted by CI, as well as a look at the CI ecosystem of DevOps tools such as SonarQube, Git, and Artifactory. Finally, we learned how CI assists us in meeting the needs of modern software development. We learned a lot about several

crucial CI/CD metrics that are used to assess the performance of CI and the productivity of software development while doing so. This taught us the *what*, *how*, and most importantly the *why* of CI.

The subsequent chapters of the book offer a practical step-by-step guide to the *how* of CI, that is, how to implement CI. And now that the concepts are clear, let's start the very next chapter by learning how to set up and configure a Jenkins server.

Questions

1. Which of the following best defines **continuous integration (CI)**?
 - A. A software development practice that advocates the integration of code changes into a shared repository, enabling frequent testing and early detection of integration issues
 - B. A methodology for deploying software frequently and reliably, ensuring that new features and updates are delivered to users quickly and seamlessly
 - C. A tool or platform that provides a centralized environment for managing code changes, builds, and tests, facilitating collaboration among developers
 - D. A mindset and set of practices that emphasize continuous improvement, rapid feedback, and the delivery of high-quality software in short cycles
2. Which of the following best describes the concept of *fail fast* and *shift left* in software development?
 - A. Identifying and fixing bugs early in the development process to minimize their impact and reduce costs
 - B. Releasing software frequently with minimal testing to ensure rapid delivery and market responsiveness
 - C. Implementing comprehensive testing and validation procedures at the end of the development cycle to guarantee software quality
 - D. Postponing bug fixes and quality checks until after software release to focus on features and functionality
3. Which of the following best describes the key steps involved in achieving CI in a software development project?
 - A. Implementing a version control system to track code changes and maintain a shared repository
 - B. Automating the build and test process to ensure frequent integration and early detection of issues
 - C. Establishing a culture of collaboration and continuous feedback among developers
 - D. All of the above
4. Which of the following is not a potential benefit of adopting CI in software development?
 - A. Increased productivity and efficiency due to automated testing and rapid feedback
 - B. Improved software quality and reliability through the early detection and fixing of bugs
 - C. Reduced time to market for new features and updates
 - D. Decreased need for manual testing and code review

5. Which of the following best describes the primary role of a CI tool in CI?
- A. To automatically merge code changes into a shared repository
 - B. To orchestrate the build, unit testing, and packaging of software code
 - C. To automate the build process
 - D. To provide a platform for storing and managing binary artifacts

Answers

1. Correct answer: **A**

Explanation of incorrect answer choices:

B. While CI is often associated with CD, CI's primary focus is on automating the integration and testing of code changes, not on the deployment process. CD, on the other hand, deals with automating the deployment of software to the staging and production environments.

C. CI tools and platforms are essential to implementing a CI practice, but they are not the defining characteristics of CI. CI is a broader concept that encompasses the practices, principles, and mindset of integrating code changes frequently and automating testing to ensure early detection of issues.

D. While CI does promote continuous improvement, rapid feedback, and the delivery of high-quality software, these are not the primary goals of CI.

2. Correct answer: **A**

Explanation of incorrect answer choices:

B. While releasing software frequently is a goal of many development teams, it should not come at the expense of thorough testing and bug fixing. Fail fast and shift left emphasize identifying and resolving issues early, not releasing untested code.

C. Comprehensive testing is essential for ensuring software quality, but it is most effective when done throughout the development process, not just at the end. Fail fast and shift left advocate for shifting testing activities leftward, meaning testing earlier and more frequently.

D. Postponing bug fixes and quality checks until after software release is counterproductive to the fail fast and shift left principles. Early detection and resolution of issues are crucial to minimize their impact and prevent them from snowballing into larger problems.

3. Correct answer: **D**

Explanation:

While each of the individual steps is crucial to achieving CI, the true effectiveness of CI lies in the synergy between them. A comprehensive CI approach encompasses all three aspects: version control, automated testing, and a collaborative mindset.

4. Correct answer: **D**

Explanation:

While CI does automate testing and code review processes, it doesn't eliminate the need for these activities altogether. Instead, it complements these processes by providing an automated foundation and facilitating more efficient and effective manual testing and code review.

5. Correct answer: **B**

Explanation of incorrect answer choices:

A. This is not the primary role of CI tools. While CI tools often trigger builds based on code changes in a repository, the actual merging of code is a function of version control systems, not CI tools.

C. This is a partial but incomplete description of the role of CI tools. Automation of the build process is an important aspect, but CI tools also extend beyond building to include testing and packaging, making their role more comprehensive.

D. Managing binary artifacts is typically handled by artifact repository managers such as JFrog Artifactory or Sonatype Nexus. While CI tools might come with some built-in systems to store artifacts, it is not their primary function.

Further reading

If you wish to know more about some of the concepts that weren't covered in depth in this chapter, we are happy to provide you with some references:

- [1] TED: *How great leaders inspire action* | Simon Sinek: https://www.ted.com/talks/simon_sinek_how_great_leaders_inspire_action
- [2] Static code analysis; *Metric definitions* | SonarQube: <https://docs.sonarqube.org/latest/user-guide/metric-definitions/>
- [3] Rules Overview: | SonarQube: <https://docs.sonarqube.org/9.7/user-guide/rules/overview/>
- [4] DevSecOps Periodic Table: <https://digital.ai/learn/devsecops-periodic-table/>
- [5] *Why every company needs to think like a tech company* | Vation Ventures: <https://www.vationventures.com/blog/thinking-like-a-tech-company>

Part 2: Engineering the CI Ecosystem

In the second part, we will create the groundwork for **Continuous Integration (CI)** with a rigorous Jenkins setup, built not just to withstand the test of time but also to adapt and thrive in the ever-changing landscape of software development. Our strategy starts with a complex Jenkins server architecture designed to balance the demands of operational efficiency, cost-effectiveness, and security. We will use infrastructure as code and Jenkins configuration as code to build a strong and flexible platform.

As we secure our manufacturing assembly line, with the precision of a locksmith, we will integrate sophisticated user authentication systems and delve into the nuances of permissions and credentials. Our goal is simple – to turn Jenkins into a fortress, protecting the critical data that passes through the veins of our CI operations. We will then turn to the extra DevOps technologies that augment Jenkins' capabilities – SonarQube for code quality, Artifactory for artifact management, and version control systems for collaborative harmony.

Finally, we ascend to the clouds, embracing virtual machines and containers' limitless possibilities. This is where our CI infrastructure takes flight, delivering a level of scalability that responds to the flow of demand with agility. We will provide our CI system with the ability to expand and contract in tune with the pulse of development needs, by orchestrating the dance of cloud services and container orchestration.

This part has the following chapters:

- [Chapter 2](#), *Planning, Deploying, and Maintaining Jenkins*
- [Chapter 3](#), *Securing Jenkins*
- [Chapter 4](#), *Extending Jenkins*
- [Chapter 5](#), *Scaling Jenkins*

Planning, Deploying, and Maintaining Jenkins

While the first chapter described the importance of using a CI tool, among other things, this chapter describes how to plan, deploy, and maintain one. Our goal here is to design and deploy a resilient, cost-effective, secure, high-performing, and operational Jenkins setup.

We will accomplish this by looking at the Jenkins server architecture, followed by evaluating a few Jenkins server deployment scenarios against the **Well-Architected Framework** (see [1] in the *Further reading* section at the end of this chapter).

Following that, we will cover the two most popular ways of deploying Jenkins and learn how to implement them step by step. While doing so, we will incorporate some key DevOps practices such as **Infrastructure as Code (IaC)**, and a Jenkins-specific approach called **Jenkins Configuration as Code (JCasC)**. Finally, we'll go through the Jenkins server's maintenance.

The CI ecosystem is larger as it contains not just the Jenkins server but also other DevOps tools. The other aspects of the CI ecosystem will be presented gradually through the next few chapters of this book.

After completing this chapter, you should be able to do the following:

- Understand the Jenkins server architecture and system requirements
- Plan a Jenkins setup by evaluating deployment scenarios and infrastructure choices
- Gain proficiency in JCasC to author and apply initial server settings for Jenkins
- Deploy your Jenkins setup on a managed Kubernetes cluster
- Deploy your Jenkins setup on a cloud instance
- Acquire knowledge on key considerations for upgrading and backing up a Jenkins server to ensure a smooth and reliable process

Technical requirements

This chapter is divided into three parts: **planning**, **deploying**, and **maintenance**. Except for the deploying part, the other two parts do not have any technical requirements. For detailed technical requirements for the deploying part, refer to the following sub-sections located under the step-by-step guides for deploying your Jenkins setup:

- *Preparatory steps*
- *Local development environment prerequisites*

These sections offer detailed information and instructions regarding the necessary technical prerequisites. Furthermore, you can access the code referenced in this chapter by visiting the following GitHub repository: https://github.com/PacktPublishing/Learning-Continuous-Integration-with-Jenkins_Third-Edition.

Planning your Jenkins setup

Planning your Jenkins setup before deployment is crucial to ensure optimal resource allocation, scalability, security, customization, and cost optimization. It sets the foundation for a stable and efficient CI/CD system that aligns with your organization's specific needs and accommodates future growth.

Within this section, you will discover a comprehensive guide on effectively planning your Jenkins setup. So, let's begin.

Understanding the Jenkins server architecture

The Jenkins design is based on a distributed approach, with the master node coordinating the build process and delegating real work to the agent nodes. *Figure 2.1* illustrates this in detail:

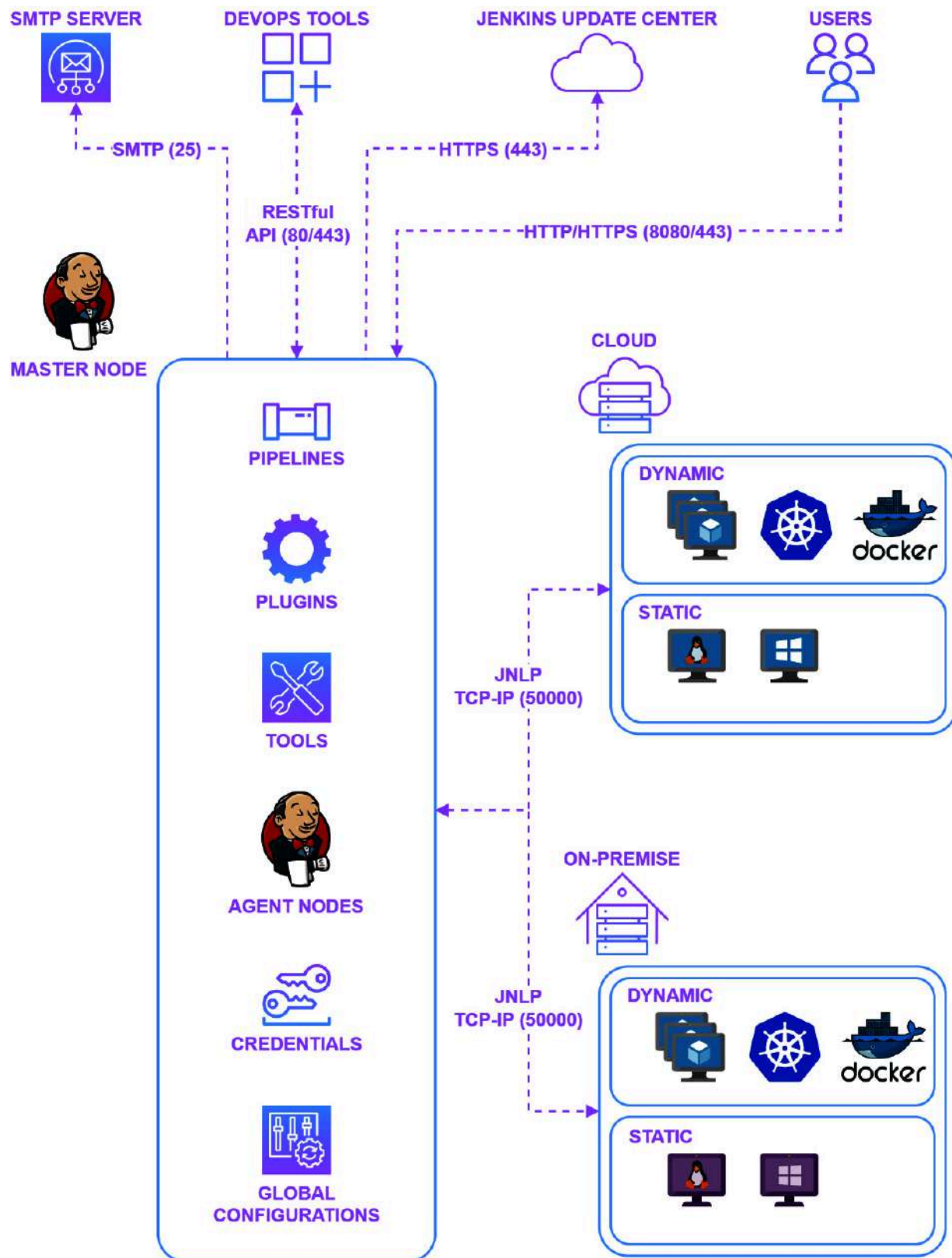


Figure 2.1 – The Jenkins server architecture

Let's go through some of the key components of the Jenkins server design depicted in the preceding figure:

- The Jenkins architecture is centered on the **master node** (also known as the Jenkins controller), or the Jenkins server itself. It runs the whole system and oversees key activities such as scheduling, distributing Pipeline jobs, monitoring agents, and providing a web interface for user interaction. The master node is responsible for coordinating the CI Pipeline process.
- **Agent nodes** (also known as worker nodes) are in charge of carrying out the Pipeline builds allocated to them by the master node. They might be containers or **virtual machines (VMs)** with different operating systems and configurations. Additionally, agent nodes can be dynamic (created during the Pipeline run and destroyed afterward) or static (pre-created and persistent). Jenkins plugins allow us to connect the agent nodes to the master node. Multiple agent nodes can be linked to a single master node, allowing builds to be executed simultaneously.
- Jenkins has a robust ecosystem of **plugins** that enhance its capabilities. Installing and configuring plugins allows you to interface with a wide range of DevOps tools, technologies, and services. They offer capabilities such as source code management, notifications, testing frameworks, and deployment options, among others.
- Jenkins supports **Pipelines**, which are sequences of interconnected stages that make up the CI/CD process. You can create a Jenkins Pipeline either using code or through the Jenkins user interface. A Pipeline often consists of numerous stages such as build, unit test, static code analysis, and publish, and a Pipeline run may be viewed and controlled using Pipeline plugins for Jenkins such as Blue Ocean. A stage is a specific operation that the Jenkins Pipeline performs, such as generating code, performing tests, or deploying an application.
- The **credentials** feature in Jenkins is used to securely store sensitive data such as passwords, API tokens, SSH keys, and other authentication credentials required by Jenkins Pipelines and plugins. The credentials feature prevents sensitive data from being exposed in plain text inside Pipeline code or build logs. The Jenkins web interface is often used to manage credentials. The credentials page on the Jenkins user interface allows you to create, edit, and remove credentials. Credentials can be assigned to specific jobs or shared across numerous jobs and plugins.
- Jenkins **Global Tools** is a feature that allows Jenkins administrators to install and configure tools globally across the Jenkins environment. These tools can include version control tools, build tools, static code analysis tools, or any other software required by the Pipeline. By installing tools globally, they can be easily accessed by Pipeline stages running on any Jenkins agent node. Administrators can specify tool installations in the Jenkins settings, and Pipelines can reference them using environment variables or specific steps in the Pipeline code. Global Tools streamlines the setup process and ensures consistency across build environments.
- Jenkins **Global Configuration** refers to the centralized settings and configurations that apply to the entire Jenkins instance. It includes various system-wide settings such as security settings, email notification configurations, global tool installations, plugin management, and other administrative preferences along with plugin configurations. Global Configuration allows administrators to define and manage these settings in one place, ensuring consistent behavior across all jobs and agents. It provides a convenient way to configure Jenkins at a global level, reducing the need for repetitive configuration tasks for individual jobs or agents. JCasC is a way to define all this Jenkins configuration using code; we will learn about this in the upcoming section of this chapter.

JCasC takes this even further. It is a utility for defining all the Jenkins configurations using code right before you set up your Jenkins server. We will learn more about it in the upcoming section of this chapter.

Jenkins system requirements

In this section, we'll explore the hardware requirements, network connectivity requirements, and **Java Virtual Machine (JVM)** configurations that are necessary for the optimal performance of the Jenkins server. By comprehending the intricacies of the Jenkins server's system requirements, administrators can make informed decisions when deploying Jenkins in different scenarios.

Determining optimal hardware requirements

Answering the hardware requirements of Jenkins is quite a challenge since it depends on what you use it for and how you use it. *Table 2.1* shows the bare minimum hardware requirements for running a Jenkins server:

CPU	A multi-core processor with a clock speed of at least 2 GHz
Memory (RAM)	A minimum of 2 GB of RAM
Disk space	A minimum of 10 GB

Table 2.1 – Jenkins server minimum hardware requirements

These requirements are sufficient if you only have a few Pipeline jobs and a few developers using Jenkins. However, as your use of Jenkins matures, you will ultimately begin to seek some formulae that provide you with the right hardware requirements. That is exactly what we shall find out in this section. Let's begin with the things that we know impact the Jenkins hardware.

The CPU is affected by the number of HTTP/HTTPS requests made to the Jenkins master at any given moment. These HTTP/HTTPS requests might come from users accessing your Jenkins master or via API calls made to your Jenkins master through webhooks from the version control tool to trigger builds. The following formula can be used to compute the estimated amount of CPU cores necessary:

$$\text{Number of CPU cores} = \text{Number of https requests} / 250$$

For instance, if your Jenkins server is powered by a quad-core processor and roughly 100 developers are using your Jenkins dashboard with 100 build triggers, your CPU use will be around 20% for that very moment.

Memory (RAM) is affected by the amount of agent nodes connected to your Jenkins master. The following formula can be used to compute the estimated amount of memory in megabytes necessary:

$$\text{Amount of memory (MB)} = \text{Number of agent nodes} * 3$$

So, if you have 100 agent nodes connected to your Jenkins master operating with 2 GB of RAM, your memory use will be around 15% for as long as the agent nodes stay connected.

The Jenkins server's disk utilization is affected by the amount of Jenkins Pipeline Jobs and their associated builds that you choose to keep. This is exacerbated if the build logs are large. It should be noted that the Jenkins master stores the build logs for Jenkins Pipelines. The following formula can be used to compute the estimated amount of additional disk space in megabytes necessary:

*Additional disk size (MB) = Number of Pipeline jobs * Number of builds (retained) * Average build logs size (MB)*

It is important to consider that we are referring to the extra disk space needed specifically when Jenkins is running in addition to the initial disk size, which encompasses the space required for the operating system, the Jenkins installation, plugins, and other essential components.

Whether you opt to operate your Jenkins server on VMs or as a container, on the cloud, or on-premises, the preceding formulas should assist you in selecting the appropriate VM sizes. Speaking of the cloud, when considering the best category of VMs for running your Jenkins server, it is advisable to focus on general-purpose VMs [2]. They provide a good balance of CPU and memory (RAM).

Because the Jenkins server handles a large number of longer-lived HTTP/HTTPS connections, prioritizing CPU performance will help guarantee efficient processing. Furthermore, while the Jenkins master does not actively run Pipelines, it does manage the whole Jenkins environment and organizes tasks. Choosing a VM with sufficient memory capacity can assist in meeting the demands of the Jenkins master and any associated services.

IMPORTANT NOTE

The presented formulae are only guidelines; real resource requirements may vary depending on your usage habits and workload. It's always a good idea to keep an eye on your Jenkins master node's resource utilization and scale it up as needed to maintain optimal performance and reliability. Furthermore, we presume that you have established agent nodes to handle your CI Pipeline workloads and that nothing is specified to execute on the Jenkins master. We also presume that there are no build artifacts stored on the Jenkins master.

Network connectivity

The Jenkins master node must have network access to the agent nodes, source code repositories, and any other services or systems that are part of your CI ecosystem. To host a Jenkins server, you must normally expose particular ports to allow communication and access. *Table 2.2* shows the ports that are commonly used by Jenkins:

Port	Description
8080	This port is used to connect to the Jenkins web interface through HTTP. It serves as the primary interface for communicating with Jenkins and managing build tasks.

Port	Description
443	If HTTPS/SSL is enabled for a secure connection with Jenkins, this port is used for HTTPS access to the Jenkins web interface.
50000	This port is used to enable communication between the Jenkins master and the agent nodes.
22	The default port for SSH allows administrators access to the Jenkins VM.

Table 2.2 – Jenkins server ports

It's worth noting that these are the default port numbers; however, you may change them during the Jenkins installation or in the Jenkins configuration files.

You may need to configure your firewall or network security groups to allow inbound connections on the listed port numbers to gain access to these ports. The specific processes for unlocking ports differ based on your operating system, network configuration, and firewall software.

Based on your individual requirements, network environment, and security standards, ensure that the relevant ports are open and accessible.

Ideal JVM configurations

Jenkins is built using the Java programming language, so a **Java Runtime Environment (JRE)** or **Java Development Kit (JDK)** is required. Jenkins supports Java 8 or later versions, with Java 11 being a common choice.

JVM settings, also known as JVM parameters, play a vital role in ensuring the smooth functioning of Jenkins. While the Jenkins server generally performs well without the need for adjusting JVM parameters right from the start, as your Jenkins server expands, the significance of JVM settings becomes increasingly crucial. Here's why:

- Jenkins users frequently claim slowness, even when there is no Pipeline activity. You may improve Jenkins' memory use, performance, and responsiveness by modifying variables such as heap size, garbage collection algorithms, and thread management.
- Specifying adequate memory limits and garbage collection settings will help prevent frequent problems such as **OutOfMemoryError** or excessive garbage collection pauses, which can cause Jenkins to become unusable or crash.
- Logging mechanisms can be enabled to collect precise information on Jenkins' internal activity to fine-tune the JVM for the best efficiency.

Having said that, here are some common JVM settings when running JDK 11:

jvm-options.txt (chopped-down version)


```
-XX:+AlwaysPreTouch
-XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=${LOGDIR}
-XX:+UseG1GC
.
.
.
-XX:ErrorFile=${LOGDIR}/hs_err_%p.log
-XX:+LogVMOutput
-XX:LogFile=${LOGDIR}/jvm.log
```

These JVM options are passed as parameters to the `JAVA_OPTS` environment variable using the `override.conf` file. In the upcoming sections of this chapter, we will delve into the process of utilizing these JVM settings when configuring our Jenkins server. However, for a comprehensive understanding of each of these JVM options, please refer to the JDK 11 Oracle documentation [3].

IMPORTANT NOTE

Experimentation, monitoring, and tweaking may be required to identify the best JVM options to meet your individual needs and ensure Jenkins runs smoothly and efficiently.

Evaluating Jenkins deployment scenarios

In this section, we will shift our attention toward the evaluation of different deployment scenarios for a Jenkins server. Whether you choose to run Jenkins as a container, VM, bare metal, serverless, or opt for cloud or on-premises setups, each approach offers unique advantages and considerations. By subjecting these scenarios to testing against the Well-Architected principles, we will gain insights into their strengths.

Running Jenkins serverless

Serverless technology is still in its early stages, but it has swiftly acquired popularity and shown to be dependable. It provides automated scalability, high availability, and a controlled infrastructure, enabling developers to concentrate on code rather than infrastructure administration. Having said that, regrettably, this book does not provide a working example of running Jenkins serverless.

However, there are two major benefits of running Jenkins serverless in the cloud. First, it removes the need for server management and infrastructure provisioning. It also eliminates operational overhead, resulting in increased productivity and a shorter time-to-market. Second, cloud-based security measures protect Jenkins from threats and vulnerabilities. A score of 9 for security and 9 for operational excellence supports our arguments, as shown here:

Serverless on Cloud



Serverless on Premises



Figure 2.2 – Evaluating serverless Jenkins on the cloud and on-premises against the Well-Architected Framework

Also, note that the on-premises option does not score well against the cloud.

Running Jenkins as a container

Running Jenkins on a cloud-managed Kubernetes cluster is the best option. For instance, it offers scalability and elasticity, allowing Jenkins to dynamically adjust resources based on workload demands because containers are easy to scale both vertically and horizontally.

Second, it guarantees high availability and fault tolerance thanks to Kubernetes' built-in redundancy and automated recovery techniques. Third, it simplifies infrastructure management as the cluster is managed by the cloud provider, reducing administrative tasks.

Finally, because containers share operating systems and hardware, it allows for effective resource usage, lowering expenses. Running Jenkins in a container is, in reality, the most cost-effective choice. As indicated in the following figure, a score of 9 for performance efficiency and reliability and an 8 for cost optimization and operational excellence back up our claims:

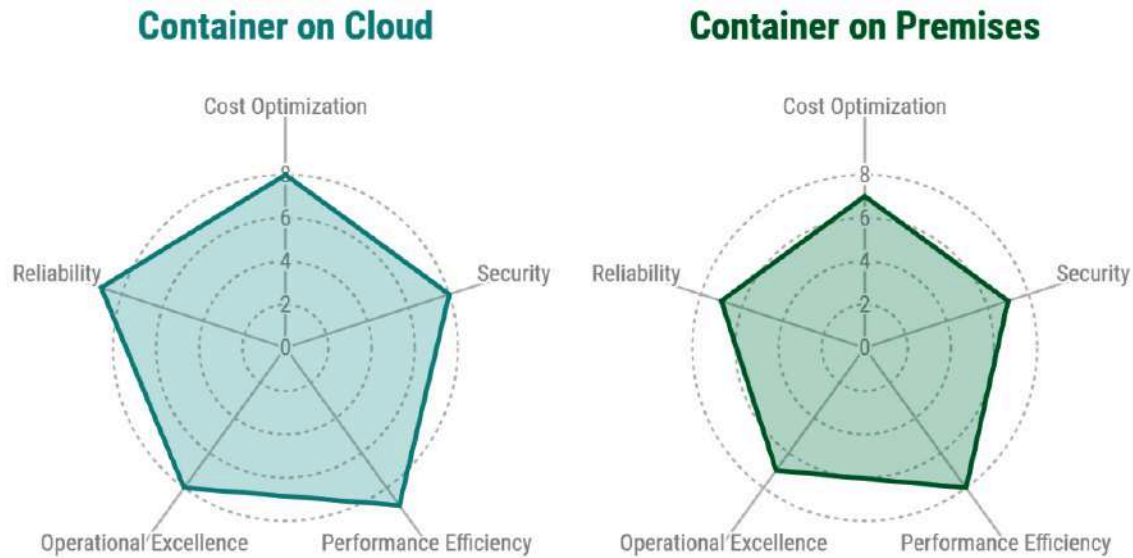


Figure 2.3 – Evaluating containerized Jenkins on the cloud and on-premises against the Well-Architected Framework

And, once again, the on-premises solution falls short of the cloud.

Running Jenkins on a VM

This solution is popular among businesses looking to migrate their on-premises Jenkins setup to the cloud. Compared to its serverless and containerized cloud contenders, the advantages of running Jenkins on a cloud-based VM appear minimal, but this solution delivers considerable benefits compared to the on-premises-based arrangement. Let's have a look at the two key advantages.

First, running Jenkins on a cloud-based VM provides vertical scalability. Cloud solutions enable you to effortlessly scale up or down VM resources based on demand. This scalability is valuable during peak hours when you may require more resources to handle a greater number of user and API requests paired with a greater number of agent node connections to the Jenkins master.

Second, despite running Jenkins on a cloud VM seeming less cost-effective than a serverless or container-based solution, it is far superior to its on-premises counterpart. With cloud services, you always pay for the resources you use, allowing you to optimize costs based on your Jenkins usage patterns. It eliminates the need for upfront hardware investments and reduces maintenance and operational costs associated with managing physical infrastructure. The following figure shows a comparison between running Jenkins inside a VM on the cloud versus on-premises. The cloud option is better:

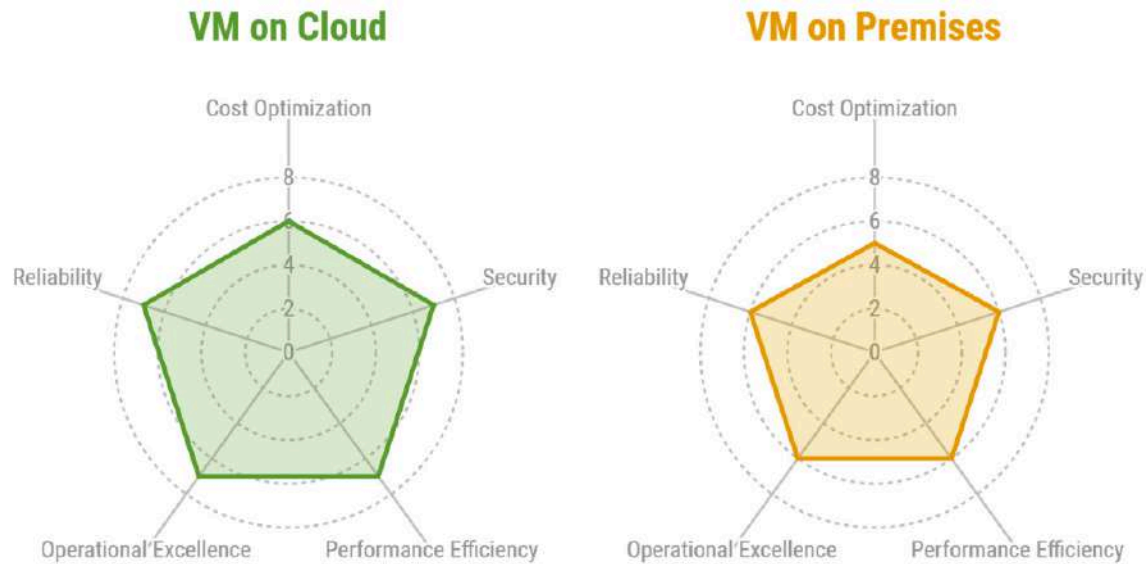


Figure 2.4 – Evaluating VM-based Jenkins on the cloud and on-premises against the Well-Architected Framework

Running Jenkins on bare metal

Running Jenkins on a physical server seems unconventional in the presence of cloud services, serverless architecture, and containers. However, this solution is by far the most secure and high-performing, which is why it is worth addressing. The only example we can think of is an organization that demands super-high security, such as an intelligence agency. Nevertheless, let's have a look at the three primary benefits.

For starters, it allows direct access to actual hardware, improving performance and lowering virtualization overhead. Second, it provides enhanced security by reducing the attack surface compared to shared virtualized environments. Finally, it enables enterprises to take advantage of cloud features such as scalability, high availability, and on-demand resource provisioning while maintaining the advantages of dedicated bare metal infrastructure. Our assertions are supported by a score of 9 for performance efficiency and security, and an 8 for reliability and operational excellence, as shown here:

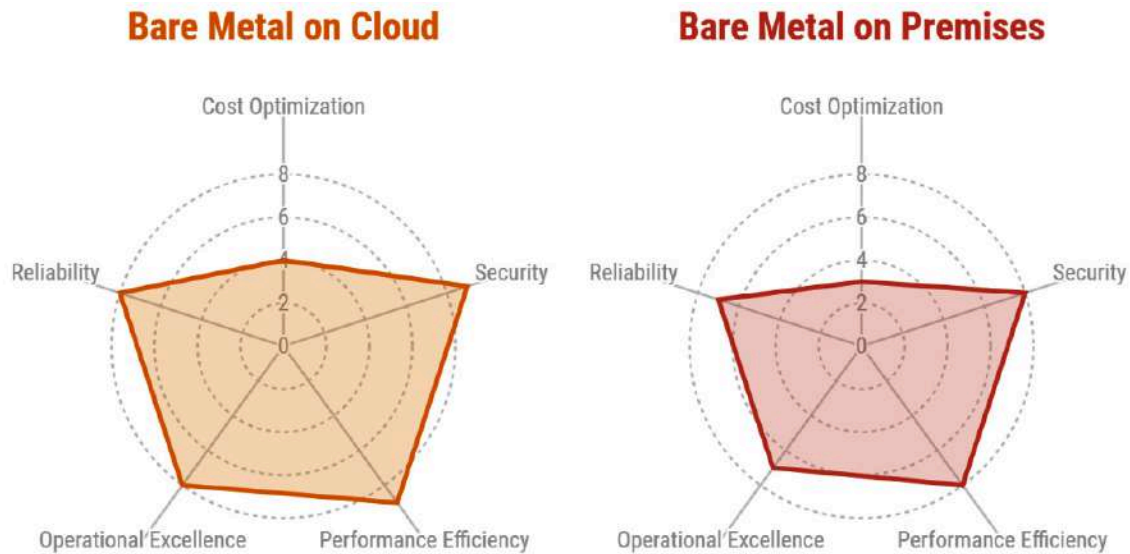


Figure 2.5 – Evaluating bare metal-based Jenkins on the cloud and on-premises against the Well-Architected Framework

As you may have noticed by looking at *Figure 2.5*, running Jenkins on bare metal is also the most expensive solution, with the on-premises solution considerably more so with a score of 3.

The following table provides concise summaries of the key deployment scenarios we’ve discussed:

Deployment scenario	Use case
Jenkins as a container on a cloud-managed Kubernetes cluster	The ideal setup for teams looking for a highly scalable, reliable, cost-effective, and manageable solution.
Jenkins on a cloud-based VM	The best bet for teams transitioning from an on-premises-based setup to a cloud setup but aren’t yet comfortable with containers.
Jenkins on an on-premises VM	The best option for teams looking for data control, network latency, and long-term cost concerns.
Jenkins on a cloud-based bare metal	The best alternative for teams who prioritize security above all at any cost.
Jenkins serverless in the cloud	A reliable approach that outperforms running Jenkins as a container on a cloud-managed Kubernetes cluster. However, this is only if your team has the time and resources to explore.

Table 2.3 – Summaries of the key Jenkins server deployment scenarios

IMPORTANT NOTE

Please bear in mind that these are general ratings that may not apply in every situation. The actual scores for each combination might differ depending on factors such as cloud providers, hardware configurations, network infrastructure, security measures, and operational procedures used.

Understanding JCasC

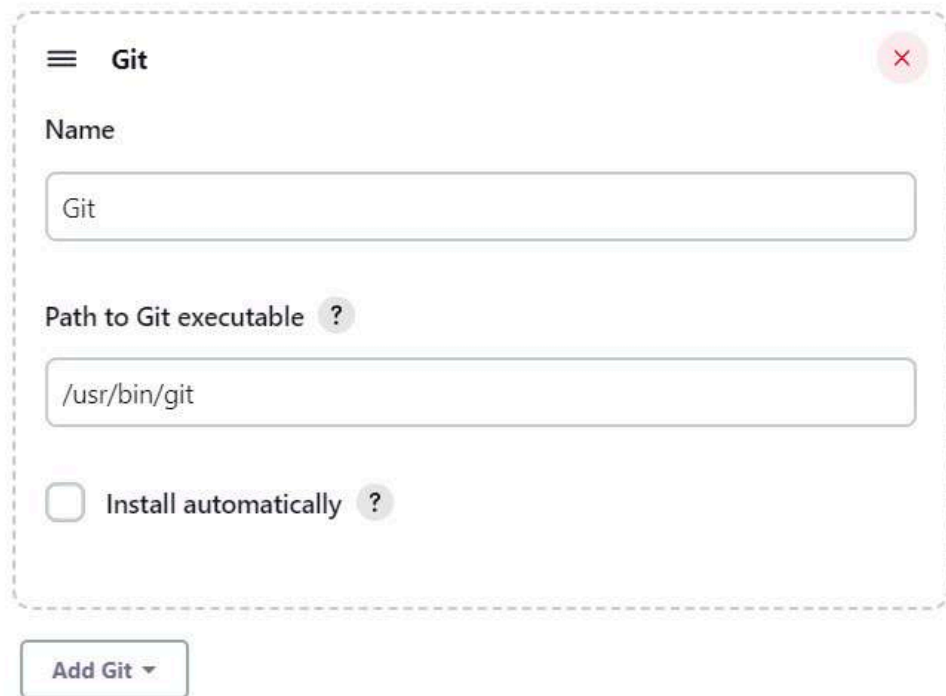
The idea of application configuration as code using XML has been around for several decades. XML-based configuration files became widely adopted in the early 2000s, particularly in enterprise software development. However, with the growing prominence of IaC in recent times, application configuration as code has once again gained significant traction among the masses, and so has JCasC. This section will teach you about JCasC and how it functions behind the scenes.

What is JCasC?

JCasC is a Jenkins plugin designed specifically to streamline the Jenkins configuration process. By reading Jenkins-specific configuration code and automatically applying it to the Jenkins server, this plugin eliminates the need for manual configuration changes, providing a more efficient and streamlined approach.

JCasC adheres to the YAML syntax, with each global configuration element on the Jenkins server having a corresponding declarative code. The following screenshot, for example, depicts the Git tool settings within Jenkins:

Git installations



Git

Name

Git

Path to Git executable ?

/usr/bin/git

☐ Install automatically ?

Add Git ▼

Figure 2.6 – Git installation configuration inside Jenkins

And here's the equivalent JCasC:

```
jenkins:
  tool:
    git:
      installations:
        - name: git
          home: /usr/bin/git
```

JCasC applies not only to core Jenkins configurations but also to numerous essential Jenkins plugins, which provide their corresponding configuration codes. To see the full list of available JCasC templates, visit the JCasC GitHub repository [4].

How does JCasC work?

A JCasC file is explicitly supplied as a value to the `CASC_JENKINS_CONFIG` variable during the Jenkins server startup procedure. The value can be a permanent link to a JCasC file from a source code repository, a full path to a JCasC file, or a path to a folder containing several JCasC files on the Jenkins server. The process is illustrated in *Figure 2.7*:

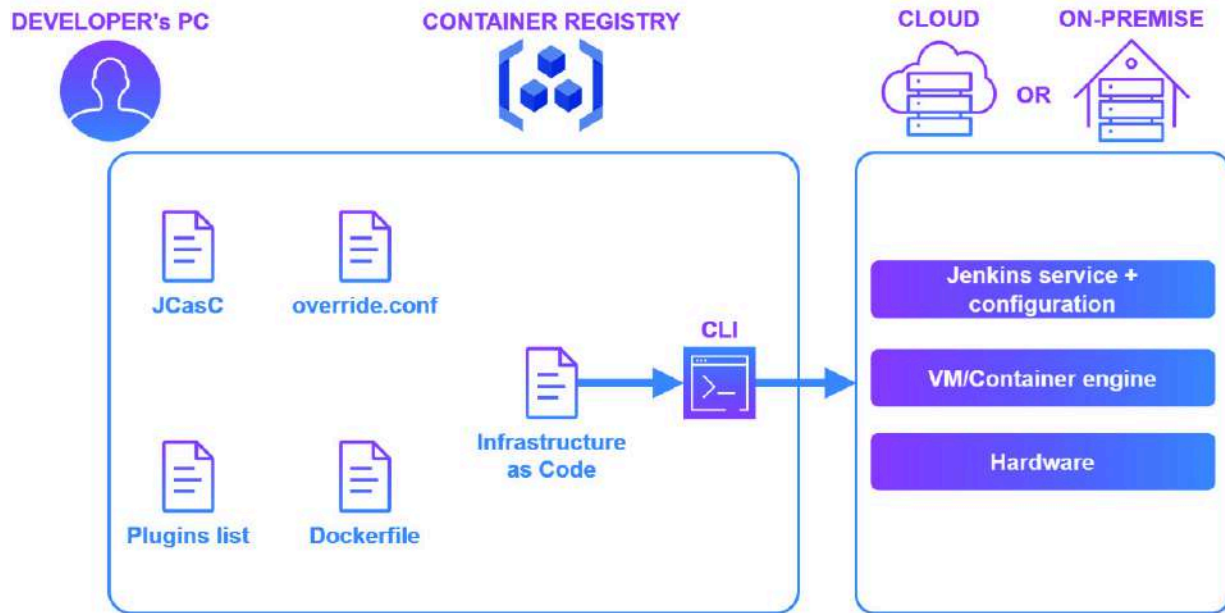


Figure 2.7 – The inner workings of JCasC

Here are the steps for this process:

1. You begin by writing a JCasC YAML configuration file that captures the desired Jenkins server initial configurations, such as security settings, system preferences, and global tools.
2. You also create a `plugins.txt` file with a list of plugins you want to install, including the JCasC plugin.
3. The next step depends on how you plan to run Jenkins. To run Jenkins on a Linux VM, you must update the `override.conf` script with information about the JCasC file and prepare a `plugins.txt` file to install the desired plugins first and then apply the configurations. But if you plan to run Jenkins as a container, you must update the `Dockerfile` instead with an additional step of generating the Docker image and publishing it to a container registry.
4. Next, you write your IaC using a tool such as Terraform or AWS CloudFormation to provision the infrastructure required for the Jenkins server, such as VMs or containers. Inside your IaC file, you must also include steps to run the script or a link to the Docker image.
5. Finally, you execute the IaC script to provision the Jenkins server, which automatically deploys the server infrastructure and applies the initial configurations defined in the JCasC file.

Maintaining the JCasC configuration file in version control and making necessary updates whenever configuration changes are required ensures a consistent and reproducible configuration management process. In the next section, you will learn how to practically configure Jenkins using a JCasC file.

This concludes the planning phase. Now, let's practice deploying a Jenkins server.

Deploying your Jenkins setup

In this section, we will explore two widely adopted methods for setting up a Jenkins server. The first approach involves leveraging a cloud-managed Kubernetes cluster such as **Azure Kubernetes**

Service (AKS), while the second approach entails employing a cloud-managed VM. Both of these methodologies represent robust and dependable options for configuring your Jenkins server in a production-ready manner.

IMPORTANT NOTE

While Azure Cloud has been selected as the focus of this book, it is worth noting that the steps outlined for infrastructure creation can be easily adapted to any other cloud provider.

Deploying Jenkins on a managed Kubernetes cluster

A managed Kubernetes cluster is a fully hosted and maintained environment where the Kubernetes control plane is taken care of by a cloud provider. It allows users to deploy and manage containerized applications without worrying about the underlying infrastructure, ensuring scalability, reliability, and ease of operations.

In this section, we'll explore the process of deploying Jenkins on a managed Kubernetes cluster, including an overview of the architectural design, prerequisites, and a brief walk-through of using the Jenkins Helm chart to deploy Jenkins on Kubernetes.

CAUTION – POTENTIAL COSTS INCURRED IN THE AZURE CLOUD

Please be aware that following the steps outlined in this section may result in costs being incurred on your Azure cloud account. To prevent unexpected charges, we recommend that you familiarize yourself with Azure pricing, regularly review resource usage, and, most importantly, delete the resource group after you are done experimenting.

Though the following exercise may cost you approximately up to 5 USD, remember that it is your responsibility to monitor and manage your Azure cloud usage to ensure cost efficiency.

Architectural design

The architectural design of running Jenkins on AKS involves several key components, as illustrated in *Figure 2.8*:

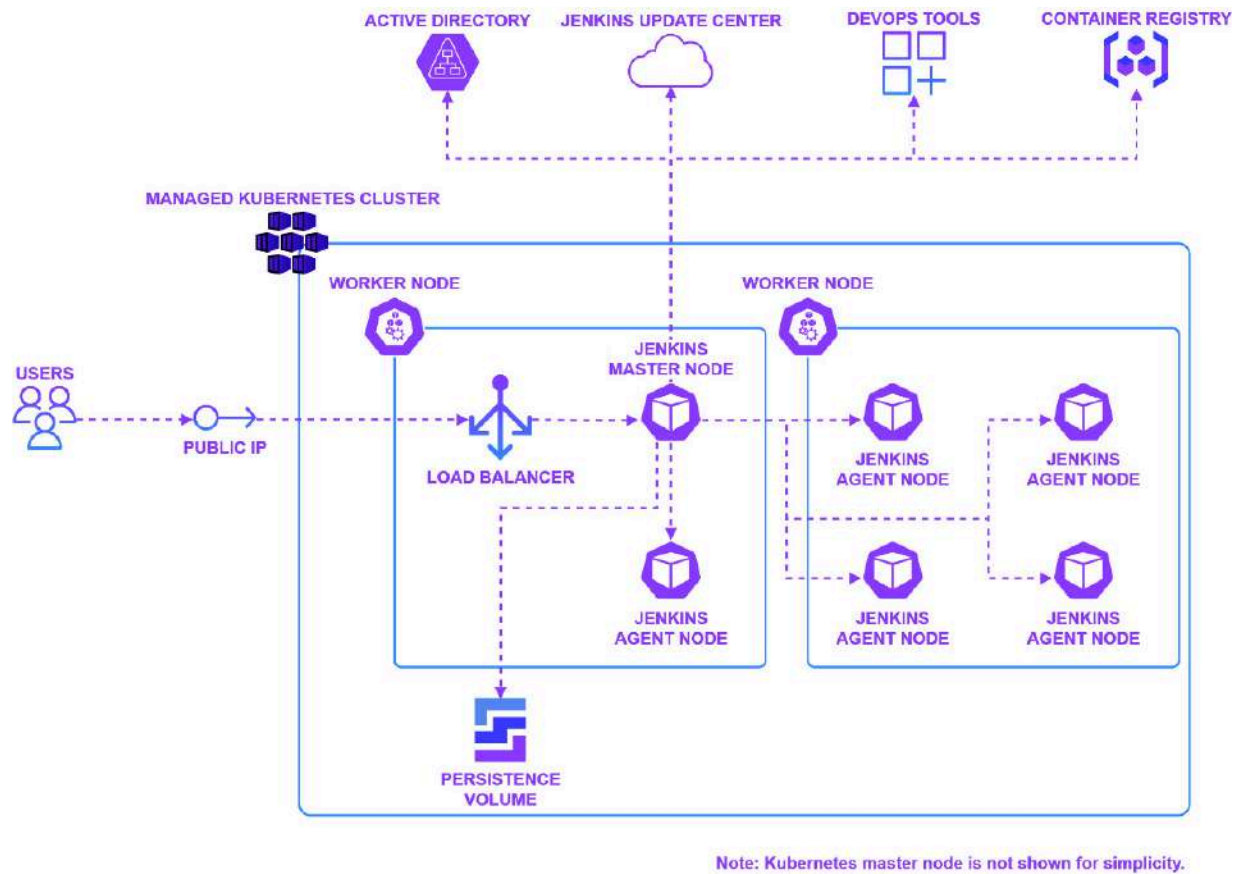


Figure 2.8 – Architectural design to run Jenkins on AKS

First, a public IP address is assigned to the AKS cluster, allowing external access to Jenkins. This enables users to interact with Jenkins through a web browser or API calls.

To ensure scalability and redundancy, a load balancer is utilized. This load balancer allows for easy scalability by enabling the addition of multiple Jenkins instances in the future. Also, if one Jenkins instance fails, the load balancer can automatically redirect traffic to other healthy instances, ensuring uninterrupted service.

Persistent storage is employed to store Jenkins data, such as job configurations, build artifacts, and plugins. By leveraging Azure Disk or Azure Files, Jenkins data is persistently stored and can be accessed by Jenkins instances, even if they are restarted or relocated within the cluster.

Additionally, Jenkins worker nodes are deployed as Kubernetes pods within the same AKS cluster. These nodes are responsible for executing Jenkins build jobs and can dynamically scale up or down based on workload demands.

By combining AKS, public IP, load balancing, persistent storage, and Jenkins nodes within a single Kubernetes cluster, this architectural design provides a robust, scalable, and reliable environment for

running Jenkins, facilitating efficient CI/CD Pipelines and enhancing developer productivity.

Let's see what we need to realize this architectural design.

Preparatory steps

First, you will require a subscription-enabled Azure account. Try creating one for free from the Azure portal [5].

This guide assumes that you have a basic understanding of Kubernetes principles. For more information, please refer to the *Learn Kubernetes Basics* documentation [6].

Local development environment prerequisites

Here are the prerequisites:

- Make use of an IDE such as Visual Studio Code [7] or anything you choose
- Download and install Git [8]
- Download and install the Azure CLI [9] so that you can create and administer Azure cloud resources
- Install kubectl [10], the Kubernetes command-line tool, so that you can perform commands against your Kubernetes cluster
- Download and install Helm [11] so that you can deploy workloads on the Kubernetes cluster

Creating a managed Kubernetes cluster

Creating a Kubernetes cluster in the cloud is a straightforward process, regardless of the chosen method. You have multiple options, including the Azure portal dashboard, Azure Cloud Shell, or the Azure CLI. In this guide, we will focus on using the Azure CLI.

Follow these steps to build a cluster with a single worker node that can auto-scale up to three worker nodes:

IMPORTANT NOTE

Ensure that the commands are executed with administrator rights.

1. Run the following command in a command prompt to connect to your Azure account. The Azure CLI will launch a web browser and ask you to log in to your Azure account:

```
az login
```

2. Next, we need a resource group to effectively organize the Azure cloud resources. To create one, run the **az group create** command. Use the **-n** and **-l** options to add a valid name and location for your resource group:

```
az group create -n rg-nikhil-sbox -l westeurope
```

SUGGESTION

*Choose a location that is close to you. Run the **az account list-locations --output table** command to acquire a list of all possible locations.*

3. Now, create a Kubernetes cluster using the **az aks create** command with the **-g** option to place it inside the resource group we created in the previous step. Use the **-n** option to give your Kubernetes cluster a name. The **--node-count** value is set to **1**; this specifies the number of worker nodes that the cluster will start with. However, in real-world scenarios, this value should be higher since Jenkins is not the only server that will use the cluster – there could be more DevOps tools. The **--enable-cluster-autoscaler --min-count 1 --max-count 3** option enables auto-scaling of the worker nodes up to a maximum of three nodes. We have chosen the free tier using the **--tier** option; however, choose **standard** as the value when you're running a production setup to get support around the Kubernetes master node:

```
az aks create -g rg-nikhil-sbox -l westeurope \  
-n kubernetes-dev --tier free --node-count 1 \  
--enable-cluster-autoscaler --min-count 1 --max-count 3 \  
--network-plugin kubenet --generate-ssh-keys
```

IMPORTANT NOTE

*AKS automatically generates a second resource group to hold AKS resources when you start a new cluster. It usually starts with **MC_**.*

4. The creation of the Kubernetes cluster will require a few minutes. Once the process is complete, it is important to establish a successful connection to your Kubernetes cluster. To obtain the necessary credentials, run the **az aks get-credentials** command. This command will generate a Kubernetes configuration (**kubeconfig**) file located at **~/.kube/config** that contains the required authentication details to connect to your Kubernetes cluster:

```
az aks get-credentials -g rg-nikhil-sbox -n kubernetes-dev
```

5. Once the Kubernetes CLI has been configured to utilize the Kubernetes configuration file, validate the connection to your cluster by executing the **kubectl get** command. This command will provide a comprehensive list of all the currently attached worker nodes within your cluster:

```
kubectl get nodes
```

6. Your new Kubernetes cluster on the cloud is equipped with a load balancer named **kubernetes** that's associated with a public IP address. To locate the public IP address, navigate to the Azure dashboard, select your subscription, and then access the second resource group generated by the system. It typically adheres to a specific naming convention in the format of **MC_<resource group name>_<cluster name>_<location>**. The following figure shows all the resources that have been created under the second resource group:

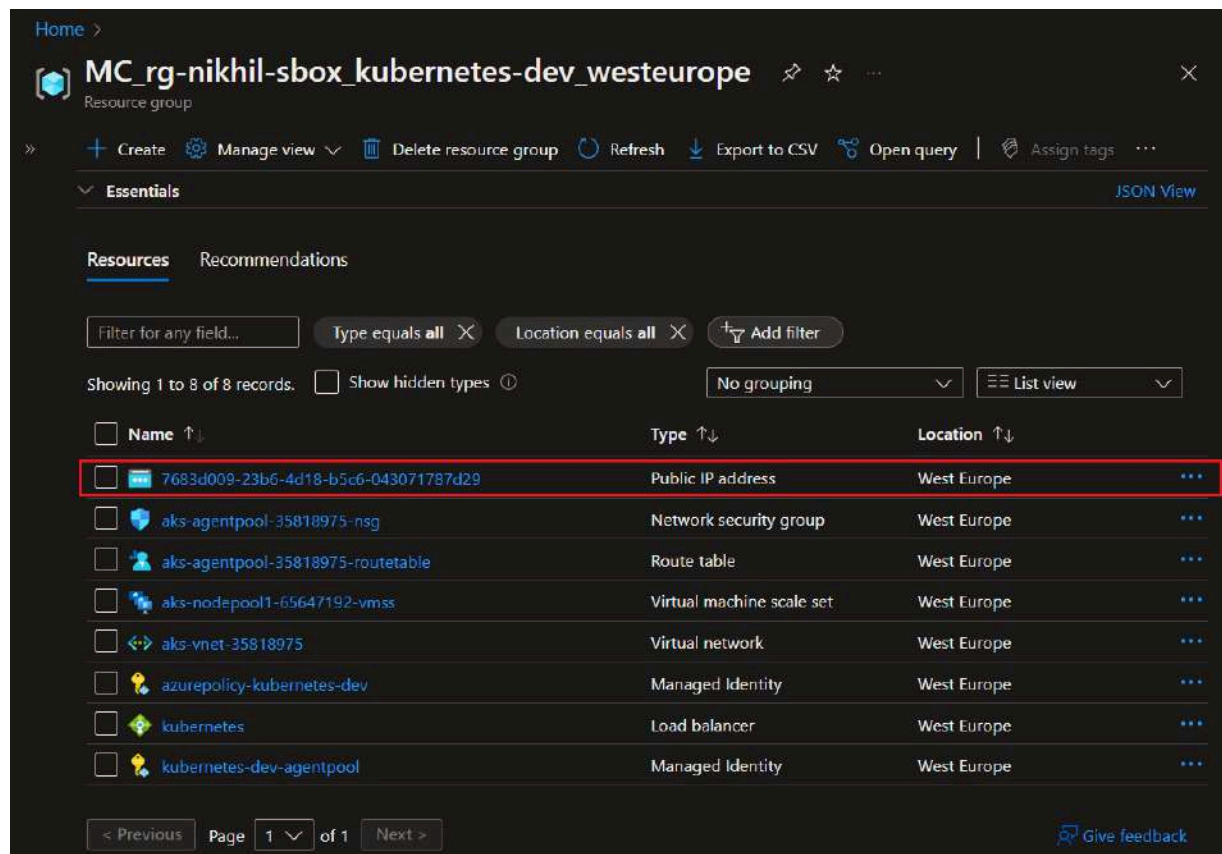


Figure 2.9 – List of Kubernetes cluster resources

7. By clicking on the public IP address resource, you will be directed to its corresponding page, where you can view the IP address:

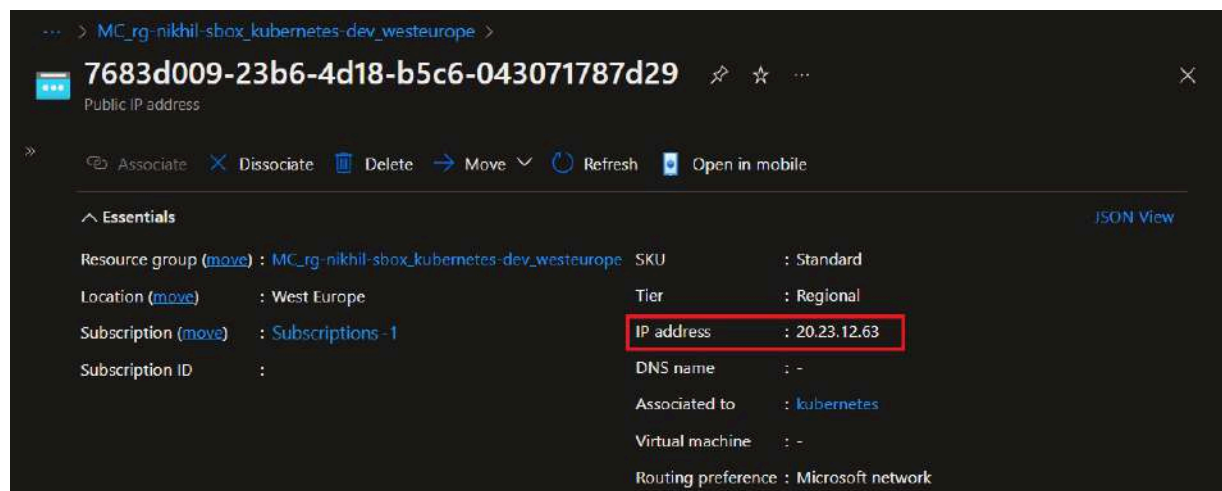


Figure 2.10 – The public IP address under the public IP address resource

If we want, we can use this public IP address to access our Jenkins server. However, let's explore a more effective method to expose our Jenkins service to the internet.

Deploying an Nginx Ingress controller for Jenkins

An Nginx Ingress controller for Jenkins facilitates optimized routing and external access to the Jenkins application deployed on Kubernetes. By implementing this setup, teams can ensure secure and dependable access to Jenkins while harnessing the powerful load balancing and routing capabilities of Nginx. Recall the public IP and load balancer shown in *Figure 2.8*; the Nginx Ingress controller for Jenkins does that for you. Although the configuration of the Nginx Ingress controller can be intricate, in this section, we will emphasize a simpler setup to achieve seamless accessibility to Jenkins. Follow these steps:

1. Connect to your AKS cluster. Use the Azure CLI (**az**) or the Azure portal to authenticate and connect to your AKS cluster.
2. The easiest way to deploy an Nginx Ingress controller on Kubernetes is using Helm. Execute the following two commands one after the other to add the **ingress-nginx** repository as a source of Helm charts and update the local cache to have the latest information about the available charts from all the added repositories:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
```

3. Use the **helm install** command to install the **Ingress-Nginx** chart with specific configurations and options to set annotations, define the replica count, and associate the **Ingress-Nginx** controller with a specific Ingress class resource and annotation value, as shown here:

```
helm install ingress-nginx-jenkins ingress-nginx/ingress-nginx \
--create-namespace -n jenkins \
--set controller.service.annotations."service\.beta\.kubernetes\.io/azure-load-
balancer-health-probe-request-path"=/healthz \
--set controller.replicaCount=1 \
--set controller.ingressClassResource.name=jenkins-nginx \
--set controller.ingressClassResource.controllerValue=k8s.io/jenkins-nginx
```

Let's understand these commands:

- The following command uses Helm to install the **Ingress-Nginx** chart from the **ingress-nginx/ingress-nginx** repository. The release name is set to **ingress-nginx-jenkins**, which identifies the specific deployment:

```
helm install ingress-nginx-jenkins ingress-nginx/ingress-nginx
```

- This option creates a new namespace called **jenkins** and associates the new Nginx Ingress controller with it:

```
--create-namespace -n jenkins
```

- This flag sets an annotation for the **Ingress-Nginx** controller's service. The annotation being set is **service.beta.kubernetes.io/azure-load-balancer-health-probe-request-path**. The value of the annotation is **/healthz**, which represents the health check path for the Azure load balancer:

```
--set controller.service.annotations."service\.beta\.kubernetes\.io/azure-
load-balancer-health-probe-request-path"=/healthz
```

- This flag sets the number of replicas for the **Ingress-Nginx** controller to **1**. It specifies that only one replica of the controller should be deployed:

```
--set controller.replicaCount=1
```

- This flag sets the name of the Ingress class resource associated with the **Ingress-Nginx** controller. Its name is set to **jenkins-nginx**, which identifies the specific Ingress class resource:

```
--set controller.ingressClassResource.name=jenkins-nginx
```

- This flag sets the value of the `k8s.io/jenkins-nginx` annotation in the Ingress class resource. The annotation is used to specify additional information or behavior associated with the **Ingress-nginx** controller:

```
--set controller.ingressClassResource.controllerValue=k8s.io/jenkins-nginx
```

4. From the Azure dashboard, look for the public IP associated with the Nginx Ingress controller, as shown here:

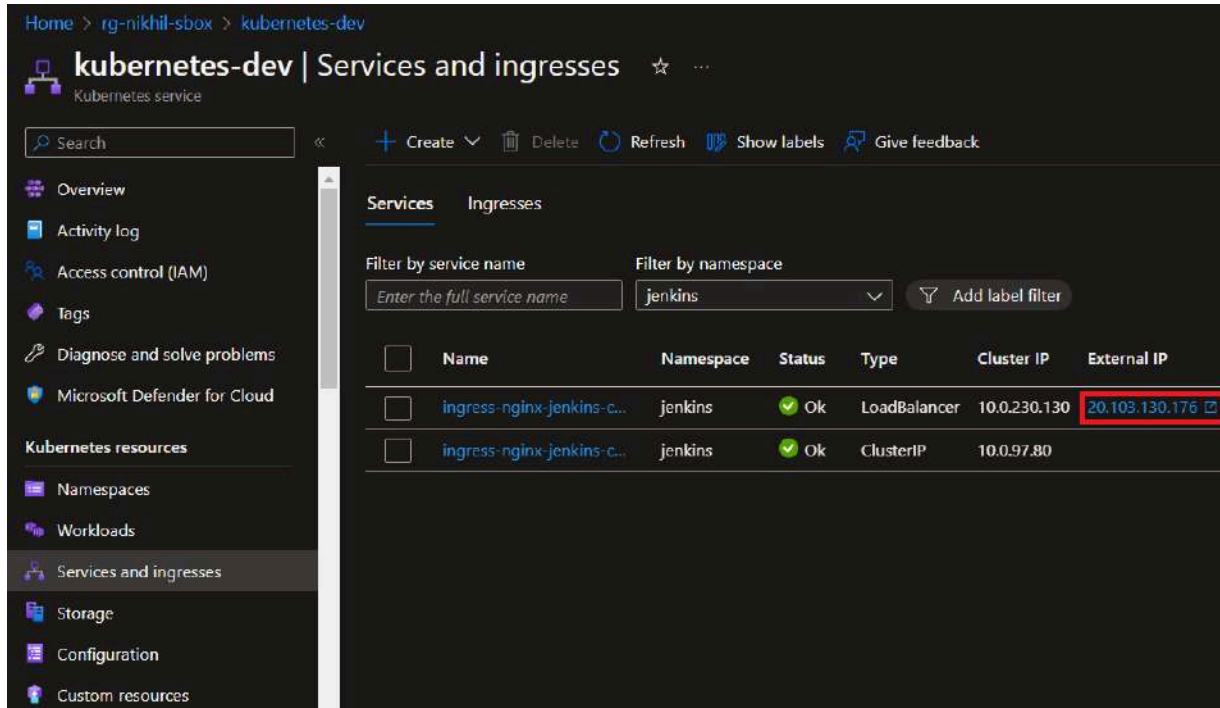


Figure 2.11 – Nginx Ingress controller services for Jenkins

To learn more about the Nginx Ingress controller Helm chart, visit its respective GitHub repository [12].

Using the Jenkins Helm chart

A Jenkins Helm chart is a package specifically designed to simplify the deployment and configuration of Jenkins on Kubernetes. It encompasses all the necessary Kubernetes manifests, configurations, and dependencies required to set up and manage Jenkins within a Kubernetes cluster. In addition to the basic Jenkins installation, the Helm chart also includes the ability to configure Jenkins using JCasC.

By utilizing the Jenkins Helm chart, users can efficiently provision a scalable and resilient Jenkins environment on Kubernetes while leveraging the benefits of JCasC for streamlined and reproducible configuration management.

Follow these steps to run a Jenkins server on your Kubernetes cluster using the Helm chart for Jenkins:

1. First, fetch the Git repository information of the Jenkins Helm chart using the **helm repo add** and **helm repo update** commands:

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

2. While the Jenkins Helm chart can be deployed as-is, we prefer to customize a few default settings before proceeding. To accomplish this, create a **values.yaml** file with the following content. Before saving the file, ensure that you replace **<Jenkins Ingress Public IP>** with the actual public IP of your Jenkins Nginx Ingress controller (refer to *Figure 2.11*):

values.yaml

```
controller:
  jenkinsUrl: http://<Jenkins Ingress Public IP>/
  JCasC:
    configScripts:
      welcome-message: |
        jenkins:
          systemMessage: Welcome to our Continuous Integration server.
```

After saving the file, we can take a moment to examine its contents.

We want to configure our Jenkins URL to utilize the public IP of our Jenkins Nginx Ingress controller, which can be achieved by setting the **jenkinsUrl** parameter. Lastly, we must incorporate a system message through JCasC, which will be visible on the Jenkins dashboard. It is important to note that JCasC is integrated within the Helm chart, eliminating the need for a separate **jenkins.yaml** file for JCasC configuration.

3. To proceed, execute the **helm install** command. Use the **-f** option to specify the **values.yaml** file we created in the previous step. Include the **-n jenkins** option to create the Jenkins workload inside the **jenkins** namespace. Additionally, provide a name of your preference for the Jenkins instance, such as **myjenkins**, and then specify the Helm chart in use, which in our case is **jenkins/jenkins**:

```
helm install -f values.yaml \
-n jenkins myjenkins jenkins/jenkins
```

4. If everything goes smoothly, you will receive a success message containing instructions on how to access your Jenkins server, including the Jenkins server URL. Please note that it may take some time for the Jenkins server pod to fully initialize.

Now, you should have a Jenkins server up and running. So, let's try to access it. But first, we need to create an Ingress resource for Jenkins.

Deploying an Ingress file for Jenkins

To access Jenkins, we need to expose it using an Ingress resource. The purpose of an Ingress resource in Kubernetes is to define rules for routing external HTTP and HTTPS traffic to services within the cluster. It acts as a configuration file that specifies how incoming requests should be directed to the appropriate backend services based on the requested host, path, or other criteria. Follow these steps to deploy an Ingress to expose our Jenkins service:

1. First, create a file called **jenkins-ingress.yaml** with the following content:

jenkins-ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: jenkins-ingress
  annotations:
spec:
  ingressClassName: jenkins-nginx
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: myjenkins
            port:
              number: 8080
```

The preceding code is an Ingress resource definition for Jenkins. Let's look at some of its key elements:

- **ingressClassName: jenkins-nginx** specifies the class name of the Ingress controller associated with this Ingress resource as **jenkins-nginx**.
- **name: myjenkins** sets the name of the backend service to **myjenkins**.
- **number: 8080** specifies the port configuration for the backend service by setting the port number to **8080**.

In summary, this Ingress resource definition sets up routing rules for Jenkins so that incoming traffic to the root path (/) will be forwarded to the backend service named **myjenkins** running on port **8080**. The annotations and **ingressClassName** provide additional customization and association with the specific Ingress controller for Jenkins.

2. Next, save the file and run the **kubectl apply** command to apply the **jenkins-ingress.yaml** manifest file in the **jenkins** namespace:

```
kubectl apply -f .\jenkins-ingress.yaml -n jenkins
```

It takes a while for the Nginx Ingress controller's public IP to associate with the Jenkins Ingress. Use the Azure dashboard to check the association:

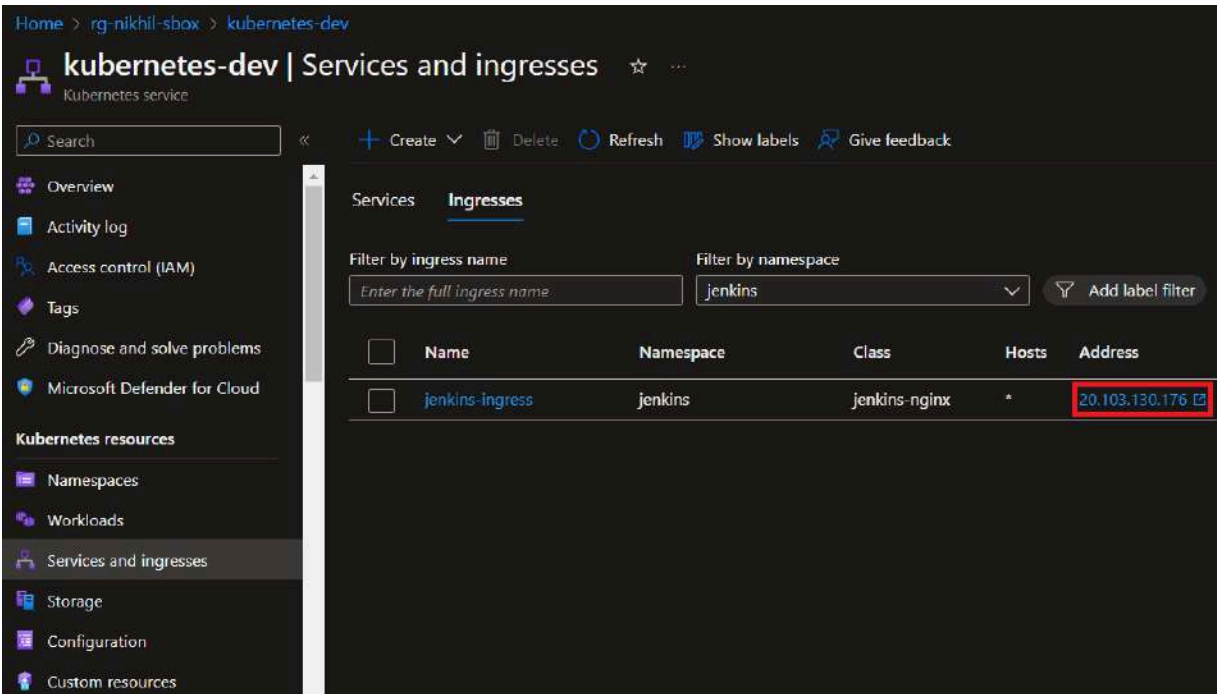


Figure 2.12 – Jenkins Ingress with its public IP address

That's it! You've successfully deployed Jenkins on AKS using a Helm chart. To access Jenkins, go to `http://<Public IP>/`, where the `<Public IP>` is the public IP of the Jenkins Nginx Ingress controller.

By default, the Jenkins Helm chart creates an `admin` user with a temporary password. To retrieve the password for the `admin` user, execute the `kubectl exec` command:

```
kubectl exec --namespace jenkins \
  -it svc/myjenkins -c jenkins \
  -- /bin/cat /run/secrets/additional/chart-admin-password
```

Accessing your Jenkins instance

To access Jenkins, simply enter the Jenkins server URL in your preferred browser. Utilize the user and password you obtained from the previous command to log in. The following figure showcases the Jenkins dashboard with the customized welcome message implemented through JCasC:

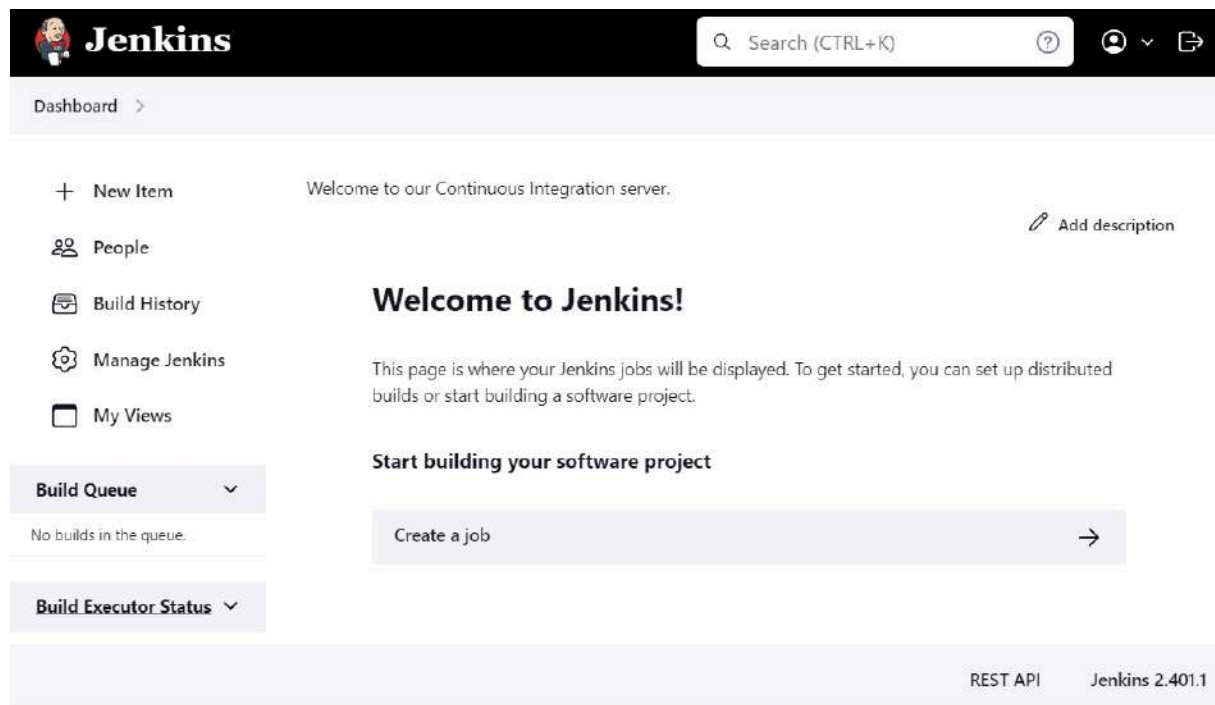


Figure 2.13 – The Jenkins dashboard

IMPORTANT NOTE

Once you have completed this exercise, utilize the `az group delete` command to remove all the resources created thus far. For instance, execute the command as follows: `az group delete --name <resource group>`. A confirmation prompt will appear, and it will take a few minutes for the system to delete all the resources.

In summary, deploying Jenkins on a managed Kubernetes cluster offers a hassle-free experience as the cloud provider takes care of the Kubernetes control plane and underlying infrastructure. This ensures seamless deployment and management of containerized applications while guaranteeing scalability, reliability, and operational simplicity.

Deploying Jenkins to a cloud instance

Deploying Jenkins on a cloud-based VM using Terraform is generally simpler compared to setting up and managing a Kubernetes cluster. It requires fewer components and configurations, making it more straightforward for users who are new to Kubernetes. Also, Jenkins has been traditionally designed to run on VMs, and many Jenkins plugins and extensions are well suited for VM-based deployments. If you are someone who has decided to adopt the cloud, migrating an existing Jenkins setup to a cloud-based VM requires less reconfiguration.

In addition to that, the VM ecosystem, including its infrastructure tools and services, is mature and well established. One big advantage of running Jenkins on a cloud VM is more straightforward

access to persistent storage options, such as attaching and managing disk volumes directly to the VM. This can be beneficial for storing Jenkins artifacts, logs, and configurations.

In this section, we'll explore the process of deploying Jenkins on a cloud-based VM, including an overview of the architectural design, prerequisites, and a brief walkthrough of using Terraform to deploy Jenkins on a VM.

CAUTION – POTENTIAL COSTS INCURRED IN THE AZURE CLOUD

Please be aware that following the steps outlined in this section may result in costs being incurred on your Azure cloud account. To prevent unexpected charges, we recommend that you familiarize yourself with Azure pricing, regularly review resource usage, and, most importantly, delete the resource group after you are done experimenting.

Though the following exercise may cost you approximately up to 5 USD, remember that it is your responsibility to monitor and manage your Azure cloud usage to ensure cost efficiency.

Architectural design

The architectural design, as illustrated in *Figure 2.14*, showcases a comprehensive setup for running Jenkins on a cloud-based VM. In this design, Jenkins is deployed on an Azure VM residing within a dedicated subnet. The Jenkins server is made accessible to users via a public IP address, enabling easy access for administration and usage.

To ensure scalability, Jenkins agent nodes are deployed using an Azure **VM Scale Set (VMSS)** or using an AKS cluster and more. This allows for the dynamic provisioning and scaling of Jenkins agents based on workload demands, improving the overall efficiency of CI/CD processes. You will learn how to implement this part in detail in the upcoming chapter.

To provide persistent storage for Jenkins data, Azure-managed disks are utilized, ensuring Jenkins historical build data is stored reliably. By leveraging Azure VMs, subnets, VMSS, AKS, and managed disks, this architectural design provides a robust and scalable infrastructure for running Jenkins on the Azure platform, facilitating efficient CI/CD Pipelines, and enhancing developer productivity:

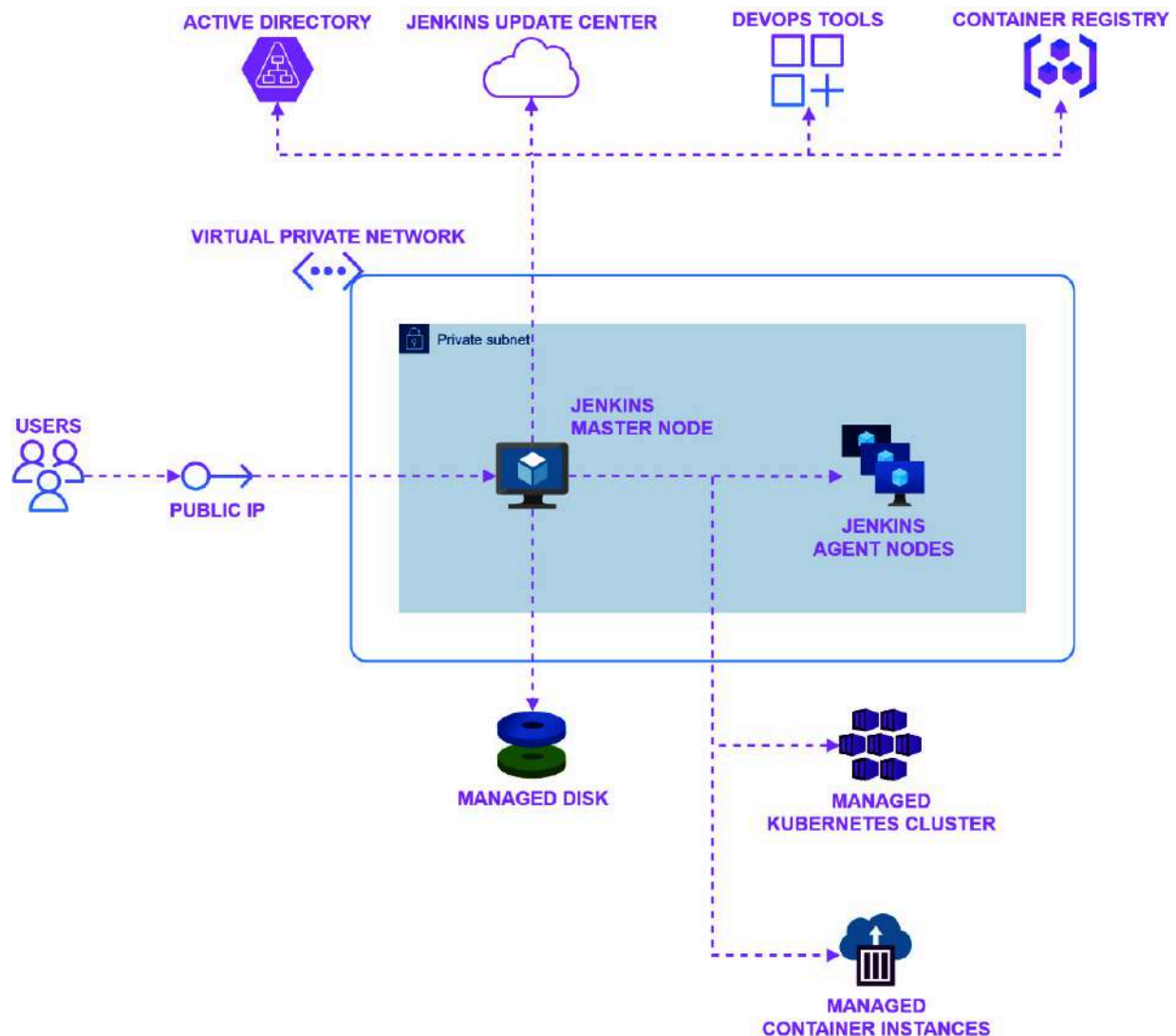


Figure 2.14 – Architectural design to run Jenkins on a cloud-based VM

Let's see what we need to realize this architectural design.

Preparatory steps

First, you will require a subscription-enabled Azure account. Try creating one for free from the Azure portal [5].

This guide assumes that you have a basic understanding of Terraform principles. In the following section, we will use the Azure provider to create resources on the Azure cloud through Terraform. For more information, please refer to the Azure provider documentation [13].

Local development environment prerequisites

Here are the prerequisites:

- Make use of an IDE such as Visual Studio Code [7] or anything you choose

- Download and install Git [8]
- Download and install the Azure CLI [9]
- Download and install Terraform CLI [14] so that you can run Terraform scripts to create and administer Azure cloud resources

Preparing the Jenkins configuration code

Creating the `jenkins.yaml` file is the initial step in configuring Jenkins. However, before Jenkins can consume this file, the JCasC plugin needs to be installed. To achieve this, a `plugins.txt` file is created, listing the necessary plugins, with the JCasC plugin being a vital one. Once the plugins have been installed, an `override.conf` file is put in place to set the `CASC_JENKINS_CONFIG` environment variable pointing to the location of the `jenkins.yaml` file. This ensures that the JCasC plugin correctly identifies and utilizes the Jenkins configuration file during the startup process. The sequence of these steps is crucial as the `jenkins.yaml` file cannot be consumed without the JCasC plugin, and the plugin needs the correct configuration file defined by the `override.conf` file. So, let's get started:

1. Start by creating a file named `jenkins.yaml` to define your Jenkins configuration. Inside the file, you can specify various aspects of your Jenkins setup, such as security, global tools, agent configurations, and more. Refer to the JCasC plugin's documentation to understand all the available options. However, to keep things simple, in the following exercise, we will use the following code for our `jenkins.yaml` file:

`jenkins.yaml`

```
jenkins:
  systemMessage: "Jenkins configured automatically by Jenkins Configuration as Code
  plugin\n\n"
  securityRealm:
    local:
      users:
        - id: jenkins-admin
          password: password
```

In the preceding code, the `systemMessage` configuration lets you define a text message to display on the Jenkins dashboard; the `securityRealm` configuration refers to the authentication and user management mechanism inside Jenkins. When set to `local`, it means that Jenkins will handle authentication and user management internally, without relying on any external authentication providers. Remember the values for `id` and `password`. You'll need them to log in to Jenkins.

2. Next, create another file named `plugins.txt` containing the list of plugins to be installed during the Jenkins setup process. This is because, to consume the Jenkins configuration defined in the `jenkins.yaml` file, we need the JCasC plugin to be installed on the Jenkins server. The following code is what we will use in the current exercise:

`plugins.txt`

```
configuration-as-code
git
sonar
```

IMPORTANT NOTE

The `plugins.txt` file used by Jenkins follows a specific syntax for listing the plugins that need to be installed. Each line in the `plugins.txt` file represents a plugin and should follow the `<plugin-short-name>:<plugin-version>` format or just `<plugin-short-name>`.

3. Lastly, create a file named `override.conf`. In Jenkins, the `override.conf` file is used to set environment variables that override the default settings of the Jenkins process. It allows you to customize certain aspects of Jenkins' behavior by providing specific environment variable values. The `override.conf` file is typically used to set environment variables related to JCasc. One common use case is setting the `CASC_JENKINS_CONFIG` environment variable to specify the location of the `jenkins.yaml` file. Here is what our `override.conf` file looks like:

`override.conf`

```
[Unit]
Description=My Company Jenkins Controller
[Service]
# Add JVM configuration options
Environment="JAVA_OPTS=-Djava.awt.headless=true -XX:+UseStringDeduplication"
# Configuration as code directory
Environment="CASC_JENKINS_CONFIG=/tmp/jenkins.yaml"
```

Preparing helper scripts

Terraform is primarily used for infrastructure provisioning rather than application configuration. Therefore, to install Jenkins on the VM and perform other essential configurations, we will develop specific scripts. Terraform will assist us in seamlessly transferring and executing these scripts on the VM. So, let's get started:

1. Begin by creating a shell script named `partition-managed-disk.sh`. This shell script will partition the newly created managed disk named `sdsc`. The creation and attachment of the managed disk to the Jenkins-hosting VM will be done using Terraform. Remember, as per the architecture design outlined earlier in this exercise, we have opted to store all the content of the Jenkins home directory (`/var/lib/jenkins`) on this managed disk:

`partition-managed-disk.sh`

```
#!/bin/sh
sudo parted /dev/sdsc --script mklabel gpt mkpart xfs xfs 0% 100%
sudo mkfs.xfs /dev/sdc1
sudo partprobe /dev/sdc1
```

IMPORTANT NOTE

The managed disk, typically referred to as `sdc`, is created and attached when you run the Terraform code. However, since the infrastructure hasn't been created yet, the specific disk name won't be known in advance. A workaround is to refrain from executing the shell scripts initially and solely focus on creating the infrastructure. Afterward, log in to the VM to verify the disk name and subsequently update the Terraform `main.tf` file so that it incorporates executing the shell scripts.

2. Next, create another shell script named `mount-managed-disk.sh`. This script will pre-create the `/var/lib/jenkins` directory on the VM, mount the `/dev/sdc1` partition to it, and run the necessary commands to make sure the mount persists even after a reboot. Note that `/var/lib/jenkins` is the default Jenkins home directory. We will use the following code for our `mount-managed-disk.sh` shell script:

`mount-managed-disk.sh`

```
#!/bin/sh
sudo mkdir /var/lib/jenkins
sudo chown jenkins:jenkins /var/lib/jenkins
sudo mount /dev/sdc1 /var/lib/jenkins
uuid=$(sudo blkid -s UUID -o value /dev/sdc1)
append_string="    /datadrive    xfs    defaults,nofail    1    2"
echo "UUID=$uuid$append_string" | sudo tee -a /etc/fstab
```

It's a clever script that has been tested on an Ubuntu 18.04-LTS.

3. Lastly, create a shell script named **jenkins-installation.sh**. First, this script installs JDK 11, followed by the Jenkins application itself. Next, it proceeds to install the Jenkins plugins listed in the **plugins.txt** file. Lastly, the script sets the **CASC_JENKINS_CONFIG** environment variable using the **override.conf** file, directing Jenkins to use the **jenkins.yaml** file to apply the desired configuration. By executing these steps in sequence, the **jenkins-installation.sh** script ensures the complete installation and configuration of the Jenkins server:

jenkins-installation.sh

```
#!/bin/sh
sudo apt-get update
sudo apt-get install -y default-jdk
curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | sudo tee
/usr/share/keyrings/jenkins-keyring.asc > /dev/null
echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc]
https://pkg.jenkins.io/debian-stable binary/ | sudo tee
/etc/apt/sources.list.d/jenkins.list > /dev/null
sudo apt-get update
sudo apt-get install -y jenkins
wget https://github.com/jenkinsci/plugin-installation-manager-
tool/releases/download/2.12.11/jenkins-plugin-manager-2.12.11.jar -P /tmp/
sudo java -jar /tmp/jenkins-plugin-manager-2.12.11.jar --war /usr/share/java/jenkins.war -
d /var/lib/jenkins/plugins --plugin-file /tmp/plugins.txt
sudo systemctl restart jenkins.service
sudo mkdir /etc/systemd/system/jenkins.service.d
sudo cp /tmp/override.conf /etc/systemd/system/jenkins.service.d/override.conf
sudo systemctl daemon-reload
sudo systemctl restart jenkins.service
```

This script uses **plugin-installation-manager-tool**. It is a utility that's used to automate the installation and management of Jenkins plugins. It simplifies the process of adding, updating, and removing plugins in a Jenkins instance. To learn more, refer to its GitHub repository [15].

To install a specific version of Jenkins use the **sudo apt-get install jenkins=<version>** command instead.

IMPORTANT NOTE

At the time of writing this book, I am utilizing Jenkins Plugin Manager version 2.12.11.jar. However, when you undertake this exercise, make sure you use the latest available version.

Preparing the Terraform code

In preparation for the Terraform code, we will create essential files to define the infrastructure configuration. These files include **providers.tf**, **main.tf**, **variables.tf**, and **outputs.tf**. Each file

plays a crucial role in specifying the providers, resources, variables, and outputs necessary to provision and manage the desired infrastructure. So, let's get started:

1. In the same directory where you worked in the previous sections, create a file named **providers.tf**. The **providers.tf** file in Terraform is used to declare the provider configurations. It specifies the target cloud platform or infrastructure provider, along with any required authentication credentials or settings necessary for Terraform to interact with the provider's API and provision resources. Here is our **providers.tf** file:

providers.tf

```
provider "azurerm" {  
  features {}  
}
```

The preceding code uses **azurerm**, which is the official Terraform provider for Microsoft Azure. It enables infrastructure provisioning and management on the Azure cloud platform, allowing users to define and deploy Azure resources using Terraform configuration files.

2. Next, we move on to creating a **main.tf** file. It is the primary Terraform configuration file that defines the desired infrastructure and resources. It contains resource declarations, configurations, and dependencies necessary for Terraform to provision and manage the specified infrastructure components. Let's see some of the important resource blocks:

- The first code snippet creates an *Azure Resource Group* using the **azurerm_resource_group** resource. The name and location of the resource group are defined using the **resource_group_name** variable and the **location** variable, whose values come from the **variables.tf** file. This resource group serves as a logical container for organizing related Azure resources:

```
# Resource Group  
resource "azurerm_resource_group" "jenkins_rg" {  
  name      = var.resource_group_name  
  location  = var.location  
}
```

- The fourth code snippet creates an *Azure public IP* using the **azurerm_public_ip** resource. We will use this public IP to access our Jenkins server. Notice that the IP is static. Again, all values for the variables come from the **variables.tf** file:

```
# Public IP  
resource "azurerm_public_ip" "jenkins_public_ip" {  
  name              = var.public_ip_name  
  location          = var.location  
  resource_group_name = azurerm_resource_group.jenkins_rg.name  
  allocation_method = "Static"  
}
```

- The sixth code snippet configures an *Azure Linux VM* that uses the **azurerm_linux_virtual_machine** resource in Terraform. It specifies various attributes, such as the VM's name, location, size, and networking details. You will also notice that the resource uses an *Ubuntu Server 18.04-LTS* image. It also sets up the operating system disk and defines the following:

```
# Virtual Machine  
resource "azurerm_linux_virtual_machine" "jenkins_vm" {  
  name              = var.vm_name  
  location          = var.location  
  resource_group_name = azurerm_resource_group.jenkins_rg.name  
  network_interface_ids = [azurerm_network_interface.jenkins_nic.id]
```

```

        size = "Standard_D2s_v3"
    source_image_reference {
        publisher = "Canonical"
        offer     = "UbuntuServer"
        sku       = "18.04-LTS"
        version   = "latest"
    }
    os_disk {
        storage_account_type = "Standard_LRS"
        caching               = "ReadWrite"
    }
    .
    .
}

```

- The **azurerm_linux_virtual_machine** resource block also contains an SSH connection with authentication details, followed by multiple provisioners to transfer necessary files, including **override.conf**, **jenkins.yaml**, **plugins.txt**, **partition-managed-disk.sh**, **mount-managed-disk.sh**, and **jenkins-installation.sh**, to specific destinations on the VM. These provisioners facilitate the installation and configuration of the managed disk and the Jenkins application on the VM:

```

    provisioner "file" {
        source      = "override.conf"
        destination = "/tmp/override.conf"
    }
    provisioner "file" {
        source      = "jenkins.yaml"
        destination = "/tmp/jenkins.yaml"
    }
    provisioner "file" {
        source      = "plugins.txt"
        destination = "/tmp/plugins.txt"
    }
    provisioner "file" {
        source      = "partition-managed-disk.sh"
        destination = "/tmp/partition-managed-disk.sh"
    }
    provisioner "file" {
        source      = "mount-managed-disk.sh"
        destination = "/tmp/mount-managed-disk.sh"
    }
    provisioner "file" {
        source      = "jenkins-installation.sh"
        destination = "/tmp/jenkins-installation.sh"
    }
}

```

- **azurerm_virtual_machine_extension** is a Terraform resource that's used to deploy extensions on Azure VMs. It enables the installation and configuration of additional software, scripts, or agents on the VM instances. Extensions can be used to customize the VM environment, enable monitoring, deploy security agents, and perform various other tasks. They are defined within the Terraform configuration and applied to specific VMs, extending their functionality, and enabling post-deployment customizations. In our **main.tf** file, we have three blocks of **azurerm_virtual_machine_extension** resources. The first one is to run the **partition-managed-disk.sh** shell script, as shown here. For aesthetic reasons, only part of the code is shown:

```

resource "azurerm_virtual_machine_extension" "jenkins_extension_1" {
    name = "partition-managed-disk"
    .
    .
    settings = <<SETTINGS
    {
        "commandToExecute": "chmod 755 /tmp/partition-managed-disk.sh &&

```

```

/tmp/partition-managed-disk.sh"
    }
    SETTINGS
}

```

IMPORTANT NOTE

Run the resource named `jenkins_extension_1` only once, during the initial infrastructure creation. Afterward, comment it out as we don't want to repartition an existing managed disk that contains data.

- The second one is to run the `mount-managed-disk.sh` shell script, as shown here:

```

resource "azurerm_virtual_machine_extension" "jenkins_extension_2" {
  name = "mount-managed-disk"
  .
  .
  settings = <<SETTINGS
  {
    "commandToExecute": "chmod 755 /tmp/mount-managed-disk.sh && /tmp/mount-managed-disk.sh"
  }
  SETTINGS
}

```

- The third one is to run the `jenkins-installation.sh` shell script, as shown here:

```

resource "azurerm_virtual_machine_extension" "jenkins_extension_3" {
  .
  .
  settings = <<SETTINGS
  {
    "commandToExecute": "chmod 755 /tmp/jenkins-installation.sh && /tmp/jenkins-installation.sh"
  }
  SETTINGS
}

```

- The second-to-last code snippet, `azurerm_managed_disk`, is a Terraform resource that's used to create and manage managed disks in Azure. With `azurerm_managed_disk`, users can define disk properties such as size, SKU, and storage account type. These disks provide persistent storage for VMs. Managed disks offer features such as data redundancy, snapshots, and encryption, ensuring reliable and secure storage for VMs running in Azure:

```

# Managed Disk
resource "azurerm_managed_disk" "jenkins_disk" {
  name = var.disk_name
  location = var.location
  resource_group_name = azurerm_resource_group.jenkins_rg.name
  storage_account_type = "Standard_LRS"
  create_option = "Empty"
  disk_size_gb = "20"
}

```

- The last code snippet, `azurerm_virtual_machine_data_disk_attachment`, is a Terraform resource that's used to attach managed disks to Azure VMs:

```

resource "azurerm_virtual_machine_data_disk_attachment" "jenkins_disk_attachment" {
  managed_disk_id = azurerm_managed_disk.jenkins_disk.id
  virtual_machine_id = azurerm_linux_virtual_machine.jenkins_vm.id
  lun = "10"
}

```

```

    caching              = "ReadWrite"
  }
}

```

3. Next, create another file named **variables.tf**. This is a file that's used in Terraform to define input variables. It allows users to parameterize their Terraform configuration, providing flexibility and reusability. By specifying variables and their types in this file, users can easily customize and provide values during Terraform initialization and execution, enabling the configuration to adapt to different environments or requirements without modifying the underlying code. The following is a portion of our **variables.tf** file, which contains the location, resource group name, and credentials of the VM:

variables.tf (chopped down version)

```

variable "location" {
  type    = string
  default = "West Europe"
}
variable "resource_group_name" {
  type    = string
  default = "rg-nikhil-sbox-2"
}
.
.
variable "admin_username" {
  type    = string
  default = "adminuser"
}
variable "admin_password" {
  type    = string
  default = "P@ssw0rd1234"
}

```

4. The last Terraform code file is the **outputs.tf** file. This file is used to define outputs from the infrastructure configuration. It allows users to specify values that they want to extract or display after the Terraform deployment. These outputs can include information such as IP addresses, resource IDs, or any other data that needs to be accessible for further use or reference. Our **outputs.tf** file is small, as shown here. It will print out the public IP address, which will be used to access our Jenkins server:

output.tf

```

output "jenkins_public_ip" {
  value = azurerm_public_ip.jenkins_public_ip.ip_address
}

```

With this, we have finished preparing all the Terraform files that are required to construct infrastructure on the Azure cloud and set up a Jenkins server. Let's move on to the execution.

Running the Terraform code

Terraform code is run in a sequence of commands. These commands are **terraform init**, **terraform plan**, **terraform apply**, and **terraform destroy**. Note that **terraform destroy** is used to destroy the resources created by the Terraform configuration. This step removes all the resources managed by Terraform and prompts for confirmation before deleting it. To create our Jenkins setup on an Azure VM with all the necessary resources, follow these steps:

1. Open a command prompt and, from your working directory, execute the **terraform init** command. This command initializes the Terraform working directory, downloads the necessary provider plugins, and sets up the backend configuration:

```
terraform init
```

2. Next, run the **terraform plan** command. This command is used to create an execution plan. Terraform analyzes the configuration, compares it with the current state, and displays a preview of the resources to be created, modified, or destroyed:

```
terraform plan -out main.tfplan
```

3. Lastly, we run the **terraform apply** command. This is the actual command that applies the changes defined in the configuration. Terraform provisions or modifies the infrastructure based on the execution plan. It prompts for confirmation before making any modifications:

```
terraform apply main.tfplan
```

Accessing your Jenkins instance

To access Jenkins, open your favorite web browser and enter the Jenkins server URL. Utilize the user and password specified in the `jenkins.yaml` file to log in. *Figure 2.15* showcases the Jenkins dashboard featuring the customized welcome message that was implemented through JCasC:

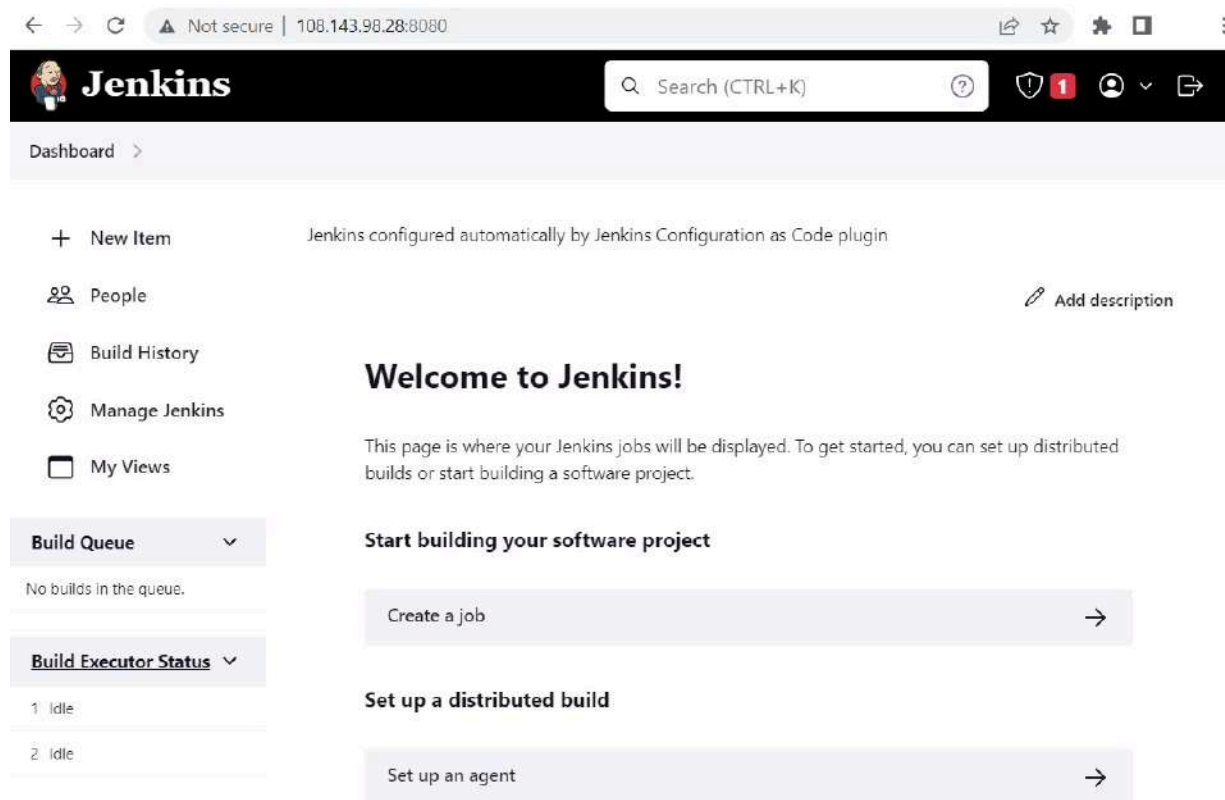


Figure 2.15 – The Jenkins dashboard

IMPORTANT NOTE

Once you have completed this exercise, utilize the `az group delete` command to remove all the resources that have been created thus far. For instance, execute the command as follows: `az group delete --name <resource group>`. A confirmation prompt will appear, and it will take a few minutes for the system to delete all the resources.

In this section, we looked at two common and dependable techniques for installing a Jenkins server in a production-ready configuration. The first way makes use of a cloud-managed Kubernetes cluster, such as AKS, which simplifies deployment and management while also ensuring scalability and stability. The second technique, on the other hand, entails using a cloud-managed VM, which provides a simple and quick alternative for setting up your Jenkins server. Whether you choose Kubernetes for its versatility or a VM for its simplicity, both techniques provide solid options for building a durable and efficient Jenkins system to match your development and CI/CD demands. Now, let's spend some time looking at the maintenance aspect of the Jenkins setup.

Maintaining your Jenkins setup

To ensure the longevity and stability of a Jenkins environment, ongoing maintenance is essential. Upgrading Jenkins involves careful planning and execution to leverage new features, security enhancements, and bug fixes effectively. Backup strategies and disaster recovery best practices safeguard critical Jenkins server data and ensure business continuity. Troubleshooting techniques and community support resources aid in addressing common issues and seeking timely assistance when needed.

Following a structured approach to upgrades

Upgrading Jenkins, or any other DevOps tool for that matter, is critical if you want to take advantage of new features, bug fixes, security updates, and performance improvements. IT security teams in any organization are well aware of this and they do so by frequently sending proactive emails to users and administrators informing them about the availability of a security patch related to the tools in use. These emails act as notifications, highlighting the need to keep up to date to prevent potential security vulnerabilities and safeguard company assets.

Having said that, establishing an upgrade strategy for your DevOps tools, as well as discussing and organizing upgrade operations, is critical in sprint planning.

Tools such as Jenkins provide security patches, bug fixes, and enhancements monthly, so it is strongly advised that you plan for a Jenkins upgrade at least once a month. The upgrade cycle should consist of four parts:

- Watching for new updates, especially security updates
- Informing users about an upcoming Jenkins upgrade activity
- Executing the upgrade activity
- Monitoring for issues

Let's take a closer look.

Watching for new Jenkins updates

This can be done by regularly visiting the Jenkins **LTS Changelog** page [16], as shown here:

Jenkins Search

LTS Changelog

Legend: ● **security fix** ● **major bug fix** ● **bug fix** ● **major enhancement** ● **enhancement**

Community feedback: ☀ no major issues ☁ notable issues ☁🌧 required rollback 📡 [RSS](#)

See the [LTS upgrade guide](#) for advice on upgrading Jenkins.

What's new in 2.401.1 (2023-05-31)

178 ☀ 0 ☁ 0 ☁🌧

Changes since 2.401:

- **Important security fix. (2023-06-14 security advisory)**
 - Fix the writing of emojis to XML (regression in 2.403). ([issue 71182](#))
 - Do not write NUL values to XML files. A technically illegal `#x0` (NUL) could be written to Jenkins XML files but could no longer be read. Now the write will fail as well (regression in 2.398). ([issue 71139](#))
 - Remove "undefined" trailing text from system dropdown menu. ([issue 71345](#))
 - Fix the warning icon in the workspaces temporary directory message. ([issue 71160](#))
 - Show full width filter field for builds on pages less than 970 pixels wide. ([issue 71115](#))

Figure 2.16 – The Jenkins LTS Changelog page

Legend: on the Jenkins changelog page provides a key to understanding the labels used in the changelog entries, such as bug fixes, new features, improvements, and security updates. **Community feedback:** highlights changes driven by the Jenkins community, showcasing the collaborative nature of the project and the importance of user feedback in shaping Jenkins' ongoing development.

Informing users

Communicating planned tool upgrade activity with users promotes transparency, collaboration, and a smoother transition. It ensures users are well informed, prepared, and actively engaged throughout the upgrade process, leading to successful adoption and utilization of the upgraded tool.

The best way to do this is by using the Jenkins **System Message** configuration, available under **Dashboard | Manage Jenkins | System**. It allows you to write messages in **Safe HTML**, as shown here:

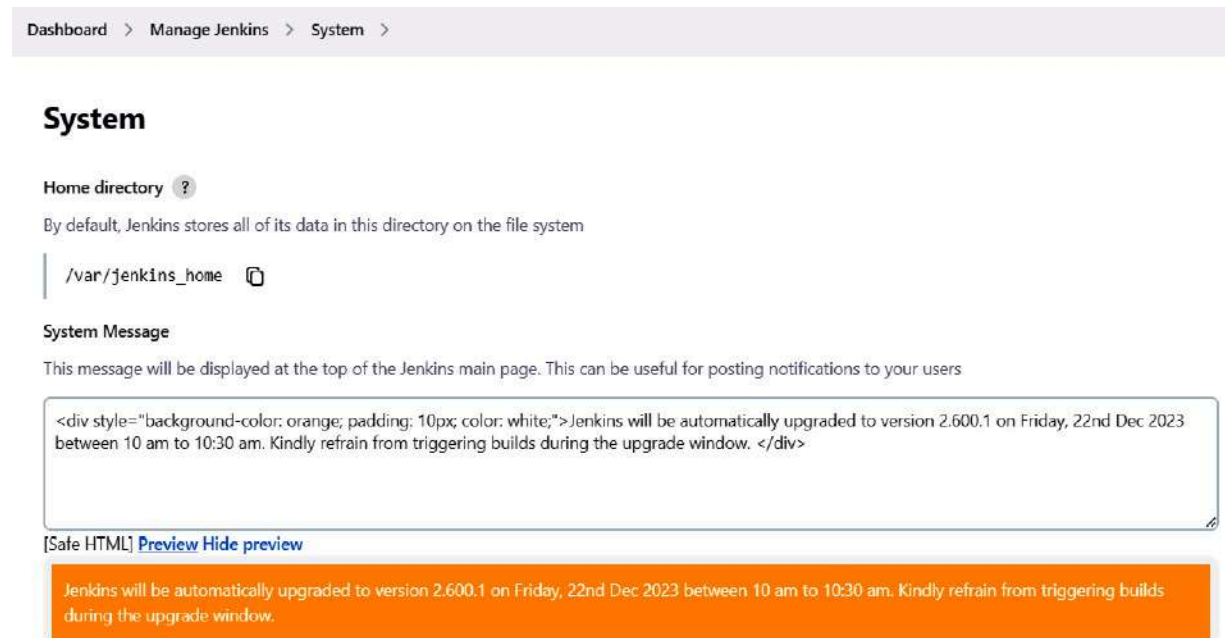


Figure 2.17 – Creating a system message from Jenkins' web interface

If you want to use HTML to customize the background color of a system message, Jenkins requires the use of **Safe HTML** to ensure that the HTML code you're inputting is sanitized. This means you can use basic HTML styling (such as changing the background color) without posing a security risk. The **Safe HTML** setting will strip out any unsafe tags or attributes, ensuring that only non-malicious, style-related HTML is used. You can set the value for the **Markup Formatter** setting to **Safe HTML** under the **Security** page. To do the same through JCasc, use the following code.

Configuration when using the Jenkins Helm chart (`values.yaml`):

```
controller:
  markupFormatter: rawHtml
```

Configuration when using a regular JCasc file (`jenkins.yaml`):

```
jenkins:
  markupFormatter:
    rawHtml:
      disableSyntaxHighlighting: true
```

A system message will appear on the Jenkins dashboard, as shown here:

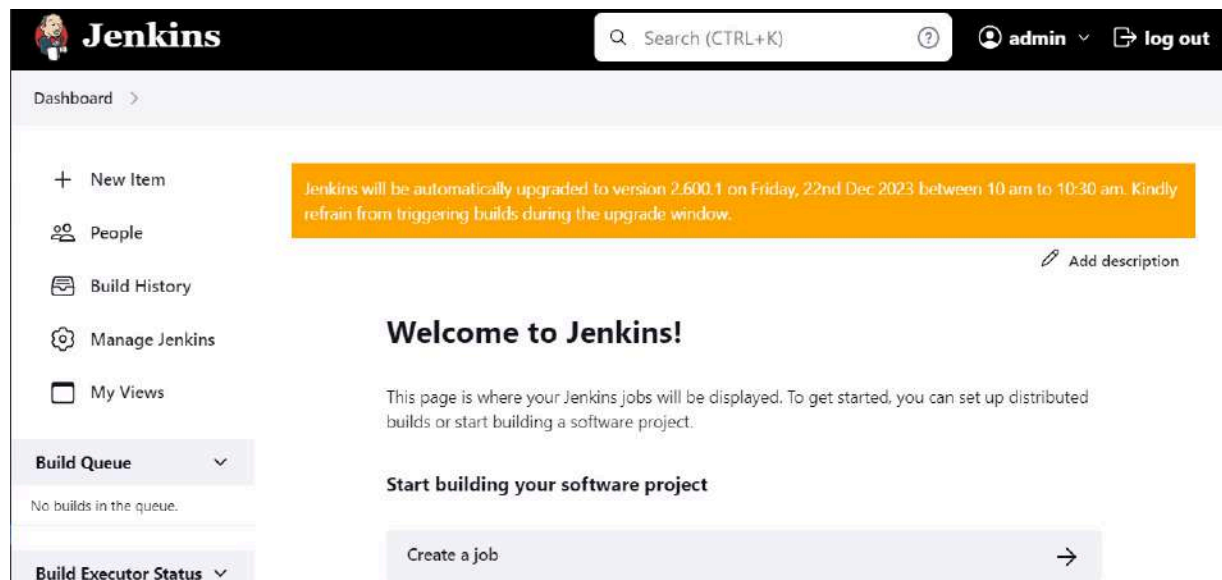


Figure 2.18 – Jenkins system message in SafeHTML

Executing a Jenkins upgrade

The approach for upgrading Jenkins varies based on the setup of your Jenkins server. In this chapter, we'll explore two different installation methods: using the Jenkins Helm chart and employing shell scripts. Let's delve into the upgrade process for both setups, starting with the Jenkins Helm chart.

Upgrading Jenkins using the Jenkins Helm chart

To upgrade Jenkins using the Jenkins Helm chart, follow these steps:

1. Update the **values.yaml** file with the desired Jenkins container image tag information, as depicted here. Refer to the Docker Hub page for Jenkins [17] to explore the available tags:

```
controller:
  tag: "2.401.1-jdk11"
```

2. Next, run the **helm upgrade** command to upgrade Jenkins:

```
helm upgrade -f values.yaml myjenkins jenkins/jenkins
```

Upgrading Jenkins using a shell script

To upgrade Jenkins on a cloud-based VM, follow these steps:

1. Create a shell script called **upgrade-jenkins.sh** on the VM running your Jenkins server.
2. Copy and paste the provided code into the script, replacing **<jenkins version>** with the appropriate value for your Jenkins version:

Upgrade-jenkins.sh

```
#!/bin/sh
# Stop the Jenkins service.
sudo systemctl stop jenkins.service
# Delete the last jenkins.war backup
sudo rm -r jenkins.war.old
# Take the backup of the current jenkins.war binary.
cd /usr/share/jenkins
mv jenkins.war jenkins.war.old
# Download the new jenkins.war file.
sudo wget https://get.jenkins.io/war-stable/<jenkins version>/jenkins.war
chown jenkins:jenkins jenkins.war
# Start the Jenkins service:
sudo systemctl start jenkins.service
```

3. Run the script to initiate the Jenkins upgrade process.

By using these methods, you can seamlessly upgrade your Jenkins installation while ensuring the correct version is specified.

Upgrading Jenkins plugins

Jenkins plugins can be upgraded from the Jenkins dashboard; however, we suggest not doing so. Instead, use the configuration as code method, which is more reliable in handling situations where a plugin update may cause issues. Also, update Jenkins plugins when updating Jenkins.

Upgrading Jenkins plugins using the Jenkins Helm chart

To upgrade Jenkins plugins using the Jenkins Helm chart, follow these steps:

1. The **values.yaml** file allows you to specify the Jenkins plugins to install during the Jenkins setup. There are two places inside the **values.yaml** file where plugin versions are listed: the **installPlugins:** section and the **additionalPlugins: section**. To update the plugins, just update the version information for each and save the **values.yaml** file:

```
controller:
  installPlugins:
    - kubernetes:3937.vd7b_82db_e347b_
    - workflow-aggregator:596.v8c21c963d92d
    - git:5.1.0
    - configuration-as-code:1647.ve39ca_b_829b_42
  additionalPlugins:
    - sonar:3.1.6
```

2. Next, run the **helm upgrade** command to upgrade your Jenkins plugins:

```
helm upgrade -f values.yaml myjenkins jenkins/jenkins
```

Upgrading Jenkins plugins using a shell script

To upgrade Jenkins on a cloud-based VM, follow these steps:

1. Download the *Plugin Installation Manager Tool* for Jenkins. If you have followed the steps in the *Deploying Jenkins to a cloud instance* section, then you will have already downloaded it to the **/tmp/** directory of the VM running your Jenkins server.

Nevertheless, you can run the following command to download it:

```
wget https://github.com/jenkinsci/plugin-installation-manager-  
tool/releases/download/2.12.11/jenkins-plugin-manager-2.12.11.jar -P /tmp/
```

2. Next, update your `plugins.txt` file with the new version for each Jenkins plugin that you wish to upgrade:

```
configuration-as-code:1647.ve39ca_b_829b_42  
git:2.9.0  
sonar:4.1.0
```

3. Next, run the following command to download the newer versions of the Jenkins plugins to the Jenkins plugin directory:

```
sudo java -jar /tmp/jenkins-plugin-manager-2.12.11.jar \  
--war /usr/share/java/jenkins.war \  
-d /var/lib/jenkins/plugins \  
--plugin-file /tmp/plugins.txt
```

4. For the changes to take effect, simply restart Jenkins.

By using these methods, you can seamlessly upgrade your Jenkins installation while ensuring the correct version is specified.

IMPORTANT NOTE

The process of upgrading Jenkins and Jenkins plugins is described separately, but it is recommended to perform them together for a comprehensive upgrade.

It is crucial to version control both the `values.yaml` file and the `plugins.txt` file. This practice proves invaluable in the event of a rollback requirement, providing the ability to revert to a previous state if needed.

Monitoring for issues

An effective approach is to execute a dedicated Jenkins CI Pipeline that's designed to identify potential issues resulting from the Jenkins or Jenkins plugin upgrade. This Pipeline, acting as a smoke test, may not provide comprehensive testing, but it aids in the early detection of immediate issues, enabling prompt resolution.

Implementing backup strategies for Jenkins build data

So far, we've seen almost every aspect of the Jenkins setup being managed as code, allowing it to be version-controlled and reproducible. We saw infrastructure being defined as code using Terraform, the Jenkins application being installed through code using a Helm chart, and the Jenkins configuration being defined as code using the JCasC plugin. In the upcoming chapters, we will see the Jenkins Pipeline as Code too. See *Figure 2.19*. These DevOps practices greatly enhanced the resilience of our solution as they allow us to reproduce and deploy Jenkins instances consistently across different environments:

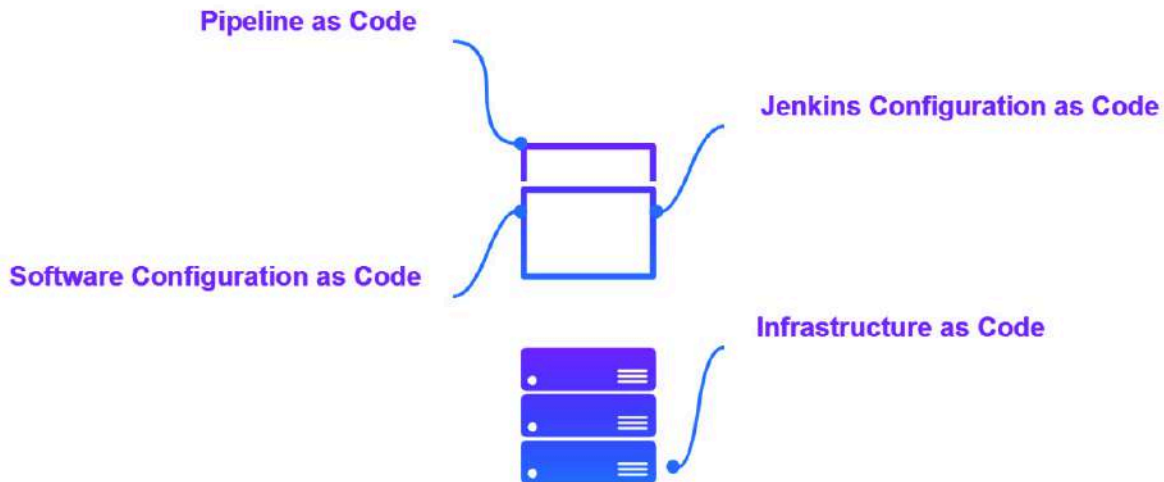


Figure 2.19 – Various aspects of the Jenkins server as code

Having said that, the only thing that remains to be backed up is the Jenkins build logs. However, these logs are only useful during debugging build failures. Once the problem has been identified, the build logs lose their relevance.

The retention period for Pipeline build logs can vary depending on the specific needs and preferences of each team. There is no universal standard or fixed duration for how long teams should keep these logs. The decision on retention duration often depends on the maturity of the CI/CD process, and the overall software agility of the team.

However, based on industry observations and best practices, teams with frequent releases often keep build logs for a shorter duration, typically ranging from a few days to a couple of weeks. Since these teams have a high release cadence, the need for retaining build logs diminishes quickly after a successful deployment. In such cases, the emphasis is often on monitoring real-time metrics and logs for immediate issue detection and resolution rather than relying heavily on historical build logs.

Having said that, when it comes to backing up Jenkins data, the two methods discussed for setting up Jenkins make sure Jenkins data is stored on a persistent volume or a managed disk. Both these storage options can be regularly backed up using the Azure Backup feature available on the Azure cloud. To learn more about backing up the persistent volume, refer to the documentation on backing up AKS using **Azure Backup** [18], and for backing up the managed disk, refer to the documentation on backing up **Azure managed disks** [19].

Summary

In this chapter, we looked at the three most important components of running a Jenkins setup: planning, deploying, and maintenance. We began the planning process by delving into the

complexities of the Jenkins server architecture. Furthermore, we investigated critical aspects for selecting the appropriate hardware specs, including network connections and JVM settings. Our analysis continued with an in-depth examination of numerous deployment scenarios, revealing the distinct advantages of each technique. In addition, we learned about JCasC and its purpose and functionality.

Shifting our focus to deployment, we gained hands-on experience by practically deploying Jenkins on a managed Kubernetes cluster. We leveraged the user-friendly Jenkins Helm chart to streamline the setup process. Additionally, we explored deploying Jenkins on a cloud-based VM using Terraform and customized scripts, expanding our deployment knowledge.

With the setup complete, we turned to managing the Jenkins setup. We gained great insights into the complexities of upgrades, as well as the need to keep Jenkins plugins up to date. Finally, we looked into backup options to protect Jenkins build data.

Having said that, for the upcoming chapter and beyond, we will leverage the Jenkins setup running on a Kubernetes cluster using a Helm chart as our foundation to further enhance and expand its capabilities to perform CI.

In the next chapter, you will learn how to extend Jenkins by configuring additional critical components of the CI/CD infrastructure, while also being introduced to the array of features offered by Jenkins.

Questions

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

1. Which of the following statements is false?
 - A. The Jenkins master node is primarily responsible for executing build jobs, while the agent nodes handle scheduling and management tasks.
 - B. Plugins are used to extend the capabilities of Jenkins, allowing for customization and integration with other tools.
 - C. Pipelines define the sequence of steps for the build, test, and deployment processes, usually written in a domain-specific language.
 - D. Credentials in Jenkins are utilized for secure access to external systems and services, such as source code repositories, artifact repositories, and deployment environments, by providing necessary authentication and authorization details.
2. What is the primary purpose of JCasC?
 - A. To present Jenkins Pipelines as code.
 - B. To provide a way to define and manage Jenkins configurations using human-readable text files.
 - C. To enhance the security of Jenkins by encrypting its configuration files.

- D. To enable real-time monitoring and logging of Jenkins.
3. Which of the following platforms is suitable for running Jenkins?
- A. Any platform with JRE support, including Windows, Linux, and macOS.
 - B. Exclusively on Linux-based systems due to its dependency on specific Linux kernel features.
 - C. Only on cloud-based platforms such as AWS, Azure, and Google Cloud Platform.
 - D. Solely on dedicated hardware servers due to its resource-intensive nature.
4. What is a recommended practice when upgrading Jenkins to a new version?
- A. Directly overwrite the existing Jenkins installation with the new version.
 - B. Create a backup of Jenkins data and configurations before initiating the upgrade process.
 - C. Upgrade all plugins before upgrading Jenkins to ensure compatibility with the new version.
 - D. Temporarily disable all security settings in Jenkins to avoid conflicts during the upgrade.
5. Which of the following makes Jenkins more resilient to failures?
- A. Jenkins Pipeline as Code.
 - B. JCasC.
 - C. IaC.
 - D. All the above.

Answers

Here are the answers to this chapter's questions:

1. Correct answer: **A**

Explanation:

In Jenkins, the master node is responsible for scheduling build jobs, managing the Jenkins UI, and overall process management. Though the master node can also run Jobs, it's the agent node that is primarily responsible for running the build jobs.

2. Correct answer: **B**

Explanation of incorrect answer choices:

A. This is incorrect. Jenkins Declarative or Scripted Pipeline syntax is used to write Jenkins Pipeline as Code, not JCasC.

C. Security is important in Jenkins but, JCasC's main role is not to encrypt configuration files. It's about managing configuration as code, although it can include aspects of security configuration.

D. Real-time monitoring and logging are separate concerns, typically handled by other tools through Jenkins plugins. JCasC aims to manage the configurations on the Jenkins server.

3. Correct answer: **A**

Explanation of incorrect answer choices:

B. This is incorrect because Jenkins does not rely on Linux-specific features and can run on any platform that supports Java, including Windows and macOS.

C. While Jenkins can be installed on cloud platforms, it is not restricted to them. Jenkins can be installed on local servers, VMs, or even personal computers, so long as they support Java.

D. Jenkins does not require dedicated hardware servers. It can run on various types of hardware, including personal computers, containers, and VMs. The necessity for dedicated hardware depends more on the scale and requirements of the specific Jenkins setup rather than a general requirement of the tool.

4. Correct answer: **B**

Explanation of incorrect answer choices:

A. This is not recommended because directly overwriting could lead to loss of data or configuration settings if the upgrade encounters issues. It's safer to back up first.

C. Keeping plugins updated is important, but it's generally recommended to upgrade Jenkins first and then deal with plugins. Upgrading plugins first can lead to compatibility issues if they require a newer Jenkins version than you have.

D. Disabling security settings is not advisable as it can expose the system to potential risks. Proper upgrade procedures should not require turning off security features.

5. Correct answer: **D**

Explanation:

Using Jenkins Pipeline as Code ensures the reproducibility of Jenkins Pipelines, while JCasC achieves the same for the server's configuration, and IaC replicates this for the underlying Jenkins infrastructure. The notable exception in this reproducibility model is the build logs.

Further reading

If you wish to learn more about some of the concepts that weren't covered in-depth in this chapter or are outside the scope of this book but relevant, we are happy to provide you with some references:

- [1] *Well-Architected Framework*: <https://learn.microsoft.com/en-us/azure/well-architected/>

- [2] *General-purpose virtual machine sizes*: <https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-general>
- [3] *JDK 11 Oracle documentation*: <https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE>
- [4] *Jenkins Configuration as Code demos*: <https://github.com/jenkinsci/configuration-as-code-plugin/tree/master/demos>
- [5] *Azure portal*: <https://azure.microsoft.com/en-us/free/>
- [6] *Learning Kubernetes Basics*: <https://kubernetes.io/docs/tutorials/kubernetes-basics/>
- [7] *Download and Install Visual Studio Code*: <https://code.visualstudio.com/docs/setup/setup-overview>
- [8] *Downloading Git*: <https://git-scm.com/downloads>
- [9] *Downloading and Installing the Azure CLI*: <https://learn.microsoft.com/en-us/cli/azure/install-azure-cli>
- [10] *Downloading and Installing the Kubernetes CLI (kubectl)*: <https://kubernetes.io/docs/tasks/tools/#kubectl>
- [11] *Downloading and Installing Helm*: <https://helm.sh/docs/intro/install/>
- [12] *Nginx Ingress controller Helm chart documentation*: <https://github.com/kubernetes/ingress-nginx/tree/main/charts/ingress-nginx>
- [13] *Azure Provider Documentation*: <https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs>
- [14] *Downloading and Installing the Terraform CLI*: <https://developer.hashicorp.com/terraform/downloads>
- [15] *Plugin Installation Manager Tool*: <https://github.com/jenkinsci/plugin-installation-manager-tool>
- [16] *Jenkins LTS Changelog page*: <https://www.jenkins.io/changelog-stable/>
- [17] *Jenkins Docker Hub page*: <https://hub.docker.com/r/jenkins/jenkins/tags>
- [18] *Back up Azure Kubernetes Service using Azure Backup*: <https://learn.microsoft.com/en-us/azure/backup/azure-kubernetes-service-cluster-backup>
- [19] *Back up Azure Managed Disks*: <https://learn.microsoft.com/en-us/azure/backup/backup-managed-disks>

Securing Jenkins

In this chapter, the key aspects of securing Jenkins will be presented. Security is of paramount importance as Jenkins deals with sensitive data and plays a critical role in the software delivery pipeline. In the previous chapter, we partially explored how to secure Jenkins, including the server setup process, the underlying infrastructure, and plugin management, all aimed at safeguarding Jenkins installations from potential threats and vulnerabilities. In this chapter, we will explore vital measures to enhance the security around who gets to do what on your Jenkins instance. First, we will delve into user authentication and permissions by integrating Jenkins with Azure **Active Directory (AD)** (see [1] in the *Further reading* section at the end of this chapter). This integration ensures a centralized and secure user management system, leveraging Azure AD's robust authentication capabilities. This section will guide you through the setup process and demonstrate how to enforce fine-grained access controls and permissions within Jenkins. After that, we will take a glance at **cross-site request forgery (CSRF)** protection settings inside Jenkins. Lastly, we will explore the powerful Jenkins Credentials feature, which allows for secure storage and usage of sensitive information, such as passwords, API keys, and certificates. By mastering these security practices, you can safeguard your Jenkins environment and protect valuable assets within your CI/CD pipeline.

After completing this chapter, you should be able to gain proficiency in configuring user authentication and permissions within Jenkins, ensuring secure access control to the system. You will have also explored the versatile Credentials feature in Jenkins, understanding the different types and their scopes, enabling seamless and secure integration with external systems.

As such, we will be covering the following main topics in the chapter:

- Configuring user authentication and permissions
- Understanding CSRF protection in Jenkins
- Using Jenkins Credentials

Technical requirements

Starting from this chapter, we will work on a Jenkins server running on **Azure Kubernetes Service (AKS)** within the Azure platform. However, it's important to note that the steps mentioned can also be applied to a Jenkins server running on any type of Kubernetes cluster, regardless of the underlying infrastructure.

To proceed with this chapter, please ensure that you have a Jenkins server up and running, and the following tools installed on your local development environment:

- An IDE such as Visual Studio Code or anything you choose
- **kubect1**, the Kubernetes command-line tool
- The Helm CLI, to deploy workloads on the Kubernetes cluster

Configuring user authentication and permissions

When you set up Jenkins for the first time, Jenkins asks you to create an admin account and install some basic plugins through the Jenkins Setup Wizard. If you read the previous chapter, you will know that the Jenkins Helm chart performs this for you automatically. Jenkins initiates its own user database out of the box to allow you to create users. However, using Azure AD or LDAP-based authentication to grant people access to Jenkins is a popular choice among organizations.

The following section will walk you through the process of integrating Jenkins with Azure AD to establish a robust and secure authentication and permissions management system.

Registering Jenkins on Azure AD

This section provides a step-by-step guide to registering your Jenkins application in Azure AD. This involves navigating to the Azure portal, accessing the **App registrations** section, and creating a new registration. You will learn how to set up a reply URL and generate a client secret, which will be utilized for authentication purposes in Jenkins.

Follow these steps:

1. Navigate to the Azure portal (portal.azure.com) and sign in with your Azure account.
2. From the dashboard, click on **Azure Active Directory** and then select **App registrations** from the menu.
3. Click on the **New registration** button to create a new app registration. This can be seen in the following screenshot:

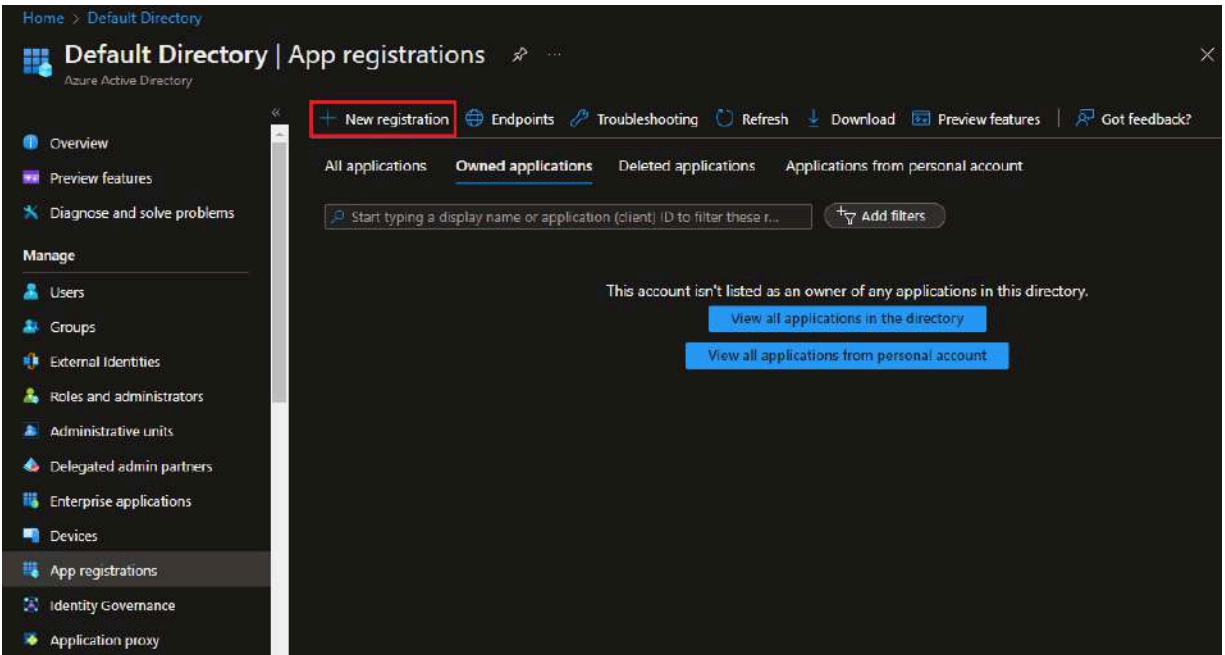


Figure 3.1: Initiating a new app for registration

4. Provide a name for your Jenkins app using the **Name** field and ensure that **Supported account types** is set to the appropriate option based on your requirements (for example, **Accounts in this organizational directory only**).
5. In the **Redirect URI** section, set the app type to **Web** and set the reply to the URL to `https://<your_jenkins_host>/securityRealm/finishLogin`. Make sure you replace `<your_jenkins_host>` with the actual host name of your Jenkins instance.
6. Click on **Register** to create the app registration:

Home > Default Directory | App registrations >

Register an application ...

Name

The user-facing display name for this application (this can be changed later).

Jenkins-dev ✓

Supported account types

Who can use this application or access this API?

- ☒ Accounts in this organizational directory only (Default Directory only - Single tenant)
- ☐ Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- ☐ Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)
- ☐ Personal Microsoft accounts only

[Help me choose...](#)

Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web ✓ https://20.76.118.213/securityRealm/finishLogin ✓

Register an app you're working on here. Integrate gallery apps and other apps from outside your organization by adding from [Enterprise applications](#).

By proceeding, you agree to the [Microsoft Platform Policies](#) ⓘ

Register

Figure 3.2: Registering Jenkins on Azure AD

- Next, on the **App registration** page, click on **Certificates & secrets**. On the resultant page, under the **Client secrets** section, click on + **New client secret** to generate a new secret key. On the pop-up page, provide a description and set an expiration period for your new secret key. Finally, click on the **Add** button:

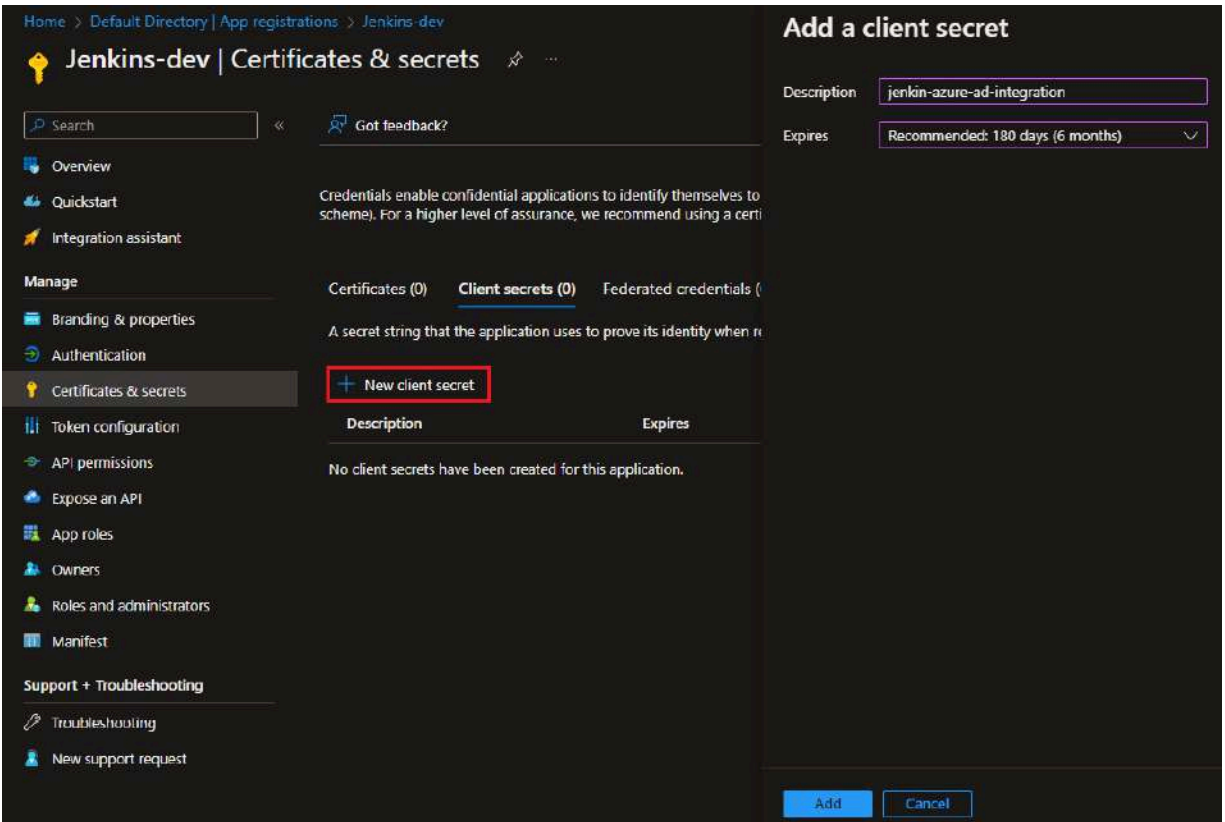


Figure 3.3: Creating a client secret

8. Once the client secret has been generated, copy and securely store the value as it will be used in Jenkins to configure the Azure AD plugin.
9. Next, from the left-hand side menu, click on **Authentication** and, under **Implicit grant and hybrid flows**, enable the **ID tokens** option:

Home > Default Directory | App registrations > Jenkins-dev

Jenkins-dev | Authentication

Search < Got feedback?

- Overview
- Quickstart
- Integration assistant

Manage

- Branding & properties
- Authentication**
- Certificates & secrets
- Token configuration
- API permissions
- Expose an API
- App roles
- Owners
- Roles and administrators
- Manifest

Support + Troubleshooting

- Troubleshooting
- New support request

Web
Redirect URIs: 1

Quickstart Docs

Front-channel logout URL

This is where we send a request to have the application clear the user's session data. This is required for single sign-out to work correctly.

e.g. <https://example.com/logout>

Implicit grant and hybrid flows

Request a token directly from the authorization endpoint. If the application has a single-page architecture (SPA) and doesn't use the authorization code flow, or if it invokes a web API via JavaScript, select both access tokens and ID tokens. For ASP.NET Core web apps and other web apps that use hybrid authentication, select only ID tokens. [Learn more about tokens.](#)

Select the tokens you would like to be issued by the authorization endpoint:

☐ Access tokens (used for implicit flows)

☒ ID tokens (used for implicit and hybrid flows)

Supported account types

Who can use this application or access this API?

☒ Accounts in this organizational directory only (Default Directory only - Single tenant)

☐ Accounts in any organizational directory (Any Azure AD directory - Multitenant)

Save Discard

Figure 3.4: Selecting ID tokens for implicit and hybrid flows

- (Optional) If you want to enable Azure AD group support, go to the **Manifest** tab and change the `groupMembershipClaims` value from `null` to `SecurityGroup`. Then, save your changes:

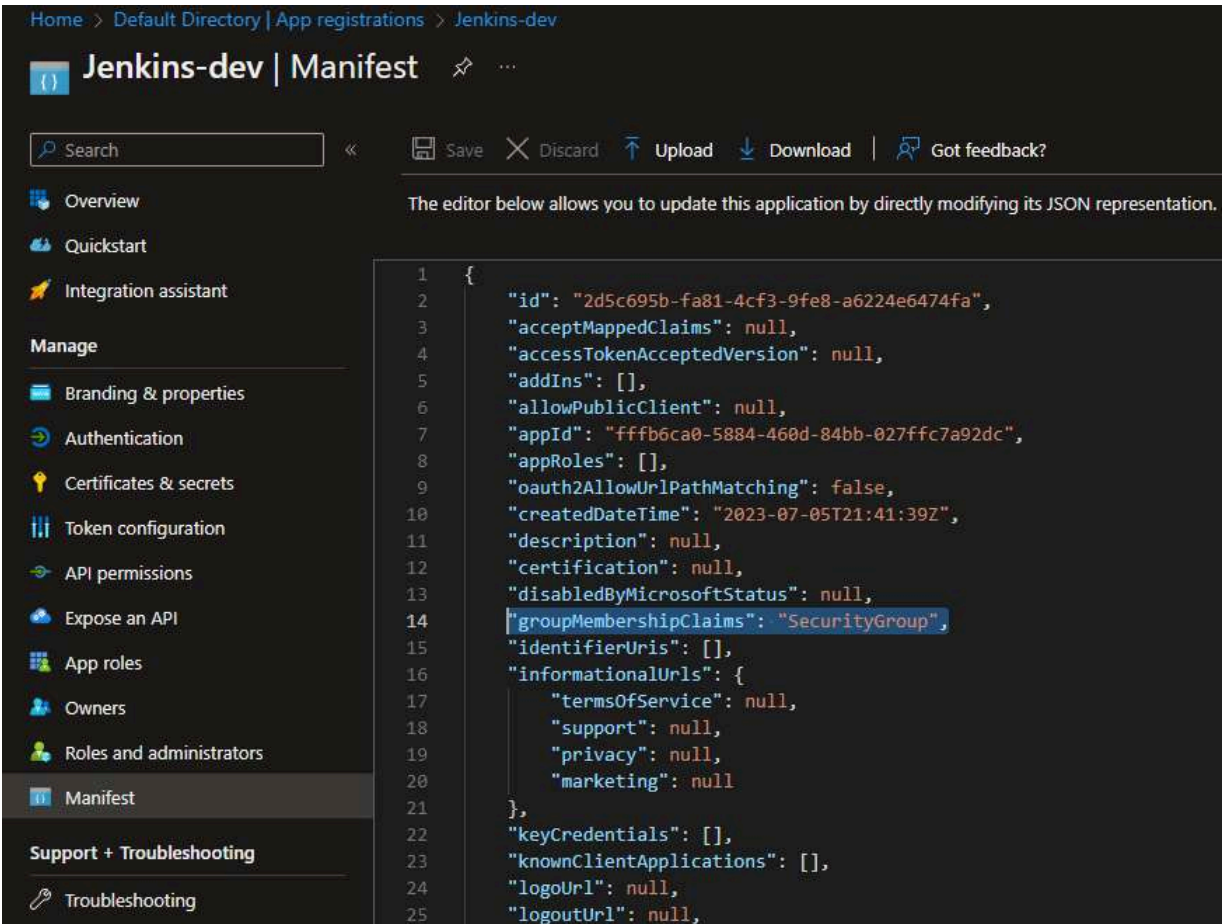


Figure 3.5: Enabling Azure AD group support

- Next, navigate to the **API permissions** section and click on + **Add a permission | Microsoft Graph | Application permissions**. Add permissions such as **User.Read.All**, **Group.Read.All**, and **People.Read.All**, as shown here:

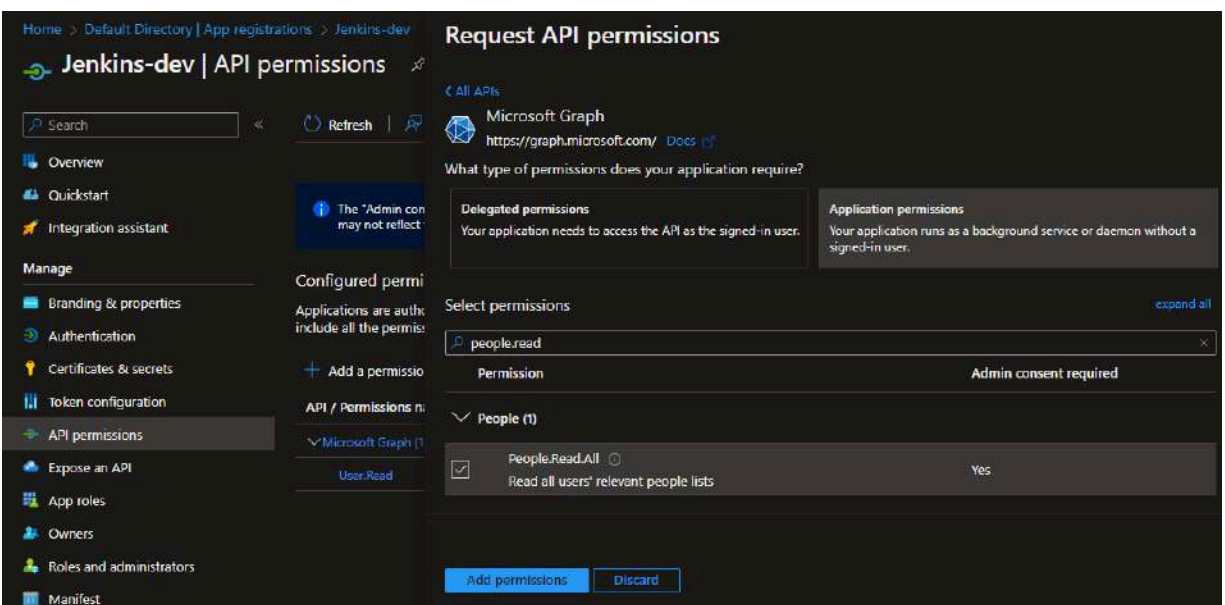


Figure 3.6: Adding API permissions to User, Group, and People

12. Next, grant admin consent for these permissions, ensuring Jenkins has the required access to retrieve user and group information. This can be seen here:

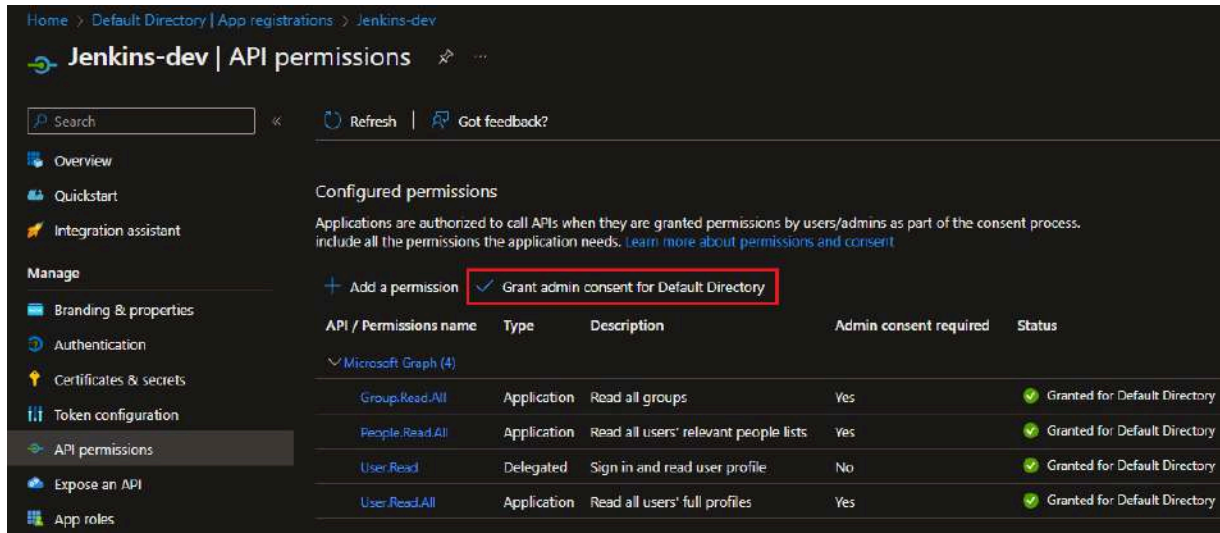


Figure 3.7: Granting API permissions and admin consent for Default Directory

13. With that, your Jenkins app will be registered in Azure AD.
14. Before we move on, make note of the **Application (client) ID** and **Directory (tenant) ID** values, as you will need them to configure Jenkins. You will find these details on the **Overview** page of your app, as shown here:

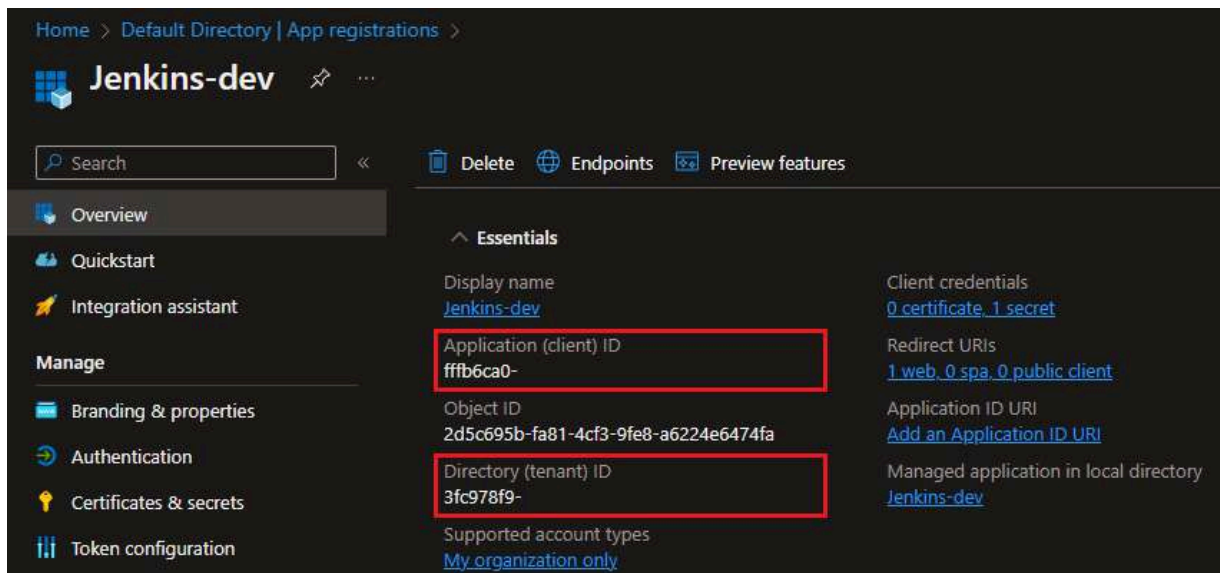


Figure 3.8: The app's overview page

15. Additionally, visit the **Users** section and take note of the **Object ID** value of a user from the Azure AD. We will use this to test the Azure AD configuration inside Jenkins to see whether Jenkins can retrieve user information from Azure AD.

Installing the Jenkins plugin for Azure AD

To install the Azure AD Jenkins plugin, you have two options: use the Jenkins dashboard or install it via the `values.yaml` file with the Jenkins Helm chart. Since we already know how to install or upgrade a Jenkins plugin using the `values.yaml` file, let's explore the alternative process. Follow these steps to install the plugin manually:

1. Log in to your Jenkins instance as an administrator and, from the Jenkins dashboard, click on **Manage Jenkins** in the left-hand menu.
2. From the options that appear, select **Manage Plugins**. On the resultant page, in the **Available** tab, locate the **Azure AD** plugin by either scrolling through the list or using the search bar:

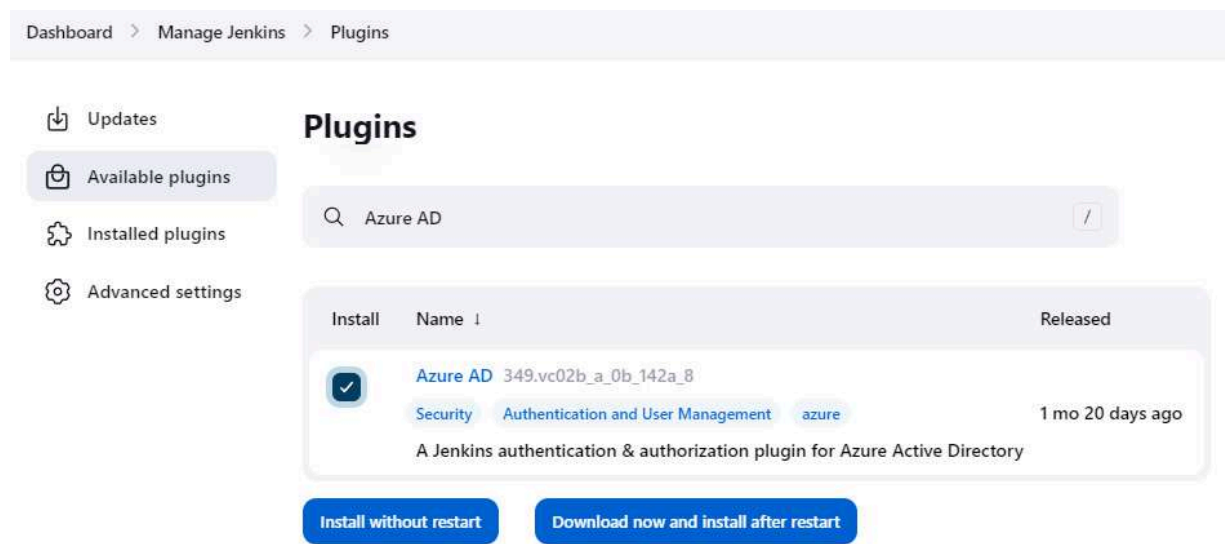


Figure 3.9: Installing the Jenkins plugin for Azure AD

3. Check the checkbox next to the **Azure AD** plugin to select it for installation and click on the **Install without restart** button to begin the installation process. Wait for Jenkins to download and install the Azure AD plugin. This may take a moment, depending on your internet connection and the size of the plugin. Once the installation is complete, you will see a notification at the top of the page indicating the success of the installation.
4. You do not necessarily need to restart Jenkins after installing the Azure AD plugin. The plugin should be ready to use immediately after the installation is complete. However, in some cases, Jenkins may recommend or require a restart to ensure proper functioning or to activate certain plugin features.

SUGGESTION

*It is possible to install, uninstall, disable, enable, and update plugins directly from the Jenkins dashboard. However, it is recommended to avoid doing so and instead adopt the **Jenkins Configuration as Code (JCasC)** approach for the same. This approach ensures reproducibility of the Jenkins configuration and provides better control over plugin versions. By managing plugins through configuration files, you can maintain consistency across different Jenkins instances and easily replicate the desired plugin setup.*

Configuring Azure AD settings in Jenkins

Moving on to the Jenkins configuration, we will explore the security settings. Here, you will learn how to enable Azure AD authentication, input the necessary credentials, and verify the application's validity within Jenkins. The following steps will guide you through the process of saving the configuration and testing the login functionality with Azure AD:

1. Log in to your Jenkins instance as an administrator and, from the Jenkins dashboard, click on **Manage Jenkins** in the left-hand menu. From the options that appear, select **Configure Global Security**.
2. On the **Security** page, scroll down to the **Security Realm** section and select the **Azure Active Directory** option from the drop-down menu.
3. Fill in the required Azure AD credential information:
 - **Client ID**: Enter the application (client) ID of your registered Jenkins app in Azure AD.
 - **Client Secret**: Enter the client secret key generated during the Azure AD app registration.
 - **Tenant**: Enter the Azure AD tenant ID:

The screenshot shows the Jenkins 'Security' configuration page. At the top, there is a breadcrumb trail: 'Dashboard > Manage Jenkins > Security'. Below this, the 'Security' title is followed by the 'Authentication' section. A checkbox labeled 'Disable remember me' is present. The 'Security Realm' section features a dropdown menu currently set to 'Azure Active Directory'. Below the dropdown are three input fields: 'Client ID' with a help icon and a value 'fffb6ca0-', 'Client Secret' with a help icon and a masked value '*****', and 'Tenant' with a help icon and a value '3fc978f9-'. Each field has a corresponding help icon (a question mark in a circle).

Figure 3.10: Configuring the Azure AD settings in Jenkins

4. Now, verify your settings and see whether you can also find an AD user. To do that, add the **Object ID** property of an AD user under the **Test user principal name or object id** field, and click on the **Verify Application** button to ensure that the provided configuration is valid. Jenkins will attempt to authenticate with Azure AD using the provided credentials. If the verification is successful, you will see a message indicating that the application was verified successfully:

Dashboard > Manage Jenkins > Security

Cache Duration ?

3600

☐ Callback URL from request ?

☐ Enable Single Logout ?

☐ Disable graph integration ?

Test user principal name or object id

c8b68c66-

Save any configuration changes before configuring authorization settings

Successfully verified, found display name: NIKHIL PATHANIA

Verify Application

Figure 3.11: Verifying the Azure AD configuration in Jenkins

Configuring user permissions in Jenkins

Now, let's explore the authorization aspect, which involves utilizing the Azure AD matrix-based security option. This option enables fine-grained permissions management based on user and group object IDs. This section demonstrates how to configure authorization rules by searching for users or groups, assigning appropriate permissions, and applying the changes:

1. Scroll down to the **Authorization** section on the **Security** page. You'll see that the default setting is **Logged-in users can do anything**; we don't want that. Replace it with **Azure Active Directory Matrix-based security**.
2. First, grant overall administrative privileges to **Authenticated Users** while ensuring **Anonymous** users have no access to the Jenkins dashboard. This ensures that users who are not logged in will see no content on the Jenkins dashboard. Click the **Save** button to apply the changes:

Dashboard > Manage Jenkins > Security

Authorization

Azure Active Directory Matrix-based security

User/group	Overall	Credentials	Agent	Job	Run	View	SCM	Metrics
	Administrator	Read Create Delete Update View	ManageDomains Configure Connect Build Cancel Build Disconnect Delete Create	Configure Cancel Build Disconnect Delete Create Discover Move Read Workspace Delete	Replay Update Configure Delete	View Create Delete Read Tag	HealthCheck ThreadDump View	
Anonymous	☐	☐	☐	☐	☐	☐	☐	☐
Authenticated Users	☒	☐	☐	☐	☐	☐	☐	☐

Azure User/group to add

Start typing a name

Add

Figure 3.12: Providing admin privileges to Authenticated Users

- Now, log out from Jenkins, and log in using an AD user. Return to the **Authorization** section on the **Security** page.
- To add a user or group for authorization, in the search field next to **Azure User/group to add**, enter the display name of the user or group you want to add and click on the **Add** button:

Dashboard > Manage Jenkins > Security

Authorization

Azure Active Directory Matrix-based security

User/group	Overall	Credentials	Agent	Job	Run	View	SCM	Metrics
	Administrator	Read Create Delete Update View	ManageDomains Configure Connect Build Cancel Build Disconnect Delete Create	Configure Cancel Build Disconnect Delete Create Discover Move Read Workspace Delete	Replay Update Configure Delete	View Create Delete Read Tag	HealthCheck ThreadDump View	
Anonymous	☐	☐	☐	☐	☐	☐	☐	☐
Authenticated Users	☒	☐	☐	☐	☐	☐	☐	☐
NIKHIL PATHANIA	☒	☐	☐	☐	☐	☐	☐	☐

Azure User/group to add

Start typing a name

Add

Figure 3.13: Providing admin privileges to an AD user

- In the permission column, provide overall administrative privileges to the logged-in AD user, which is the new admin user on Jenkins, and change the **Authenticated Users** permissions to read overall. If you tilt your head and read the columns of the following matrix, you will notice various privileges segregated based on the Jenkins items. This mechanism allows you to control

who can do what. In the preceding figure, you can see I have given zero access to **Anonymous** users, read access to **Authenticated Users**, and admin privileges to myself.

6. Continue adding users or groups as needed, assigning the desired permissions for each. To remove a user or group, click the **X** icon next to their entry in the list.
7. Once you have configured the authorization rules, click on the **Save** button to apply the changes.

After completing these steps, AD users can now log in to Jenkins using their Azure AD credentials, and permissions can be managed based on the Azure AD user and group settings.

Speaking of **JCasC**, while it's not available for configuring the Azure AD plugin, it is possible and highly useful for configuring Azure AD matrix-based security authorization. As the number of Azure AD users and groups accessing Jenkins grows, maintaining extensive lists of user and group permissions as code becomes crucial. The following is an example that demonstrates the configuration described in *Figure 3.13*:

values-1.yaml

```
controller:
  jenkinsUrl: http://<Jenkins Ingress Public IP>/
  JCasC:
    configScripts:
      welcome-message: |
        jenkins:
          systemMessage: Welcome...
      authorizationStrategy: |
        azureAdMatrix:
          permissions:
            - "GROUP:Overall/Read:authenticated"
            - "USER:Overall/Administer:<Group ID>"
```

If you wish to use this code, save it as `values.yaml`. Now, let's quickly look at the CSRF protection settings in Jenkins to prevent malicious attacks.

Understanding CSRF protection in Jenkins

CSRF protection settings in Jenkins are designed to prevent malicious attacks that can trick users into performing unintended actions. CSRF attacks occur when an attacker tricks a user's browser into making unintended requests to a target website, potentially causing unauthorized actions to be executed. In Jenkins, CSRF protection is enabled by default to safeguard against such attacks. The CSRF protection settings ensure that requests made to Jenkins are verified and validated to prevent unauthorized access or actions. Let's understand this with an example.

In certain scenarios, you may need to trigger a Jenkins pipeline using an external webhook. For instance, when a CI pipeline uploads a new artifact to Artifactory, you may want the Artifactory webhook plugin to trigger a specific performance testing pipeline in Jenkins. External webhooks

involving Jenkins typically utilize Jenkins' REST API, which requires authentication. However, it is crucial to protect against CSRF attacks that could exploit Jenkins REST APIs through methods such as spear phishing and trigger undesired activities within Jenkins.

To prevent such attacks, the REST API call in your webhook must include a crumb along with the usual authentication to confirm its legitimacy. This crumb acts as an additional security measure, verifying that the request originates from an authenticated source and is not a malicious attempt to manipulate Jenkins.

The CSRF protection settings in Jenkins are as follows:

- **Crumb issuer:** Jenkins uses a unique value called a *crumb* to verify requests. The **Default Crumb Issuer** option generates and manages these crumbs. By default, Jenkins generates the crumb using a random string, but you can also configure a custom crumb issuer if needed.
- **Proxy compatibility:** This option allows you to configure Jenkins to work with reverse proxies or load balancers that modify the request headers. Enabling this setting ensures that CSRF protection functions correctly in such environments:



Figure 3.14: CSRF Protection settings inside Jenkins

In the upcoming chapters, you will learn how to generate a Jenkins crumb and use it with Jenkins APIs.

Using Jenkins Credentials

Jenkins collaborates with a multitude of tools to facilitate automation, CI, CD, and more. To ensure smooth integration between Jenkins and these third-party tools, Jenkins offers a dedicated feature called **Credentials**. This feature allows you to securely manage and configure the necessary authentication details for seamless interaction with external tools. In this section, we will learn about the default credentials provided by Jenkins and their scope.

Types of credentials

To suit various needs, Jenkins provides you with at least five different kinds of credentials to choose from. They are as follows:

- Username and password
- Secret text
- Secret file
- SSH username with private key
- Certificate

In addition to these, when certain plugins are installed in Jenkins, other types of credentials become available. For example, installing the Jenkins plugin for Kubernetes adds a new credential type:

Kubernetes Service Account.

Having said that, let's take a closer look at the **Username and password** credential. It is one of the simplest and most widely used credentials in Jenkins. As its name suggests, the **Username and password** credentials allow you to save a username and a password in Jenkins. Follow these steps to create one:

1. From the Jenkins dashboard, navigate to **Manage Jenkins | Credentials | System | Global credentials (unrestricted)** and click **+ Add Credentials**.
2. On the resultant page, you will find the following settings to configure:
 - The **Kind** field allows you to choose the type of credential you want to create. Notice that the **Username and password** credential is selected by default.
 - The **Username** and **Password** fields enable you to specify a username and password, respectively.
 - The **ID** field allows you to assign a unique string to your credentials, which are used within your Jenkins pipeline code for identification and retrieval.
 - Following that is the **Description** field, which provides an opportunity to include a brief note about the credentials. It is considered good practice to include a description as it aids in comprehending the purpose of the credentials for others.
3. To save the credentials, click the **Create** button located at the bottom of the page:

The image shows the 'Create Username with password' form in Jenkins. At the top, there is a 'Kind' dropdown menu set to 'Username with password'. Below this is a 'Scope' dropdown menu set to 'Global (Jenkins, nodes, items, all child items, etc)'. The 'Username' field is a text input. Below it is a checkbox labeled 'Treat username as secret'. The 'Password' field is a text input. Below that is an 'ID' text input. At the bottom is a 'Description' text input. A blue 'Create' button is located at the bottom left of the form.

Figure 3.15: Creating Username and Password credentials in Jenkins

Once saved, you will find your newly created credentials listed under the **Global credentials (unrestricted)** page. The credentials are now accessible and can be utilized throughout Jenkins for various purposes. To edit the credentials, simply click on the **Update** button. However, it's important to note that once credentials are created, the ID cannot be modified.

As you continue reading this chapter, you will gain further insight into creating and utilizing credentials. Now, let's investigate the scope of credentials.

Scope of credentials

Credentials in Jenkins are categorized based on their scope. Global credentials are accessible throughout Jenkins, while specific credentials are reserved for individual items such as folders or pipelines. Jenkins credentials can be segregated into three categories, as outlined here:

- Global credentials

- Folder/pipeline-level credentials
- User-level credentials

Global credentials

Global credentials in Jenkins refer to credentials that are available and accessible across the entire Jenkins instance. These credentials can be used by any job or pipeline within Jenkins, regardless of their location or configuration. Additionally, various global settings under the **Manage Jenkins** page also use global credentials.

To create global credentials, as we saw earlier, navigate to **Manage Jenkins | Credentials | System | Global credentials (unrestricted)** and click **+ Add Credentials**.

Folder/pipeline-level credentials

Credentials at the folder or pipeline level provide a way to scope credentials to specific folders or individual pipelines within Jenkins. This allows for more granular control over the usage and visibility of credentials within a Jenkins environment.

At the folder level, you can create credentials that are specific to that particular folder. These credentials are only available to jobs or pipelines residing within that folder. This ensures that the credentials are contained within the folder and not accessible to other jobs or pipelines outside of it. See *Figure 3.16* for more details.

Similarly, at the pipeline level, you can define credentials that are specific to a particular pipeline. These credentials are only accessible within that pipeline and cannot be accessed by other pipelines or jobs.

This scoping of credentials to folders or pipelines provides an additional layer of security and isolation. It allows you to restrict the usage of sensitive credentials to specific areas of your Jenkins instance, ensuring that they are only available to the jobs or pipelines that genuinely require them. It also helps in managing and organizing credentials by associating them with the relevant folders or pipelines.

By utilizing folder or pipeline-level credentials, you can maintain tighter control over sensitive information and reduce the risk of accidental exposure or unauthorized access across your Jenkins environment:

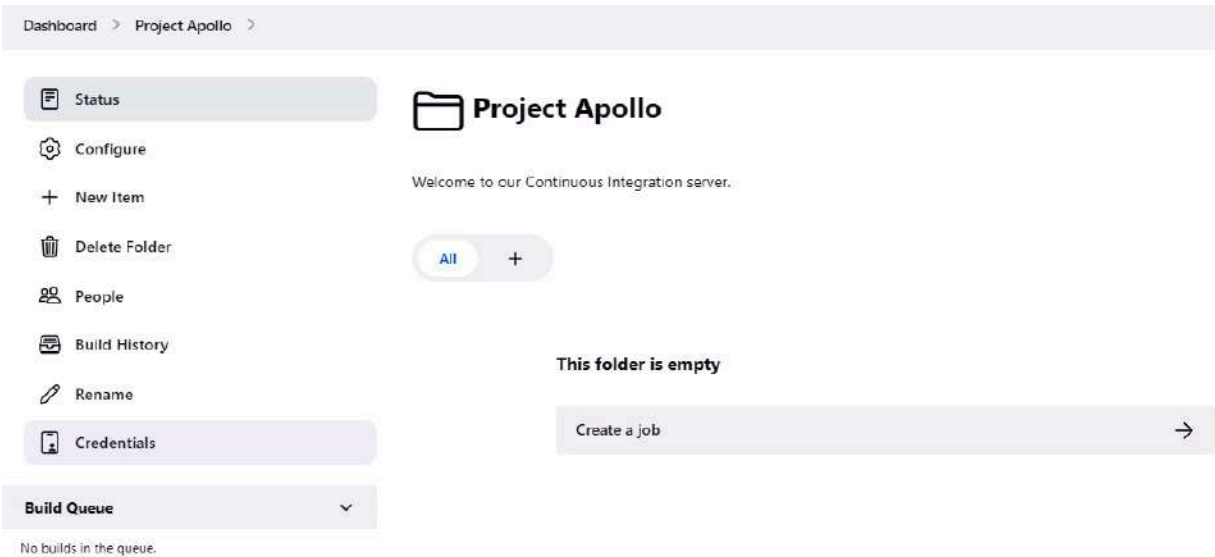


Figure 3.16: Option to create credentials at the folder level

User-level credentials

Lastly, user-level credentials refer to credentials that are associated with individual users or accounts. These credentials are specific to each user and can be used for various purposes within the Jenkins instance. However, we should try to avoid creating Jenkins credentials at the user level for the following reasons:

- **Dependency on individual users:** User-level credentials are tied to individual user accounts. This can lead to dependencies on specific users for certain operations or workflows. If a user leaves the organization or their account becomes inactive, it can cause issues with the associated user-level credentials and potentially disrupt automated processes.
- **Limited visibility and manageability:** User-level credentials are typically managed by individual users themselves, making it challenging for administrators to centrally manage and oversee the credentials. It can be difficult to track, rotate, or revoke user-level credentials, when necessary, which can introduce security risks and hinder credential management practices.
- **Lack of consistency and standardization:** User-level credentials are specific to each user and may not follow consistent naming conventions or usage patterns. This lack of standardization can make it difficult to enforce best practices, maintain a clear overview of credentials, or implement consistent security policies across the Jenkins environment.
- **Limited control and auditability:** User-level credentials may lack the centralized control and auditability that comes with global or folder/pipeline-level credentials. This can make it more challenging to monitor and track credential usage, enforce access controls, or maintain compliance with security policies and regulations.

Securing secrets in JCasC

The previous section outlined the steps for creating a username and password credential within Jenkins using the Jenkins dashboard, which served its purpose of familiarizing ourselves with Jenkins' Credentials feature. However, as we intend to practice configuration as code, we would now like to achieve the same outcome using JCasC through the `values.yaml` file using the Jenkins Helm

chart. The `values.yaml` file for our Jenkins server, including a username and password credential, appears as follows:

```
controller:
  jenkinsUrl: http://<Jenkins Ingress Public IP>/
  JCasC:
    configScripts:
      welcome-message: |
        jenkins:
          systemMessage: Welcome...
    jenkinsCredentials: |
      credentials:
        system:
          domainCredentials:
            - credentials:
                - usernamePassword:
                    scope: GLOBAL
                    description: "username and password"
                    id: "username_pass"
                    password: password
                    username: username
```

Do you see anything peculiar in this file? Notice that the username and password, which are sensitive data, are simply kept in plain text. When this file is placed in a source code repository, it becomes available to more people than is required. Therefore, it is critical to prioritize protecting sensitive data within the `values.yaml` file, such as credentials.

There are several approaches; in this section, we will look at one that includes leveraging Kubernetes secrets [2]. Follow these steps:

1. Use the `kubectl create secret generic` command to create a new secret with two key-value pairs. Here is an example:

```
kubectl create secret generic mysecret \
  --from-literal=username_key=username \
  --from-literal=password_key=password \
  -n jenkins
```

2. After executing this command, Kubernetes will create a secret named `mysecret` in the `jenkins` namespace with two literals: `username_key` and `password_key`. Just ensure that the `jenkins` namespace exists before running this command.
3. Now, update the `values.yaml` file, as shown here:

values-2.yaml

```
controller:
  jenkinsUrl: http://<Jenkins Ingress Public IP>/
  existingSecret: mysecret
  JCasC:
    configScripts:
      welcome-message: |
        jenkins:
          systemMessage: Welcome...
    jenkinsCredentials: |
      credentials:
        system:
```

```
domainCredentials:
- credentials:
  - usernamePassword:
    scope: GLOBAL
    description: "username and password"
    id: "username_pass"
    password: ${password_key}
    username: ${username_key}
```

In the preceding code, the inclusion of the `existingSecret: mysecret` line instructs the Jenkins Helm chart to utilize the previously created Kubernetes secrets. Furthermore, the usage of plain text for the username and password has been replaced with variables corresponding to the respective key names.

4. Run the `helm upgrade` command to put the changes into effect:

```
helm upgrade -n jenkins -f values.yaml \
myjenkins jenkins/jenkins
```

Running this command will create a Jenkins credential of the username and password type, all using JCasC, securely. This is how you secure sensitive data inside JCasC.

Our exploration of security doesn't end here. Security is a multi-faceted aspect, and in the upcoming chapters, we will delve into additional methods to secure various aspects of the CI/CD system.

Summary

In this short chapter, we started by exploring the process of configuring user access and permissions by seamlessly integrating Jenkins with Azure AD. Additionally, we delved into the essential concept of credentials within Jenkins by creating and managing them throughout this chapter. To ensure enhanced security, we acquired knowledge on safeguarding credentials within the `values.yaml` file by leveraging Kubernetes secrets. These topics collectively constituted the comprehensive security aspect that was covered in this chapter.

Moving forward, in the next chapter, our focus will be on extending the functionalities of Jenkins, wherein we will embark on setting up SonarQube and Artifactory through the utilization of a Helm chart, accompanied by an Nginx Ingress controller and a public IP for each service. Subsequently, we'll explore the necessary configurations within Jenkins to seamlessly integrate it with SonarQube, Artifactory, and GitHub.

Questions

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

1. Assuming Jenkins is integrated with Azure AD, which of the following practices is most appropriate?

A. All authenticated users should be granted overall administrator privileges.

- B. Anonymous users should have, at a minimum, overall read privileges.
 - C. Only a select group of authenticated users should be granted overall administrator privileges.
 - D. Authenticated users may have all permissions, except for deletion permissions.
2. Which of the following is not a type of Jenkins credential?
- A. User token
 - B. Certificate
 - C. Secret file
 - D. SSH username and private key
3. Which of the following statements is incorrect?
- A. Global-level credentials are available and accessible across the entire Jenkins instance.
 - B. Folder-level credentials are accessible to all subfolders and pipelines within that folder.
 - C. Pipeline-level credentials are available only to the respective pipeline.
 - D. User-level credentials in Jenkins, while scoped to the individual user who created them, can be used inside any Jenkins pipeline.
4. How should credentials be secured when using JCasC?
- A. Credentials should be directly embedded in the JCasC YAML file for easy access.
 - B. Credentials should be stored in a separate, encrypted secrets file and referenced in JCasC.
 - C. Credentials are automatically encrypted by JCasC so that they can be plainly written in the configuration.
 - D. It is recommended to use environment variables within JCasC for storing credentials.
5. Enabling CSRF protection in Jenkins is optional since modern browsers inherently block all CSRF attacks:
- A. True
 - B. False

Answers

Here are the answers to this chapter's questions:

1. Correct answer: C

Explanation of incorrect answer choices:

A. Granting overall administrator privileges to all authenticated users means that every individual and group from an organization's AD would have administrative access. This is a significant security risk, as it gives too much control to every user.

B. Providing anonymous users with at least overall read privileges would allow anyone with access to the Jenkins server URL to view pipelines, build logs, and other sensitive information. This is particularly risky if the Jenkins URL is accessible over the internet, potentially exposing internal data to external parties.

D. Allowing every person and group from an organization's AD to have all permissions except deletion rights still permits excessive access. This approach enables users to perform and see actions beyond what is typically necessary or safe for standard users.

2. Correct answer: **A**

Explanation:

There is no specific Jenkins credential type named **user token**. To store a user token, you should use the **secret text** credential type.

3. Correct answer: **D**

Explanation:

User-level credentials in Jenkins, although scoped to the individual user who created them, can be used in the Jenkins pipeline to which the user has access.

4. Correct answer: **B**

Explanation of incorrect answer choices:

A. This is incorrect because embedding credentials directly in the JCasc file can pose a significant security risk as it may expose sensitive information in plain text.

C. This is not correct. JCasc does not automatically encrypt credentials; therefore, placing them plainly in the configuration file is not secure.

D. While using environment variables is a common practice for managing credentials, it's not the most secure method in the context of JCasc. The best practice is to use an encrypted secrets management system and reference these secrets in the JCasc file, not to store them as environment variables, which might be less secure and could be exposed in certain contexts.

5. Correct answer: **A**

Explanation:

While modern browsers have improved in mitigating some types of CSRF attacks, they do not block all CSRF attacks inherently. CSRF protection in Jenkins, or any web application, is an essential security measure. It ensures that web requests are made by the user intentionally and not by a malicious site or script. Relying solely on browser security can leave applications vulnerable

to certain types of CSRF attacks. Jenkins, like many web applications, provides its own CSRF protection mechanism that should be enabled to safeguard against these vulnerabilities.

Further reading

If you wish to learn more about some of the concepts that weren't covered in depth in this chapter or are outside the scope of this book but relevant, we are happy to provide you with some references:

- [1] *Azure Active Directory Fundamentals*: <https://learn.microsoft.com/en-us/azure/active-directory/fundamentals/>
- [2] *Kubernetes Secrets*: <https://kubernetes.io/docs/concepts/configuration/secret/>

Extending Jenkins

This chapter explores the expansive world of Jenkins enhancements, enabling you to tailor its functionalities for specific needs such as **continuous integration (CI)**. Enhancing Jenkins for CI demands the integration of additional tools and services, such as **SonarQube**, **Artifactory**, and a **version control system (VCS)**.

At its core, a VCS such as GitHub offers effortless source code version control, fostering seamless collaboration among developers. Meanwhile, SonarQube conducts static code analysis to pinpoint code quality issues, ensuring compliance with coding standards. Artifactory acts as a repository manager, handling artifacts and dependencies efficiently for builds. By integrating SonarQube for code analysis, Artifactory for artifact management, and a VCS for version control, Jenkins evolves into a comprehensive CI solution. This ensures heightened code quality, efficient artifact management, and seamless team collaboration.

After completing this chapter, you should be able to integrate Jenkins seamlessly with a version control tool, streamlining code management and enabling efficient collaboration within the CI pipeline. You will learn how to set up SonarQube, a powerful static code analysis tool, and learn how to integrate it with Jenkins for comprehensive code quality checks and analysis. By the end of this chapter, you will be able to establish a connection between Jenkins and Artifactory, a reliable binary repository manager, enabling efficient artifact management within the CI/CD process.

As such, we will be covering the following main topics in the chapter:

- Connecting Jenkins to a version control system
- Connecting Jenkins to SonarQube
- Connecting Jenkins to Artifactory

Technical requirements

To proceed with this chapter, please ensure that you have the following tools installed on your local development environment:

- An IDE such as Visual Studio Code or anything you choose
- Git
- The Azure CLI to interact with the Azure cloud
- **kubect1**, the Kubernetes command-line tool

- The Helm CLI to deploy workloads on the Kubernetes cluster

Connecting Jenkins to a version control system

Jenkins acts as a powerful automation tool that seamlessly integrates with version control platforms such as GitHub or GitLab. The following section demonstrates connecting Jenkins to a GitHub account using the GitHub plugin and making all the necessary configurations to allow Jenkins to run pipelines either by monitoring repositories for code changes or leveraging features such as webhooks and triggering automated CI pipelines driven by the events in the version control system.

Installing the Jenkins plugin for GitHub

The GitHub plugin enables seamless integration and interaction between Jenkins and GitHub. It offers a range of features and functionalities that enhance the CI/CD pipelines when working with GitHub repositories:

- **Webhook support:** The GitHub plugin enables Jenkins to receive webhook notifications from GitHub. Webhooks allow Jenkins to be notified automatically when certain events occur in the GitHub repository, such as code pushes, pull requests, and branch creations. This triggers Jenkins jobs or pipelines to start builds and perform automated CI/CD tasks.
- **Pull request integration:** The GitHub plugin provides seamless integration with GitHub pull requests. Jenkins can automatically build and validate code changes proposed in pull requests, run tests, and provide feedback on the pull request itself. This helps ensure the quality of the code before it gets merged into the main branch.
- **Build status and reporting:** Jenkins can report build statuses, test results, and other information back to GitHub. This allows developers and team members to quickly see the status of builds, track test results, and view other build-related information directly within the GitHub interface.
- **Integration with GitHub API:** The GitHub plugin leverages the GitHub API to perform various operations, such as creating comments on pull requests, setting commit statuses, and interacting with other GitHub-specific features.

Take the following steps to install the Jenkins plugin for GitHub:

1. Log in to your Jenkins instance as an administrator and navigate to **Manage Jenkins | Plugins | Available plugins**.
2. On the resultant page, locate the **GitHub** plugin by either scrolling through the list or using the search bar, as shown:

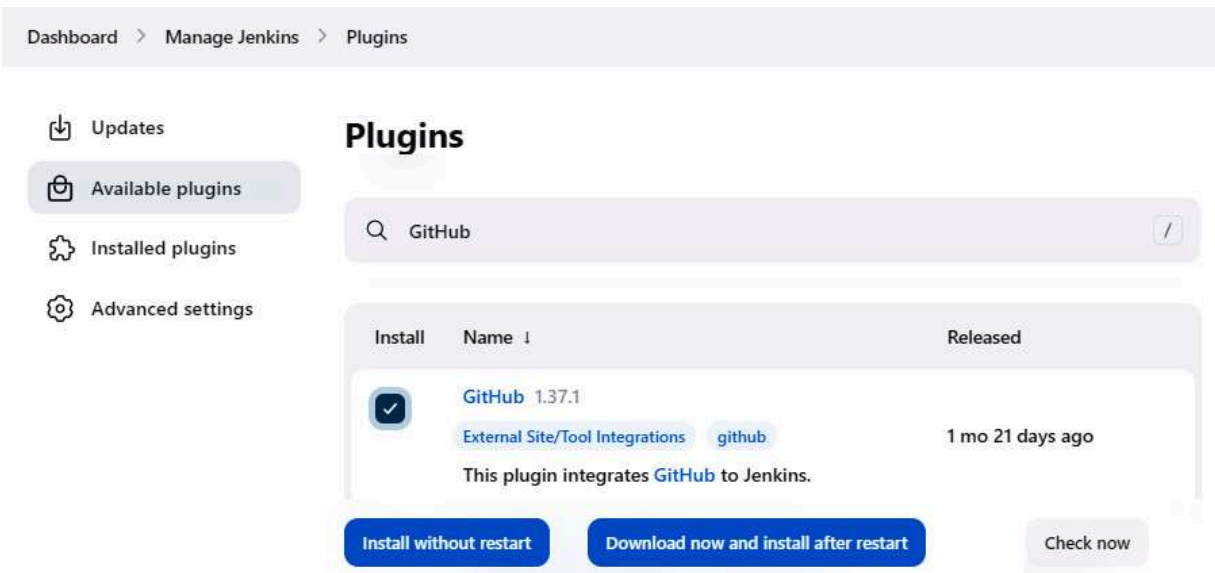


Figure 4.1 – Installing the Jenkins plugin for GitHub

3. Check the checkbox next to the **GitHub** plugin to select it for installation and click on the **Install without restart** button to begin the installation process. Wait for Jenkins to download and install the GitHub plugin. Once the installation is complete, you will see a notification at the top of the page indicating the success of the installation.
4. If you select the **Install without restart** option, you may be prompted to restart Jenkins. Follow the instructions to restart Jenkins for the plugin to take effect.

After the installation, the GitHub plugin will be available to configure on the Jenkins **System** page, enabling you to integrate Jenkins with GitHub and utilize its features for CI/CD workflows, including GitHub webhook triggers, pull request integration, and more.

Creating a personal access token on GitHub

A personal access token obtained from GitHub acts as a secure authentication mechanism, enabling Jenkins to interact with your GitHub account and repositories. By utilizing personal access tokens, an additional layer of security is established, as they separate the authentication process from your primary credentials, such as your username and password. These tokens can be generated, revoked, scoped, and managed independently, significantly reducing the risk associated with exposing your primary credentials in Jenkins configurations or logs.

With the personal access token, Jenkins acquires the necessary permissions to execute actions on your behalf, such as cloning repositories, creating webhooks, or posting comments on pull requests. To create a personal access token on GitHub with the specific scopes (`admin:repo_hook` and `repo`), follow these steps:

1. Log in to your GitHub account and click on your profile picture in the top-right corner of the page to select **Settings** from the drop-down menu.
2. In the left sidebar of the **Settings** page, scroll all the way down and click on **Developer settings**, and on the **Developer settings** page, click on **Personal access tokens**.
3. Once on the **Personal access tokens** page, click on the **Generate new token** button.
4. Enter a meaningful description for the token using the **Note** field to help you identify its purpose later. Additionally, set **Expiration** to anything, but not **No expiration**.
5. Next, in the **Select scopes** section, check the boxes for **admin:repo_hook** and **repo** to grant the required permissions to the token. The following table explains what the two scopes can do:

Token's scope	What it can do
admin:repo_hook	• This scope grants administrative access specifically for repository hooks. • Repository hooks are mechanisms that allow external systems, such as Jenkins, to receive notifications or trigger actions when certain events occur within a repository. • With the admin:repo_hook scope, Jenkins can create, modify, and delete repository hooks. • It provides full control over the hooks associated with the repositories.
repo	• The repo scope provides access to the repository-related actions and data. It covers various operations related to repositories, including read and write access. • With the repo scope, Jenkins can clone, fetch, and interact with the repository and perform actions such as creating branches, pushing changes, and managing pull requests. • It enables Jenkins to retrieve code, trigger builds, and perform CI/CD operations effectively.

Table 4.1 – GitHub personal access token scopes and permissions

6. Lastly, scroll down and click on the **Generate token** button.

After the token is generated, it will be displayed on the screen. Make sure to copy the token, as it will not be shown again. Using this token, we will now create two credentials inside Jenkins: a secret text credential and a username and password credential.

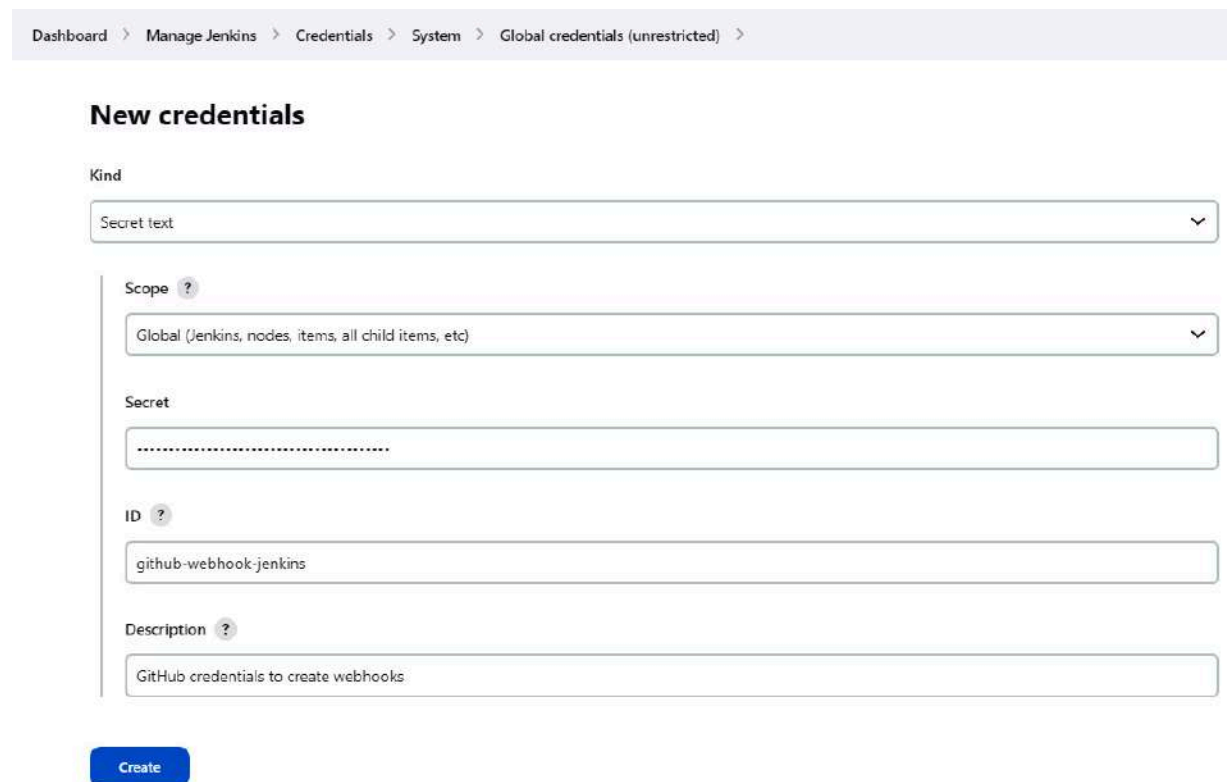
Please note that the personal access token provides access to your GitHub account, so it should be treated as sensitive information. Keep the token secure and avoid sharing it publicly or committing it to version control systems.

Creating a secret text credential inside Jenkins

By creating a secret text credential for the GitHub personal access token, you can safely and securely store and manage the token within Jenkins, ensuring that it is readily available for the GitHub plugin configuration that requires authentication when interacting with GitHub repositories and services.

Take the following steps to create a secret text credential inside Jenkins:

1. To create one, from the Jenkins dashboard, navigate to **Manage Jenkins | Credentials | System | Global credentials (unrestricted)** and click + **Add Credentials**.
2. Configure the credential settings as follows:
 - I. Using the **Kind** field, choose **Secret text** from the drop-down list.
 - II. In the **Secret** field, enter the personal access token that we copied earlier.
 - III. Fill in **ID** and **Description** to help identify and describe the credential.



The screenshot shows the Jenkins 'New credentials' page. At the top is a breadcrumb trail: Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >. Below this is the title 'New credentials'. The form has several fields: 'Kind' is a dropdown menu with 'Secret text' selected; 'Scope' is a dropdown menu with 'Global (Jenkins, nodes, items, all child items, etc)' selected; 'Secret' is a text input field with a masked value '.....'; 'ID' is a text input field with the value 'github-webhook-jenkins'; and 'Description' is a text input field with the value 'GitHub credentials to create webhooks'. At the bottom left of the form is a blue 'Create' button.

Figure 4.2 – Creating a secret text credential in Jenkins

3. Lastly, click on the **Create** button to save the secret text credential. Once saved, you can use the credential inside Jenkins by referencing its **ID**.

Creating a username and password credential inside Jenkins

You may wonder why there is a need to create a username and password credential inside Jenkins. Although the GitHub personal access token stored as a secret text credential is sufficient for the GitHub plugin to connect to a GitHub account and manage webhooks, it is not adequate for repository cloning. For Jenkins pipeline jobs and multi-branch pipeline jobs, a GitHub personal access token saved as a username and password credential will do. An alternative approach to this involves generating SSH keys, adding the public key to GitHub, and then creating an **SSH Username with private key** credential in Jenkins using the private key. Additionally, this method necessitates specifying the GitHub repository URL in the Jenkins pipeline job's configuration, formatted as `git@github.com:username/repository.git`. However, for simplicity and convenience in this book, we will primarily focus on using a GitHub personal access token stored as a username and password credential inside Jenkins.

Take the following steps to create a username and password credential inside Jenkins:

1. From the Jenkins dashboard, navigate to **Manage Jenkins | Credentials | System | Global credentials (unrestricted)** and click + **Add Credentials**.
2. Configure the credential details as follows:
 - I. Using the **Kind** field, choose **Username with password** from the drop-down list.
 - II. The **Username** field isn't of much significance; add any name.
 - III. In the **Password** field, enter the personal access token that we copied earlier.
 - IV. Fill in **ID** and **Description** to help identify and describe the credential.

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

New credentials

Kind

Username with password

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Username ?

nikhil

☐ Treat username as secret ?

Password ?

.....

ID ?

github-repo-jenkins

Description ?

GitHub credentials to interact with repositories

Create

Figure 4.3 – Creating a username and password credential in Jenkins

3. Lastly, click on the **Create** button to save the credential. Once saved, you can use this credential inside Jenkins by referencing its **ID**.

Configuring the GitHub plugin

Finally, we can now configure the GitHub plugin settings inside Jenkins. Take the following steps:

1. From the Jenkins dashboard, click on **Manage Jenkins** in the left-hand side menu and click on **Configure System**.
2. On this page, scroll down until you find the **GitHub** section.
3. Configure the **GitHub** section as follows:
 - I. Add a meaningful name to your configuration using the **Name** field.
 - II. Leave the **API URL** field to its default value of <https://api.github.com> if you intend to use **public github.com**. However, if you are utilizing GitHub Enterprise, please provide the API endpoint specific to your GitHub Enterprise instance (e.g., <https://your.organization.com/api/v3/>).
 - III. Using the **Credentials** field, choose an existing secret text credential from the drop-down list.
 - IV. Make sure to check the **Manage hooks** checkbox.

Dashboard > Manage Jenkins > System >

GitHub

GitHub Servers ?

GitHub Server ?

Name ?

Default GitHub Server

API URL ?

https://api.github.com

Credentials ?

GitHub credentials to create webhooks

Add

Credentials verified for user nikhilpathania, rate limit: 4998

Test connection

☒ Manage hooks

Advanced

Figure 4.4 – Configuring the GitHub plugin settings inside Jenkins

4. After saving the configuration, you can click on the **Test Connection** button to verify that Jenkins can successfully connect to GitHub using the provided credentials.
5. Finally, scroll to the bottom of the page and click on **Apply** and then **Save** to save the configuration changes.

Once the GitHub plugin is configured, you can now use features such as configuring webhooks, triggering builds on specific GitHub events, integrating with pull requests, and more. You can configure these settings within your Jenkins pipeline configurations based on your project requirements. In the upcoming chapters, we will delve into this further, exploring the details in depth. Additionally, to learn more about GitHub, visit the GitHub documentation (see [1] in the *Further reading* section at the end of the chapter).

Connecting Jenkins to SonarQube

This section describes setting up a SonarQube server using a Helm chart. We will deploy it on the same AKS cluster where our Jenkins server runs but in a different namespace. Our SonarQube server will have an Nginx Ingress controller in front of it with its own public IP. To keep things simple, we will avoid going into the details of setting up a production-grade SonarQube setup with detailed user access and permissions. So, let's start.

Deploying an Nginx Ingress controller for SonarQube

An Nginx Ingress controller for SonarQube facilitates optimized routing and external access to the SonarQube application deployed on Kubernetes. By implementing this setup, teams can ensure secure and dependable access to SonarQube while harnessing the powerful load balancing and routing capabilities of Nginx. Although the configuration of the Nginx Ingress controller can be intricate, in the current section, we will emphasize a simpler setup to achieve seamless accessibility to SonarQube. Take the following steps:

1. Connect to your AKS cluster. Use the Azure CLI ('az') or Azure portal to authenticate and connect to your AKS cluster.
2. The easiest way to deploy an Nginx Ingress controller on Kubernetes is using Helm. Execute the following two commands one after the other to add the **ingress-nginx** repository as a source of Helm charts and update the local cache to have the latest information about the available charts from all the added repositories:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
```

3. Use the **helm install** command to install the **Ingress-Nginx** chart with specific configurations and options to set annotations, define the replica count, and associate the Nginx Ingress controller with a specific ingress class resource and annotation value, as shown:

```
helm install ingress-nginx-sonarqube ingress-nginx/ingress-nginx \
--create-namespace -n sonarqube \
--set controller.service.annotations."service\.beta\.kubernetes\.io/azure-load-
balancer-health-probe-request-path"/healthz \
--set controller.replicaCount=1 \
--set controller.ingressClassResource.name=sonarqube-nginx \
--set controller.ingressClassResource.controllerValue=k8s.io/sonarqube-nginx
```

Let's understand these commands step by step:

- This command uses Helm to install the **ingress-nginx** chart from the repository **ingress-nginx/ingress-nginx**. The release name is set to **ingress-nginx-sonarqube**, which identifies the specific deployment.

```
helm install ingress-nginx-sonarqube ingress-nginx/ingress-nginx
```

- The next command creates a new namespace **sonarqube** and associates the new Nginx Ingress controller to it:

```
--create-namespace -n sonarqube
```

- This flag sets an annotation for the Nginx Ingress controller's service. The annotation being set is **service.beta.kubernetes.io/azure-load-balancer-health-probe-request-path**. The value of the annotation is **/healthz**, which represents the health check path for Azure Load Balancer:

```
--set controller.service.annotations."service\.beta\.kubernetes\.io/azure-
load-balancer-health-probe-request-path"/healthz
```

- This flag sets the number of replicas for the Nginx Ingress controller to **1**. It specifies that only one replica of the controller should be deployed:

```
--set controller.replicaCount=1
```

- This flag sets the name of the ingress class resource associated with the Nginx Ingress controller. The name is set to **sonarqube-nginx**, which identifies the specific Ingress class resource:

```
--set controller.ingressClassResource.name=sonarqube-nginx
```


- This flag sets the value of the `k8s.io/sonarqube-nginx` annotation in the Ingress class resource. The annotation is used to specify additional information or behavior associated with the Nginx Ingress controller:

```
--set controller.ingressClassResource.controllerValue=k8s.io/sonarqube-nginx
```

4. From the Azure dashboard, look for the public IP associated with the Nginx Ingress controller, as shown:

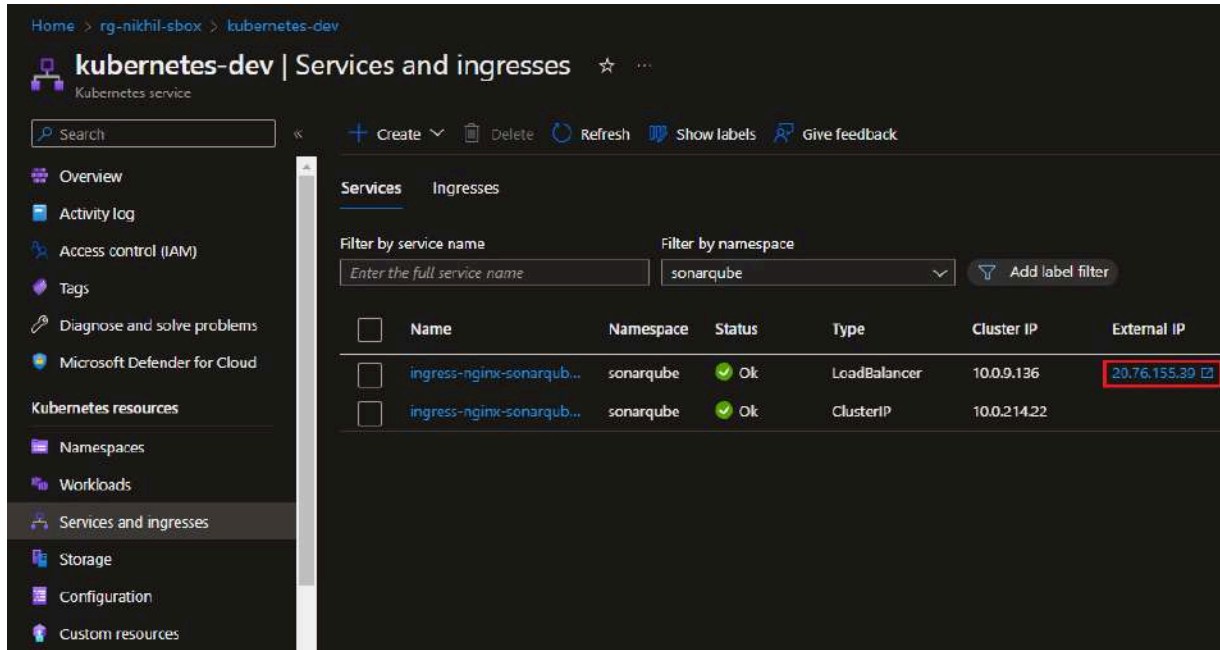


Figure 4.5 – Nginx Ingress controller services for SonarQube

To learn more about the Nginx Ingress controller Helm chart, visit its respective GitHub repository [2].

Deploying a SonarQube server using a Helm chart

Here's a step-by-step guide to deploying a SonarQube server on **Azure Kubernetes Service (AKS)** using a Helm chart:

1. Run the following command to download and update the SonarQube Helm repository:

```
helm repo add sonarqube https://SonarSource.github.io/helm-chart-sonarqube
helm repo update
```

2. You can modify the default configuration values of the SonarQube chart as per your requirements. To see the available configuration options, run the upcoming command. The available options will be printed to a YAML file. In the current chapter, however, we will stick to the default settings.

```
helm show values sonarqube/sonarqube > sonarqube-values.yaml
```

3. Next, to deploy SonarQube, run the following command:

```
helm upgrade --install -n sonarqube --version ~8 sonarqube sonarqube/sonarqube
```

Let's take a moment to understand some of the options in the previous command:

- **-n sonarqube**: This flag specifies the namespace **sonarqube** where the SonarQube server will be installed.
- **--version ~8**: This flag specifies the version of the SonarQube chart to be installed. In this case, **~8** indicates a version that starts with **8**. At the time of writing this book, version **8.x.x** installs SonarQube 9.9 LTS.
- **sonarqube**: This is the name of the deployment. It can be customized to your preference.
- **sonarqube/sonarqube**: This is the chart repository and chart name. It tells Helm to use the SonarQube chart from the **sonarqube** repository.

4. Monitor the deployment and check that the SonarQube pods are **Running** using the Azure dashboard. It usually takes a while for the SonarQube pods and stateful sets to come up.

To learn more about the SonarQube Helm chart, visit its GitHub repository [3].

Deploying an Ingress file for SonarQube

To access SonarQube, we need to expose it using an Ingress resource. The purpose of an Ingress resource in Kubernetes is to define rules for routing external HTTP and HTTPS traffic to services within the cluster. It acts as a configuration file that specifies how incoming requests should be directed to the appropriate backend services based on the requested host, path, or other criteria. Take the following steps to deploy an Ingress file to expose our SonarQube service:

1. First, create a file called **sonarqube-ingress.yaml** with the following content:

sonarqube-ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: sonarqube-ingress
  annotations:
spec:
  ingressClassName: sonarqube-nginx
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: sonarqube-sonarqube
            port:
              number: 9000
```

This code is an Ingress resource definition for SonarQube. Let's see some of its key elements:

- **ingressClassName: sonarqube-nginx** specifies the class name of the Ingress controller associated with this Ingress resource as **sonarqube-nginx**.
- **name: sonarqube-sonarqube** sets the name of the backend service to **sonarqube-sonarqube**.

- **number:** 9000 specifies the port configuration for the backend service by setting the port number to 9000.

In summary, this Ingress resource definition sets up routing rules for SonarQube, such that incoming traffic to the root path (/) will be forwarded to the backend service named `sonarqube-sonarqube` running on port 9000. The annotations and `ingressClassName` provide additional customization and association with the specific Ingress controller for SonarQube.

2. Next, save the file and run the `kubectl apply` command to apply the `sonarqube-ingress.yaml` manifest file in the `sonarqube` namespace:

```
kubectl apply -f ./sonarqube-ingress.yaml -n sonarqube
```

It takes a while for the Nginx Ingress controller's public IP to associate with the SonarQube ingress. Use the Azure dashboard to check the association:

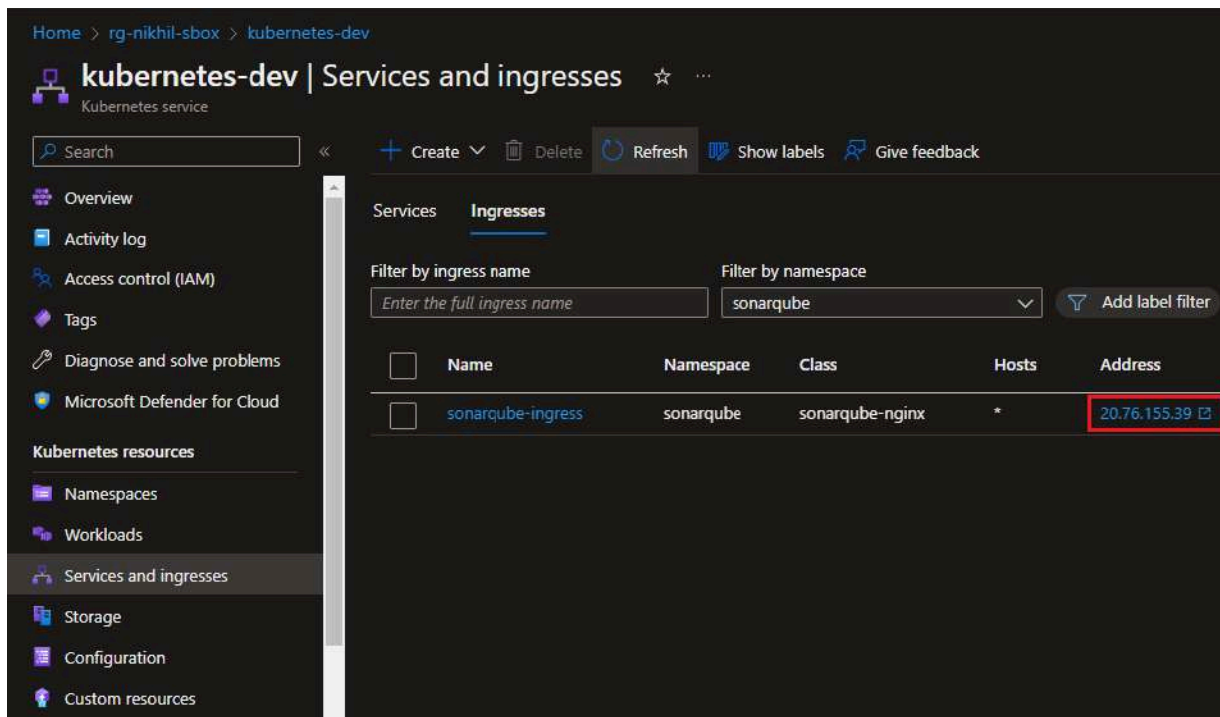


Figure 4.6 – SonarQube ingress with its public IP address

That's it! You've successfully deployed SonarQube on AKS using a Helm chart. To access SonarQube, visit the URL `http://<Public IP>/`, where `<Public IP>` is the public IP of the SonarQube Nginx Ingress controller.

The default login credentials to SonarQube are `admin/admin`. You will be asked to change them when you log in to SonarQube for the first time.

Installing the Jenkins plugin for SonarQube

The SonarQube Scanner plugin enables seamless integration between Jenkins and SonarQube, allowing Jenkins to run code analysis using the SonarQube platform and providing valuable insights and metrics about code quality, security, reliability, and maintainability.

With the SonarQube Scanner plugin, you can configure Jenkins jobs to trigger code analysis during the pipeline process. Additionally, the plugin also facilitates the analysis results to be displayed within Jenkins and contributes to the overall build pipeline.

Take the following steps to install the Jenkins plugin for SonarQube:

1. Log in to your Jenkins instance as an administrator and navigate to **Manage Jenkins | Plugins | Available plugins**.
2. On the resultant page, locate the **SonarQube Scanner** plugin by either scrolling through the list or using the search bar:

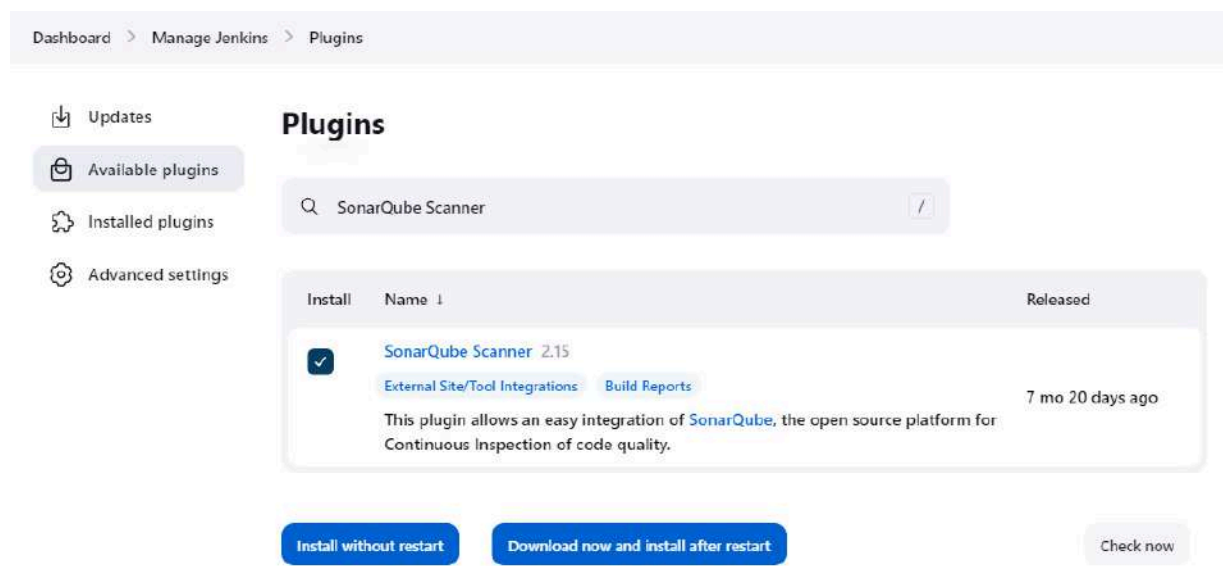


Figure 4.7 – Installing the Jenkins plugin for SonarQube

3. Check the checkbox next to the **SonarQube Scanner** plugin to select it for installation and click on the **Install without restart** button to begin the installation process. Wait for Jenkins to download and install the SonarQube plugin. Once the installation is complete, you will see a notification at the top of the page indicating the success of the installation.
4. If you select the **Install without restart** option, you may be prompted to restart Jenkins. Follow the instructions to restart Jenkins for the plugin to take effect.

Creating a global analysis token on SonarQube

To create a global analysis token for the admin user on SonarQube, you can follow these steps:

1. Log in to SonarQube using the credentials of the admin user.
2. Click on the default admin user profile icon at the top-right corner of the SonarQube interface. Select **My Account** from the drop-down menu.
3. In the **User Account** settings, click on the **Security** tab and configure the token properties as follows:

- I. Provide a name for the token in the **Name** field.
- II. Select **Global Analysis Token** using the **Type** field. This ensures the token has sufficient privileges for code analysis. Additionally, we would like our token to only have privileges to run analysis and nothing else.
- III. Set the **Expires in** field appropriately.

Tokens

If you want to enforce security by not providing credentials of a real SonarQube user to run your code scan or to invoke web services, you can provide a User Token as a replacement of the user login. This will increase the security of your installation by not letting your analysis user's password going through your network.

Generate Tokens

Name **Type** **Expires in**

sonarqube-analysis-jenkins Global Analysis Token 90 days **Generate**

Name	Type	Project	Last use	Created	Expiration
No tokens					

Figure 4.8 – Creating a global analysis token for the admin user on SonarQube

4. Click on the **Generate** button to create the token. Once the token is generated, a token value will be displayed on the screen. Copy the token value and save it in a secure location. Note that this value will not be visible again in the SonarQube interface.

With the global analysis token created for the admin user, you can now use this token in your CI/CD pipelines or for integration with Jenkins to authenticate SonarQube.

Creating a secret text credential inside Jenkins

By creating a secret text credential for the SonarQube token, you can safely and securely store and manage the token within Jenkins, ensuring that it is readily available for the SonarQube plugin configuration that requires authentication when interacting with the SonarQube server. Take the following steps to create a secret text credential inside Jenkins:

1. From the Jenkins dashboard, navigate to **Manage Jenkins** | **Credentials** | **System** | **Global credentials (unrestricted)** and click + **Add Credentials**.
2. Configure the credential details as follows:
 - I. Using the **Kind** field, choose **Secret text** from the drop-down list.
 - II. In the **Secret** field, enter the global analysis token that we copied earlier.
 - III. Fill in **ID** and **Description** to help identify and describe the credential.

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

New credentials

Kind

Secret text

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Secret

.....

ID ?

sonarqube-analysis-jenkins

Description ?

SonarQube token to run Static Code Analysis

Create

Figure 4.9 – Creating a secret text credential inside Jenkins

3. Lastly, click on the **Create** button to save the credential. Once saved, you can use the secret text credential inside Jenkins by referencing its **ID**.

Configuring the SonarQube plugin

To configure your SonarQube server in Jenkins, follow these steps:

1. From the Jenkins dashboard, click on **Manage Jenkins** in the left-hand side menu and then click on **System**.
2. On the resultant page, scroll down to the **SonarQube servers** section and click on **Add SonarQube**.
3. Configure the SonarQube installations section as follows:
 - Give a meaningful name to the current SonarQube server configuration using the **Name** field.
 - Add the SonarQube server URL using the **Server URL** field. Make sure there is no trailing **/** at the end of the SonarQube Server URL.
 - Use the secret text credential created earlier as the value for the **Server authentication token** field.

Dashboard > Manage Jenkins > System >

SonarQube servers

If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build.

☒ **Environment variables**
Enable injection of SonarQube server configuration as build environment variables

SonarQube installations

List of SonarQube installations

Name

Default SonarQube Server

Server URL
Default is http://localhost:9000

http://20.76.155.39

Server authentication token
SonarQube authentication token. Mandatory when anonymous access is disabled.

SonarQube token to run Static Code Analysis

Add

Advanced

Figure 4.10 – Configuring SonarQube settings inside Jenkins

4. Finally, click on **Apply** and then **Save** to save the configuration.

By following the provided steps to configure the SonarQube server in Jenkins, you establish the essential connection and settings required for the seamless integration of SonarQube within your Jenkins environment. This integration empowers your Jenkins pipelines with the capabilities to perform code analysis and generate comprehensive quality reports. As we progress through the upcoming chapters and delve into creating a CI pipeline, we will further explore the practical aspects of leveraging these SonarQube-enabled pipelines to enhance code quality and ensure efficient software development practices.

Connecting Jenkins to Artifactory

This section describes setting up an Artifactory server using a Helm chart. We will deploy it on the same AKS cluster where our Jenkins and SonarQube server runs but in a different namespace. Our Artifactory server will have an Nginx Ingress controller in front of it with its own public IP. To keep things simple, we will avoid going into the details of setting up a production-grade Artifactory setup with detailed user access and permissions. So, let's start.

Deploying an Nginx Ingress controller for Artifactory

Just like for SonarQube, we will create an Nginx Ingress controller for Artifactory too. Take the following steps:

1. Connect to your AKS cluster. Use the Azure CLI (`az`) or Azure portal to authenticate and connect to your AKS cluster.
2. Execute the following two commands one after the other to add the **ingress-nginx** repository as a source of Helm charts and update the local cache to have the latest information on the available charts from all the added repositories:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
```

3. Use the **helm install** command to install the Ingress-Nginx chart with specific configurations and options to set annotations, define the replica count, and associate the Nginx Ingress controller with a specific Ingress class resource and annotation value:

```
helm install ingress-nginx-artifactory ingress-nginx/ingress-nginx \
--create-namespace -n artifactory \
--set controller.service.annotations."service\.beta\.kubernetes\.io/azure-load-
balancer-health-probe-request-path"=/healthz \
--set controller.replicaCount=1 \
--set controller.ingressClassResource.name=artifactory-nginx \
--set controller.ingressClassResource.controllerValue=k8s.io/artifactory-nginx
```

4. From the Azure dashboard, look for the public IP associated with the Nginx Ingress controller:

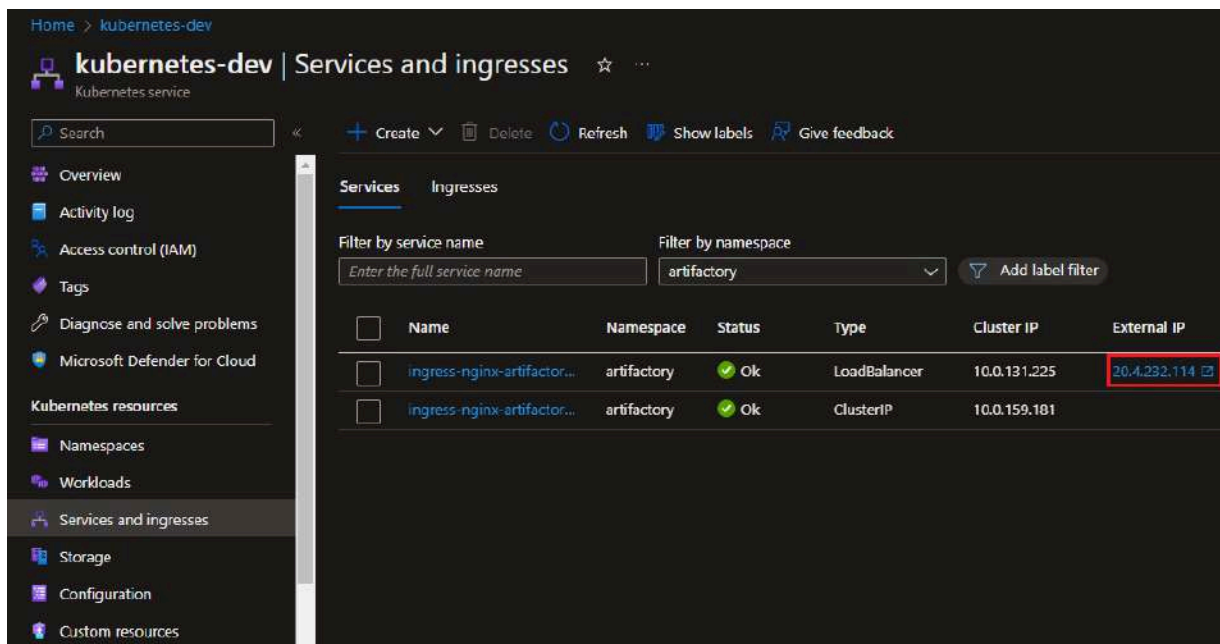


Figure 4.11 – Nginx Ingress controller services for Artifactory

Deploying an Artifactory server using a Helm chart

Here's a step-by-step guide to deploying an Artifactory server on AKS using a Helm chart:

1. Run the following command to download and update the Artifactory Helm repository:

```
helm repo add jfrog https://charts.jfrog.io
helm repo update
```

2. You can modify the default configuration values of the Artifactory chart as per your requirements. To see the available configuration options, run the following command. The available options will be printed to a YAML file. In the current chapter, however, we will stick to the default settings.

```
helm show values jfrog/artifactory > artifactory-values.yaml
```

3. Next, to deploy Artifactory, run the following command. Note that we are installing the open source version of Artifactory:

```
helm upgrade --install artifactory \
--set artifactory.nginx.enabled=false \
--set artifactory.ingress.enabled=false \
-n artifactory \
jfrog/artifactory
```

Let's take a moment to understand some of the options in this command:

- **artifactory**: This is the name given to the Helm release. You can customize it as per your preferences.
- **artifactory.nginx.enabled=false**: This **--set** flag disables the deployment of the Nginx component within Artifactory.
- **artifactory.ingress.enabled=false**: This **--set** flag disables the creation of an Ingress resource for accessing Artifactory via HTTP/HTTPS.
- **-n artifactory**: This flag specifies the namespace **artifactory** where the Artifactory server will be installed.
- **jfrog/artifactory**: This is the chart repository and chart name. It tells Helm to use the Artifactory Pro chart from the JFrog chart repository, which is a licensed version of JFrog Artifactory. To get a one-month free trial for Artifactory Pro, visit <https://jfrog.com/start-free/>. You need to complete a form with your contact information, including your name, email address, company name, and other relevant details. Once you submit the form, check your email inbox for a confirmation email from JFrog. This email will usually contain further instructions on how to activate your free trial license. Make sure to check your spam or junk folder if you do not see the email in your inbox.

4. Monitor the deployment and check that the Artifactory pods are **Running** using the Azure dashboard. It usually takes a while for the Artifactory pods and stateful sets to come up.

To learn more about the Artifactory Helm chart, visit its GitHub repository [4].

Deploying an Ingress file for Artifactory

Just like SonarQube, to access Artifactory, we need to expose it using an Ingress resource. Take the following steps to deploy an Ingress to expose our Artifactory service:

1. First, create a file called **artifactory-ingress.yaml** with the following content:

artifactory-ingress.yaml

```
apiVersion: networking.k8s.io/v1
```

```

kind: Ingress
metadata:
  name: artifactory-ingress
  annotations:
    nginx.ingress.kubernetes.io/proxy-body-size: 2g
spec:
  ingressClassName: artifactory-nginx
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: artifactory-oss
            port:
              number: 8082

```

This code is an Ingress resource definition for Artifactory. Let's see some of its key elements:

- The **nginx.ingress.kubernetes.io/proxy-body-size: 2g** annotation is used to ensure that the NGINX Ingress in the Kubernetes cluster will allow client requests with a body size of up to 2 GB, helping to prevent clients from sending excessively large request bodies that could overwhelm the server. If a client sends a request with a body size larger than 2 GB, NGINX will reject the request with a **413 Request Entity Too Large** error.
- **ingressClassName: artifactory-nginx** specifies the class name of the Ingress controller associated with this Ingress resource as **artifactory-nginx**.
- **name: artifactory-oss** sets the name of the backend service to **artifactory-oss**.
- **number: 8082** specifies the port configuration for the backend service by setting the port number to **8082**.

In summary, this Ingress resource definition sets up routing rules for Artifactory, such that incoming traffic to the root path (/) will be forwarded to the backend service named **artifactory-oss** running on port **8082**. The annotations and **ingressClassName** provide additional customization and association with the specific Ingress controller for Artifactory.

2. Next, save the file and run the **kubectl apply** command to apply the manifest file **artifactory-ingress.yaml** in the **artifactory** namespace:

```
kubectl apply -f ./artifactory-ingress.yaml -n artifactory
```

It takes a while for the Nginx Ingress controller's public IP to associate with the Artifactory ingress. Use the Azure dashboard to check the association:

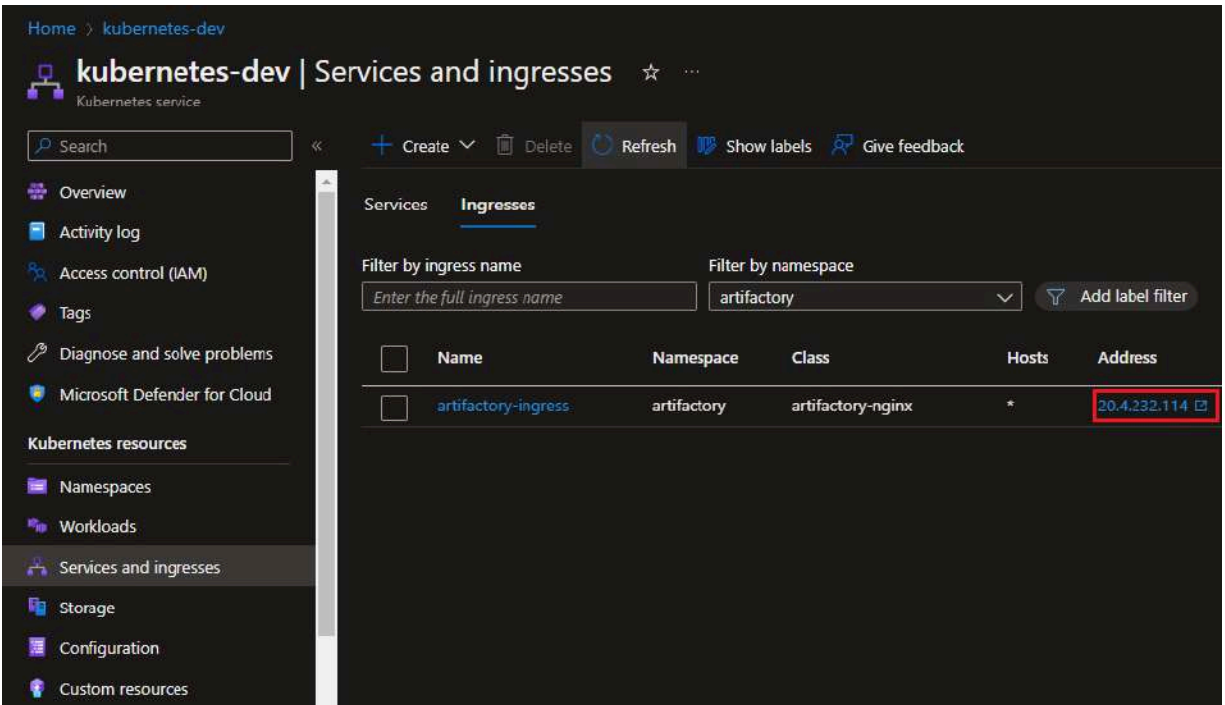


Figure 4.12 – Artifactory ingress with its public IP address

That's it! You've successfully deployed Artifactory on AKS using a Helm chart. To access Artifactory, visit the URL `http://<Public IP>/`, where `<Public IP>` is the public IP of the Artifactory Nginx Ingress controller.

The default login credentials to Artifactory are `admin/password`. You will be asked to change them when you log in to Artifactory for the first time. Immediately after logging in, configure the trial license for Artifactory.

Installing the Jenkins plugin for Artifactory

The Jenkins plugin for Artifactory provides powerful integration between Jenkins and Artifactory. With the Artifactory plugin, Jenkins jobs can seamlessly interact with Artifactory repositories, ensuring the reliable and efficient handling of artifacts. It facilitates artifact management, allowing you to resolve dependencies, publish build artifacts, and deploy artifacts to specific repositories. Take the following steps to install the Jenkins plugin for Artifactory:

1. Log in to your Jenkins instance as an administrator and navigate to **Manage Jenkins | Plugins | Available plugins**.
2. On the resultant page, locate the **Artifactory** plugin by either scrolling through the list or using the search bar:

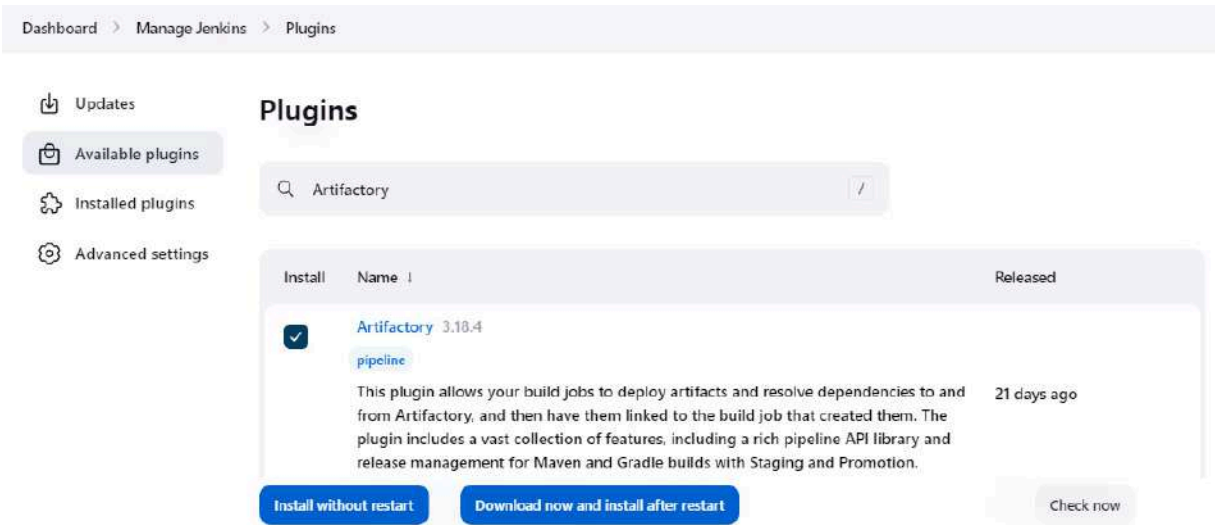


Figure 4.13 – Installing the Jenkins plugin for Artifactory.

3. Check the checkbox next to the **Artifactory** plugin to select it for installation and click on the **Install without restart** button to begin the installation process. Wait for Jenkins to download and install the Artifactory plugin. Once the installation is complete, you will see a notification at the top of the page indicating the success of the installation.
4. If you select the **Install without restart** option, you may be prompted to restart Jenkins. Follow the instructions to restart Jenkins for the plugin to take effect.

Creating a user in Artifactory

To create an internal user in Artifactory, you can follow these steps:

1. Log in to Artifactory using your Artifactory administrator credentials.
2. To create a new user, access the **User Management** section by clicking on the **Administration** tab in the top-left navigation menu. Select **Users** from the drop-down menu.
3. Create a new user by clicking on the + **New user** button.
4. In the user creation form, provide the following necessary details for the new internal user:
 - I. **Username**: Choose a unique username for the user. For example, this could be **Jenkins**.
 - II. **Password**: Set a secure password for the user.
 - III. **Email**: Provide the email address of the administrator.
 - IV. Make sure to unselect all options under the **Roles and Options** section.
 - V. By default, new users may get added to some groups. Make sure that this new user is not related to any group. Check this in the **Related Groups** section.

JFrog Platform All Search admin resources Welcome, admin

Administration

- Projects
- Environments
- Repositories
- User Management**
 - Settings
 - Users**
 - Groups
 - Global Roles
 - Permissions
 - Access Tokens
- Authentication Providers
- License Details
- General
- Proxies

Add new user

User Settings

* User Name: jenkins

* Email Address: emailid@domainname.com

Roles

- ☐ Administer Platform
- ☐ Manage Resources

Options

- ☒ Can Update Profile
- ☐ Disable UI Access
- ☐ Disable Internal Password

Password

* Password: [masked]

* Retype Password: [masked]

Figure 4.14 – Creating a user in Artifactory

5. Once done, click on the **Save** button.
6. Next, we move on to the **Permissions** section under **User Management**. On the **Permissions** page, click on the + **New Permission** button to create a new permission.
7. In the permission creation form, choose a unique name for the new permission using the **Name** field. For example, this could be **Jenkins**.
8. Next, move to the **Resources** tab and click **Add Repositories**. Provide the following necessary details:
 - I. Select **Any Local Repository**, **Any Remote Repository**, and **Any Distribution Repository**.
 - II. Leave the rest of the settings blank.
 - III. Once done, click on the **Save** button.
9. Now, move to the **Users** tab and provide the following necessary details:
 - I. Click on the + button under the **Selected Users** section.
 - II. Select the user we created in the previous step and click the **OK** button.

III. Under the **Selected Users Repositories** section, choose **Read**, **Annotate**, and **Deploy/Cache**.

10. Once done, click on the **Create** button to save the new permission.

The new internal user is now created in Artifactory. This user can be used to authenticate and access Artifactory resources based on the assigned permissions.

Creating a username and password credential inside Jenkins

By creating a username and password credential for the Artifactory user, you can safely and securely store and manage the credentials within Jenkins, ensuring that it is readily available for the Artifactory plugin configuration that requires authentication when interacting with the Artifactory server to push built artifacts to it. Take the following steps to create a username and password credential inside Jenkins:

1. From the Jenkins dashboard, navigate to **Manage Jenkins | Credentials | System | Global credentials (unrestricted)** and click **+ Add Credentials**.
2. Configure the credential details:
 - I. Using the **Kind** field, choose **Username with password** from the drop-down list.
 - II. In the **Username** field, add the name of the user we created earlier in Artifactory.
 - III. In the **Password** field, enter the corresponding password.
 - IV. Fill in **ID** and **Description** to help identify and describe the credential.

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

New credentials

Kind

Username with password

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Username ?

jenkins

☐ Treat username as secret ?

Password ?

ID ?

artifactory-publish-jenkins

Description ?

Artifactory user credentials to publish artifacts to Artifactory

Create

Figure 4.15 – Creating a username and password credential inside Jenkins

3. Lastly, click on the **Create** button to save the credential. Once saved, you can use this credential inside Jenkins by referencing its **ID**.

Configuring the Artifactory plugin

Here are the detailed steps to configure the Artifactory plugin in Jenkins:

1. From the Jenkins dashboard, click on **Manage Jenkins** in the left-hand side menu and then click on **System**.
2. On the resultant page, scroll down to the **JFrog** section and select **Use the Credentials Plugin**. This will allow us to use the username and password credential created earlier to authenticate to Artifactory.
3. Next, click on **Add JFrog Platform Instance** and fill in the following details for the JFrog platform instances (Artifactory server):
 - I. **Instance ID**: Provide a unique ID for the Artifactory server configuration.
 - II. **JFrog Platform URL**: Enter the URL of your Artifactory instance.
 - III. Under the **Default Deployer Credentials** section, choose the username and password credential we created earlier using the **Credentials** field.

IV. Test the connection by clicking on the **Test Connection** button to verify the connection to Artifactory.

Dashboard > Manage Jenkins > System >

JFrog

☒ Use the Credentials Plugin ?

JFrog Platform Instances

List of JFrog platform instances that projects will want to deploy artifacts and build info

JFrog instance details

Instance ID ?

Default Artifactory Server

JFrog Platform URL ?

http://20.4.232.114/

Advanced Configuration ▾

Default Deployer Credentials

Credentials

jenkins/***** (Artifactory user credentials to publish artifacts to Artifactory) ▾

Add ▾

Found JFrog Artifactory 7.59.12 at http://20.4.232.114/artifactory
JFrog Distribution not found at http://20.4.232.114/distribution

☐ Use Different Resolver Credentials

Test Connection

Figure 4.16 – Configuring the Artifactory plugin inside Jenkins

4. Finally, click on **Apply** and then **Save** to save the configuration.

Now, your Jenkins instance is configured to integrate with the Artifactory server using the Artifactory plugin. You can use this configuration to deploy artifacts, resolve dependencies, and perform other Artifactory-related operations within your Jenkins jobs.

Summary

In this chapter, we commenced our journey with a detailed exploration into setting up SonarQube and Artifactory using a Helm chart paired with an Nginx Ingress controller and a public IP for each service. Following this, the necessary configurations within Jenkins for seamless integration with SonarQube, Artifactory, and GitHub were diligently explored.

In the forthcoming chapter, we will delve deeply into the crucial topic of horizontally scaling Jenkins. A comprehensive guide will cover the two most popular methods, employing both the pre-configured Kubernetes plugin and the Azure VM Agent plugin. This exploration will enable the effective scaling of Jenkins by creating on-demand build agents, seamlessly enhancing our CI setup and paving the way for further advancements in Jenkins pipeline utilization.

Questions

1. When integrating Jenkins with a GitHub repository, which of the following steps must be performed last?
 - A. Performing necessary GitHub configurations inside Jenkins global settings
 - B. Creating a personal access token on GitHub
 - C. Creating a secret text credential inside Jenkins
 - D. Installing the Jenkins plugin for GitHub
2. Choose whether the following statement true or false: *The GitHub plugin for Jenkins supports triggering builds when a commit or merge happens but does not support GitHub pull requests. However, it cannot report build statuses and test results back to GitHub.*
3. When integrating Jenkins with SonarQube, which of the following steps must be performed last?
 - A. Installing the Jenkins plugin for SonarQube
 - B. Performing necessary SonarQube configurations inside Jenkins global settings
 - C. Creating a personal access token on SonarQube
 - D. Creating a secret text credential inside Jenkins
4. When integrating Jenkins with Artifactory, which of the following steps must be performed last?
 - A. Creating a service user on Artifactory
 - B. Installing the Jenkins plugin for Artifactory
 - C. Performing necessary Artifactory configurations inside Jenkins' global settings
 - D. Creating a username and password credential inside Jenkins
5. Choose whether this statement is true or false: *To manage webhooks in a GitHub repository when integrating with Jenkins, your user access token must have the `admin:repo_hook` scope.*

Answers

1. Correct answer: **A**

Explanation:

The Jenkins plugin for GitHub must be installed first to enable the integration. Then, a personal access token is created on GitHub, which is used for authenticating Jenkins with GitHub. This token is added to Jenkins as a secret text credential. Only after these steps are completed can the final configuration in Jenkins' global settings be performed, establishing the full integration.

2. Correct answer: **False**

Explanation:

The GitHub plugin for Jenkins does indeed provide seamless integration with GitHub pull requests. It allows Jenkins to automatically build and validate code changes proposed in pull requests.

3. Correct answer: **B**

Explanation:

The Jenkins plugin for SonarQube must be installed first to enable the integration. Then, a personal access token is created on SonarQube, which is used for authenticating Jenkins with SonarQube. This token is added to Jenkins as a secret text credential. Only after these steps are completed can the final configuration in Jenkins global settings be performed, establishing the full integration.

4. Correct answer: **C**

Explanation:

The Jenkins plugin for Artifactory must be installed first to enable the integration. Then, a user whose credentials are used for authenticating Jenkins with Artifactory is created on Artifactory. These credentials are added to Jenkins as a username and password credential. Only after these steps are completed can the final configuration in Jenkins global settings be performed, establishing the full integration.

5. Correct answer: **True**

Explanation:

When integrating Jenkins with GitHub, specifically for automatically managing webhooks on GitHub, the user access token used needs to have the appropriate scope. The `admin:repo_hook` scope allows Jenkins to create, view, and manage hooks in public and private repositories on GitHub.

Further reading

If you wish to know more about some of the concepts that aren't covered in depth in this chapter or are out of the scope of this book but relevant, we are happy to provide you with some references:

- [1] GitHub documentation: <https://docs.github.com/en>
- [2] Nginx Ingress controller Helm chart documentation: <https://github.com/kubernetes/ingress-nginx/tree/main/charts/ingress-nginx>
- [3] SonarQube Helm chart documentation: <https://github.com/SonarSource/helm-chart-sonarqube/tree/master/charts/sonarqube>
- [4] Artifactory Helm chart documentation: <https://github.com/jfrog/charts/tree/master/stable/artifactory-oss>

Scaling Jenkins

In the ever-evolving realm of **continuous integration and continuous deployment (CI/CD)**, ensuring scalability and optimal resource utilization remains paramount. This chapter revisits the Jenkins distributed build architecture, shedding light on how to best configure it for distributed workloads and scalability. We'll delve into the intricacies of connecting Jenkins with a Kubernetes cluster, tapping into its dynamic provisioning capabilities to scale build agents as demand fluctuates. Additionally, we'll explore the synergy between Jenkins and Azure, harnessing Azure's prowess to dynamically provision build agents and fine-tune resource use within our CI/CD pipelines. Let us learn the secrets of scaling Jenkins seamlessly and efficiently, setting the stage for a robust and resilient CI/CD infrastructure.

In the following chapter, we will revisit the Jenkins distributed build architecture, understanding the optimal configuration for scaling and distributing workloads across multiple instances. We will then learn how to connect Jenkins with a Kubernetes cluster, leveraging the dynamic provisioning capabilities to efficiently scale build agents based on demand. We will also explore the seamless integration of Jenkins with Azure, leveraging the platform's capabilities to dynamically provision build agents and optimize resource utilization within the CI/CD pipeline.

As such, we will cover the following main topics in the chapter:

- Understanding Jenkins' distributed build architecture
- Creating on-demand build agents on Kubernetes
- Creating on-demand build agents on Azure

Technical requirements

To proceed with this chapter, there are no intricate technical prerequisites. However, please ensure that your Jenkins server is operational and you have access to an Azure account.

Understanding Jenkins' distributed build architecture

In the realm of distributed computing and CI, certain architectural models have emerged as a critical driver of efficiency and scalability. This small section delves into two pivotal aspects of such a model. Firstly, we explore the foundational **master-agent model**, a design paradigm that centralizes task distribution while leveraging multiple agents to perform concurrent operations. This model, by

virtue of its design, enhances scalability and flexibility. Following this, we transition to examine two of the most effective methods to construct a **distributed build farm**. This farm serves as the practical embodiment of the master–agent concept, allowing for parallel task execution across different environments.

An overview of the master–agent model

The master–agent (sometimes called master–slave or master–worker) model is a fundamental design paradigm in distributed computing systems, and it is prevalent in various applications, Jenkins being one of them. In the master–agent model, there are typically two types of entities: the **master node** and the **agent node**. To visualize this, imagine a Jenkins master node at the center connected to multiple agent nodes radiating outward like spokes on a wheel. Each agent node would have its own OS and environment specifics labeled.

The primary Jenkins server, commonly referred to as the **master**, is responsible for scheduling build jobs, dispatching builds to the agents for actual job execution, and monitoring the agents. It also retains the configuration details, build logs, and artifacts to ensure traceability. Here is what it does:

- **Task distribution:** The master delegates tasks to agents based on certain criteria such as load, capabilities, or priority.
- **Monitoring:** It keeps an eye on the performance and health of the agents.
- **Resource management:** It ensures that tasks are allocated in an optimized manner, so no single agent is overburdened while others are idle.
- **Aggregation:** After agents complete their tasks, the master might be responsible for aggregating results or data.
- **Communication:** It acts as the primary communication hub. Agents often report back to the master, and in many cases, inter-agent communication may also be routed through the master.

Agents, on the other hand, are essentially machines, VMs, or containers that are set up to offload build projects from the master. An agent can be situated on different operating systems or platforms, which is particularly helpful when projects need to be built or tests need to be run on different environments. These are the entities that do the actual work, and they do the following:

- **Execute tasks:** They carry out the jobs or tasks assigned by the master
- **Report back:** Once a task is completed or in case of any issues, agents communicate the status/results back to the master
- **Wait for instructions:** Agents often remain in standby mode until they receive their next set of instructions from the master

Some of the benefits of the master–agent model are:

- **Scalability:** You can easily add more agents to the system to handle a larger workload.
- **Flexibility:** Agents can be specialized. For instance, some might be optimized for certain tasks, while others might be equipped for different types of jobs.

- **Fault tolerance:** If one agent fails, the task can be rerouted to another agent. The master can monitor the health of agents and ensure that work is evenly distributed.

In earlier times, the intricacies of the master–agent paradigm were viewed as pioneering, often requiring detailed clarification. Today, it has gracefully transitioned into a commonly understood and practiced methodology within the development community. The evolution of best practices means we no longer find ourselves in the routine of pushing developers away from building directly on the master.

Having said that, in essence, the master–agent model optimizes resource utilization, facilitates parallelism, and allows for flexibility in the system’s operational environment. It’s a key reason why Jenkins is so powerful and versatile in handling CI/CD pipelines across different scenarios.

Methods for building a distributed farm

There are several ways available using both cloud and on-premises systems that may be used to horizontally scale your Jenkins server for CI and CD. These methods use fixed and limited agents, as well as dynamic agent production based on demand. The approach you take is determined by the configuration of your infrastructure. Scaling Jenkins on the cloud with dynamically produced build agents on both **virtual machines (VMs)** and containers is more popular nowadays and it adds flexibility and scalability to CI/CD workflows. It allows teams to efficiently handle variable workloads by dynamically provisioning build agents as needed.

If you choose VMs, cloud providers such as AWS, Azure, or Google provide services such as AWS EC2, Azure VMs, and Google Compute Engine. These services enable the creation and termination of VM-based build agents on demand. When demand increases, more VMs are provisioned; when demand lowers, VMs can be scaled down or terminated, improving resource use.

Containerization is an even better method for dynamic agent provisioning. Containers offer lightweight, discrete runtime environments that can be deployed and scaled quickly. Jenkins can be connected with tools such as Docker or Kubernetes to construct and manage dynamic build agents as containers. Kubernetes, in particular, excels at orchestration, allowing Jenkins to scale up or down dynamically depending on the workload.

Both VMs and containers allow organizations to leverage the strengths of each approach. VMs provide flexibility and compatibility with existing infrastructure, while containers offer efficient resource utilization and faster deployment times. These approaches are illustrated further in the following figure:

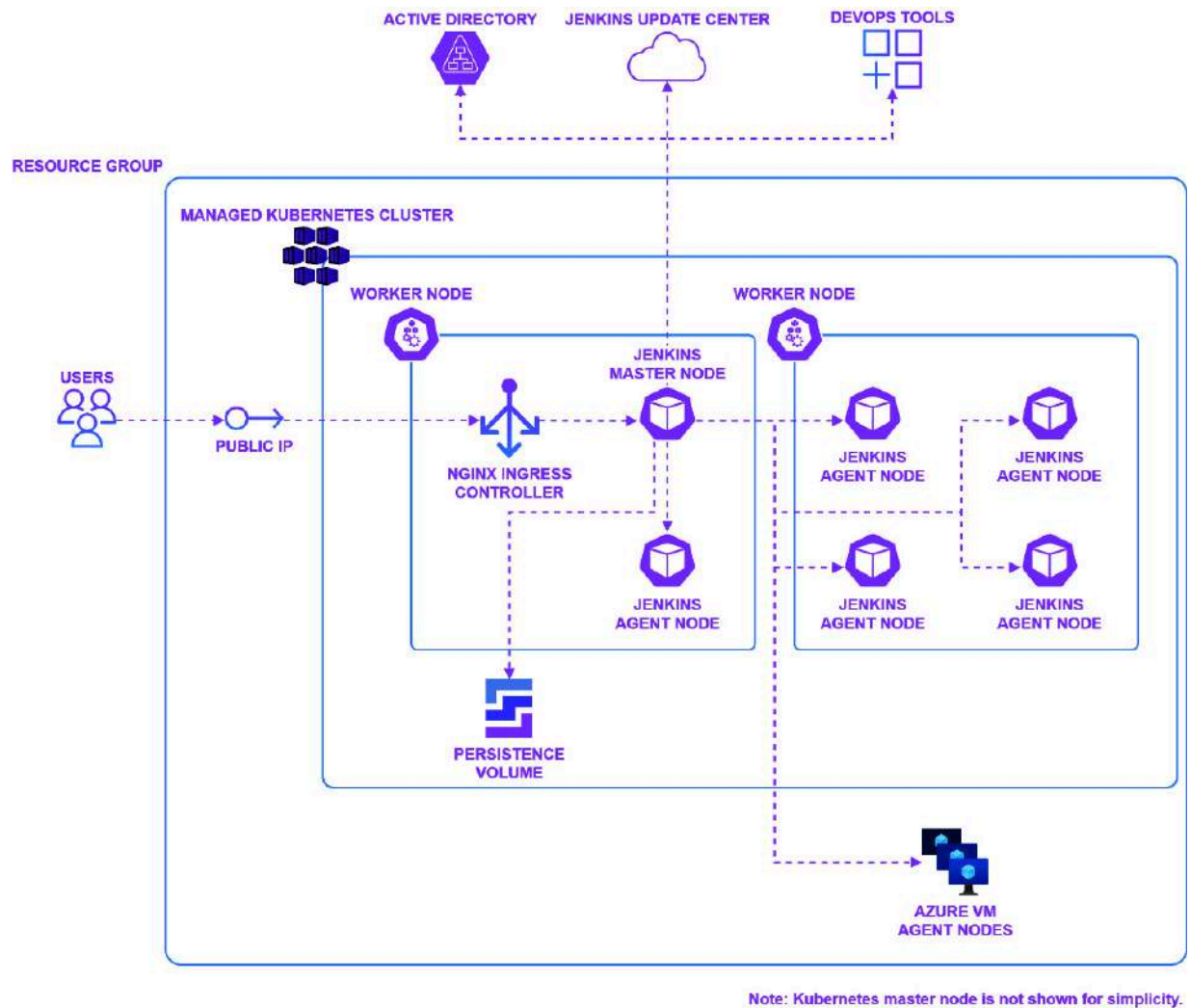


Figure 5.1 – Jenkins distributed build architecture

With that in mind, in this chapter, we'll delve into two prominent methods of scaling Jenkins: running Jenkins within Kubernetes on AKS and employing on-demand Azure VMs. Let us start with exploring what our Jenkins setup already has: the Kubernetes plugin to create on-demand build agents on AKS.

Creating on-demand build agents on Kubernetes

The Kubernetes plugin for Jenkins enables the creation of on-demand build agents on a Kubernetes cluster, providing scalability and flexibility for Jenkins-based CI/CD pipelines. By leveraging Kubernetes as the underlying orchestration platform, the plugin dynamically provisions Jenkins agent pods as containers on the cluster. These agent pods are created and scaled based on the workload, ensuring optimal resource utilization.

With the plugin's configuration options, users can define container templates, resource limits, environment variables, and other settings to tailor the build agent's runtime environment. The cherry on top is that when you create a Jenkins server using the Jenkins Helm chart, the Kubernetes plugin comes pre-configured as part of the default setup. This simplifies the setup process for users who want to utilize Jenkins in a Kubernetes environment. It eliminates the need for manual installation and configuration of the plugin, saving time and effort. Users can further customize the Kubernetes plugin settings to match their specific requirements and environments. This includes defining Kubernetes URLs, configuring pod templates, specifying resource limits, configuring container templates, and more.

This on-demand provisioning of build agents on Kubernetes empowers the team to efficiently handle varying workloads, reduce resource costs, and enable fast, parallelized builds in a containerized environment. Having said that, let's take a look at some of the important Kubernetes plugin configurations inside Jenkins.

Understanding the Kubernetes plugin configuration

The Kubernetes plugin includes several settings for configuring the Jenkins–Kubernetes integration. The present section will cover some of the most significant ones; many self-explanatory settings will be skipped to keep the length of this chapter under control.

Before we begin, from the Jenkins dashboard, navigate to **Manage Jenkins | Nodes and Clouds | Clouds**. You should see the **Kubernetes** section. When you set up Jenkins using its official Helm chart, the Kubernetes plugin is automatically installed. Furthermore, basic settings such as the Kubernetes cloud details come pre-configured, making it easier to spin up containers in the Kubernetes environment right out of the box. Let's look at some of the main settings. To make things simple, the settings have been categorized:

- **Kubernetes cloud details:** This is a set of configurations for connecting Jenkins to the Kubernetes cluster. We can categorize them rationally into two groups. Some are necessary at all times, while others are only required when the Jenkins server is not running within the Kubernetes cluster to which it seeks to connect. Let's look at the mandatory ones:
 - **Kubernetes URL:** This specifies the URL of the Kubernetes cluster API server.
 - **Kubernetes Namespace:** This defines the Kubernetes namespace where Jenkins agents will be provisioned.
 - **Jenkins URL:** This is the Kubernetes internal URL for accessing Jenkins running within a Kubernetes cluster. In a typical Kubernetes setup, the `myjenkins.jenkins.svc.cluster.local` part of the URL represents the service name (`myjenkins`) and domain name (`jenkins.svc.cluster.local`) within the Kubernetes cluster. This DNS format allows other pods or services within the cluster to communicate with the Jenkins service. The port `8080` in the URL represents the default port used by Jenkins for web access.

- **Jenkins tunnel:** This is the internal URL for the Jenkins agent running within a Kubernetes cluster. `myjenkins-agent` is the name of the Jenkins agent pod or container. `jenkins.svc.cluster.local` represents the cluster-local domain name for accessing services within the same Kubernetes namespace. `50000` is the default port used by the Jenkins controller to communicate with Jenkins agents.
- **Pod templates:**
 - **Name:** This assigns a name to the pod template.
 - **Namespace:** This defines the Kubernetes namespace where Jenkins agents will be provisioned.
 - **Labels:** This specifies labels to identify and filter the Jenkins agent pods.
 - **Usage:** This defines when Jenkins should provision new agent pods. It has two options: **Use this node as much as possible** (default) and **Only build jobs with label expressions matching this node**. In summary, the first mode allows Jenkins to freely use the node for any build, while the second mode restricts builds on the node based on label expressions, allowing you to reserve the node for specific job types and control their execution.
- **Container template:** In the Kubernetes Jenkins plugin, container templates refer to the configuration that defines the Docker containers used for running Jenkins agents. It allows you to define the necessary runtime environment and dependencies required for executing your Jenkins build jobs within Kubernetes:
 - **Docker Image:** This specifies the Docker image for the Jenkins agent container. This image typically includes the necessary build tools and dependencies required for your build jobs.
 - **Working directory:** When a build job runs on a Jenkins agent provisioned by the Kubernetes plugin, Jenkins sets up a workspace within the agent container where the build job's code and artifacts are stored. This field allows you to specify the location of this workspace or working directory inside the container.
 - **Environment Variable:** This field allows you to define environment variables that are passed to the Jenkins agent container at runtime. These variables can be used to configure and customize the behavior of your build jobs. Environment variables come in the form of a key-value pair. They can be accessed by the build scripts, build tools, or any processes executed during the build job.
 - **Advanced section:** The advanced section deals more with the compute specifications of the container such as CPU, memory, etc.
 - **Concurrency Limit:** This field sets the maximum number of agent pods that can be spun up using this template and run simultaneously in the Kubernetes cluster. This restriction ensures that the number of active agents never exceeds the global concurrency limit specified in the cloud configurations. If the limit is set to empty, zero, or a negative value, it means that there is no limit on the number of agent pods that can operate concurrently.
 - **Pod Retention:** This determines whether Jenkins should retain or terminate the Jenkins agent pods after the completion of build jobs. The options are **Always**, **Default**, **Never** (default), and **On Failure**.

These are some of the key pre-configured settings available in the Kubernetes plugin for Jenkins when installed using the Jenkins Helm chart. To learn more about the rest of the Kubernetes plugin configurations that we skipped, visit its GitHub repository (see [1] in the *Further reading* section at the end of the chapter).

Creating on-demand build agents on Azure

Creating on-demand build agents on Azure using the Azure VM Agents plugin is a highly efficient and flexible approach to running your pipelines on Windows-based agents. By leveraging the power of Azure VM, organizations can establish a robust build infrastructure that scales effortlessly, ensures optimal resource utilization, and enhances overall development productivity.

At the core of this solution is the Azure VM Agents plugin, which seamlessly integrates with Jenkins, providing a dynamic and customizable framework for creating on-demand build agents. The plugin harnesses the extensive capabilities of Azure VMs, enabling developers to spin up and tear down VM instances as per their specific requirements.

The process begins with installing and configuring the Azure VM Agents plugin and establishing the necessary connections between Azure and Jenkins. This involves setting up the Azure credentials, specifying the target Azure subscription, and defining the resource group that will house the build agents. Additionally, the plugin allows for fine-grained control over VM provisioning options, such as choosing the VM size, specifying the base image, defining the network configuration, and configuring authentication mechanisms.

Once the plugin is properly configured, developers can effortlessly create build agents on demand. Furthermore, the Azure VM Agents plugin offers advanced features to optimize the build agent provisioning process. It supports image caching, allowing for the reuse of base images across multiple build agent instances, significantly reducing provisioning time. Additionally, it provides support for managed disks, enabling persistent storage for build agents, ensuring data integrity, and eliminating the need for manual configuration.

To ensure optimal resource utilization, the Azure VM Agents plugin includes features such as idle agent shutdown and scale-down rules. These rules define the criteria for shutting down or scaling down the build agent instances based on factors such as inactivity periods, load patterns, and predefined schedules. By automatically managing the lifecycle of build agents, organizations can optimize resource allocation, reduce costs, and maintain an efficient build infrastructure.

Furthermore, the plugin provides comprehensive monitoring and logging capabilities, allowing developers and administrators to gain insights into the performance and health of the build agents. It offers real-time metrics on resource utilization, build queue status, and agent availability, empowering organizations to proactively manage and troubleshoot any issues that may arise during the build process.

Installing the Jenkins Azure VM Agents plugin

Let's begin with installing the Azure VM Agents plugin. Take the following steps:

1. As usual, log in to your Jenkins instance as an administrator and navigate to **Manage Jenkins | Plugins | Available plugins**.
2. On the resultant page, locate the **Azure VM Agents** plugin by either scrolling through the list or using the search bar, as shown:

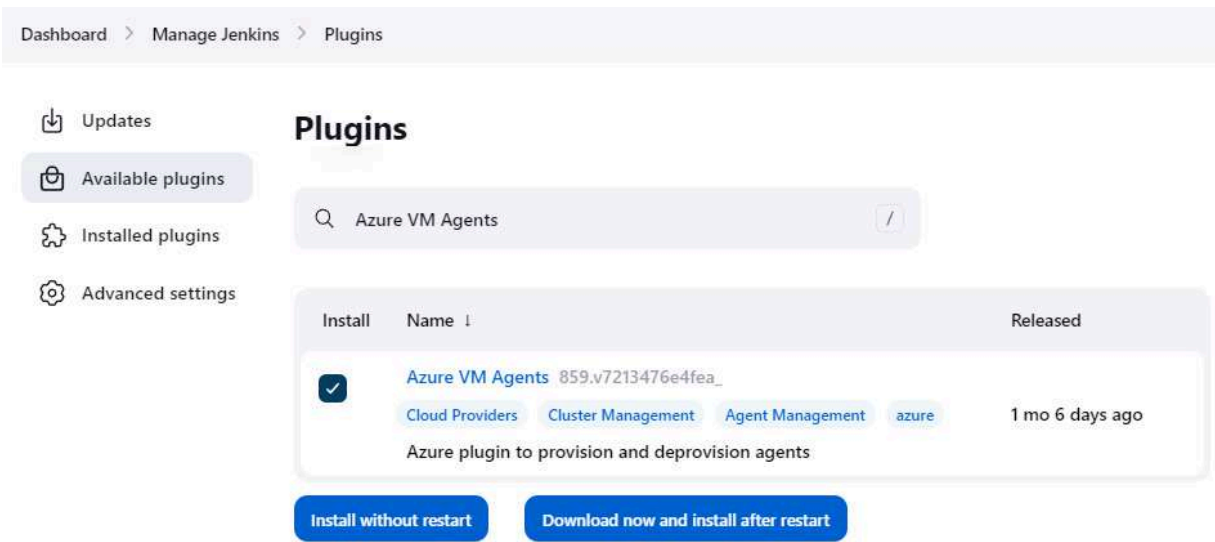


Figure 5.2 – Installing the Jenkins Azure VM Agents plugin

3. Check the checkbox next to the **Azure VM Agents** plugin to select it for installation and click on the **Install without restart** button to begin the installation process. Wait for Jenkins to download and install the plugin. Once the installation is complete, you will see a notification at the top of the page indicating the success of the installation.
4. If you select the **Install without restart** option, you may be prompted to restart Jenkins. Follow the instructions to restart Jenkins for the plugin to take effect.

Creating an Azure service principal in Azure

An Azure service principal is a security identity that you can create in Azure **Active Directory (AD)** and assign specific permissions to. It allows non-human entities, such as applications, services, or automation tools, to authenticate and access Azure resources securely.

In essence, an Azure service principal is a form of application registration in Azure AD. When you create a service principal, it is assigned a unique identifier, also known as an object ID or application ID. This identifier, along with other authentication credentials, can be used to authenticate the service principal and authorize it to interact with Azure resources.

Take the following steps to create an Azure service principal:

1. Sign in to the Azure portal by navigating to the Azure portal (<https://portal.azure.com>) and signing in with your Azure account credentials.
2. Open Azure AD by searching for and selecting **Azure Active Directory** in the search bar at the top.
3. Navigate to the app registrations section from the Azure AD page by selecting **App registrations** from the left-hand menu. This is where you will create the service principal.

4. Create a new application registration by clicking on the **New registration** button to create a new application registration for your service principal.
5. In the **Register an application** form, provide the following information:
 - I. Enter a name for your service principal in the **Name** field.
 - II. Select the appropriate option for the **Supported account types** section based on your requirements. Leaving it as default will also do.
 - III. Leave the **Redirect URI** field blank unless you have specific requirements for authentication flows.
6. Next, click on the **Register** button to create the application registration.
7. After the application registration is created, you will be redirected to the overview page of the application. Make note of the **Application (client) ID** and **Directory (tenant) ID** values, as these are the unique identifiers for your service principal and will be required for authentication by the Azure VM Agents plugin.
8. To authenticate the service principal, we must create a client secret. To do so, from the application overview page, select **Certificates & secrets** from the left-hand menu. Then, click on the **New client secret** button and provide a description for the secret. Make note of the generated secret value, as it will also be required for authentication by the Azure VM Agents plugin.
9. To allow our Azure service principal to create VMs on the Azure cloud, we need to assign it an appropriate role. To do so, switch from the Azure home page and click on your subscription.
10. From your subscription page, click on the **Access control (IAM)** section from the left-hand menu. From there, click on **Add** and then select **Add role assignment**.
11. On the **Role** tab, select the role from the list that you want to assign to the application. For this example, we will pick the **Contributor** role from the **Privileged administrator** roles tab.
12. Once you are done selecting a role, click on the **Next** button to proceed to the next step.
13. On the **Members** tab, select **Assign access to** and then choose **User, group, or service principal**.
14. Next, click on the + **Select members** link and search for your Azure service principal application using the **Select** field:

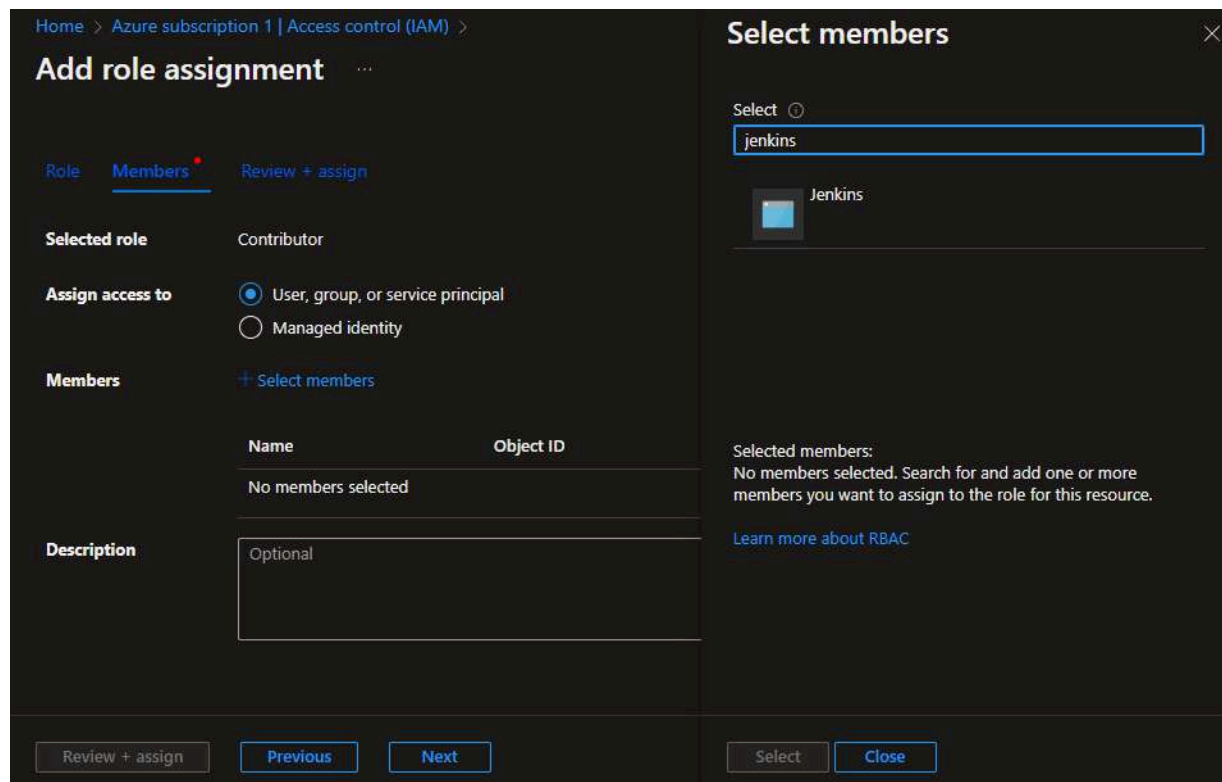


Figure 5.3 – Assigning a contributor role to your Azure service principal

When you find it, select it and click on the **Select** button at the bottom. Finally, click on **Review + assign** to complete the process.

You can now use the service principal in your applications, scripts, or automation tools to authenticate and interact with Azure resources.

Creating an Azure service principal credential inside Jenkins

By creating an Azure service principal credential for the Azure service principal that we created earlier in Azure, you can safely and securely store and manage the credentials within Jenkins, ensuring that they are readily available for the Azure VM Agents plugin configuration, which requires authentication when interacting with the Azure cloud to create on-demand VMs acting as Jenkins build agents. Take the following steps to create an Azure service principal credential inside Jenkins:

1. From the Jenkins dashboard, navigate to **Manage Jenkins | Credentials | System | Global credentials (unrestricted)**, and click **+ Add Credentials**. Configure the credential details as follows:
 - I. In the **Kind** drop-down menu, select **Azure Service Principal**.

II. Leave the **Scope** value as the default.

III. In the **Subscription ID** field, enter the unique identifier of your Azure subscription.

IV. Fill in the **Client ID**, **Client Secret**, and **Tenant ID** fields with the respective values obtained when creating the Azure service principal in the previous section.

V. Provide an **ID** and **description** to help identify and describe the credential.

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

New credentials

Kind

Azure Service Principal

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Subscription ID ?

4fb30eea-

Client ID ?

be1835de-

Client Secret ?

.....

Tenant ID ?

3fc978f9-

Azure Environment ?

Azure

ID ?

azure-vm-jenkins

Description ?

Azure service principal to create on demand VM on Azure

Successfully verified the Microsoft Azure Service Principal.

Verify Service Principal

Figure 5.4 – Creating an Azure service principal credential inside Jenkins

2. Lastly, click on the **Create** button to save the credential. Once saved, you can use this credential inside Jenkins by referencing its **ID**.
3. Once saved, you can utilize this credential in your Azure VM Agents plugin configuration, as well as the Jenkins pipeline, which requires authentication with Azure using the service principal. The credential provides the necessary authentication information for Jenkins to interact with Azure resources securely.

Configuring the Azure VM Agents plugin

In this section, we will create a cloud connection to Azure to allow Jenkins to spin up VMs on demand to run pipelines. Take the following steps:

1. From the Jenkins dashboard, navigate to **Manage Jenkins | Nodes and Clouds | Clouds**.
2. On the **Clouds** page, click on the **Add a new cloud** button and select **Azure VM Agents**.

You will be presented with a long list of fields to be filled, which we will do in two parts. The first part is the general Jenkins-to-Azure connection and some generic settings, and the second part is related to the VMs template.

3. Let's configure the general settings as follows:

- I. Add a meaningful name to the current configuration using the **Cloud name** field.
- II. The **Azure credentials** field will be auto-populated with the only Azure service principal credential we have on our Jenkins server.
- III. The rest of the fields are self-explanatory. For example, the **Max Jenkins agents limit** field limits the total number of VMs you are allowed to create on the Azure cloud, and the **Deployment timeout (seconds)** field tells a pipeline job how long it should wait for the VM to get ready.
- IV. The **Resource group name** section has two options: **Create new** and **Use existing**. When you choose the first one, Jenkins, using the Azure service principal, will try to create all VMs under a new resource group on Azure. The name of the new resource group needs to be specified using the **Name** field. The latter option allows you to use an existing resource group on Azure.
- V. The **Verify configuration** button lets you verify that the Azure service principal credential is good enough to create a VM on the Azure cloud.

Azure VM Agents

[View the log](#)

Cloud name ?

Default Azure Build Farm

Azure credentials ?

azure-vm-jenkins (Azure service principal to create on demand VM on Azure) ▼

Add ▼

Max Jenkins agents limit ?

10

Deployment timeout (seconds) ?

1200

Custom tag ?

Add

Resource group name

Create new

Name ?

Use existing

Successfully verified Azure configuration.

Verify configuration

Add Azure Virtual Machine template

Figure 5.5 – Azure VM Agents general settings

With this, the general settings are done. Now let's look at the VM templates settings:

1. Click on the **Add Azure Virtual Machine template** button to add a new VM template. You will be asked to provide a name for your template using the **Name** field:

The image shows a dashed rectangular box representing a form. At the top left, the label 'Name' is followed by a question mark icon. At the top right, there is a red circular button with a white 'X' icon. Below the label, there is a text input field containing the text 'default-azure-vm-template'. At the bottom left, there is a button labeled 'Agent template details' with a downward-pointing chevron icon. At the bottom right, there is a pencil icon followed by the text 'Edited'.

Figure 5.6 – Naming your Azure VM template

2. Next, click on **Agent template details**. We will only discuss the important ones:
 - The **Labels** field allows you to put a unique tag on all VM nodes created using the current configuration on the Azure cloud. This allows Jenkins pipelines to specifically run on an Azure VM-based Jenkins agent.
 - The **Admin credentials** field allows you to use a **username and password** or an **SSH username with a private key-based** credential to log in to the VM created on Azure. Use the **Add** button to create one.

General configuration

Description ?

Default Azure Windows VM Template To Build C++/C# Code

Labels ?

azure-windows-agent

Admin credentials ?

nikhilpathania/***** (Log in credentials for Azure VM Jenkins agents.)



Add ▾

Max Jenkins agents limit ?

10

Figure 5.7 – General settings for the VM template

- The **Agent workspace** field allows you to define the location inside the VM to execute the steps of a stage of a pipeline. Add OS-appropriate paths.
- **Region**, **Availability options**, and **Virtual machine size** are all self-explanatory. However, choosing the **No infrastructure redundancy required** option for the **Availability options** field is a better choice since build agents must be used and discarded, meaning they need not be resilient.

Agent workspace ?

c:\jenkins

Region ?

West Europe

Availability options ?

No infrastructure redundancy required

Virtual machine size ?

Basic_A0

Figure 5.8 – Configuring Agent workspace, Region, Availability options, and Virtual machine size for the VM template

- The **Storage account type** field allows you to define the type of storage for your VM agent. The options available depend on the **Virtual machine size** and **Region** options you choose.
- The **Storage account name** section has two options: **Create new** or **Use existing**. Selecting the first one will create a new storage account. Use the **Name** field to provide a unique name. If you leave it blank, Jenkins will automatically name and create a new storage account for you. If you are using an existing storage account, make sure to select the **Use existing** option to avoid potential errors related to typos and replication strategy.

Storage account type ?

Standard_LRS ▼

Storage account name

☒ Create new

Name ?

☐ Use existing

Figure 5.9 – Configuring Storage account type and Storage account name settings for the Azure VM template

- The **Choose disks type** field has two options: **Managed Disk** and **Unmanaged Disk**. In Azure, managed and unmanaged disks refer to different storage options for VMs. Here's an explanation of them:
 - **Managed disks**: Azure managed disks provide a simplified and managed storage solution for VMs. With managed disks, Azure takes care of the underlying storage infrastructure, including disk creation, management, and maintenance.
 - **Unmanaged disks**: Unmanaged disks, on the other hand, require manual management of storage accounts and the VM's **virtual hard disks (VHDs)**. With unmanaged disks, you must create and manage your storage accounts separately, and then attach the VM's VHDs to the VM.

Choose disks type ?

Unmanaged Disk ▼

OS disk storage account type ?

Standard_LRS ▼

☐ Use Ephemeral OS disk

OS disk size in GB (Optional) ?

20

☐ Enable encryption at host

Figure 5.10 – Configuring disk options for the Azure VM template

- **Retention strategy** has the following options:
 - **Azure VM Idle Retention Strategy:** By default, agents will be deleted or shut down after a specific retention time if they remain idle.
 - **Azure VM Pool Retention Strategy (Experimental):** This retention strategy helps maintain a specific number of agents in a pool. When saving the configuration page with templates that use this strategy, a specified number of agents will be provisioned. If a retention time is specified (not zero), agents will be deleted after that time and replaced with new ones to maintain the pool size. Changing cloud or template settings will result in the deletion and provisioning of agents accordingly, while changes to retention time or pool size will only adjust scaling.
 - **Azure VM Once Retention Strategy:** With this strategy, agents are used only once. Once a job is completed, the agent is marked as offline and suspended until a periodic cleanup task is executed. It is recommended to use a single executor with this strategy, but if multiple executors are used, the agent will be marked as offline and suspended when the first job finishes. The agent is not truly deleted until it becomes completely idle, and any remaining running jobs are not terminated.

Retention strategy ?

Azure VM Idle Retention Strategy



Retention Time In Minute ?

60

☐

Shutdown instead of deleting ?

Figure 5.11 – Retention strategy settings for the Azure VM template

- The **Usage** field has the following options:
 - **Use this node as much as possible:** This is the default setting. Jenkins freely schedules builds on this node whenever possible. If a build can be performed on this node, Jenkins will use it.
 - **Only build jobs with label expressions matching this node:** In this mode, Jenkins restricts building a project on this node to cases where the project has specified label expressions that match this node's name and/or labels. This allows you to reserve a node for specific types of jobs. For example, if certain jobs, such as performance tests, should only run on a specially configured machine, a label expression matching that machine can be used to restrict those jobs to that node. By setting the number of executors to 1, only one performance test will execute on that machine at a time, preventing interference from other builds.

Usage ?

Use this node as much as possible



Maximum deployment size (Optional) ?

10

Figure 5.12 – Usage settings for the Azure VM template

3. In the **Image configuration** section, you have two options. You can either go with pre-cooked images, such as **Windows Server 2019** or **2022**, by selecting **Use built-In image**, or you can employ a custom image using the **Use advanced image** option:

Image configuration

☒ Use built-in image

Windows Server 2019 ▼

Pre-installed tools

☒ Install Git

☐ Install Maven

☐ Install Docker

☐ Use advanced image

Verify template

Figure 5.13 – Image configuration settings for the Azure VM template

4. Finally, click on **Apply** and then **Save** to save the configuration.
5. Now, your Jenkins instance is configured to integrate with the Azure cloud using the Azure VM Agents plugin. You can use this configuration to deploy on-demand VMs on Azure to run your pipelines.

Summary

In this chapter, we dedicated our attention to the crucial topic of horizontally scaling Jenkins. We comprehensively covered the two most popular methods for achieving this, starting with the utilization of the pre-configured Kubernetes plugin, followed by the employment of the Azure VM Agents plugin. These strategies enabled us to effectively scale Jenkins by creating on-demand build agents on the fly. With these accomplishments, we have successfully established a minimal CI setup, marking a significant milestone in our journey.

In the upcoming chapter, we'll delve deep into Jenkins' **Pipeline as Code**, focusing specifically on the declarative pipeline syntax. Mastering this will prime us to harness AI for enhancing and optimizing our pipeline creation.

Questions

1. Which of the following is not an advantage of the master-agent model?
 - A. Scalability
 - B. Flexibility
 - C. Fault tolerance
 - D. High availability
2. Which of the following statements is false about using Jenkins agents on a Kubernetes cluster compared to fixed VM-based agents?
 - A. Kubernetes-based agents offer better and more efficient resource utilization than fixed VM-based agents
 - B. Kubernetes-based agents provide less elasticity and scalability compared to fixed VM-based agents
 - C. Kubernetes-based agents require less manual intervention for scaling compared to fixed VM-based agents
 - D. Kubernetes-based agents have less limited integration with cloud-native tools than fixed VM-based agents
3. Choose whether this statement is true or false: *One advantage of using VM-based on-demand Jenkins agents on the cloud is their suitability for running Windows-based legacy applications that may not be compatible with containerized environments such as Kubernetes.*
4. Choose whether this statement is true or false: *Jenkins can be configured to connect with multiple Kubernetes clusters to manage and distribute workloads across different environments.*
5. What is the purpose of a pod template in Jenkins' Kubernetes plugin configuration?
 - A. To specify the resource limits for the Jenkins master node
 - B. To define the Jenkins agent configuration, including the container template used for running builds
 - C. To outline the backup and recovery procedures for Jenkins data
 - D. To set global environment variables for all Jenkins jobs

Answers

1. Correct answer: **D**

Explanation:

While the master-agent model improves scalability, flexibility, and fault tolerance, it does not inherently guarantee high availability, especially regarding the master node.

2. Correct answer: **B**

Explanation:

Kubernetes-based agents can be spun up quickly when needed and decommissioned when not, optimizing resource usage much more effectively than fixed VM-based agents, which typically have static resource allocation and less flexibility in scaling.

3. Correct answer: **True**

Explanation:

VMs provide a more traditional and isolated environment, offering the full functionality of an OS. This makes them more suitable for legacy applications that require specific configurations or are not yet optimized for containerization.

4. Correct answer: **True**

Explanation:

Jenkins has the capability to be configured to connect with multiple Kubernetes clusters.

5. Correct answer: **B**

Explanation of incorrect answer choices:

A: This option is incorrect because the pod template in Jenkins' Kubernetes configuration is not used to define resource limits for the Jenkins master node. Instead, it's used to configure the Jenkins agents.

C: The pod template has nothing to do with outlining backup and recovery procedures. Backup and recovery strategies for Jenkins data are managed through external backup tools, not through the Kubernetes pod template.

D: The pod template is specifically for configuring agent environments, not for setting global environment variables for all Jenkins jobs. While environment variables can be specified in a pod template, they apply only to the agents created using that template.

Further reading

If you wish to learn more about some of the concepts that aren't covered in depth in this chapter or are out of the scope of this book but relevant, here is a reference:

- [1] Kubernetes plugin documentation: <https://github.com/jenkinsci/kubernetes-plugin>

Part 3: Crafting the CI Pipeline

In the third part of our journey through the complicated machinery of **Continuous Integration (CI)**, our story unfolds on the CI factory floor, where Jenkins pipeline code becomes our tools of the trade. We will begin our study with the Jenkins pipeline syntax and structure, preparing us to create the blueprints of complicated automation processes with engineering accuracy. Our production line then welcomes a game-changing collaborator – ChatGPT’s artificial intelligence. This collaboration is analogous to adding cutting-edge robotics to traditional manufacturing, transforming the possibilities of our assembly line.

With the theoretical foundation in place, we will prepare for the real work – laying the framework for our first CI pipeline. We will meticulously design each stage, ensuring that each component fits flawlessly into the next. Our quality control and material management stations, SonarQube and Artifactory, respectively, will ensure that every artifact is up to standard and kept safe. The actual implementation of the CI pipeline code is the climax of our efforts. This is where our rough sketches come to life. We will methodically construct our pipeline code in the same way that a machinist would craft a custom component, ensuring that each stage of our CI process is tested, analyzed, and seamlessly integrated into the overall workflow.

This part has the following chapters:

- [*Chapter 6, Enhancing Jenkins Pipeline Vocabulary*](#)
- [*Chapter 7, Crafting AI-Powered Pipeline Code*](#)
- [*Chapter 8, Setting the Stage for Writing Your First CI Pipeline*](#)
- [*Chapter 9, Writing Your First CI Pipeline*](#)

Enhancing Jenkins Pipeline Vocabulary

In our learning journey so far, we focused on building a robust **continuous integration (CI)** setup by integrating Jenkins and an array of DevOps tools. Having concluded the setup phase, we now embark on a new and exciting endeavor of exploring the capabilities of Jenkins for CI. Our journey into this realm begins with an exploration of the art of writing pipeline code, a pivotal aspect that this chapter broadly addresses. Interestingly, the topic of pipeline code resonates across a broader audience, empowering both DevOps practitioners and software developers alike to orchestrate the software delivery process. It undoubtedly forms the backbone of the continuous integration, delivery, and deployment processes.

This chapter offers theoretical knowledge about the Jenkins pipeline syntax. Here, we will dive deep into the realm of Jenkins Declarative Pipeline code vocabulary where you will familiarize yourself with the various essential elements of pipeline code, focusing primarily on the declarative approach and its advantages. This is followed by the contemporary method of constructing pipeline code that involves leveraging the built-in tools within Jenkins.

We will cover the following main topics in the chapter:

- Building vocabulary for pipeline code
- Using built-in tools to construct pipeline code

Technical requirements

To proceed with this chapter, please ensure that you have a Jenkins server up and running and an IDE, such as Visual Studio Code, on your local machine.

Building vocabulary for pipeline code

Social media has unfortunately fostered a misguided perception of AI tools such as ChatGPT and Google Bard, leading many to believe that these technologies may threaten job security within the software community. The truth is that AI tools serve as valuable assistants rather than replacements. A prime example of this is the undeniable necessity of having a solid Jenkins pipeline code vocabulary, even when leveraging AI such as ChatGPT to aid in writing the code. Here is why:

- **Effective communication with AI:** To utilize AI effectively, you need to provide clear and structured input. Understanding the vocabulary of Jenkins pipeline code enables you to communicate your requirements precisely to the AI. The AI model can then

use this knowledge to generate more accurate and relevant code.

- **Interpreting AI output:** When the AI generates code, it might include specific terms and concepts related to the Jenkins pipeline. Having a solid vocabulary allows you to understand and interpret the AI's output correctly. This helps you review, modify, or enhance the generated code if needed, ensuring that it aligns with your project's requirements.
- **Fine-tuning AI suggestions:** While AI can provide helpful suggestions, you may want to customize or fine-tune the generated code to suit your unique needs. Understanding the pipeline code vocabulary empowers you to tweak the AI's output effectively and optimize the resulting code.
- **Debugging and troubleshooting:** In the process of working with AI-generated code, you might encounter issues or errors. Having a good grasp of the pipeline code vocabulary aids in debugging and troubleshooting any problems that arise, allowing you to identify and resolve issues faster.

Having said that, let us start with a quick introduction to Jenkins pipeline code.

Introducing Jenkins pipeline code

In the first chapter, we briefly introduced the concept of **Pipeline as Code**. Now, let's delve deeper into it within the context of Jenkins. There are three different ways you can write your pipeline code in Jenkins: in plain Groovy, using Scripted Pipeline syntax, and using Declarative Pipeline syntax.

Initially, Jenkins adopted the Groovy programming language as the basis for creating a **domain-specific language (DSL)** to define pipelines. This led to the birth of the **Scripted Pipeline syntax**, which was met with widespread acceptance and continued to be a popular choice. Its appeal lay in its flexibility and adaptability, offering users the freedom to construct pipelines according to their specific needs. However, mastering the Scripted Pipeline syntax requires a solid understanding of Groovy programming, making it more accessible to experienced Jenkins users.

Recognizing the need for a more approachable and structured pipeline syntax, Jenkins introduced the **Declarative Pipeline syntax**. This syntax simplified the process of defining pipelines by providing a more straightforward and user-friendly approach. It aimed to make the concept of Pipeline as Code accessible to a wider range of Jenkins users, including those without advanced programming skills. As a result, the Declarative Pipeline syntax became the foundation for Jenkins features such as Blue Ocean, offering a **visual pipeline editor** to facilitate easy pipeline creation. Let us learn a little bit more about the Jenkins Scripted Pipeline syntax.

Jenkins Scripted Pipeline syntax

The Scripted Pipeline syntax is an example of a domain-specific and imperative approach to writing pipeline code. For those unfamiliar with these programming concepts, let's briefly define imperative programming and DSL:

- **Imperative programming:** In imperative programming, the program specifies both what needs to be achieved and how to achieve it. Programming languages such as C, C++, Java, and Groovy fall under this category.

- **Domain-specific language (DSL):** A DSL is designed to address specific problem areas. Examples include Apache Ant, Apache Maven, and Gradle. Unlike general-purpose languages, DSLs are tailored to work within distinct domains.

The Scripted Pipeline syntax, being unrestricted, allows for the inclusion of Groovy code within it. This syntax is powerful but comes with a steep learning curve, making it less desirable for many users. It can also be challenging to maintain and comprehend. Imagine a situation where you need to debug or modify a scripted pipeline code written by a former colleague who is no longer with the team. Such scenarios become less desirable when there are numerous Jenkins pipelines written using the Scripted Pipeline syntax that require ongoing maintenance. This challenge prompted Jenkins engineers to develop a more user-friendly DSL, leading to the creation of the Declarative Pipeline syntax.

While the Declarative Pipeline syntax may not match the Scripted Pipeline syntax in terms of versatility and flexibility, it compensates for these limitations by leveraging Jenkins' shared libraries. This combination offers the best of both worlds, enabling users to benefit from the strengths of the Declarative Pipeline while still accessing the capabilities of the Scripted Pipeline syntax through shared libraries. We will learn more about Jenkins' shared libraries in upcoming chapters. Nevertheless, here is an example of a scripted pipeline code:

```
node {
    stage('Build') {
        echo 'Building the project.'
        sh 'mvn clean package'
    }
    def testResult
    stage('Test') {
        echo 'Running tests.'
        testResult = sh(returnStatus: true, script: 'mvn test')
    }
    stage('Deploy') {
        echo 'Deploying the application.'
        if (testResult == 0) {
            echo 'Tests passed. Deploying...'
            sh 'scp target/my-app.jar user@deploy-server:/opt/app/'
        } else {
            echo 'Tests failed. Skipping deployment.'
        }
    }
}
```

This example showcases the key features that differentiate the Scripted Pipeline syntax from the Declarative Pipeline syntax. They are explained as follows:

- **Defining variables:** In the scripted pipeline, you can define variables using **def**. Here, we define a variable called **testResult** to store the result of the test execution.
- **Using environment variables:** We use the **testResult** variable to store the exit status of the test execution, which we then use in the conditional statement during the **deploy** stage.

- **Custom scripting:** The deploy stage demonstrates more complex scripting, where we check the value of `testResult`. If the tests pass (exit status 0), we proceed with deployment using SCP. Otherwise, if the tests fail, we skip the deployment step.

These features are more evident in the Scripted Pipeline code, where you have more flexibility and control over the flow of the pipeline. However, keep in mind that with this flexibility, the scripted pipeline can become more verbose and harder to maintain as the pipeline complexity increases.

Jenkins Declarative Pipeline syntax

The Declarative Pipeline utilizes a domain-specific and declarative syntax, emphasizing a different approach compared to scripted pipelines. For those unfamiliar with declarative programming, it is a coding standard that focuses on describing the desired outcome rather than explicitly stating how to achieve it. The compiler is responsible for determining the necessary steps to accomplish the task. Examples of declarative programming include SQL, HTML, XML, and CSS.

The Declarative Pipeline syntax is well-structured and boasts a smoother learning curve, making it easy to compose and maintain. However, it does have certain limitations in terms of available options for pipeline authors. Like the Scripted Pipeline syntax, the Declarative Pipeline allows the usage of Groovy code, but it enforces a more structured approach. Here's an example of Declarative Pipeline code:

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        echo 'Building the project.'
      }
    }
    stage('Test') {
      steps {
        echo 'Running tests.'
      }
    }
    stage('Deploy') {
      steps {
        echo 'Deploying the application.'
      }
    }
  }
  post {
    always {
      echo 'Will always run, irrespective of success or failure'
    }
  }
}
```

In this example, we can observe several differences from the scripted pipeline:

- **Declarative structure:** The Declarative Pipeline uses a structured approach with keywords such as **pipeline**, **agent**, **stages**, **steps**, and **post**. This helps in creating a more organized and readable pipeline.

- **Built-in steps:** The Declarative Pipeline provides built-in steps, such as **echo**, for simple operations without the need to use the **sh** step for every command. This simplifies the syntax and makes the pipeline code more concise.
- **Simplified stages:** The Declarative Pipeline uses the **stages** block to define stages and their corresponding **steps**. This helps in maintaining clear separations between the different phases of the pipeline.
- **Post-build actions:** The Declarative Pipeline uses the **post** block to define post-build actions, such as actions that should always run (**always**), actions for successful builds (**success**), and actions for failed builds (**failure**). This simplifies handling various outcomes of the pipeline.

Overall, the Declarative Pipeline provides a more straightforward and structured approach to defining pipelines, enhancing readability, comprehension, and maintenance, particularly for simpler use cases. However, in scenarios demanding extensive customization and scripting, the Scripted Pipeline syntax may be favored due to its higher flexibility, though such situations are rare. The software development landscape has evolved, with advancements in frameworks for code building, testing, and other processes. Moreover, the shift away from monolithic software development has contributed to the growing popularity of the Declarative Pipeline, as it aligns well with modern software development practices. Consequently, in the current chapter and throughout this book, our focus will only be on the Declarative Pipeline syntax.

Understanding the components of Declarative Pipelines

The section is very important. Here, you will gain an in-depth understanding of each element of the Declarative Pipeline syntax, accompanied by illustrative examples of pipeline code. Let's begin, starting with **sections**.

Sections

In Jenkins Declarative Pipeline syntax, sections define different aspects of the pipeline configuration, organize the pipeline code, and provide a structured and readable way to define the various stages, steps, post-build actions, and more. The sections in the Jenkins Declarative Pipeline syntax include **pipeline**, **agent**, **stages**, **steps**, and **post**. Let's see each one of them.

Pipeline

In the Jenkins Declarative Pipeline syntax, the **pipeline** section is the heart of the pipeline configuration. It serves as the main block that encapsulates and defines the entire pipeline structure. It's the first thing you type when formulating a pipeline code and is a mandatory block. Within the **pipeline** section, you can specify various aspects of the pipeline, such as the agent where the pipeline

will be executed, the stages that make up the CI/CD process, global options, post-build actions, and more.

The pipeline section provides a clear and organized way to represent the pipeline's flow and logic. It acts as a container that holds all the other sections, directives, and steps, helping to maintain a well-structured and readable pipeline definition. Here are some quick rules about the `pipeline` section that you must always remember:

- Jenkins pipeline code should always be encapsulated within a pipeline section: `pipeline { }`
- The pipeline section must contain only sections, directives, and steps, and no other imperative code is allowed

Agent

In the Jenkins Declarative Pipeline syntax, the `agent` section is used to specify where the pipeline will be executed. It defines the environment where the different stages and steps of the pipeline will run. The agent section allows you to allocate specific resources, such as Jenkins nodes or Docker containers, to execute the pipeline steps. There are several ways to use the `agent` section:

- Agent types:
 - **agent any:** This allows the pipeline to run on any available agent in the Jenkins environment. It is the most straightforward option but may lead to different agents executing different runs of the pipeline.
 - **agent none:** This allows you to define a completely custom execution environment for the pipeline. It is useful for creating “dummy” pipelines or to conditionally start a pipeline later in the process.
 - **agent { label 'labelName' }:** This lets you specify a specific label (node) in Jenkins where the pipeline should run. The node with the specified label will be dynamically allocated to execute the pipeline.
 - **agent { docker { ... } }:** This enables running the pipeline inside a Docker container. You can specify the Docker image to use, along with any additional settings related to the Docker environment.
 - **agent { kubernetes { ... } }:** This enables running the pipeline inside a pod on a Kubernetes cluster. You can specify the Docker image to use, along with any additional settings related to the Kubernetes environment.

- Agent configuration:

The agent section can include additional configurations, such as `environment variables`, and `tools`. For example, you can define environment variables specific to the agent or you can specify the tools and versions required for the pipeline execution.

Here's a simple example using the `agent` section:

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        sh 'mvn clean package'
```



```

    }
  }
}

```

In this example, the **agent any** statement ensures the pipeline will execute on any available agent in the Jenkins environment. The **build** stage's steps will run within this agent, performing the Maven build. This is just one way to use the agent section, and you can choose different options based on your specific requirements and Jenkins setup. Defining an **agent** section inside your pipeline is mandatory. Optionally, the **agent** section can also be placed inside a **stage** directive to allow individual stages of a pipeline to run on different machines/containers.

Stages

The **stages** section allows you to define one or more **stage** directives, each representing a distinct phase of the CI/CD pipeline process. The **stages** section is defined at least once inside your pipeline section. It exists solely to describe the sequential flow of execution of any **stage** directives included inside it. Here's a simple example of the **stages** section:

```

pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        echo 'Building the project.'
        sh 'mvn clean package'
      }
    }
    stage('Test') {
      steps {
        echo 'Running tests.'
        sh 'mvn test'
      }
    }
    stage('Deploy') {
      steps {
        echo 'Deploying the application.'
        sh 'scp target/my-app.jar user@deploy-server:/opt/app/'
      }
    }
  }
}

```

In this example, the **stages** section defines three **stage** directives: **Build**, **Test**, and **Deploy**. When the pipeline runs, it will execute these stages in sequence, ensuring a controlled and structured pipeline flow. The stages section also allows you to include conditional execution of stages inside it using the **when** directive. This allows you to run specific stages based on certain conditions, making your pipeline more flexible and adaptable to different scenarios. You will learn more about the **when**

directive in a while. Nevertheless, the only thing you need to remember about the **stages** section while composing a pipeline code is to have one.

Steps

In the Jenkins Declarative Pipeline syntax, the **steps** section is used to define the individual tasks or commands that need to be executed within a specific stage of the pipeline. Each stage can have its own steps block, allowing you to specify the commands or actions required to perform the tasks related to that stage. Inside the **steps** section, you can use various built-in step functions or plugin-provided steps by Jenkins to execute commands, perform tests, deploy artifacts, publish reports, and more. These steps are the building blocks that enable you to create a fully automated and configurable pipeline. The commands inside the **steps** section can only be executed in sequence. Here's a simple example of a **steps** section:

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        echo 'Building the project.'
        sh 'mvn clean package'
      }
    }
    stage('Test') {
      steps {
        echo 'Running tests.'
        sh 'mvn test'
      }
    }
    stage('Deploy') {
      steps {
        echo 'Deploying the application.'
        sh 'scp target/my-app.jar user@deploy-server:/opt/app/'
      }
    }
  }
}
```

In this example, each stage contains its own set of steps defined using the **sh** step, which allows you to run shell commands. The **echo** step is used to display messages during the pipeline execution for better visibility. Besides shell commands, the Declarative Pipeline provides a wide range of built-in steps for various purposes, such as **git**, **input**, **archive**, and more.

A very useful and popular pipeline step that is worth mentioning is the **script** step. It is used to execute arbitrary script code within a specific stage or block of the pipeline. It allows you to include custom logic, complex calculations, or interactions with Jenkins APIs that may not be directly supported by other pipeline steps. The **script** step has the following structure:

```

pipeline {
  agent any
  stages {
    stage('Custom Script') {
      steps {
        script {
          // Groovy code to be executed
          def variable = 'Hello, World!'
          echo variable
        }
      }
    }
  }
}

```

In this example, the `script` step is used within the `Custom Script` stage. Inside the `script` block, you can write any code you need. In this case, a variable named `variable` is defined and then echoed using the `echo` step. Keep in mind that while the `script` step provides flexibility, it's important to strike a balance between using built-in pipeline steps and custom scripting, as excessive scripting may make the pipeline harder to maintain and understand.

To get a complete list of available steps, visit the *Pipeline Step reference* documentation (see [1] in the *Further reading* section at the end of the chapter). The document refers to the extensive collection of built-in functions or methods that can be utilized within your pipeline code. These steps perform a variety of functions, ranging from interfacing with source code repositories and developing software to deploying apps and providing notifications. The *Pipeline Steps reference* documentation is essentially a toolbox that allows you to create complicated automation workflows without writing custom code for each operation. It contains a large number of steps classified into many groups.

In addition to the documentation, in the upcoming sections, you will also learn about an integrated Snippet Generator tool. This tool is useful for creating code fragments for individual stages, experimenting with innovative steps provided by plugins, and evaluating various settings for a specific step. The Snippet Generator dynamically displays a list of available steps from the Jenkins instance. The number of accessible steps is determined by the plugins that have been installed that explicitly make steps available for use in the Pipeline.

Post

In the Jenkins Declarative Pipeline syntax, the `post` section is used to define post-build actions that should be executed after the pipeline or a stage has been completed, regardless of whether it succeeded or failed. The `post` section allows you to specify different steps and conditions based on the outcome of the pipeline or a stage, such as `success`, `failure`, `unstable`, `aborted`, or even `always`.

Within the `post` section, you can use different directives to specify the actions you want to perform based on the pipeline result. The common directives include:

- **always:** This specifies actions that should always be executed, regardless of the pipeline's result.
- **success:** Actions defined under this directive will be executed only if the pipeline completes successfully (exit code 0).
- **failure:** Actions defined under this directive will be executed only if the pipeline fails (non-zero exit code).
- **unstable:** Actions defined under this directive will be executed only if the pipeline result is marked as unstable. An unstable pipeline is when a build is successful but one or more publishers report it as unstable. This can happen if certain conditions are not met, such as if a specific test fails or if there are warnings in the build logs. Publishers are plugins that provide additional functionality to Jenkins, such as reporting test results or collecting artifacts.
- **changed:** Actions defined under this directive will be executed only if the pipeline result has changed. Here's how the `changed` block functions. It compares the current pipeline run's completion status with the previous one. If the current run's status differs from the previous run (e.g., the previous run was successful but this run is unstable), the steps within the `changed` block are executed. If the current run's status is the same as the previous one, the steps within the `changed` block are skipped.

Here's an example of a `post` section:

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        echo 'Building the project.'
        sh 'mvn clean package'
      }
    }
  }
  post {
    always {
      echo 'This will always run, regardless of success or failure'
    }
    success {
      echo 'This will run only if the pipeline succeeds'
    }
    failure {
      echo 'This will run only if the pipeline fails'
    }
  }
}
```

In this example, the **always** directive ensures that the specified step will be executed no matter what the pipeline result is. The **success** and **failure** directives specify steps that will be executed only if the pipeline succeeds or fails, respectively. Depending on the outcome of the pipeline, the corresponding steps will be displayed in the console log.

Using the `post` section, you can define additional actions after the pipeline execution, such as sending notifications, archiving artifacts, publishing reports, cleaning up resources, or triggering downstream jobs. This section enhances the flexibility and automation capabilities of your pipeline, enabling you to handle various post-build tasks effectively.

Directives

In the Jenkins Declarative Pipeline syntax, directives are powerful instructions that control the flow and behavior of the pipeline. They are used within the `pipeline` or `stage` blocks to customize the behavior of the pipeline steps and stages. Directives enable users to define conditions, loop structures, parallel execution, and more, making the pipeline code more flexible and adaptable to different scenarios. Some commonly used directives include `when`, `environment`, `options`, and `parameters`.

With the versatility of directives, users can create sophisticated and automated pipelines that cater to various development and deployment needs. In Declarative Pipeline syntax, directives act as a provisioner to all your pipeline steps, helping them accomplish their goals. Let's learn about all the directives.

Environment

In the Jenkins Declarative Pipeline syntax, the `environment` directive is used to define environment variables that will be available to steps within various stages of a pipeline. This allows you to set up environment-specific configurations or provide essential data that multiple steps in a stage may require. The `environment` directive can be set either at a global level or inside a `stage` directive. The scope of an `environment` directive is decided by where it's defined. Here's an example of how to use the `environment` directive in Declarative Pipeline:

```
pipeline {
  agent any
  stages {
    stage('Build') {
      environment {
        MAVEN_HOME = '/usr/local/maven'
        JAVA_HOME = '/usr/lib/jdk11'
      }
      steps {
        sh 'echo "Maven Home: $MAVEN_HOME"'
        sh 'echo "Java Home: $JAVA_HOME"'
        sh 'mvn clean package'
      }
    }
  }
}
```

In this example, we have a `Build` stage with two environment variables specified in the `environment` block: `MAVEN_HOME` and `JAVA_HOME`. These variables will be accessible to all the steps within the `Build` stage. Inside the `steps` block of the `Build` stage, we use the `sh` step to execute shell commands. We can use the defined environment variables (`$MAVEN_HOME` and `$JAVA_HOME`) in these shell commands. When the pipeline runs, the shell commands will display the values of the environment variables. The

variables will be set for all the commands executed within the `Build` stage, ensuring consistent and predictable behavior throughout the stage.

The `environment` directive is beneficial when you want to avoid redundant declarations of environment variables in multiple steps and keep the pipeline code more organized and maintainable. It allows you to establish a consistent environment for a specific stage, making it easier to manage complex pipelines with multiple stages and steps.

Options

In the Jenkins Declarative Pipeline syntax, the `options` directive is used to specify global options that apply to the entire pipeline or specific stages. These options help customize the behavior and execution of the pipeline as a whole. Here are some examples of how to use the `options` directive in the Declarative Pipeline:

- **timeout**: The `timeout` option sets a maximum time limit for the entire pipeline or specific stages. If the pipeline or a particular stage exceeds the specified time limit, it will be automatically aborted.
- **retry**: The `retry` option specifies the number of retries for the entire pipeline or specific stages. If the pipeline or a particular stage fails, it will be retried the specified number of times.
- **skipDefaultCheckout**: This option allows you to skip the default checkout of the source code repository at the beginning of the pipeline. You can use this option when you want to perform a custom checkout within your pipeline:

```
pipeline {
  agent any
  options {
    timeout(time: 1, unit: 'HOURS')
    retry(3)
    skipDefaultCheckout()
  }
  stages {
    stage('Build') {
      steps {
        // Your build steps go here
      }
    }
    stage('Test') {
      steps {
        // Your test steps go here
      }
    }
  }
}
```

The `options` directive is useful for tailoring the behavior of your Jenkins pipeline to match specific requirements. To see the complete list of available `options` directives, visit the *Options* section from the Jenkins Declarative Pipeline syntax documentation [2].

Parameters

In the Jenkins Declarative Pipeline syntax, the **parameters** directive is used to define and declare parameters that can be provided during the pipeline run. Parameters allow you to make your pipeline more dynamic and configurable, enabling users to pass values when triggering the pipeline rather than hardcoding them in the pipeline script. There are several types of parameters that can be defined using the **parameters** directive:

- **string:** This allows users to input a simple string value when starting the pipeline:

```
pipeline {
  agent any
  parameters {
    string(name: 'ENVIRONMENT', defaultValue: 'dev', description: 'Specify the
environment for deployment')
  }
  stages {
    stage('Deploy') {
      steps {
        echo "Deploying to ${params.ENVIRONMENT} environment."
        // Your deployment steps go here
      }
    }
  }
}
```

- **booleanParam:** This allows users to specify a **true/false** value when triggering the pipeline:

```
pipeline {
  agent any
  parameters {
    booleanParam(name: 'RUN_TESTS', defaultValue: true, description: 'Run tests
as part of the build')
  }
  stages {
    stage('Test') {
      when {
        expression {
          params.RUN_TESTS == true
        }
      }
      steps {
        echo 'Running tests.'
        // Your test steps go here
      }
    }
  }
}
```

- **choice:** This provides a list of options for users to choose from:

```
pipeline {
  agent any
  parameters {
    choice(name: 'BUILD_TYPE', choices: ['Debug', 'Release'], description:
'Select the build type')
  }
  stages {
    stage('Build') {
      steps {
        echo "Building ${params.BUILD_TYPE} version."
      }
    }
  }
}
```

```

        // Your build steps go here
    }
}
}
}

```

- **file:** This allows users to upload a file when triggering the pipeline:

```

pipeline {
    agent any
    parameters {
        file(name: 'CONFIG_FILE', description: 'Upload the configuration file')
    }
    stages {
        stage('Process Config') {
            steps {
                echo "Processing ${params.CONFIG_FILE}."
                // Your processing steps go here
            }
        }
    }
}

```

These examples showcase the versatility of the **parameters** directive in creating dynamic and interactive pipelines. However, in the context of building a CI pipeline in Jenkins, it is essential to avoid features that pause the pipeline for user input before execution. A CI pipeline triggered by a code change should operate seamlessly from start to finish, without any interruption or human intervention. Despite this, it demonstrates that Jenkins goes beyond being a mere CI/CD solution. By incorporating parameters, you empower users to customize the pipeline's behavior according to their requirements, enhancing automation adaptability and user-friendliness.

Triggers

In the Jenkins Declarative Pipeline syntax, triggers are used to specify how the pipeline should be automatically triggered or scheduled to run. There are various trigger options available to suit different use cases, and you can find the complete list by visiting the *Triggers* section in the Jenkins Declarative Pipeline syntax documentation [3]. For pipelines integrated with a source control system such as GitHub or GitLab, explicit triggers may not be necessary since webhooks-based integration is likely to be already in place. Webhooks facilitate the automatic triggering of pipeline runs whenever changes are pushed to the repository, making separate trigger configurations redundant.

Stage

In the Jenkins Declarative Pipeline syntax, the **stage** directive is used to define a distinct phase or section of the pipeline. Each **stage** represents a logical division of the pipeline workflow, allowing you to organize and visualize the steps involved in the CI/CD process. The **stage** directive has a straightforward structure:


```

pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        // Build steps go here
      }
    }
    stage('Test') {
      steps {
        // Test steps go here
      }
    }
    // Additional stages can be defined here
  }
}

```

In this example, the pipeline has two stages: **Build** and **Test**. Each stage contains its corresponding steps block, where you can define the actual actions to be performed for that stage, such as building the project, running tests, deploying artifacts, and so on.

The stages are executed sequentially, meaning that the pipeline progresses through each stage in the order they are defined. If any stage fails, the pipeline execution is halted, and no further stages are executed. This allows you to identify and isolate issues at specific stages, making it easier to troubleshoot and maintain the pipeline.

The **stage** directive enhances pipeline visualization by providing a clear breakdown of the workflow into manageable chunks. It also facilitates parallel execution of stages using the **parallel** directive, which allows you to run stages concurrently when appropriate, reducing overall pipeline execution time.

Overall, the **stage** directive is a vital component in the Jenkins Declarative Pipeline syntax, enabling you to structure and orchestrate the CI/CD process effectively, ensuring a smooth and organized automation flow.

Tools

In the Jenkins Declarative Pipeline syntax, the **tools** directive is used to specify the tools or software installations required for a specific stage or block of the pipeline. It allows you to define which version of a tool should be used during the pipeline execution. The **tools** directive can be placed at either the pipeline level or within a specific stage. When used at the pipeline level, it applies to all stages in the pipeline. When used within a stage, it applies only to that particular stage and overrides any tool configuration set at the pipeline level. Here's an example of how to use the **tools** directive in the Declarative Pipeline:

```

pipeline {

```

```

agent any
tools {
    maven 'Maven-3.8.1'
    jdk 'JDK-11'
}
stages {
    stage('Build') {
        tools {
            maven 'Maven-3.6.3'
        }
        steps {
            sh 'mvn clean package'
        }
    }
    stage('Test') {
        steps {
            sh 'mvn test'
        }
    }
}
}

```

In this example, we have defined two tools at the pipeline level: **Maven-3.8.1** and **JDK-11**. These tools will be used by default for all stages unless overridden within a specific stage. In the **Build** stage, we use the **tools** directive again to specify a different version of Maven (**Maven-3.6.3**). This overrides the default Maven version defined at the pipeline level, and the specified version will be used only for the steps within the **Build** stage. For the **Test** stage, no specific **tools** directive is provided, so the default Maven version defined at the pipeline level (**Maven-3.8.1**) will be used for the test steps.

By using the **tools** directive, you can ensure that the correct versions of tools or software are used at each stage of the pipeline, making your CI/CD process more reliable and consistent. It also allows you to keep tool configurations separate from the pipeline script, simplifying maintenance and enabling easy updates to tool versions across multiple pipelines. However, when using containers as Jenkins agents, the tools used to run the various steps of your pipeline are built into the containers themselves, rendering the **tools** directive feature useless.

Input

In the Jenkins Declarative Pipeline syntax, the **input** directive is used to pause the pipeline and prompt for user input during its execution. It allows you to add a manual approval step or to interact with users to make decisions before the pipeline continues. The **input** directive has a simple structure:

```

pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                // Build steps go here
            }
        }
    }
}

```

```

    }
    stage('Deploy') {
        steps {
            // Deployment steps go here
        }
        input {
            message "Do you want to proceed with deployment?"
            ok "Yes, proceed!"
            submitter "user1", "user2"
        }
    }
    stage('Test') {
        steps {
            // Test steps go here
        }
    }
}

```

In this example, after the deploy stage's steps have been executed, the pipeline will pause and display a prompt to the user(s) specified by the `submitter` parameter. The user(s) will see the message specified by the `message` parameter and will have the option to either approve or reject the prompt using the button labeled with the text specified in the `ok` parameter. Once the user approves the input, the pipeline will proceed to the next stage (`Test` in this case). If the user rejects the input, the pipeline will be halted at that point.

The `input` directive is useful when you need human intervention to make critical decisions during the pipeline execution, such as approving a production deployment or releasing a new version. It provides a way to introduce manual control points in an automated pipeline, ensuring that specific actions are authorized before they are executed. Again, this feature is more for continuous delivery and should be avoided for CI and continuous deployment.

Keep in mind that using input directives may introduce potential delays in your CI/CD process, as the pipeline will pause and wait for user input. Therefore, it's essential to use this feature judiciously and only when necessary for specific stages that require manual approval or decision-making.

When

In Jenkins Declarative Pipeline syntax, the `when` directive is used to conditionally execute stages or individual steps within a pipeline based on specified conditions. It allows you to define rules that determine when a stage or step should be run, providing flexibility and control over the pipeline's flow.

The `when` directive can be used at both the pipeline level and within individual stages or steps. It supports various conditional expressions and operators to evaluate the conditions. Here's an example of how to use the `when` directive in the Declarative Pipeline:

```

pipeline {
  agent any
  parameters {
    booleanParam(
      name: 'RUN_TESTS',
      defaultValue: true,
      description: 'Choose to run tests'
    )
  }
  stages {
    stage('Build') {
      when {
        branch 'master'
      }
      steps {
        // Build steps go here
      }
    }
    stage('Test') {
      when {
        expression { params.RUN_TESTS == true }
      }
      steps {
        // Test steps go here
      }
    }
  }
}

```

In this example, the **Build** stage will run only if the pipeline is triggered by a commit to the master branch. If the pipeline is triggered by commits to other branches, the **Build** stage will be skipped. Similarly, the **Test** stage will run only when the **RUN_TESTS** parameter is set to **true**. If the parameter is not true, the **Test** stage will be skipped.

The **when** directive supports various conditional expressions, including **branch**, **changeset**, **environment**, and **expression**. These expressions allow you to control the pipeline's behavior based on branch names, changes in the repository, environment variables, and custom evaluations.

By using the **when** directive, you can create a more dynamic and intelligent pipeline that adapts to different scenarios and conditions. This feature is particularly useful when you need to skip certain stages or steps based on specific conditions, optimizing the pipeline's execution and resource usage.

As we conclude our exploration of sections and directives in the Jenkins Declarative Pipeline syntax, it's paramount to underscore their technical significance. The syntax's structured architecture—be it through the encapsulation of stages, steps, or post-actions—enables the precise execution of CI/CD tasks. Moreover, understanding the intricacies of directives, from environment variables to agent specifications, directly impacts resource allocation and job execution paths. Familiarity with these foundational elements ensures that pipeline code not only adheres to best practices but also optimizes system resources and reduces build times. By mastering these facets, developers are better equipped

to tackle complex CI/CD challenges, ensuring that pipelines are robust, efficient, and highly maintainable in a technical ecosystem that demands precision.

Managing complexity using nested stages

In the realm of CI and CD, the Jenkins Declarative Pipeline stands out as a powerful tool for orchestrating complex software delivery processes. One of the distinguishing features that contribute to its flexibility and organization is the concept of nested stages. Nested stages allow developers to further refine and structure their pipeline by breaking down larger stages into smaller, more manageable sub-stages, enhancing the clarity and modularity of the pipeline code.

Nested stages enable developers to hierarchically organize the various steps of a pipeline, providing a clear visual representation of the workflow. This not only enhances readability but also simplifies debugging and troubleshooting by narrowing down the scope of potential issues. The structure becomes particularly valuable as pipelines grow in complexity, involving multiple steps, tests, and deployments. Nested stages can either be sequential, parallel, or both. Let's see some examples and use cases for each scenario.

Working with sequential nested stages

Consider a scenario where you have a comprehensive CI/CD pipeline for a web application.

Traditionally, you might have stages named **Build**, **Test**, **Deploy to Staging**, and **Deploy to Production**. With nested stages, you can further break down these stages into logical components:

```
pipeline {
  agent any
  stages {
    stage('Build and Test') {
      stages {
        stage('Build') {
          steps {
            // Build steps
          }
        }
        stage('Unit Tests') {
          steps {
            // Unit testing steps
          }
        }
      }
    }
    stage('Deploy') {
      stages {
        stage('Deploy to Staging') {
          steps {
            // Deployment to staging steps
          }
        }
        stage('Deploy to Production') {
          steps {
            // Deployment to production steps
          }
        }
      }
    }
  }
}
```

```

    // Deployment to production steps
}
}
}
}
}
```

In this example, the pipeline has two main stages: **Build and Test** and **Deploy**. Each main stage is further divided into nested sub-stages that encapsulate specific tasks. The **Build and Test** stage contains **Build** and **Unit Tests** sub-stages, while the **Deploy** stage consists of **Deploy to Staging** and **Deploy to Production** sub-stages. This hierarchical organization provides a granular view of the pipeline's execution flow.

In conclusion, nested stages in the Jenkins Declarative Pipeline provide a powerful mechanism to structure, visualize, and manage complex CI/CD workflows. By breaking down larger stages into smaller, focused sub-stages, developers can enhance the readability, maintainability, and reusability of their pipeline code. As pipelines become more intricate, nested stages serve as a crucial tool for streamlining the development and delivery process, ultimately contributing to the efficiency and success of software projects.

Working with parallel nested stages

Parallel nested stages allow you to execute multiple sub-stages concurrently within a main stage. This can significantly improve the efficiency of your pipeline by running independent tasks in parallel, reducing overall execution time. Here's an example of how you can use parallel nested stages in Jenkins Declarative Pipeline:

```
pipeline {
    agent any
    stages {
        stage('Build and Test') {
            parallel {
                stage('Build') {
                    steps {
                        echo 'Building.'
                        // Build steps
                    }
                }
                stage('Unit Tests') {
                    steps {
                        echo 'Running unit tests.'
                        // Unit testing steps
                    }
                }
                stage('Integration Tests') {
                    steps {
                        echo 'Running integration tests.'
                        // Integration testing steps
                    }
                }
            }
        }
    }
}
```

```

stage('Deploy') {
    parallel {
        stage('Deploy to Staging-1') {
            steps {
                echo 'Deploying to staging area 1.'
                // Deployment steps
            }
        }
        stage('Deploy to Staging-2') {
            steps {
                echo 'Deploying to staging area 2.'
                // Deployment steps
            }
        }
    }
}
}
}

```

In this example, the pipeline has two main stages: **Build and Test** and **Deploy**. Each main stage contains parallel sub-stages that run concurrently. Within the **Build and Test** stage, the sub-stages **Build**, **Unit Tests**, and **Integration Tests** execute simultaneously, taking advantage of the parallel execution feature. Similarly, within the **Deploy** stage, the sub-stages **Deploy to Staging-1** and **Deploy to Staging-2** run concurrently.

Parallel stages are particularly beneficial when you have tasks that can be executed independently and do not depend on each other's outcome. For example, while building the code, you can simultaneously run unit tests and integration tests, significantly reducing the time required to complete the pipeline.

By using parallel stages, you can optimize the utilization of available resources and achieve faster pipeline execution, leading to quicker feedback and faster software delivery. It's important to design your pipeline carefully to ensure that tasks executed in parallel are truly independent and can run concurrently without causing conflicts or dependencies.

With this, we conclude the vocabulary segment of the Jenkins Declarative Pipeline. The extensive array of sections, directives, and other features elucidated here encapsulates the entirety of its syntax. The Declarative Pipeline syntax's strength lies in its simplicity and maintainability, and while there may be instances where the declarative syntax falls short, the incorporation of the `script` step or harnessing the capabilities of Jenkins' shared libraries in tandem with your Declarative Pipeline can effectively bridge any gaps. In the final chapter of this book, we will delve deeply into harnessing the capabilities of Jenkins' shared libraries. But before we get there, let's explore some of the built-in Jenkins tools that facilitate the creation of pipeline code.

Using built-in tools to construct pipeline code

One of the key strengths of Jenkins is its Declarative Pipeline, a user-friendly approach to defining Pipeline as Code. While the Declarative Pipeline syntax simplifies pipeline creation, the process can still involve understanding and correctly implementing various sections and directives that we saw earlier. To alleviate this challenge, Jenkins introduces a simple tool known as the **Declarative Directive Generator**. Additionally, as development teams strive for efficient and error-free pipeline creation, Jenkins offers another powerful feature known as the **Snippet Generator**. This tool, embedded within the Jenkins user interface, serves as a guiding light, enabling users to generate customized pipeline code snippets effortlessly. In the following section, we will look at both tools, beginning with the Declarative Directive Generator. So, let's get started.

Introducing the Declarative Directive Generator

In Jenkins Declarative Pipeline syntax, sections and directives serve as the building blocks that define the structure and behavior of the pipeline. These sections and directives encompass a range of functionalities, such as specifying agent requirements, defining stages, setting environment variables, and more. While each section and directive brings its unique capabilities, collectively, they form the blueprint of a pipeline's execution.

The Declarative Directive Generator is a web-based utility embedded within the Jenkins user interface. Its primary purpose is to assist pipeline creators, especially those less familiar with the intricacies of the Declarative Pipeline syntax, in generating valid pipeline code by interactively configuring sections and directives. This intuitive tool bridges the gap between pipeline novices and the power of the Jenkins Declarative Pipeline.

To access the Declarative Directive Generator, log in to your Jenkins server and click on the **Pipeline Syntax** option from the list of available options for your pipeline job, as shown:

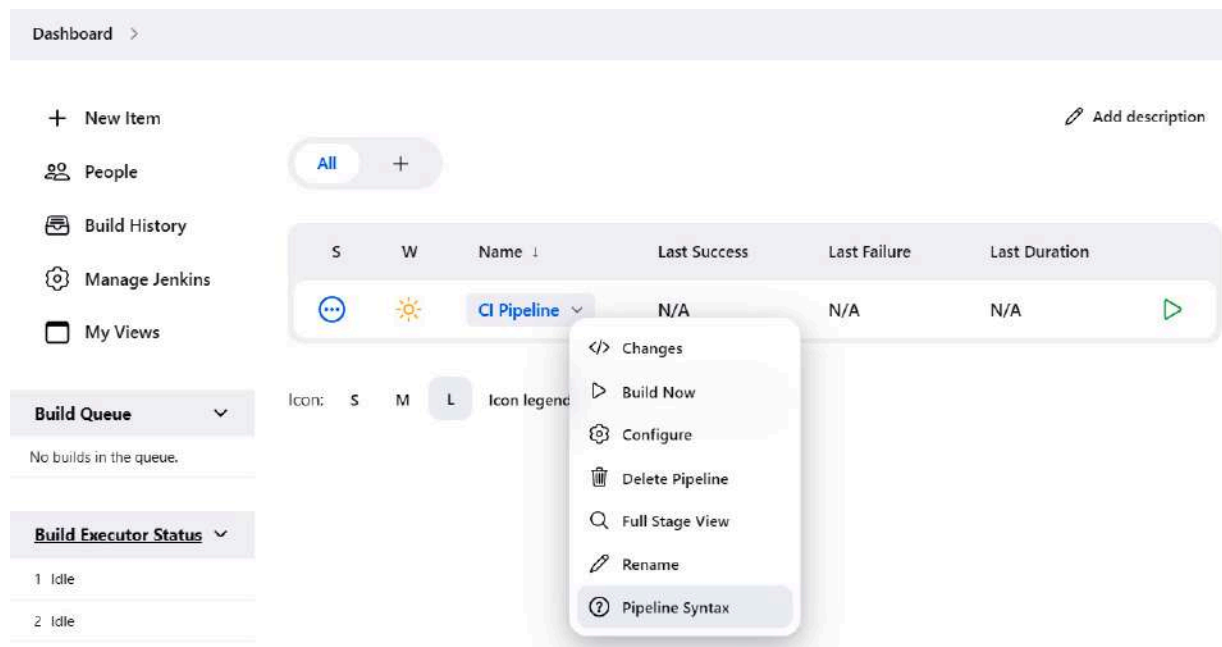


Figure 6.1 – Accessing the Declarative Directive Generator from the Jenkins dashboard

On the resultant page, you will find the generator under the **Declarative Directive Generator** tab. Upon clicking this tab, a user-friendly interface emerges, presenting a series of input fields and options that correspond to different directives, as shown:

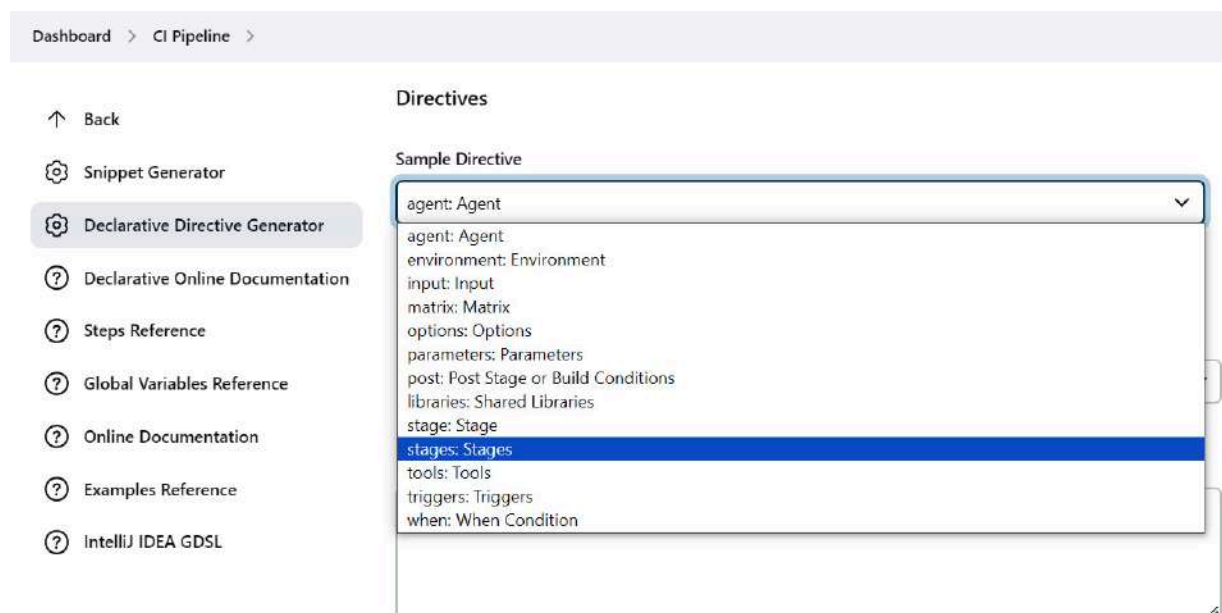


Figure 6.2 – The Declarative Directive Generator page

Let's delve into a scenario where the Declarative Directive Generator proves invaluable. Imagine a development team embarking on a new project with the goal of establishing a robust CI/CD pipeline. While the team possesses diverse skills, not everyone is an expert in the Jenkins Pipeline syntax. This

is precisely where the Declarative Directive Generator shines. The team can start by accessing the generator and specifying basic pipeline settings. They can input the desired agent type, whether it's a specific label, Docker image, or any other supported agent. With this information, the generator dynamically generates the corresponding `agent` directive.

Next, the team can define stages for their pipeline. For instance, they might have stages for code checkout, building, testing, and deployment. The generator allows them to effortlessly add and configure these stages, saving time and ensuring accurate syntax. As each stage is added, the generator populates the pipeline script with the necessary `stage` directives and placeholders for `steps`. In addition to stages, the generator facilitates the configuration of other directives. It offers fields for setting environment variables, specifying post-build actions, and even incorporating parallel execution. These options help the team tailor the pipeline to their project's specific needs without grappling with the intricacies of the underlying syntax.

What sets the Declarative Directive Generator apart is its dynamic nature. As users input their preferences and choices, the generator instantaneously updates the generated pipeline script in real time. This dynamic interaction provides immediate feedback, enabling users to visualize the impact of their selections and make adjustments as needed. Take a look at *Figure 6.3*; it demonstrates using the Declarative Directive Generator:

Dashboard > CI Pipeline >

↑ Back

Snippet Generator

Declarative Directive Generator

Declarative Online Documentation

Steps Reference

Global Variables Reference

Online Documentation

Examples Reference

IntelliJ IDEA GDSL

Directives

Sample Directive

stage: Stage

?
See [the online documentation](#) for more information on the stage directive.

Name ?

Build

This stage will contain... ?

Steps

Add ?

Generate Declarative Directive

```
stage('Build') {
  steps {
    // One or more steps need to be included within the steps block.
  }
}
```

Figure 6.3 – Generating pipeline code using the Declarative Directive Generator

In this figure, we first select the directive that interests us from the **Sample Directive** drop-down list. We are immediately presented with a form that gets displayed right below it. The new form may or may not contain further settings. It all depends on the directive we choose. Once we fill out the options for the chosen directive, clicking on the **Generate Declarative Directive** button will generate a pipeline code in the text field.

The Declarative Directive Generator empowers both newcomers and experienced users to create efficient, error-free, and standardized pipelines. By eliminating potential syntax errors and reducing the learning curve, the generator accelerates pipeline creation and encourages best practices in pipeline development. However, the Declarative Directive Generator cannot generate code for any of the pipeline steps. To do that, you should look at the Snippet Generator utility in Jenkins, which is what we are going to look at next.

Introducing Jenkins Snippet Generator

As we have learned so far, pipeline code defines the stages, tasks, and logic that orchestrate the journey of code from development to deployment. While Jenkins provides a vast library of built-in steps, constructing pipelines that align with specific project requirements can be intricate, especially for users still acquainting themselves with the Jenkins Declarative Pipeline syntax. This is where Snippet Generator steps in, illuminating the path to precise pipeline configuration.

Snippet Generator is accessed in the same way as Declarative Pipeline Generator is. Log in to your Jenkins server and select **Pipeline Syntax** from the list of available options for your pipeline job, as shown in *Figure 6.1* earlier. Snippet Generator serves as a dynamic utility that generates pipeline code snippets based on user-defined criteria. From fundamental steps such as code checkout to advanced tasks such as deploying to Kubernetes, Snippet Generator offers a versatile solution for crafting accurate and tailored pipeline code, as shown:

[↑ Back](#)[Snippet Generator](#)[Declarative Directive Generator](#)[Declarative Online Documentation](#)[Steps Reference](#)[Global Variables Reference](#)[Online Documentation](#)[Examples Reference](#)[IntelliJ IDEA GDSL](#)

Steps

Sample Step

echo: Print Message

archiveArtifacts: Archive the artifacts

bat: Windows Batch Script

build: Build a job

catchError: Catch error and set build result to failure

checkout: Check out from version control

cleanWs: Delete workspace when build is done

deleteDir: Recursively delete the current directory from the workspace

dir: Change current directory

echo: Print Message

emailx: Extended Email

emailxrecipients: Extended Email Recipients

error: Error signal

fileExists: Verify if file exists in workspace

findBuildScans: Find published build scans

fingerprint: Record fingerprints of files to track usage

git: Git

input: Wait for interactive input

isUnix: Checks if running on a Unix-like node

junit: Archive JUnit-formatted test results

library: Load a library on the fly

Figure 6.4 – The Snippet Generator page

And just like the Declarative Directive Generator, the hallmark of Snippet Generator is its real-time interaction and feedback loop. As users make selections and input values, the generator instantaneously reflects the changes in the generated code snippet. This dynamic interaction empowers users to visualize the impact of their choices and adjust on the fly. This not only expedites the process but also enhances users' understanding of how different parameters affect the pipeline's behavior. The steps available to configure through Snippet Generator are proportional to the installed plugins that support Scripted Pipeline or Declarative Pipeline. *Figure 6.5* demonstrates using Snippet Generator:

Dashboard > CI Pipeline > Pipeline Syntax

↑ Back

Snippet Generator

Declarative Directive Generator

Declarative Online Documentation

Steps Reference

Global Variables Reference

Online Documentation

Examples Reference

IntelliJ IDEA GDSDL

Steps

Sample Step

echo: Print Message

echo

Message ?

hello

Generate Pipeline Script

echo 'hello'

Figure 6.5 – Generating pipeline code using Snippet Generator

As depicted in this figure, from the **Sample Step** drop-down list, we first select the desired step that we are interested in. Doing so will result in a form to fill out. The form may or may not contain settings to configure. It all depends on the step you choose. Once you've filled out all the options in the form for the chosen step, clicking on the **Generate Pipeline Script** button will generate a pipeline code in the text field below. If the step does not support Declarative Pipelines, use the generated code inside a `script {}` step.

Snippet Generator opens the door to pipeline customization without the need for deep expertise in Jenkins Pipeline scripting. This proves especially advantageous for development teams comprising members with varying levels of familiarity with the intricacies of Jenkins syntax. By simplifying the process, Snippet Generator transforms pipeline creation into an intuitive and user-friendly experience. Consider a scenario where a development team is setting up a comprehensive CI/CD pipeline for a web application. The pipeline needs to include steps for code checkout, running unit tests, and deploying the application to a staging environment. Using Snippet Generator, team members can swiftly configure these steps by selecting the desired options and parameters. The generator dynamically generates the corresponding pipeline code snippet, effectively removing the guesswork and potential syntax errors.

Novice users can leverage the generator to explore different options, experiment with parameters, and observe the resulting pipeline code. This hands-on learning experience accelerates their familiarity with Jenkins Pipeline syntax, empowering them to confidently contribute to pipeline development.

The power of Snippet Generator extends beyond basic pipeline steps. Jenkins' vibrant ecosystem boasts a multitude of plugins that offer specialized functionalities and integrations. Snippet Generator seamlessly integrates with these plugins, provided they support Pipeline as Code, allowing users to effortlessly incorporate advanced features into their pipelines. For instance, imagine a scenario where a team is integrating Slack notifications into their pipeline. By leveraging Snippet Generator in conjunction with the Slack Notification plugin, the team can swiftly generate the necessary pipeline code to send notifications at key stages of the pipeline's execution. This integration showcases the extensibility and adaptability of Snippet Generator to cater to a wide range of pipeline requirements.

As developers navigate the complex landscape of CI/CD, after a while, they might become accustomed to all the sections and directives and may no longer need to use the Declarative Directive Generator; nevertheless, Snippet Generator emerges as an indispensable companion.

Summary

In the current chapter, we learned about the intricacies of constructing efficient pipeline code using the Jenkins Declarative Pipeline syntax, offering readers a comprehensive understanding of its various aspects. The chapter started with building a vocabulary for pipeline code, emphasizing the significance of a solid foundation in terminologies. This was achieved by carefully understanding the elements of the Declarative Pipeline code and unraveling the core components that compose the pipeline's structure. We learned in detail about sections, exploring how stages, steps, and other constructs work in harmony to achieve seamless automation. Simultaneously, directives exemplified how to navigate complex workflows and employ conditional logic, further enhancing the code's efficiency. We also saw how to address complex pipeline structures using nested stages, facilitating comprehensibility and maintainability. The discussion then advanced to built-in Jenkins tools, highlighting how native Jenkins tools can help you construct your pipeline code.

Having mastered the Jenkins pipeline syntax vocabulary, you're now poised to harness AI in crafting your pipeline code. The next chapter takes an innovative twist, unraveling the power of ChatGPT. In this segment, we unveiled the potential of AI in assisting code generation, particularly ChatGPT's role in producing pipeline code.

Questions

1. In the Jenkins Declarative Pipeline syntax, which of the following statements is true about sections?
 - A. Sections can only be defined globally and cannot be used within individual stages of a pipeline
 - B. Sections are used for defining stages, setting environment variables, and specifying post-build actions
 - C. Sections in a pipeline are exclusively for defining agent configurations and do not support other functionalities

- D. Sections are used for detailing specific actions to be taken when a stage fails, such as retrying or sending notifications
2. In the Jenkins Declarative Pipeline syntax, which of the following is true for directives?
- A. Directives are optional elements that can be used only at the beginning of a pipeline code
 - B. Directives can be used to define environment variables, agent configurations, and post-build actions
 - C. Directives are exclusively used for defining post-build actions and cannot be used for any other purpose
 - D. Directives are used to control the execution flow of scripts within stages, such as programming control including if-else or loops
3. In the Jenkins Declarative Pipeline syntax, which of the following is true for using sequential and parallel nested stages?
- A. Sequential and parallel nested stages cannot be used within the Jenkins Declarative Pipeline
 - B. Parallel stages can only be nested within sequential stages, not the other way around
 - C. Sequential stages can be nested within parallel stages, and parallel stages can also be nested within sequential stages
 - D. Sequential and parallel nested stages are automatically converted to a flat structure by Jenkins
4. Choose whether this statement is true or false: *The Declarative Directive Generator, which is not part of the Jenkins user interface, is mainly aimed at experienced Jenkins users to write complex pipeline scripts without the need to understand the Declarative Pipeline syntax.*
5. Choose whether this statement is true or false: *The Jenkins Snippet Generator can only be used for basic pipeline steps and does not support advanced features provided by Jenkins' multitude of plugins.*

Answers

1. Correct answer: **B**

Explanation of incorrect answer choices:

A. This option is incorrect because sections in the Jenkins Declarative Pipeline syntax are not limited to global definitions. They can also be used within individual stages, allowing for more granular control and customization of pipeline behavior.

C. This option is incorrect, as sections are multifunctional and not restricted to defining agent configurations. They cover a range of functionalities, including but not limited to setting environment variables, defining stages, and managing post-build actions.

D. This option is incorrect because sections are not solely used for handling failures within stages. While they can include error-handling strategies, their primary purpose extends to a broader range of functionalities, including stage definition, environment setup, and post-build actions.

2. Correct answer: **B**

Explanation of incorrect answer choices:

A. This option is incorrect because directives in the Jenkins Declarative Pipeline are not restricted to the beginning of the script. They can be employed throughout the pipeline to apply various configurations and settings.

C. This option is incorrect, as directives are multifunctional in the Jenkins Declarative Pipeline syntax. They are not limited to defining post-build actions; they also include setting up environment variables, configuring agents, and more.

D. This option is incorrect because directives in the Jenkins Declarative Pipeline syntax do not control the execution flow within stages such as programming control structures (e.g., if-else, loops). Control flow within stages is managed by different scripting mechanisms.

3. Correct answer: C

Explanation of incorrect answer choices:

A. This option is incorrect because the Jenkins Declarative Pipeline syntax does allow the use of sequential and parallel nested stages.

B. This option is incorrect because Jenkins allows for both configurations: parallel stages nested within sequential stages and sequential stages nested within parallel stages.

D. This option is incorrect as Jenkins does not automatically convert sequential and parallel nested stages into a flat structure. Jenkins preserves the nested structure as defined in the pipeline script, allowing for the execution of stages in the specified nested manner, which can be crucial for certain build and deployment processes.

4. Correct answer: **False**

Explanation:

The Declarative Directive Generator is indeed a web-based utility that is embedded within the Jenkins user interface. Its primary purpose is to help pipeline creators, especially those who are less familiar with the intricacies of the Declarative Pipeline syntax. It aids in generating valid pipeline code by allowing users to interactively configure sections and directives.

5. Correct answer: **False**

Explanation:

The Jenkins Snippet Generator does indeed extend its capabilities beyond basic pipeline steps. Thanks to Jenkins' vibrant ecosystem, which includes a multitude of plugins offering specialized functionalities and integrations, Snippet Generator can integrate with these plugins, provided they support Pipeline as Code.

Further reading

If you wish to know more about some of the concepts that are not covered in depth in this chapter, or are out of the scope of this book but relevant, we are happy to provide you with some references:

- [1] *Pipeline Steps* reference: <https://www.jenkins.io/doc/pipeline/steps/>
- [2] *Options* directive documentation: <https://www.jenkins.io/doc/book/pipeline/syntax/#options>
- [3] *Triggers* directive documentation: <https://www.jenkins.io/doc/book/pipeline/syntax/#triggers>

Crafting AI-Powered Pipeline Code

In the preceding chapter, we delved deep into Jenkins' Declarative Pipeline syntax, arming ourselves with a comprehensive vocabulary. This was not just an exercise in familiarization but a strategic move to prepare ourselves well for the current chapter. By grasping the nuances of the syntax, we've paved the way for the integration of an exciting collaborator in our coding journey: artificial intelligence.

The realm of AI is vast and diverse, and among its stellar achievements is ChatGPT, a product of OpenAI's groundbreaking research. This model stands out for its ability to understand, process, and produce human-like text. When combined with our Jenkins expertise, ChatGPT becomes a potent ally, making the task of crafting pipeline code not just simpler but also innovative.

Effective collaboration with ChatGPT requires a systematic approach. Therefore, the current chapter introduces the **Describe > Run > Troubleshoot > Refine** methodology. Instead of merely querying the AI for answers, this structured interaction ensures that we engage in a productive dialogue. The outcome? Precise and efficient pipeline code that resonates with our objectives.

As we proceed, it's essential to recognize the practical aspects of using ChatGPT. A hands-on guide will lead you through real-life scenarios, showcasing the seamless fusion of Jenkins' structured environment with ChatGPT's dynamic capabilities. But as we delve deeper into this synergy, understanding the boundaries of AI becomes paramount. Every tool, however powerful, has its limitations. As we explore the vast potential of ChatGPT and other AI tools, we will also shed light on areas where the human touch remains unparalleled.

With a robust foundation laid in the previous chapter, and the promise of AI collaboration in this one, we're at the point of a transformative coding experience. Let's navigate this fascinating intersection of traditional pipeline coding and AI-powered innovation. In the current chapter, you will learn about the following:

- Introduction to AI and ChatGPT
- How to use AI-based tools such as ChatGPT to assist in constructing pipeline code

Technical requirements

To proceed with this chapter, please ensure that you have a Jenkins server up and running, an IDE such as Visual Studio Code on your local machine, and a ChatGPT account. You can create a

ChatGPT account by visiting <https://openai.com/chatgpt>.

Introducing AI, language models, and ChatGPT

In the rapidly evolving landscape of technology, AI has emerged as a transformative force, reshaping the way humans interact with machines and the world around them. AI refers to the simulation of human intelligence by computer systems, enabling these systems to perform tasks that typically require human cognitive functions, such as learning, reasoning, problem-solving, and decision-making. One of the most remarkable manifestations of AI's capabilities is the development of language models, which have revolutionized communication and information processing.

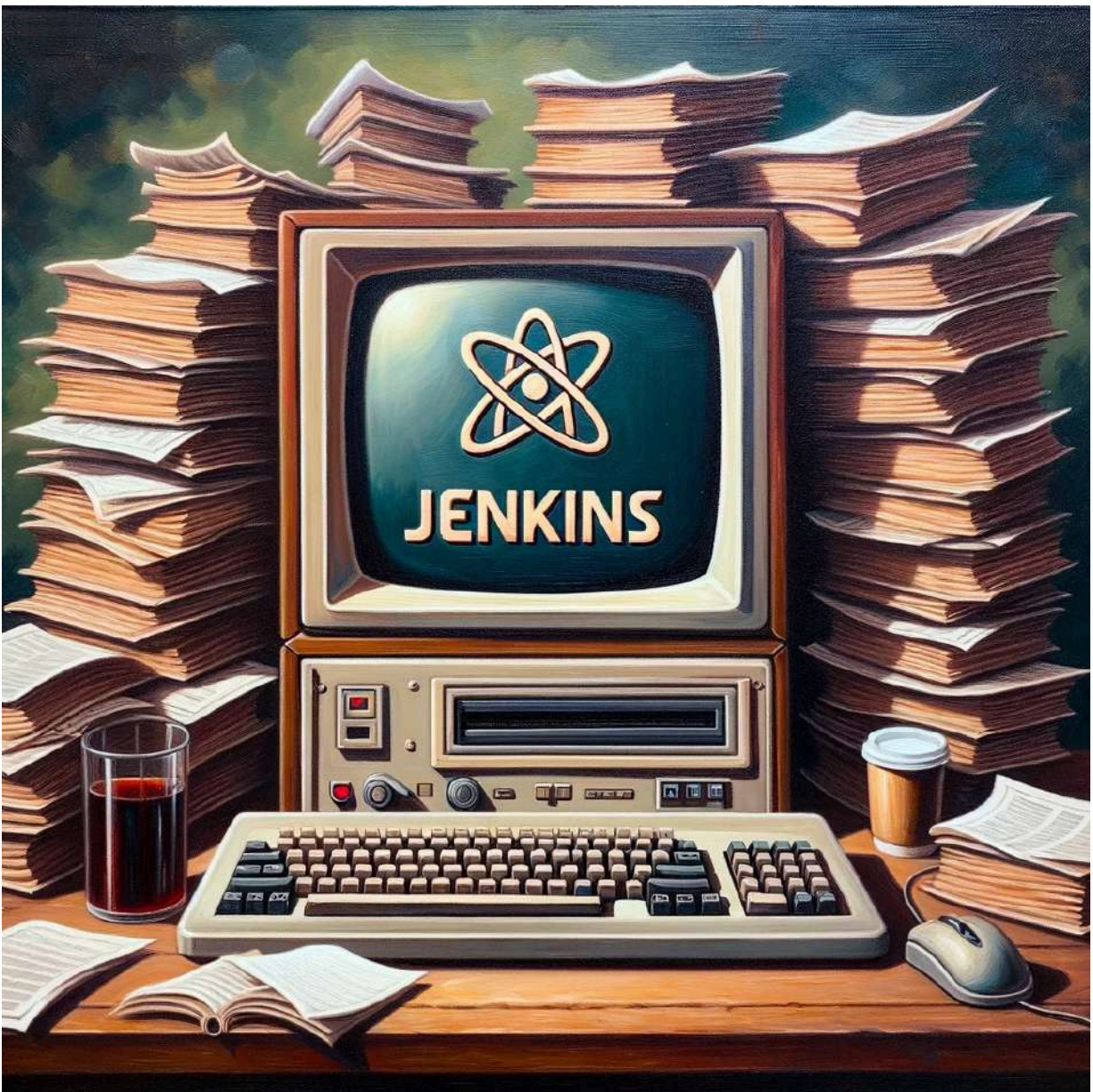


Figure 7.1 – AI-generated (DALL-E 3) oil painting of Jenkins, the CI/CD tool

Language is a fundamental means of communication for humans and replicating this intricate skill in machines has been a longstanding goal of AI research. Language models are a subset of AI technologies that focus on enabling computers to understand, generate, and manipulate human language. These models have undergone remarkable advancements, allowing machines to not only comprehend the meaning of words and sentences but also to generate coherent and contextually relevant text. At the core of modern language models is a concept known as the **transformer architecture**. This architectural paradigm revolutionized the field by introducing attention mechanisms that enable models to weigh the importance of different words in a sentence relative to one another. This breakthrough innovation significantly improved the models' ability to understand and generate human-like text.

Among the pioneering developments in the realm of language models is ChatGPT, a ground-breaking creation by **OpenAI**. ChatGPT is an exemplar of the transformer-based architecture designed for generating human-like text in a conversational manner. It is a variant of the larger GPT (short for **Generative Pre-trained Transformer**) family of models, each building on its predecessors to achieve higher levels of language understanding and generation. ChatGPT's ability to engage in contextually rich and coherent conversations has sparked a revolution in human-AI interaction. By training on a diverse range of internet text, ChatGPT has acquired an uncanny proficiency in mimicking human speech patterns and comprehending the nuances of human language. It can answer questions, offer explanations, generate creative content, and even aid in various domains, all while maintaining a natural conversational flow.

The applications of AI, language models, and ChatGPT are vast and multifaceted. They have found utility across industries ranging from customer service and content creation to healthcare and education. Customer support systems now deploy AI-powered chatbots that can provide instant responses and assistance, enhancing user experiences. Content creators leverage language models to aid in drafting articles, stories, and even code. In healthcare, AI-driven language models help analyze medical literature and patient records, contributing to diagnostic accuracy. However, as with any technological advancement, there are ethical considerations and challenges associated with the deployment of AI, especially in language-related tasks. Issues of bias in training data, potential misinformation propagation, and the risk of generating inappropriate content are some of the concerns that demand careful attention. Striking the right balance between AI capabilities and ethical usage is essential to harness the benefits of these technologies responsibly.

The journey of AI, language models, and ChatGPT is an ongoing one, characterized by continuous innovation and refinement. As research progresses, the capabilities of these systems will undoubtedly expand, leading to even more sophisticated interactions between humans and machines. The

evolution of AI in the context of language underscores the incredible potential for technology to augment human communication and understanding, making information accessible and communication seamless across linguistic and cultural barriers. In conclusion, AI, language models, and ChatGPT represent a paradigm shift in how humans and machines interact. These technologies stand as testament to the remarkable progress AI has achieved, and they hold the promise of reshaping industries, enhancing communication, and fostering new modes of creativity. As we continue to explore the potential of AI, it is imperative to remain mindful of the ethical considerations and implications, ensuring that these technologies contribute positively to our collective progress.

Guidelines for effective interaction with ChatGPT

ChatGPT can assist you in writing Jenkins pipeline code by providing guidance, suggesting syntax, and offering solutions to common problems. Things are quite simple if you know the Jenkins pipeline vocabulary. But if you don't, then the following guideline is here to help you, based on the iterative principle of **describe > run > troubleshoot > refine**, as shown in the following figure:

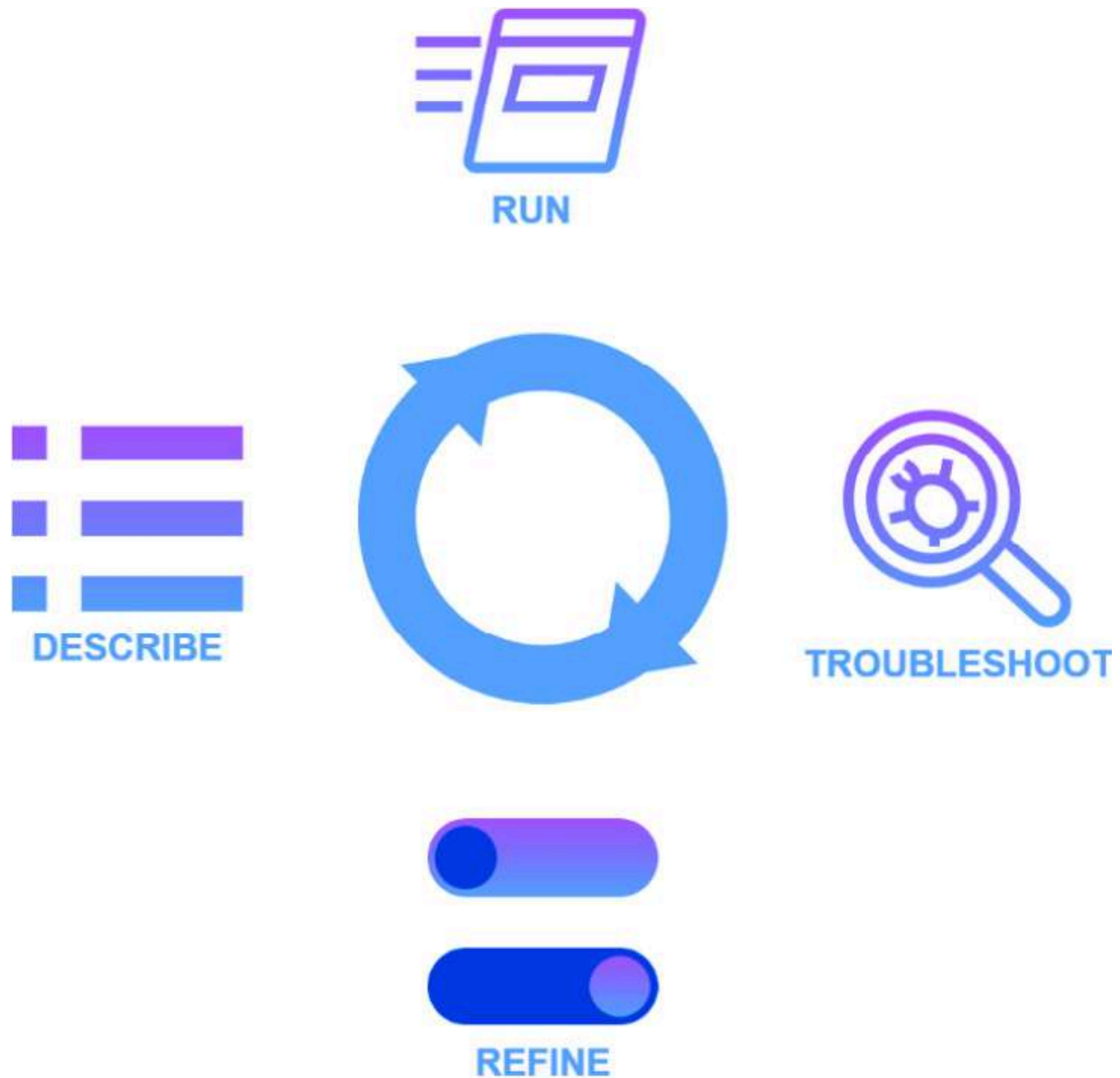


Figure 7.2 – The iterative process of perfecting a pipeline code with ChatGPT

The process in the preceding figure involves a systematic approach of “Describe, Run, Troubleshoot, and Refine.” Initially, the user articulates their project’s requirements, outlining the pipeline’s objectives, stages, integrations, and challenges. ChatGPT responds by generating code snippets and explanations based on the provided description. Upon implementation, the user runs the code in their Jenkins environment, observing its execution and identifying potential issues. In the event of errors or unexpected outcomes, the user turns to ChatGPT for troubleshooting advice, describing the encountered problems. ChatGPT then offers insights, clarifications, and potential solutions to address the issues. Armed with this guidance, the user refines the pipeline code, making the necessary adjustments and improvements. This iterative process continues until the pipeline performs as

intended, showcasing the collaborative and progressive nature of using AI to facilitate efficient pipeline development. Let's take a detailed look at one of the most important stages in the process, which is describing requirements to ChatGPT.

Describing your pipeline requirements

We start by describing our project, the tasks we want to automate using Jenkins, and the specific steps we need in our pipeline. For example, you might say:

“Hi, I’m setting up a Jenkins pipeline for a web application. I need to build the project, run tests, and deploy to a staging server. Can you help me write the Jenkins pipeline code for this?”

You can also start the question by adding some details about the questioner. The following is an example:

“Act like a DevOps engineer with over 15 years of experience working with Jenkins...<Your actual question>.”

And here is a slimmed-down version of what the output looks like:

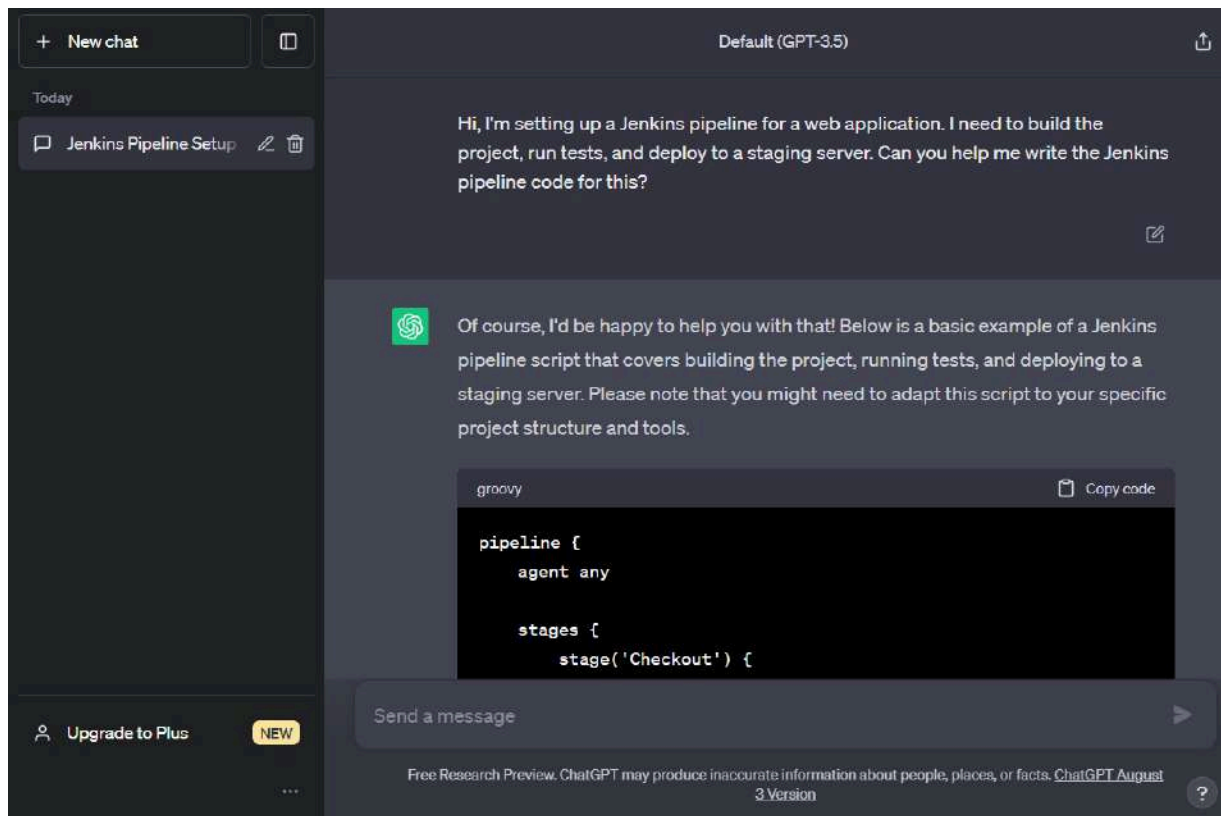


Figure 7.3 – ChatGPT user interface

When you're asking ChatGPT for assistance with writing Jenkins pipeline code, describing your requirements in detail helps the AI understand your project's context and goals. This information enables ChatGPT to provide more accurate and relevant guidance. The following are some crucial points to mention in your description:

- **Provide an overview:** Provide a brief overview of your project. Mention the type of application you're working on (web app, mobile app, etc.) and any relevant details concerning its architecture or technology stack.
- **Define a goal:** Clearly state what you want your Jenkins pipeline to achieve. Are you looking to automate the build process, run tests, deploy to specific environments, or all the above? This gives ChatGPT a clear understanding of the tasks your pipeline needs to handle.
- **Outline steps and stages:** This necessitates familiarity with the Declarative Pipeline syntax. Nevertheless, outline the steps you envision in your pipeline. Break down the process into stages such as "Check out code," "Build," "Test," and "Deploy." Mention any specific requirements or tools you plan to use in each stage.
- **List integration points:** If your project interacts with external tools or services (e.g., version control systems, databases, or deployment servers), mention how these integrations should be handled in the pipeline.
- **List environments and configurations:** If your pipeline needs to adapt to different environments (development, staging, and production), explain how you'd like the pipeline to accommodate these variations.
- **Define error handling:** Describe how you would like errors and failures to be handled within the pipeline. Are there specific notifications or actions you want to take in case of failures?
- **Define testing and validation:** If you have specific testing requirements, provide them. Do you want to run unit tests, integration tests, or other types of tests as part of the pipeline?
- **Define the pipeline trigger:** Indicate whether your pipeline should trigger automatically on code commits or follow a manual trigger. If it's automatic, describe how you want the pipeline to react to different branches or pull requests.
- **Define your continuous process:** Is it just Continuous Integration or Continuous Delivery? Does your pipeline involve deployment? If yes, then explain the deployment process. Is it a Continuous Deployment setup, or are there manual approval steps involved?
- **Describe the tools and technologies:** Mention any specific technologies, programming languages, frameworks, or tools you're using in your project. This helps ChatGPT provide more context-specific guidance.

By providing this level of detail, you're giving ChatGPT the information it needs to generate more accurate code snippets, suggestions, and advice tailored to your project. Remember that the more context you provide, the more effectively ChatGPT can assist you in achieving your desired Jenkins pipeline setup. Here's an example of how you might phrase your request:

"Hi, I'm working on a web application using [programming language] and [build tool]. My project is hosted on [version control system]. Could you please help me create a Jenkins pipeline that performs the following:

- 1. Stage "Scm": Check out the source code from the [version control system].*
- 2. Stage "Build": Build the project using [build tool].*

3. Stage “Unit testing”: Run unit tests and integration tests using [test tool].
4. Stage “SCA”: Run static code analysis using SonarQube. Fail pipeline if SonarQube quality gate check fails.
5. Stage “Publish”: Publish the application to a Artifactory server.
6. All stages must run sequentially.
7. All stages must run on a build agent with the label “Default Kubernetes Agent”.

I’d appreciate a Jenkins pipeline script in the Declarative Pipeline syntax that I can use as a starting point. Thanks!”

Consecutive to the preceding query, you can also ask ChatGPT for guidance on specific aspects of your pipeline code. Consider the following, for example:

“How do I pass my built artifacts from the “Build” stage to the “Publish” stage?”

“How can I add a “CI Stage = Pass” property to all artifacts uploaded to Artifactory?”

“Can you suggest how to publish a code coverage report to SonarQube?”

It is encouraged that you continue your conversations in the same chat session so that ChatGPT understands that all your queries are related. Additionally, request the pipeline structure first (the *sections*), followed by the *directives*, and then the individual *steps* within every stage of your pipeline.

NOTE – VARIED RESPONSES AND CAUTION ON USAGE OF CHATGPT OUTPUT

Please be aware that the code examples provided by ChatGPT are meant to be illustrative and may not always be ready for direct implementation. Users are advised to understand the code before using it, and always validate and test the code in a controlled environment to ensure it works as intended and is free from errors. The response provided may require customization and adaptation to fit the specific requirements and constraints of your project or system. Remember, the use of any code or information provided by ChatGPT is at your own risk. It is always recommended to consult with qualified professionals or the official documentation for critical implementations.

Running your pipeline code

To put your AI-generated and manually stitched pipeline code to the test, first create a pipeline job in Jenkins using the following instructions:

1. Log in to your Jenkins server as an administrator, and from the Jenkins dashboard, click on + **New Item** from the left-hand side menu to create a new Pipeline job.
2. On the resulting page, use the **Enter an item name** field to give your Pipeline job a descriptive name, and then pick **Pipeline** from the options below this field. Then click the **OK** button, as shown in the following screenshot.

Dashboard > All >

Enter an item name

CI Pipeline

» Required field

 **Freestyle project**

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

 **Pipeline**

Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

 **Multi-configuration project**

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

 **Folder**

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

 **Multibranch Pipeline**

Creates a set of Pipeline projects according to detected branches in one SCM repository.

 **Organization Folder**

Creates a set of multibranch project subfolders by scanning for repositories.

OK

Figure 7.4 – Creating a Pipeline job in Jenkins

- Next, on the resultant page, paste your pipeline code into the **Script** field.



IMPORTANT NOTE

The goal here is to test the pipeline code for linting issues and functioning, which is why we chose to put it into the Jenkins job rather than pulling it from a version control system. Any linting error in your pipeline code would immediately become evident at this point. Additionally, you may need to set up additional plugins and configurations inside Jenkins depending on the response you get from ChatGPT.

4. Once done, click on **Apply** and then **Save**.
5. Next, click the **Build Now** option in the left-hand side menu to run the pipeline.

Troubleshooting your pipeline code

If the build fails for some reason, click on the **Console Output** option to look for the error message, as shown in the following screenshot.

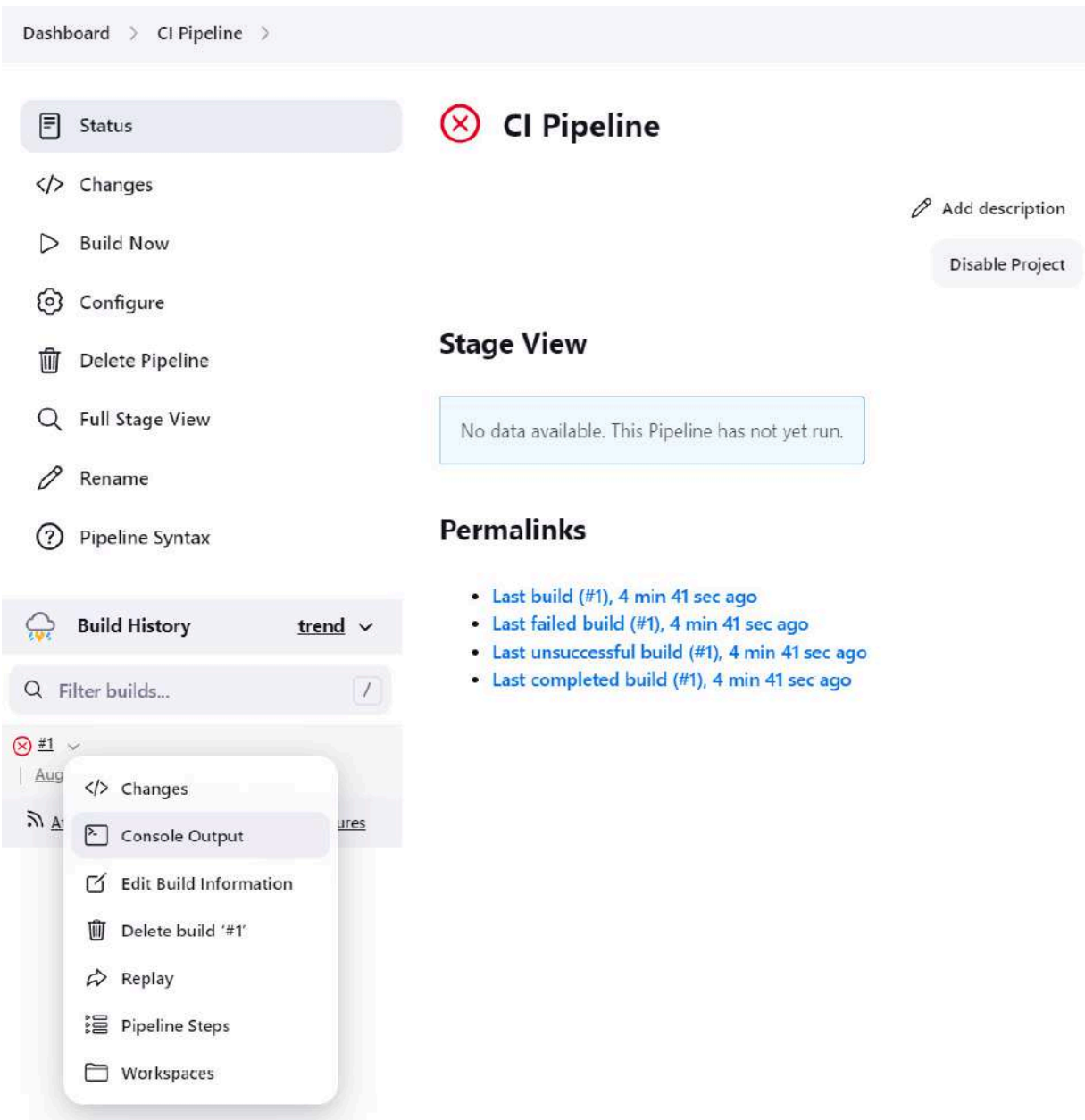


Figure 7.6 – Looking for the build error log for a Pipeline run

Next, describe the problem to ChatGPT using the copied error message. You can continue to use the same chat session that you used to get the pipeline code in the first place. Start by pasting the error followed by a statement and some context. Here are some examples:

I'm getting an error in my Jenkins pipeline. My pipeline is failing at the publish step. Can you help me identify the issue and resolve it? Here is the error message:

"org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed: WorkflowScript: 35: No steps specified for branch @ line 35, column 16.

always {

^”

If your description is good enough, ChatGPT will suggest a fix, or ask for more information.

Refining your code

After receiving suggestions or code snippets from ChatGPT, you can refine and adapt the code to fit your project’s specific requirements. To test your new and improved pipeline code, use the *Replay* option available on the Jenkins Pipeline job page. This feature can be useful for debugging and retesting purposes, without having to modify the original pipeline script saved inside the Pipeline job.

Here’s how the *Replay* option works. When you run a Jenkins Pipeline job, whether it’s manually triggered or scheduled, Jenkins executes the pipeline script according to the defined stages and steps. The original build is logged, and any artifacts or output generated during the build are stored. After the original build has completed, Jenkins provides a *Replay* option. This option allows you to rerun the pipeline build with either the same parameters and configuration as the original run or with modifications. In the case of the latter, before rerunning the pipeline, you have the option to modify the pipeline script or adjust parameters if needed using the **Main Script** field, as shown in the following screenshot.

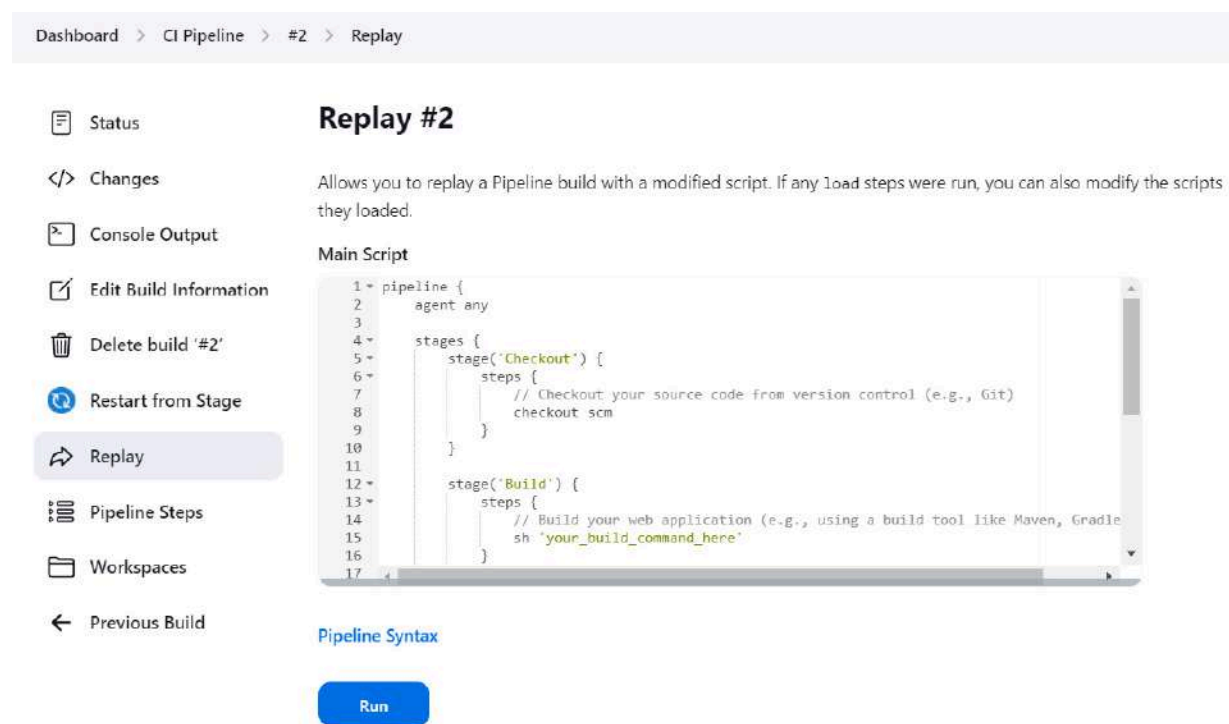


Figure 7.7 – Rerunning the build with modified pipeline code using the Replay option

Any changes made during the replay are only applied to this specific instance and do not affect the original pipeline configuration. The Replay feature is particularly useful for debugging issues that occurred during the original build or for retesting specific scenarios. It allows you to experiment with changes in a controlled manner without affecting the original pipeline setup.

Constructing a pipeline code using ChatGPT

Let us try a real experiment in the following section to harness the power of ChatGPT to build **Continuous Integration (CI)** pipeline code. We will prompt ChatGPT to produce basic pipeline code by giving a scenario and providing insights into the Jenkins environment. We will then iteratively refine the pipeline code as we enquire about individual stages and processes within the pipeline. This collaborative process yields a well-optimized CI pipeline that perfectly corresponds with project objectives and connects easily with platforms such as GitHub, SonarQube, and Artifactory. So let us get started.

The following is what we will ask ChatGPT based on the guidelines we learned earlier:

“Hi, I am working on a web application using the Java programming language and Maven to build, unit test, and package output as a .jar file. Can you create a CI pipeline that performs the following:

- *Stage “Checkout”: Check out the source code from GitHub.*
- *Stage “Build”: Build the project.*
- *Stage “Unit test”: Run unit testing.*
- *Stage “SCA”: Run static code analysis using SonarQube. Use the generic SonarScanner.*
- *Stage “Publish”: Publish the built .jar files to Artifactory.*
- *All stages must run sequentially.*
- *All stages run on a build agent with the label “docker”.*
- *Use Declarative Pipeline syntax.*
- *Pass the built .jar files from the “Build” stage to the “Publish” stage.*

I would appreciate a Jenkins pipeline code. Thanks!”

You should get something like the following output:

```
pipeline {
  agent {
    label "docker"
  }
  stages {
    stage('Checkout') {
      steps {
        checkout scm
      }
    }
  }
}
```

```

stage('Build') {
  steps {
    sh 'mvn clean package'
  }
  post {
    success {
      archiveArtifacts artifacts: '**/target/*.jar', allowEmptyArchive: true
    }
  }
}
stage('Unit test') {
  steps {
    sh 'mvn test'
  }
}
stage('SCA') {
  steps {
    sh 'sonar-scanner'
  }
}
stage('Publish') {
  steps {
    sh 'curl -u <your-artifactory-credentials> -T **/target/*.jar "your-artifactory-url"'
  }
}
}
}

```

The generated output is undeniably impressive, evoking a sense of awe at ChatGPT's ability to deliver sophisticated pipeline code. Nonetheless, there remains room for enhancement. Based on our observations, here are some areas that need improvement:

- The inclusion of the **Checkout** stage section is redundant, as we intend to store our pipeline code within the project's **Jenkinsfile**, coexisting harmoniously with the source code in a **Version Control System (VCS)**. This approach ensures a streamlined and comprehensive management of both code and pipeline configuration within the same repository, simplifying versioning, tracking, and collaborative development processes.
- The preceding pipeline code implementation features a **post** section that employs the **archiveArtifacts** step to retain the built **.jar** files following the **Build** stage, facilitating their utilization in the subsequent **Publish** stage for Artifactory upload. However, this approach essentially stores files outside the immediate pipeline run, resembling an internal Artifactory server within Jenkins, which may not align with the intended workflow. An alternative and more efficient approach involves the utilization of Jenkins' *stash* and *unstash* steps, designed specifically for passing files seamlessly between different stages of the pipeline. This not only enhances the pipeline's performance but also maintains a cleaner and more targeted transfer of artifacts.
- The pipeline code assumes the presence of the *SonarScanner* CLI on our build agent, which is evident from the *SCA* stage that utilizes a **sh** step to directly invoke the **sonar-scanner** command. However, this assumption doesn't align with our intended approach. Ideally, we should guide ChatGPT to integrate the SonarQube plugin, leveraging its dedicated Declarative Pipeline step to fulfill the same objective. This approach ensures a seamless and cohesive integration with our chosen tools, resulting in a more optimized and effective pipeline.
- The problem with ChatGPT making assumptions about our Jenkins setup is evident in the *Publish* stage too. Initially, the **.jar** files from the *Build* stage are not retrieved for use. Additionally, ChatGPT operates under the presumption that an Artifactory plugin is absent, leading to the utilization of a **curl** command to upload artifacts to the Artifactory server. To rectify this, it is essential to instruct ChatGPT to incorporate the dedicated Declarative Pipeline step embedded in the Artifactory plugin. This revised approach ensures the seamless integration of the pipeline with Artifactory, promoting efficiency and adherence to best practices within our environment.

Let's enhance our pipeline code by providing ChatGPT with a deeper understanding of our environment and requirements. Try the following query using the same chat session:

"The pipeline code looks good. However, I want you to make changes to it based on the following suggestions:

- *The pipeline will be triggered using GitHub webhooks.*
- *We will save our pipeline code in the VCS along with the source code.*
- *Use the "stash" and "unstash" step to pass files between the "Build" and the "Publish" stages.*
- *We have the GitHub, SonarQube, and Artifactory plugins installed on our Jenkins server. Use the Declarative Pipeline steps that come with these plugins.*
- *Use the "withSonarQubeEnv ()" step for static code analysis in the SCA stage.*
- *Use the "rtUpload ()" step that comes with the Artifactory plugin to upload artifacts to Artifactory.*

Thanks."

Now, you can anticipate a comparable response. Please be aware that ChatGPT may generate different responses to the same question for different users. This variability is due to several factors:

- **Dynamic learning:** ChatGPT continuously learns from interactions and evolves its understanding, which can lead to variations in its responses over time.
- **Personalization:** The model attempts to tailor responses based on the context and nuances of each individual query, which can result in different answers for seemingly similar questions.
- **Randomness in AI processing:** There is an inherent randomness in the way AI models such as ChatGPT process and generate language, contributing to the uniqueness of each response.
- **Updates and improvements:** OpenAI regularly updates the model, improving its capabilities and knowledge base. Responses may change as the model gets updated.

As a result, users may experience varied responses from ChatGPT for the same question. This is a characteristic feature of advanced AI language models and is indicative of their dynamic and evolving nature. The following is an example response:

```
pipeline {
  agent {
    label "docker"
  }
  stages {
    stage('Checkout') {
      steps {
        checkout scm
      }
    }
    stage('Build') {
      steps {
        script {
          stash name: 'builtJars', includes: '**/target/*.jar'
          sh 'mvn clean package'
        }
      }
    }
  }
}
```



```

    }
  }
  stage('Unit test') {
    steps {
      sh 'mvn test'
    }
  }
  stage('SCA') {
    steps {
      script {
        withSonarQubeEnv('Your_SonarQube_Server') {
          sh 'sonar-scanner'
        }
      }
    }
  }
  stage('Publish') {
    steps {
      script {
        rtUpload (
          serverId: 'Your_Artifactory_Server',
          spec: '**/target/*.jar',
          buildName: "${env.JOB_NAME}",
          buildNumber: "${env.BUILD_NUMBER}"
        )
      }
    }
  }
}

```

As you can see from the preceding code, the level of accuracy has significantly improved, yet there remains room for further refinement. To achieve an even more precise outcome, you can proceed by engaging in specific inquiries about individual steps with ChatGPT. This iterative process promises to yield an increasingly optimized and tailored pipeline configuration.

In this hands-on practical exercise, ChatGPT demonstrated its ability to assist with the creation of a streamlined CI pipeline code. Initially, ChatGPT was presented with descriptions of various stages and tasks within a pipeline scenario. Remarkably, it promptly generated an initial pipeline code, showcasing its ability to comprehend and act on given instructions. However, the exercise didn't end there. To emphasize the power of iterative interactions, a consecutive query was sent to ChatGPT. This query aimed to refine the initially generated code, improving its efficiency and alignment with the project's specific needs. While the example deliberately remained simple to facilitate learning, it highlights the potential of ChatGPT to tackle much more intricate and sophisticated operations. With that in mind, let's delve into the realm of the current limitations within AI technology, especially when employing ChatGPT to craft Jenkins pipeline code.

Understanding the limitations of ChatGPT

While ChatGPT has outstanding capabilities, it is vital to recognize its limitations and take them into consideration. One critical component is its ability to generate incorrect or inaccurate information, particularly when given unclear queries or in the absence of context. Furthermore, ChatGPT may occasionally generate responses that appear plausible but are factually inaccurate. It can also be sensitive to query phrasing, producing different replies to differently rephrased queries. Furthermore, because it learns from a wide spectrum of internet text, the algorithm may generate biased or improper content inadvertently. These constraints highlight the importance of users rigorously evaluating and verifying results, especially in crucial applications where precision is critical. In the realm of code generation, including pipeline code, ChatGPT possesses a few notable limitations:

- While it can produce functional code snippets, it may not consistently adhere to best practices, leading to code that might work but isn't necessarily optimal or secure.
- ChatGPT's responses might lack a deep understanding of the specific project requirements, potentially resulting in code that doesn't align perfectly with the intended functionality.
- The system's inability to grasp nuances in coding standards or architectural patterns could lead to code that requires additional modifications.
- Jenkins (along with its collection of plugins) and other open source DevOps tools operate within the open source realm and undergo frequent updates. Consequently, the Declarative Pipeline steps associated with Jenkins plugins can evolve over time, potentially leading to instances where ChatGPT lacks awareness of these modifications.

Users must exercise caution and thoroughly review the generated code, ensuring it meets industry standards and aligns precisely with the project's goals. While ChatGPT serves as a valuable aid, it's not a replacement for skilled developers and the meticulous scrutiny that software development demands.

ChatGPT continually learns from user feedback and interactions. OpenAI refines its models to enhance accuracy and offer more coherent, relevant responses. While the default (GPT-3.5) is good enough, we recommend upgrading to a subscription plan to harness the maximum benefits of ChatGPT for Jenkins pipeline code creation and beyond. Subscriptions grant access to premium features, allowing users to tap into the most up-to-date and refined AI capabilities. Upgrading to newer versions ensures access to the latest enhancements, optimized for accuracy and relevance.

Summary

In this chapter, we explored the vast capabilities of AI, particularly focusing on ChatGPT's contributions to generating Jenkins Declarative Pipeline code. This was complemented with a set of guidelines to streamline the use of ChatGPT, offering you actionable tactics to maximize AI interactions, enhance precision, and tailor the output to specific needs. As we delved deeper, the chapter highlighted ChatGPT's inherent limitations. While its feats are impressive, ChatGPT isn't

perfect and can sometimes yield less-than-ideal or inaccurate results. Such insights stress the importance of human review and validation.

Thanks to this chapter and the previous one, by enhancing our vocabulary, mastering Jenkins' built-in tools to construct pipeline code, harnessing the prowess of ChatGPT, and understanding its constraints, we have become equipped for an insightful journey into crafting top-notch Jenkins pipeline code. In the next chapter, we'll employ all the knowledge gathered from this book so far to design a holistic end-to-end CI pipeline. Teaming up with practical example source code, we'll depict a comprehensive CI pipeline, capturing every phase of its operation.

Questions

1. Which of the following best describes the most important stage in the systematic approach of Describe, Run, Troubleshoot, and Refine when using AI assistance for Jenkins pipeline development?
 - A. Running the code in the Jenkins environment to observe its execution
 - B. Troubleshooting with ChatGPT to address errors or unexpected outcomes
 - C. The initial stage of articulating the project's requirements and outlining the pipeline's objectives to ChatGPT
 - D. Refining the pipeline code based on the guidance from ChatGPT
2. Choose whether this statement is true or false: The code examples provided by ChatGPT are always ready for direct implementation without any need for further understanding, validation, or customization.
3. In a Jenkins Pipeline, what does the **Replay** option allow users to do?
 - A. The **Replay** option allows users to run a previously successful build without any changes
 - B. The **Replay** option enables users to edit and rerun a specific pipeline script from a previous build
 - C. The **Replay** option is used to automatically trigger a new build whenever a previous build fails
 - D. The **Replay** option allows users to compare the results of two different builds in a pipeline
4. How important is it to have knowledge of Jenkins Declarative Pipeline syntax when using ChatGPT to write pipeline code?
 - A. Not important at all, as ChatGPT completely eliminates the need for any understanding of Jenkins Declarative Pipeline syntax
 - B. Somewhat important, as a basic understanding can help you to communicate your requirements more effectively to ChatGPT
 - C. Very important, as a deep understanding of Jenkins Declarative Pipeline syntax is necessary to use ChatGPT effectively for writing pipeline code
 - D. Irrelevant, as ChatGPT is designed to work independently of the user's knowledge of Jenkins Declarative Pipeline syntax
5. Choose whether this statement is true or false: A **Large Language Model (LLM)** such as ChatGPT can understand and process human languages in the same way that a human brain does.

Answers

1. Correct answer: **C**

Explanation of incorrect answer choices:

A. While running the code is a crucial part of the process, it is not the most important stage. The effectiveness of this step largely depends on the accuracy and thoroughness of the initial description of the project's requirements.

B. Troubleshooting is essential when errors occur, but it is a reactive step.

D. Refining the pipeline code is a key part of the iterative process, but it builds upon the initial description of the project's requirements. Without a clear and comprehensive initial description, the refinement process may not be as effective.

2. Correct answer: **False**

Explanation:

Code examples provided by ChatGPT are meant to be illustrative and might not always be ready for direct implementation. Users are strongly advised to make sure they understand the code, validate and test it, customize and adapt it, and always consult professionals and the official documentation if in doubt.

3. Correct answer: **B**

Explanation of incorrect answer choices:

A. This option is incorrect because the **Replay** option is not about rerunning a successful build as-is. It specifically allows for editing the script of a previous build and then rerunning it, which is different from just re-executing a successful build without changes.

C. This option is incorrect as the **Replay** option does not automatically trigger a new build upon failure.

D. This option is incorrect because the **Replay** option is not used for comparing build results. Jenkins has other mechanisms and plugins for comparing build results.

4. Correct Answer: **B**

Explanation of incorrect answer choices:

A. This option is incorrect because while ChatGPT can assist in writing pipeline code, having some level of understanding of Jenkins Declarative Pipeline syntax is beneficial. It helps in better articulating the requirements and understanding the code provided by ChatGPT.

C. This option is an overstatement. While having a deep understanding is beneficial, it is not necessary to use ChatGPT effectively. Users with a basic-to-intermediate level of knowledge can still make good use of ChatGPT for writing pipeline code.

D. This option is incorrect as the user's knowledge of Jenkins Declarative Pipeline syntax is relevant. While ChatGPT can generate code independently, the user's understanding aids in customizing, validating, and effectively implementing the code.

5. Correct answer: **False**

Explanation:

Large Language Models (LLMs) such as ChatGPT process and generate language based on statistical patterns and probabilities learned from vast amounts of text data. Unlike the human brain, which understands language through complex cognitive processes involving consciousness, context, and emotional intelligence, LLMs do not possess actual understanding or consciousness. They mimic language comprehension and generation by predicting the most likely next word or sentence based on their training.

Setting the Stage for Writing Your First CI Pipeline

In this chapter, we embark on a comprehensive journey to design and set the stage for our very first **Continuous Integration (CI)** pipeline using Jenkins. We will start the chapter by delving into the conceptual realm of planning for CI and discerning the high-level design of CI processes. To do that accurately, it's pivotal to first grasp the software project and its architecture before diving into the CI process. Subsequently, we'll walk through the practical steps involved in configuring various tools and platforms, such as SonarQube for code quality and Artifactory as a Docker repository.

Each of these platforms plays a crucial role in our CI pipeline, and by the end of our journey, you will have learned how to dissect a software project for CI implementation, understand its architecture and source code intricacies, and be able to create a high-level CI design. We will discuss setting up webhooks on repositories to trigger CI processes in real time upon code changes. You will also be able to incorporate SonarQube for static code analysis and understand the importance of quality gates. You will also know how to use Artifactory as a Docker repository for containerized applications.

As such, we will cover the following main topics in this chapter:

- Understanding the software project for CI
- Understanding the high-level CI design
- Setting up a webhook on a GitHub repository
- Configuring a project inside SonarQube and Artifactory
- Creating a Kubernetes Secret on AKS

Technical requirements

Before diving into this chapter, it's crucial that you've diligently followed the instructions in *Chapters 3, 4, and 5*. By now, you should be able to meet the following requirements:

- Have a fully functional Jenkins server, a SonarQube server, and an Artifactory server
- Ensure that your Jenkins server is integrated with an AKS cluster, allowing it to dynamically deploy Jenkins agents as needed
- Possess an active GitHub account
- Have GitHub, SonarQube, and Artifactory seamlessly connected to Jenkins through the appropriate plugins

Additionally, you need the following tools installed on your local development environment:

- An IDE such as Visual Studio Code (or any of your choice)
- Git
- The Azure CLI, to interact with the Azure cloud
- kubectl, the Kubernetes command-line tool

You can download the source code of this chapter from https://github.com/PacktPublishing/Learning-Continuous-Integration-with-Jenkins_Third-Edition.

Understanding the software project for CI

In this section, we'll begin by exploring an example software project written in JavaScript, exploring its architecture and code. We are going to use this example software project to perform CI. Following that, we'll outline a high-level CI plan. Finally, we'll configure settings in GitHub, SonarQube, Artifactory, and our AKS cluster, before we start writing the CI pipeline code. So, let's start by diving into the software project.

To practically demonstrate writing and executing a CI pipeline with Jenkins, we will use a simple Hello World web application that displays `HELLO WORLD` in 20 different languages. Our web application is a three-tier, container-based solution designed to showcase a simple interaction between a frontend, a backend, and a MongoDB database. While our primary goal in the following section isn't to instruct you in coding, it's still beneficial to understand the web application's architecture and functionality. Let's start with the architecture.

Understanding the architecture

The Hello World web application is a demonstration of a modern three-tier web application that is structured in modules and containerized. The following diagram depicts its high-level architectural design.

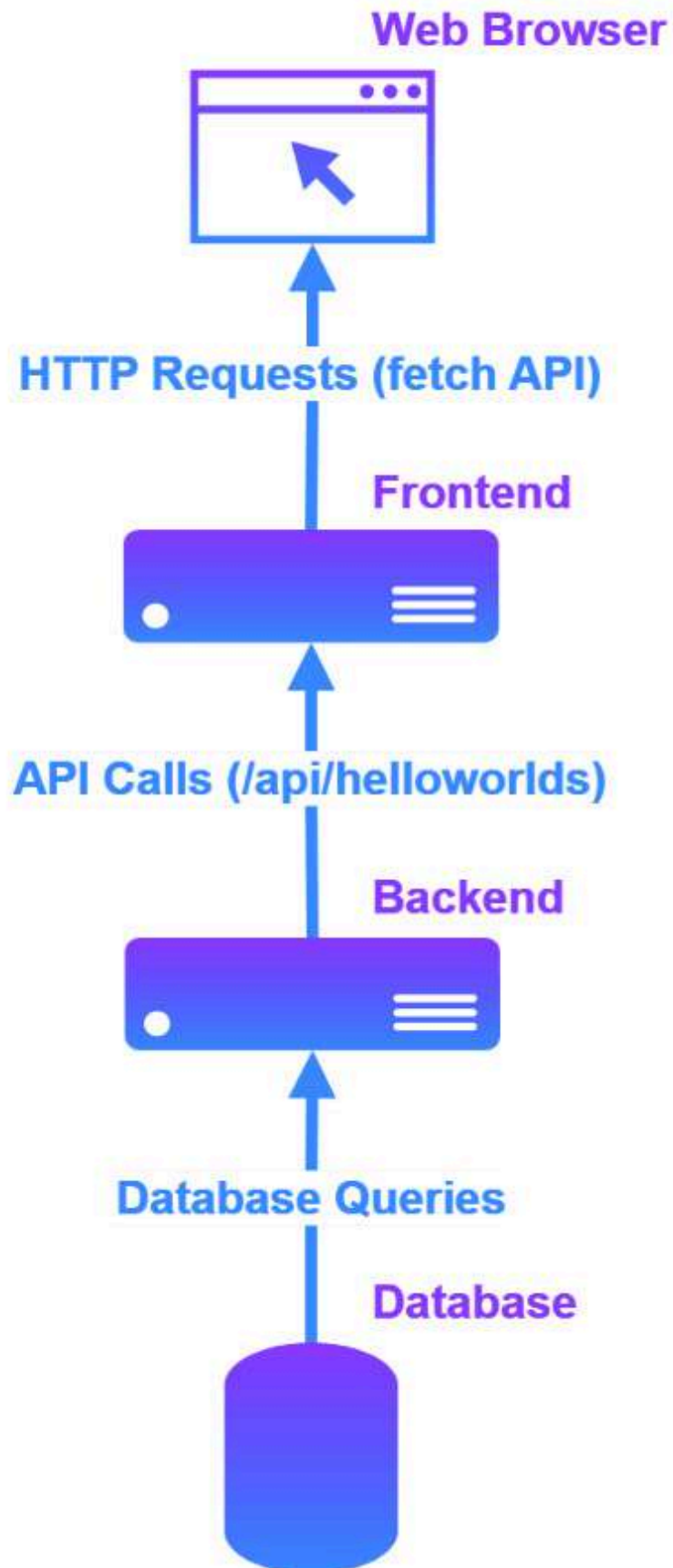


Figure 8.1: High-level architectural design of the web application

Let's delve into the intricacies of the preceding design, where we'll explore the **frontend**, **backend**, and **database architectures**, understanding their roles and how they interact within the system. I also suggest you take a high-level glance at the source code. Notice that each tier (**frontend**, **backend**, and **database**) is containerized using Docker. This means each part of the application runs in an isolated environment with its dependencies, ensuring consistency across different deployment environments:

- **The frontend layer:**
 - **Simple UI:** The UI is a basic web page written in HTML5 that fetches and displays greetings from the backend.
 - **Dynamic data retrieval:** Uses vanilla JavaScript (fetch API) to retrieve and display data dynamically from the backend API.
 - **Styling:** The appearance is controlled by a simple **Cascading Style Sheets (CSS)** file, ensuring a clean and responsive design.
 - **Docker containerization:** The frontend is packaged within a Docker container, ensuring a consistent environment for running the server, irrespective of where it's deployed.
- **The backend layer:**
 - **Node.js server:** Built with Express.js, a popular framework for building web applications in Node.js
 - **API endpoint:** A single API endpoint (`/api/helloworlds`) retrieves greetings from a MongoDB database
 - **MongoDB integration:** Uses the official MongoDB Node.js driver, allowing seamless integration and data retrieval
 - **Docker containerization:** Like the frontend, the backend is also encapsulated within its own Docker container
- **The database layer:**
 - **MongoDB:** The choice of database for this application. It's a NoSQL database, ideal for storing JSON-like documents.
 - **Initialization script:** The `init-mongo.js` script serves to set up initial collections, seed data, or establish necessary indexes, ensuring the database is primed and ready for the application's needs.
 - **Docker containerization:** There is no Dockerfile for the database. It is spun during the deployment phase of the application using either a Docker Compose file or a Helm chart depending on where you deploy your application.
- **Communication:**
 - The frontend communicates with the backend over HTTP/HTTPS, typically using AJAX or Fetch for API calls
 - The backend communicates with the database using a database driver or **Object Relational Mapping (ORM)**
- **Setup and deployment:** The application's structure, combined with Docker, allows for a straightforward setup and deployment. The Dockerfiles in both the frontend and backend directories detail the environment setup, the dependencies, and the entry point for their respective components. With tools such as Docker Compose and Kubernetes, you can orchestrate the deployment of the entire application stack, ensuring each tier communicates seamlessly with one another. The application can be deployed using

container orchestration tools such as Docker Compose, Kubernetes, or cloud-specific services such as Microsoft's AKS, AWS ECS, or **Google Kubernetes Engine (GKE)**.

This was everything about the design of our hypothetical web application; now, let's have a look at the source code, which is short enough to be discussed.

Understanding the source code

Whether you're a seasoned developer, a curious newcomer, or someone in between, this section will shed light on the structures and conventions that shape the way our web application functions. Let's gain insights into all the code files one by one.

At the root of the repository, you will find the following files:

- **.gitignore**: This file lists all the patterns of files and directories that Git should ignore and not track. Among other things, it is important to note that it ignores the **test-output** folder, which typically contains code coverage and unit testing results. Ignoring this folder helps keep generated reports and coverage data out of the Git history since they can be regenerated as needed.
- **LICENSE.md**: This file contains the licensing information for the project. Open source projects often include a license to specify how the software can be used, modified, and distributed by others.
- **README.md**: This file provides documentation for the project. It often includes a description of the project, setup instructions, usage examples, and other pertinent information for developers and users.

Understanding the backend code

The code for the backend is placed inside a folder named **backend**, and it has the files that we are going to explore in the following subsections.

server.js

The **server.js** file is the core of the backend tier. Based on typical Node.js applications, it will initialize a web server (often using libraries such as Express.js), define API routes that the frontend can call, handle any business logic, such as processing data, and establish a connection to a database (MongoDB) for data storage and retrieval. The following is part of **server.js**. Refer to the source code repository for [Chapter 8](#) to see the full code:

hello-world/backend/server.js (chopped-down version)

```
const express = require('express');
...
app.get('/api/helloworlds', async (req, res) => {
  ...
});
module.exports = app;
if (require.main === module) {
  ...
}
```

Let's understand the preceding code in a structural and cohesive way. The file establishes a backend server for an application, leveraging the Express.js framework—one of Node.js' most prominent libraries for web development. This server's primary goal is to interact with a MongoDB database to retrieve greeting messages. Initially, the script loads essential dependencies. Among them are `express` for server operations, `mongodb` to facilitate connections to the MongoDB database, and `cors` for handling cross-origin requests, ensuring that the frontend can communicate with this backend irrespective of where they're hosted.

With the foundations set, the server is configured to listen on port 3000. The MongoDB database connection string, labeled `DB_URI`, is specifically tailored to connect to a database named `helloworldapp`. Middleware configurations follow suit. By enabling **Cross-Origin Resource Sharing (CORS)**, the server can gracefully accept requests from different origins. Furthermore, with the `express.json()` middleware, it's primed to decode JSON payloads that might come with incoming requests.

A crucial aspect of this server is the `GET /api/helloworlds` endpoint. When triggered, it establishes a connection to the MongoDB database, zeroes in on the `greetings` collection, and fetches all its entries. These greetings, once retrieved, are relayed to the requester in a structured JSON format. Following this, the database connection is promptly closed to maintain optimal performance.

For testing purposes and modular development, the Express `app` instance is exported. However, when this script is the primary module being executed, the server springs to life, listening diligently on port 3000 and signaling its operational status via a console log. In essence, this backend script embodies a bridge, connecting the user's requests for greetings and the rich database reservoir where these greetings reside.

package.json

The `package.json` file is the manifest file for Node.js applications. It lists dependencies (libraries and frameworks) that the backend needs, as well as possibly defining scripts such as starting the server or running tests. The following is what `package.json` looks like:

hello-world/backend/package.json (chopped-down version)

```
{
  "name": "hello-world",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "test": "..."
  },
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.17.1",
```

```

    "mongodb": "^3.6.6"
  },
  "devDependencies": {
    "mocha": "^10.2.0",
    ...
  },
  "nyc": {
    "reporter": ["lcov", "text-summary"]
  },
  "author": "Nikhil Pathania",
  "license": "MIT"
}

```

Let's break down the preceding code. This file serves as the blueprint for the Hello World Node.js project, detailing its dependencies, scripts, and other vital configurations. Its **metadata** section encapsulates core information such as the project's name, its primary entry point via **server.js**, the version, and details about its author and the license under which it's distributed.

Diving into the **scripts** section, it defines the tasks essential for the project's life cycle. The **start** command swiftly launches the application, while the **test** command integrates the power of **mocha** and **nyc** to run unit tests and generate coverage reports, respectively.

In the realm of dependencies, the backbone of any Node.js application, three primary libraries stand out: **cors**, facilitating cross-origin requests; **express**, the beating heart of our backend server; and **mongodb**, bridging the gap between the app and MongoDB databases. Meanwhile, the development environment is buttressed by an array of **devDependencies**, such as the testing trio of **mocha**, **chai**, and **sinon**, alongside others that aid in report generation and test assertions.

Lastly, a dedicated configuration for **nyc** is present, dictating the format for the generated test coverage reports, ensuring developers have a comprehensive view of code coverage metrics.

All in all, **package.json** delineates the skeleton and sinews of the Hello World project, providing clarity on its structure, dependencies, and developmental procedures.

Dockerfile

The Dockerfile defines the environment setup for the backend. It's responsible for creating an image that can run the server-side logic. Often, for Node.js applications, this will involve setting up a Node.js environment, copying over the source code, installing dependencies, and specifying the entry point (such as **server.js**):

hello-world/backend/Dockerfile

```

FROM node:14
COPY ./backend/package*.json ./
RUN npm install
COPY ./backend ./
npm start
CMD ["npm", "start"]

```

The preceding Dockerfile is designed to containerize a Node.js application, which is our backend server. It starts by using the official Node.js version 14 image. Within the container, it sets `/app` as the main working directory. From the host machine, it then copies the application's dependency files (`package.json` and `package-lock.json`, if available) into the container and installs these dependencies using `npm install`. After that, it copies the remaining application files from the `backend` directory into the container. When the container is initiated, it will automatically run the Node.js application using the `npm start` command.

Now, let's move on to the frontend.

Understanding the frontend code

The code for the frontend is placed inside a folder named `frontend`, and it has the files explained in the following subsections.

`app.js`

This JavaScript file contains the logic for the frontend. It's responsible for making API calls to the backend, manipulating the DOM based on user interactions or data received from the backend, and handling any client-side logic, such as form validations or animations. The following is the complete `app.js` code:

hello-world/frontend/app.js

```
fetch('http://localhost:3000/api/helloworlds')
  .then(response => response.json())
  .then(data => {
    const greetingsDiv = document.getElementById('greetings');
    data.forEach(greeting => {
      greetingsDiv.innerHTML += `<p>${greeting.language}: ${greeting.message}</p>`;
    });
  })
  .catch(error => console.error('Error:', error));
```

Let us understand the preceding code. This code is designed to fetch and display greeting messages on a web page. When executed, it makes a network request to the URL

`http://localhost:3000/api/helloworlds`. Once the data is received, it's parsed into JSON format. The script then selects the `div` element with the `greetings` ID from the HTML. For each greeting retrieved from the server (consisting of a language and its corresponding message), the script appends a new paragraph to this `div`, displaying the language and message. If any errors occur during this process, they are logged to the console. In essence, this `app.js` script fetches greetings from the backend, processes them, and displays them on the frontend web page in an element with the `greetings` ID.

`index.html`

This is the main HTML file for the frontend, defining the structure and content of the web page in HTML5. It provides a simple structure for the frontend, with a dedicated area to display greetings and an included script (`app.js`) to handle the logic:

hello-world/frontend/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Hello World App</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <div id="greetings"></div>
  <script src="app.js"></script>
</body>
</html>
```

Let's explore the preceding code to understand the structure and layout of the frontend web page. The preceding HTML code sets up a basic web page titled *Hello World App*. The page is optimized for mobile and desktop viewing, ensuring it scales correctly on various device sizes. When loaded, the content within the `greetings` div is likely to be populated dynamically using the `app.js` JavaScript file. Additionally, the visual presentation and styling of the web page are defined in an external stylesheet named `styles.css`. In its current state, the page will initially appear blank, but given the structure, it's designed to display greetings or content within the `greetings` div through the linked `app.js` JavaScript.

styles.css

This file contains the styling rules for the frontend. This ensures the web page looks as intended and is responsive to different device sizes. We won't get into the code because it isn't essential.

Dockerfile

Like the backend Dockerfile, this file contains instructions to build a Docker image for the frontend part of the application. It will typically set up a web server environment (such as Nginx or another static server) to serve the frontend assets.

Now let us look at the unit testing part. For our Hello World application, we have a single unit test called `server.test.js` placed inside a folder named `unit-tests`. This file contains unit tests for the `server.js` file. Unit tests help ensure that individual parts of your code work as expected. The following is the full `server.test.js` code:

hello-world/unit-tests/server.test.js

```
const chai = require('../backend/node_modules/chai');
const chaiHttp = require('../backend/node_modules/chai-http');
```

```

const sinon = require('../backend/node_modules/sinon');
const mongodb = require('../backend/node_modules/mongodb');
const app = require('../backend/server');
chai.use(chaiHttp);
const expect = chai.expect;
describe('Hello World API', function() {
  // Create a stub for MongoDB...
  after(() => {
    connectStub.restore();
  });
  it('should return an array of greetings', async function() {
    const res = await chai.request(app).get('/api/helloworlds');
    expect(res).to.have.status(200);
    expect(res.body).to.be.an('array');
    expect(res.body[0]).to.have.property('language');
    expect(res.body[0]).to.have.property('greeting');
  });
});

```

Let us understand the preceding code. It is a unit test designed to validate the behavior of a Hello World API within a backend application. Utilizing `chai` as an assertion library and `chai-http` to facilitate HTTP requests within tests, the setup ensures that our API interacts as expected with a MongoDB database. However, instead of directly tapping into a real database, the test employs `sinon`—a mocking library—to create a stub that simulates MongoDB’s `connect` method. This stubbed connection, set up in the `before` block, ensures that whenever the API tries to fetch greetings from the database, it receives a predefined response, in this case, an English greeting: “Hello World.”

The core of the test lies within the `it` block, which asynchronously sends a `GET` request to the `/api/helloworlds` endpoint. Subsequent assertions then check the HTTP response. It confirms a successful status code (200), ensures the response contains an array, and further scrutinizes the array to make sure it has the `language` and `greeting` properties. After the tests run, the `after` block restores any modifications made to the MongoDB connection, ensuring no residual effects remain from the test.

In essence, this unit test ensures that the Hello World API functions correctly, responding with the expected format and content, all while isolating the actual database operations to provide consistent and controlled testing conditions.

Understanding the database code

Now let’s understand the database code. The code to populate the database is placed inside a folder named `database-init`, and it has a file called `init-mongo.js`:

hello-world/database-init/init-mongo.js (chopped-down version)

```

db = db.getSiblingDB('helloworldapp'); // select helloworldapp database
db.greetings.insertMany([
  { language: "English", message: "Hello World" },

```

```

    { language: "French", message: "Bonjour le Monde" },
    .
    .
    { language: "Finnish", message: "Hei Maaailma" },
    { language: "Polish", message: "Witaj świecie" },
    { language: "Hungarian", message: "Helló Világ" },
  ];

```

The preceding script is very simple and will initialize or set up a MongoDB database. It is used to seed initial data or set up collections and indexes.

As we close this section, we've meticulously explored the source code of our example three-tier web application, laying a solid foundation for understanding its inner workings. With this groundwork in place, we are poised to delve into the exciting realm of CI. In the next section, we will establish a repository on GitHub.

Creating a repository on GitHub for your project

In this section, you will learn how to create a repository on GitHub. You can skip this section if you already know how to create one. The idea is: to create a repository on GitHub and upload the cloned source code of our example Hello World web application. Creating a repository on GitHub is straightforward. Here's a step-by-step guide:

1. If you don't already have a GitHub account, you'll need to sign up (<https://github.com/join>) first. If you already have an account, just log in (<https://github.com/login>).
2. Once you're logged in, navigate to the main dashboard or your profile page. Click on the + icon at the top-right corner next to your profile picture and select **New repository** from the drop-down menu.
3. Now, configure your new repository:
 - **Repository Name:** Give your repository a name. This name should ideally reflect the project or the content you intend to store in it.
 - **Description (Optional):** Provide a brief description of your project.
 - **Visibility:** Choose whether you want your repository to be public (visible to everyone). You don't need credentials to clone the repository if it's public. Or choose for it to be private (only visible to you and any collaborators you choose). Credentials are needed if you want to clone or interact with it.
 - **Initialize with a README:** If you check this box, GitHub will create a basic **README.md** file in your repo. This is often recommended as it allows you to immediately clone the repository to your machine or elsewhere.
 - **Adding a .gitignore file:** This is an optional step, but if you know the programming language or the platform you're using, you can choose it from the drop-down list, and GitHub will create a **.gitignore** file that automatically excludes irrelevant files (such as temporary files or dependencies) from your Git tracking.
 - **Choose a LICENSE file:** This is also optional but recommended. If you want other developers to use your code, it's a good practice to add a license to your repository to clarify permissions. Common choices include the MIT license (permissive) and the GNU GPLv3 license (copyleft).

4. Once you are done, create the repository by clicking the green **Create repository** button at the bottom.
5. After creating your repository, GitHub will provide you with links to clone the repository, create a new file, upload an existing file, or open the repo in GitHub Desktop. If you want to clone it to your machine and start working locally, you can use the following command:

```
git clone https://github.com/YourUsername/YourRepositoryName.git
```

6. Now that your repository is set up, you can start adding files and folders and commence with your project development. Add the Hello-World-web-application-related files to the repository, commit, and then push. Note that the **README.md**, **LICENSE**, and **.gitignore** files will be replaced.

And that's it! With these simple steps, you've successfully created a repository on GitHub, ready for collaboration, version control, and more.

High-level CI design

Designing a CI pipeline for a three-tier web application using Node.js and aiming for a container-based solution involves several considerations. Here's a high-level guide to designing such a CI system using Jenkins, SonarQube, and Artifactory.

Source code considerations

When preparing the foundation for a CI system, especially for a three-tier web application, it's crucial to carefully evaluate several aspects of the application and its environment. These considerations ensure that the integration process is smooth and efficient and caters specifically to the application's architecture and requirements:

- **Source code structure:** The application should ideally be modularized, with separate directories or repositories for the frontend, backend, and database layers. Because our source code is small, we have chosen to divide the layers using folders.
- **Dependencies:** Ensure all Node.js dependencies are properly listed in your **package.json** files for each tier.
- **Testing:** Unit tests, integration tests, and any other relevant tests should be well defined using a framework such as Mocha, Chai, or Jest.
- **Containerization:** Each tier might require its own Dockerfile for containerization, ensuring the built containers are lightweight and optimized. This also ensures that the web application is modular.

Having gone through the source code for our software project (Hello World web application), you would agree that we have all the preceding considerations covered. Now, let's look at the high-level CI pipeline design.

High-level CI pipeline design

As we embark on constructing our CI pipeline, it's essential to conceptualize a sequence of stages that each piece of code must traverse. This orchestrated journey ensures that our code is not only functionally accurate but also adheres to the highest standards of quality, security, and performance. Here's a bird's-eye view of the CI process:

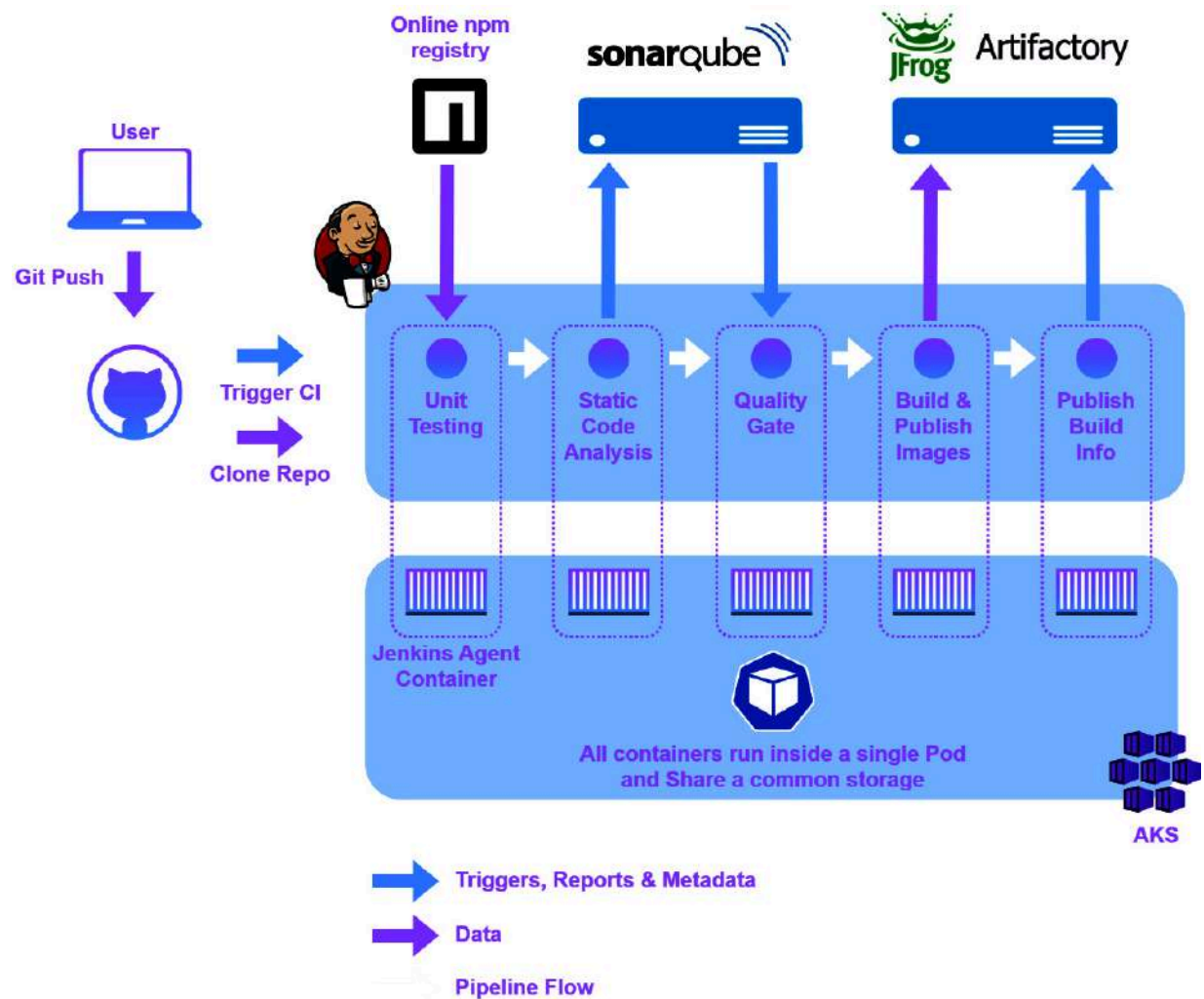


Figure 8.2: High-level CI pipeline design

Let us understand the preceding CI workflow. When a developer makes a change to the source code, changes are pushed to GitHub. With webhooks in place, GitHub will trigger a CI pipeline in Jenkins and the pipeline will follow these steps in sequence:

1. **Cloning the source code:** Pull the latest code from the GitHub repository. Basically, the pipeline will try to clone the source code repository.
2. **Unit testing:**
 - I. Use `npm install` to install the dependencies for each tier of the application. Dependencies will come from the online npm registry. They can also be configured to come from Artifactory.

II. Run unit tests.

III. If any test fails, the pipeline should halt, and the developers should be notified.

3. **Static code analysis:** Analyze the code base for code quality, vulnerabilities, bugs, and code smells. Publish the analysis report to SonarQube.

4. **Quality gate check:** Ensure the code meets the quality gate criteria set in SonarQube before proceeding. Fail the CI pipeline run if the SonarQube quality gate check fails.

5. **Build and publish Docker images:**

I. Use **docker build** or other relevant commands to create Docker images for each tier.

II. Tag the built Docker images appropriately, ideally with the build number or commit hash for traceability.

III. Push the built Docker images to the Artifactory Docker repository.

IV. Artifactory can be used as a secure private Docker registry, ensuring only authenticated pulls and pushes.

6. **Publish build info:** Publish build information (metadata) to Artifactory.

By designing your CI pipeline in this structured manner, you ensure that the code meets the quality criteria and that any changes can be swiftly and securely propagated through to your environments. Now, let us go through some configurations on GitHub, SonarQube, Artifactory, and the AKS cluster necessary for the CI pipeline.

Creating a webhook on the repository

Setting up a webhook in GitHub allows you to trigger custom callbacks when specific events happen on your repository. This is commonly used to integrate with CI/CD systems, monitoring tools, or other third-party services. Here are the steps to create a webhook on a GitHub project repository:

1. Navigate to your GitHub repository by logging in to GitHub and opening the repository where you want to add the webhook.
2. Next, click on the **Settings** tab located near the top of the page to access the repository settings. From the left sidebar on the resultant page, you'll find a section labeled **Webhooks**. Click on it.
3. On the resultant page, click on the **Add webhook** button available on the top-right side to add a new webhook.
4. You will be asked to fill in the webhook details, which are as follows:

- **Payload URL:** Enter the URL where the payloads will be delivered. This should be **<Jenkins server URL>/github-webhook/** (for example, <http://20.103.130.176/github-webhook/> or <http://myjenkins.org/github-webhook/>).
- **Content type:** Typically, you'd choose **application/json**.
- **Secret:** This is an optional field. If you provide a Secret, GitHub will use it to create a hash signature with each payload. This hash signature is sent along with each request in the headers. This is a mechanism to ensure that the requests sent to the payload URL are from GitHub. We will ignore it for now to keep things simple.
- **Which events would you like to trigger this webhook?:** Here, you can choose which GitHub events you want to trigger the webhook. You can select individual events or choose the default, which is **Just the push event**. But we

want our webhook to be triggered for both pull requests and push events. To do that, instead of selecting the **Just the push event** option, choose the **Let me select individual events** option instead. Doing so will give you a lot of sub options to select from. Check the box next to **Pushes** to trigger the webhook on push events, and also check the box next to **Pull requests** to trigger the webhook for pull request events. By selecting both **Pushes** and **Pull requests**, GitHub will send a payload to your specified URL whenever there's a push to the repository or whenever a pull request is opened, closed, or synchronized. Make sure your server is set up to handle both types of events, as the payload structure might differ between them.

5. Next, ensure that the **Active** checkbox is checked if you want the webhook to be functional immediately after creation. Once you've filled out the details, click on the **Add webhook** button. A successfully created webhook on GitHub looks something like this:

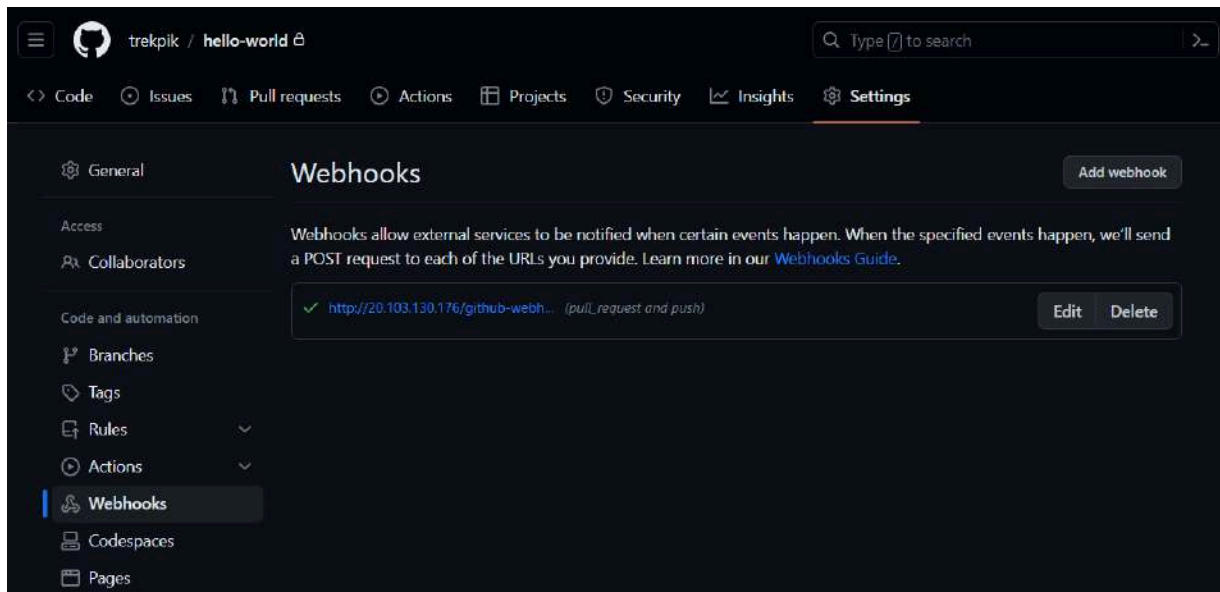


Figure 8.3: Creating a webhook on GitHub

6. After setting things up, GitHub will show a green checkmark next to each successful delivery. If there's an issue (e.g., if your server returns a non-2xx status response), you'll see a red X. You can click on each delivery to get more details.

Setting up a project in SonarQube

In this section, we'll take a step-by-step approach to initializing a new project within our established SonarQube server. This will lay the groundwork for the continuous inspection of our code quality, making future CI processes smoother and more efficient. Let's walk through the necessary steps to get our project up and running in SonarQube:

1. Navigate to your SonarQube instance in your browser and log in as an admin. In an actual project, you will log in as a non-admin user, and you'll need to have the necessary permissions to create a project. By default, logged-in users can create projects, but this might vary depending on your SonarQube configuration.
2. From the SonarQube dashboard, click on **Projects** in the main menu. You will be presented with some options; however, to keep things simple, let's go with the **Manually** option.
3. On the resultant page to create a project in SonarQube, you will be asked to fill in some information about the project, as follows:

- **Project display name:** This is the name that will be displayed in the SonarQube UI for your project.
- **Project key:** This is a unique identifier for your project. It's used internally by SonarQube and in URLs. Typically, it's something such as `orgname:projectname` or just `projectname`. It must be unique across the SonarQube instance. This value is special as it will be used inside the Jenkins pipeline code to publish the static code analysis report for the right project in SonarQube.
- **Main branch name:** This is by default set to `main`. It's the name of your repository's default branch.

Create a project

All fields marked with * are required

Project display name *



Up to 255 characters. Some scanners might override the value you provide.

Project key *



The project key is a unique identifier for your project. It may contain up to 400 characters. Allowed characters are alphanumeric, '-' (dash), '_' (underscore), '.' (period) and ':' (colon), with at least one non-digit.

Main branch name *

The name of your project's default branch [Learn More](#)

Set Up

Figure 8.4: Creating a project in SonarQube

4. Once you have filled in the required fields as suggested in the preceding figure, click on the **Set Up** button.

And that is all we need to do for now. However, there are a lot of mandatory and optional settings to configure for the SonarQube project. You can choose to configure them either manually from the

SonarQube project dashboard using the **Project Settings** section or programmatically from the Jenkins pipeline code, which we will learn shortly when we start to write the CI pipeline code. In summary, to run analysis from the CI pipeline, all we need are the SonarQube credentials (we already have them in Jenkins from [Chapter 3](#)), and the project key, which we created just now. Great! Now let's move on to making the necessary configurations in Artifactory.

Using Artifactory as a Docker registry

Artifactory, developed by JFrog, not only serves as a universal artifact repository manager but also effectively functions as a Docker registry. This means that Artifactory can securely store and manage Docker container images. By acting as a Docker registry, it provides developers with a centralized location to push and pull Docker images, streamlining the build and deployment process. Additionally, Artifactory's fine-grained access control ensures that the right Docker images are accessed by the right personnel, adding an extra layer of security.

Setting up a Docker repository acting as a Docker registry in Artifactory is a straightforward process. The following are the steps to create a local Docker repository named `docker-local`:

1. Open your Artifactory instance in a web browser and log in using your admin credentials.
2. From the main dashboard, go to the **Admin** section (typically a gear icon in the top-right corner).
3. Under the **Repositories** section in the sidebar, select the **Repositories** option. You will see a page with four tabs: **Local**, **Remote**, **Virtual**, and **Federated**. To create a new local repository in Artifactory, with the **Local** tab selected, click on the **+ Add Repositories** button, available at the top-right corner of the page, and select **Local Repository**.
4. From the resultant pop-up menu, select **Docker** as the package type.
5. Enter `docker-local` as the value for the mandatory **Repository key** field.
6. Optionally, you can adjust other settings based on your needs, such as enabling forced Docker authentication and setting up Xray integration.
7. Click the **Create Local Repository** button to create the local Docker repository. Your newly created local repository is now available under **Application | Artifactory | Artifacts**, as shown.

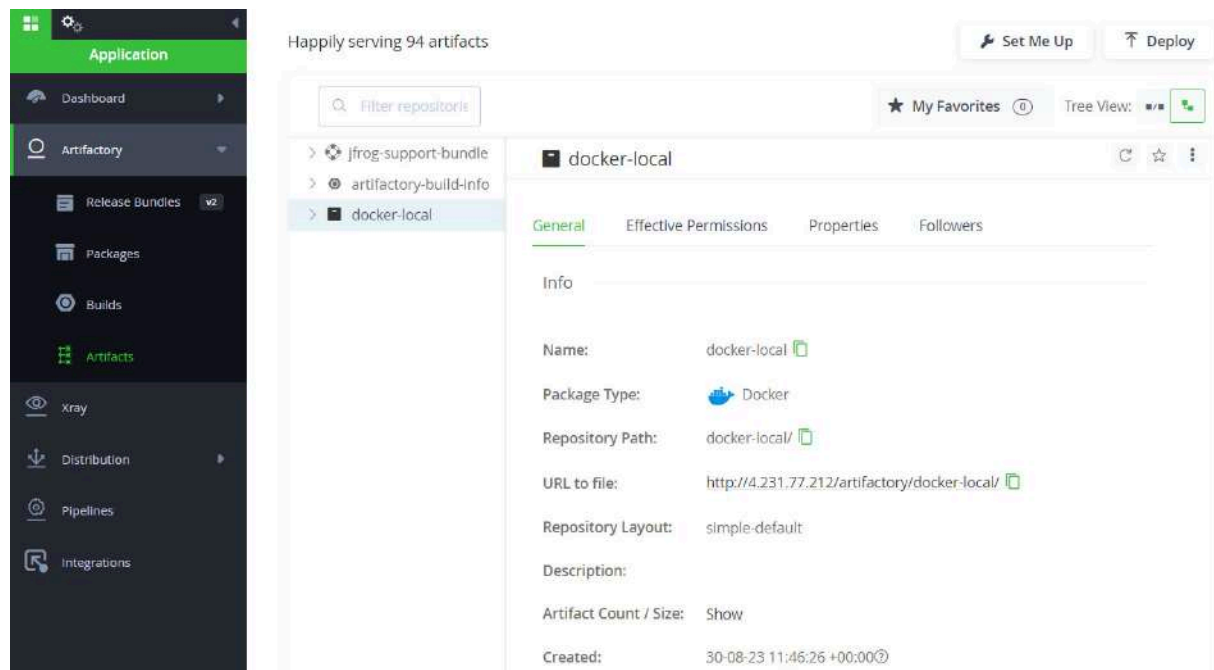


Figure 8.5: Artifactory repository

Once you've set up the local repository, you'll also want to ensure your CI/CD systems are configured to push to and pull from these Artifactory Docker repositories. For that, you need Artifactory credentials (make sure you have followed the steps in the *Creating a user in Artifactory* section from [Chapter 3](#)) and the name of the local repository in Artifactory, which is what we created just now. Next, let's create a Kubernetes Secret containing the Artifactory credentials in our Kubernetes cluster.

Creating a Kubernetes Secret on AKS

You may ask why we need to create a Kubernetes Secret. Well, in the next chapter, when we build Docker images with Kaniko and push them to Artifactory, it will be imperative to secure your Artifactory credentials. Kubernetes Secrets offer an efficient way to store sensitive information, such as authentication credentials, outside of your application code. By creating a Kubernetes Secret with your Artifactory credentials, you ensure that Kaniko can authenticate with Artifactory securely during the image push process. This approach not only safeguards your credentials from potential exposure but also upholds the integrity of your CI pipeline. Without this Secret, Kaniko would be unable to securely interact with Artifactory, risking unauthorized access or image push failures. Follow these steps to create a Kubernetes Secret on your AKS cluster:

1. Execute the **az login** command in your terminal or command prompt to authenticate with your Azure account:

```
az login
```

2. A browser window will open requesting you to sign in to your Azure account. Once authenticated in the browser, you can close the window. The terminal should then display a list of subscriptions associated with your Azure account.
3. Next, set a default resource group for your Azure CLI session by running the following command in your terminal. Remember to replace **<Your resource group>** with the name of the resource group where you have your AKS. Once executed, any subsequent Azure CLI commands that require a resource group will use the specified default unless overridden. Ignore this step if you have only one resource group in your Azure subscription:

```
az configure --defaults group=<Your resource group>
```

4. Now, use the **kubectl create secret** command to create a Secret in Kubernetes that holds the credentials for your Docker registry. In our case, the Docker registry is our Artifactory local repository. Executing the command in the terminal will create a Secret named **artifactory-credentials** in the **Jenkins** namespace:

- Replace **<IP address of your Artifactory server>** with the actual IP address or URL of your Artifactory server
- Replace **<Password for user jenkins>** with the actual password for the **jenkins** user
- Replace **<Email for user jenkins>** with the email address associated with the **jenkins** user:

```
kubectl create secret \  
  docker-registry artifactory-credentials \  
  --docker-server=<IP address of your Artifactory server> \  
  --docker-username=jenkins \  
  --docker-password=<Password for user jenkins> \  
  --docker-email=<Email for user jenkins> \  
  -n jenkins
```

Notice that we created the Kubernetes Secret for Artifactory authentication within the **jenkins** namespace, rather than the **artifactory** or **default** namespace. This is because the Jenkins server is going to employ the **artifactory-credentials** Secret to authenticate with Artifactory from a container situated within its own namespace.

Great! With our SonarQube project in place, an Artifactory repository set up for our Docker images, and the necessary Kubernetes Secret established, we've laid a strong foundation. We've also initialized a GitHub repository and set up a webhook for real-time event tracking. Couple these preparations with the Jenkins credentials, plugins, and configurations for both SonarQube and Artifactory (as detailed in [Chapter 3](#)) and we are fully prepared. Now, we're poised to craft and kick off our CI pipeline for the Hello World web application.

Summary

In this chapter, we embarked on a comprehensive journey through the world of CI using Jenkins. We began by setting the stage with the planning section, explaining the significance of a robust CI plan and shedding light on high-level design principles essential for any CI initiative. A deep dive into understanding the software project for CI, its architecture, and the intricacies of its source code equipped us with the knowledge to mold our CI pipeline effectively.

After that, by integrating key components such as webhooks for real-time code push notifications, setting up a project in SonarQube for quality checks, and establishing a Docker repository in Artifactory for container management, we seamlessly intertwined multiple tools into our CI ecosystem.

With our deep dive into configurations across GitHub, SonarQube, Artifactory, and AKS, we've set the stage for the exciting phase that lies ahead. In the next chapter, we will harness the power of Jenkins through its Declarative Pipeline code, crafting a robust CI pipeline that encapsulates our groundwork. Not only will we construct this pipeline, but we'll also journey through its execution, offering you a real-time visualization of each run. As we move on, prepare to merge theory with tangible action, bringing our CI vision to life.

Questions

1. Is this statement true or false?

In SonarQube, the project key used to identify projects is case sensitive.

2. Which of the following is the correct order of the typical stages of a **Continuous Integration (CI)** pipeline, starting from the initial code commit to deployment?

- A. Unit Test → Build → **Static Code Analysis (SCA)** → Quality Gate → Package → Publish
- B. Build → Unit Test → Package → **Static Code Analysis (SCA)** → Quality Gate → Publish
- C. Build → Unit Test → **Static Code Analysis (SCA)** → Quality Gate → Package → Publish
- D. Quality Gate → Build → Unit Test → **Static Code Analysis (SCA)** → Package → Publish

3. When configuring a webhook in GitHub, which events should you select to trigger the webhook for both push and pull request events?

- A. Just the push event
- B. Select individual events, and then check pushes only
- C. Select individual events, and then check both pushes and pull requests
- D. Check all events to ensure the webhook is triggered

4. Is this statement true or false?

To use Artifactory as a Docker registry, the package type should be set to Generic.

5. Is this statement true or false?

A Secret created in Namespace A in Kubernetes can only be accessed by services running within Namespace A, and not by services in Namespace B.

Answers

1. Correct answer: **True**

Explanation:

In SonarQube, the project key, which uniquely identifies each project within the system, is case sensitive. This means that project keys such as `ProjectKey`, `projectkey`, and `PROJECTKEY` would be recognized as different projects. It's important for users to be consistent in their use of case when dealing with project keys to avoid confusion or the misidentification of projects.

2. Correct answer: **C**

Explanation of incorrect answers:

A. This option is incorrect because the Build stage typically precedes Unit Testing. The build process compiles the code and prepares it for testing and analysis, which then follows in the subsequent stages.

B. This option is incorrect as it places the Package stage before **Static Code Analysis (SCA)** and Quality Gate. Typically, the code is analyzed and passes through a quality gate before it is packaged for release.

D. This option is incorrect because the Quality Gate stage does not come first. Typically, the quality gate is used to assess the code after it has been built, tested, and analyzed. It is a checkpoint to ensure that the code meets the set quality standards before moving on to packaging and publishing.

3. Correct answer: **C**

Explanation of incorrect answer choices:

A. This option is incorrect because choosing *just the push event* will only trigger the webhook for push events, and not for pull request events.

B. This option is also incorrect. While it involves selecting individual events, checking only *pushes* will not trigger the webhook for pull request events, which is required in this scenario.

D. This option is not ideal because checking all events would trigger the webhook for more events than necessary.

4. Correct answer: **False**

Explanation:

This statement is false. When configuring Artifactory to be used as a Docker registry, the package type should specifically be set to *Docker*, not *Generic*. The *Generic* package type in Artifactory is

used for the basic storage and handling of artifacts that do not fit into specific package types, such as Docker, Maven, or npm.

5. Correct answer: **True**

Explanation:

In Kubernetes, Secrets are namespaced resources. This means that a Secret created in one namespace (e.g., Namespace A) is accessible only to Services, Pods, and Deployments within that specific namespace. It cannot be accessed by services running in a different namespace (e.g., Namespace B) unless specific measures are taken to replicate or share the Secret across namespaces. This design is part of Kubernetes' approach to ensuring resource isolation and security within a cluster.

Writing Your First CI Pipeline

In the previous chapter, we established a solid foundation for designing **Continuous Integration (CI)** pipelines using Jenkins. This was achieved through an exploration of the conceptual terrain of CI planning and a detailed examination of the high-level design principles underpinning CI processes. In the upcoming chapters, we will pivot our focus toward the heart of our discussion: the actual creation of CI pipeline code. We will comprehensively cover each stage of the pipeline, ranging from unit testing to the publication of artifacts. Additionally, we will provide step-by-step guidance on seamlessly integrating this pipeline code into your source code, putting the CI process into immediate action. The journey will culminate in a thorough exploration of visualizing the CI pipeline, both through the classic Jenkins interface and the Jenkins Blue Ocean interface, offering valuable insights into test reports, code coverage, build artifacts, and more.

By the end of this chapter, you will be able to write the actual CI pipeline code that integrates, tests, and publishes software changes to Artifactory. We will also look at how to set up multi-branch pipeline jobs in Jenkins to handle CI across different branches of a software project. Once you have completed this chapter, you should be able to use Jenkins Blue Ocean for a visual representation of the CI pipeline, which will help you with easier monitoring and debugging.

As such, we will cover the following main topics in the chapter:

- Learning how to write the various stages of CI pipeline code
- Executing and visualizing the CI pipeline using Jenkins Blue Ocean

Technical requirements

For this chapter, you will need the following tools installed in your local development environment:

- An IDE such as Visual Studio Code or any one you choose
- Git

You can download the source code of this chapter from https://github.com/PacktPublishing/Learning-Continuous-Integration-with-Jenkins_Third-Edition.

Writing CI pipeline code

Our source code project contains a **Jenkinsfile** containing our CI pipeline code. It defines a Jenkins pipeline that can handle CI for our Hello World web application with both frontend and backend components. The pipeline integrates with Kubernetes, SonarQube, and Artifactory. Let's look at the file section by section in detail.

Configuring Jenkins agent settings

In the context of declarative CI pipeline code, the **agent { kubernetes {...} }** section is pivotal. It designates that the Jenkins job should run within a Kubernetes environment. The **agent** section is responsible for defining where and how the entire pipeline or specific stages will be executed. By specifying **kubernetes**, we're informing Jenkins to provision a fresh Kubernetes Pod for the job. This is particularly beneficial for scalability, as Pods are ephemeral and can be dynamically created and destroyed. It ensures that our CI tasks run in an isolated, consistent, and resource-efficient manner, taking full advantage of Kubernetes' container orchestration capabilities. Following is a code snippet of the **agent** section:

```
agent {
  kubernetes {
    label 'ci-template'
    yaml '''
      //Your Pod Template
    '''
  }
}
```

Let's break down this code:

- The **agent {...}** section clarifies that the execution of the pipeline will take place on a Jenkins agent, using Kubernetes. The agent will operate with a distinct Kubernetes Pod configuration, as specified in the provided YAML string. Furthermore, to differentiate this Pod template, the **ci-template** label is employed.
- The Pod consists of three container specs: *Node Container (node)* for running Node.js tasks, *Sonar Scanner CLI Container (sonar-scanner-cli)* for running SonarQube scans, and *Kaniko Container (kaniko)* for building Docker images.

Here is a short snippet of the complete Pod template (refer to the **Jenkinsfile** from the GitHub repository to view the complete code):

//Your Pod Template (chopped- down version)

```
apiVersion: v1
kind: Pod
metadata:
  name: ci-pod
spec:
  ...
  - name: kaniko
    image: gcr.io/kaniko-project/executor:debug
    command:
```

```

    - sleep
  args:
    - 99d
  volumeMounts:
    - name: kaniko-secret
      mountPath: /kaniko/.docker
  restartPolicy: Never
  volumes:
    - name: kaniko-secret
      secret:
        secretName: artifactory-credentials
        items:
          - key: .dockerconfigjson
            path: config.json

```

Let's discuss the preceding Pod deployment YAML in detail:

- The **spec: { ... }** block defines the specification for a Kubernetes Pod. A Pod in Kubernetes is the smallest and simplest unit in the Kubernetes object model that you create or deploy. It represents a single unit of deployment.
- **containers:** represents a container configuration. This section defines a list of containers that the Pod will run.
- An item in the list, for example, **- name: node**, sets the name of the container to **node**. This name is used to reference the container elsewhere in the Jenkinsfile, as you will see later in this chapter.
- **image: node:latest:** This specifies the Docker image to be used for this container. In this case, it's using the latest version of the official Node.js image from DockerHub.
- **command: - sleep:** This defines the command that will be run when the container starts. The **sleep** command is a Unix command-line utility that suspends execution for a specified period.
- **args: -99d:** This section defines the arguments passed to the command when the container starts. The **99d** argument tells the **sleep** command to pause execution for 99 days. This is a common pattern used in Jenkins Kubernetes configurations to keep a container running for a long time so that Jenkins can execute multiple steps inside it without restarting the container.
- Similarly, we have another container spec named **sonar-scanner-cli** for running our static code analysis stage. To view it, please refer to the Jenkinsfile in the code provided for [Chapter 9](#), available on GitHub. **image: sonarsource/sonar-scanner-cli:latest** specifies the Docker image to be used for this container. This image is the official SonarScanner CLI image from DockerHub. It provides the necessary environment and tools to run SonarQube scans directly from the command line.
- We also have another container spec named **kaniko** for building our frontend and backend Docker images. It uses the **gcr.io/kaniko-project/executor:debug** image. It's the debug version of the official **Kaniko** executor image, which allows building Docker images in environments without a Docker daemon.
- Additionally, we have **volumeMounts:** for the **kaniko** container spec. These sections define a volume that's mounted on the **kaniko** container. The volume is backed by a Kubernetes secret named **artifactory-credentials**. This secret contains the credentials required to push images to the specified Docker registry. The secret is mounted in a way that it can be read by **kaniko**, enabling it to authenticate with the Docker registry in Artifactory.

Performing unit testing

Next comes the first stage of our pipeline: *Unit testing*. But wait, shouldn't it be *Build*? Here is an explanation. In the context of a Node.js application, the term **build** is different from traditional

compiled languages. It refers to building Docker images, and that is why it is the second-last stage of our CI pipeline, and unit testing is the first one. Here is the code snippet for the *Unit testing* stage:

```
stage('Unit testing') {
  steps {
    container('node') {
      sh '''
        cd ./backend/
        npm install
        npm run test
      '''
    }
  }
}
```

Let's break down this code step by step:

- The **stage('Unit testing')** { ... } section defines a Jenkins pipeline stage named *Unit testing*. Within Jenkins, stages are the primary blocks of the pipeline and help in organizing the entire process into discrete steps. Each stage represents a specific phase of the CI process.
- The **steps { ... }** section within the stage describes a series of sequential tasks that Jenkins will execute inside the given stage. It provides the detailed commands and actions that need to be run.
- The **container('node')** { ... } step instructs Jenkins to execute the following steps inside a specific container labeled **node**. The container's definition and properties are described elsewhere in the *Jenkinsfile*, typically in the **agent** section, or in an external Kubernetes configuration. It ensures that the steps inside are run in an environment that matches the specified container configuration.
- The **sh '...'...** command runs shell commands. It's a way for Jenkins to execute system-level commands. The contents of the shell scripts are as follows:
 - **cd ./backend/**: This command changes the current working directory to the **backend** folder. It ensures that the following commands are executed within the **backend** directory.
 - **npm install**: The **Node Package Manager (npm) install** command is used to install all dependencies defined in the **package.json** file located in the current directory (in this case, the **backend** directory). This ensures that all the required libraries and modules are available for the application.
 - **npm run test**: This runs the test script defined in the **package.json** file. Typically, this command will execute unit tests for the backend application, and any test failures will be reported.

In essence, the *Unit testing* stage is designed to execute unit tests for the backend component of the application using a Node.js environment. The container spec ensures that a suitable environment (Node.js in this case) is available for the execution of these tests.

Performing static code analysis

Next comes the static code analysis stage. This stage is meant to analyze your code base using SonarQube to identify issues, technical debt, vulnerabilities, and more. Here is the code snippet of the

SonarQube analysis stage:

```
stage('SonarQube analysis') {
  steps {
    container('sonar-scanner-cli') {
      withSonarQubeEnv('Default SonarQube Server') {
        sh """
          sonar-scanner \
          -Dsonar.projectKey=Hello-World \
          -Dsonar.sources=backend/,frontend/,database-init/ \
          -Dsonar.exclusions=integration-tests/,test-output/ \
          -Dsonar.tests=unit-tests/ \
          -Dsonar.host.url=${env.SONAR_HOST_URL} \
          -Dsonar.login=${env.SONAR_AUTH_TOKEN} \
          -Dsonar.javascript.lcov.reportPaths=./test-output/coverage/lcov.info
        """
      }
    }
  }
}
```

Let's delve into the *SonarQube Analysis* stage and its associated specification:

- The **stage('SonarQube analysis')** {...} section initiates a Jenkins pipeline stage named **SonarQube analysis**.
- The **steps** {...} section within the stage describes a series of sequential tasks that Jenkins will execute inside the given stage.
- The **container('sonar-scanner-cli')** {...} step instructs Jenkins to execute the following steps inside a specific container labeled **sonar-scanner-cli**. The properties of this container are defined in the specified container spec. This makes the sonar-scanner CLI available.
- **withSonarQubeEnv('Default SonarQube Server')** {...} is a Jenkins step provided by the SonarQube Jenkins plugin. It sets up the required environment variables to communicate with a SonarQube server. The **Default SonarQube Server** argument is the name of a SonarQube server configuration saved in Jenkins.
- Next, the **sh "..."** command runs a multi-line shell. Inside it, the **sonar-scanner** command is used to run the SonarQube analysis with the following parameters:
 - **-Dsonar.projectKey=Hello-World**: Assigns a unique key for the project in SonarQube. Remember the project creation section in SonarQube.
 - **-Dsonar.sources=backend/,frontend/,database-init/**: Specifies the directories containing the source code to be analyzed.
 - **-Dsonar.exclusions=test-output/**: Excludes specific directories from the analysis.
 - **-Dsonar.tests=unit-tests/**: Indicates the directory containing the test code.
 - **-Dsonar.host.url=\${env.SONAR_HOST_URL}**: Sets the URL for the SonarQube server, fetched from an environment variable.
 - **-Dsonar.login=\${env.SONAR_AUTH_TOKEN}**: Authentication token for the SonarQube server.
 - **-Dsonar.javascript.lcov.reportPaths=./test-output/coverage/lcov.info**: Specifies the path to the coverage report for JavaScript code.

Refer to the *Analysis parameters* documentation (see [1] in the *Further reading* section at the end of the chapter) to see the full list of all the mandatory and optional parameters that you can pass to the sonar-scanner CLI. In summary, the *SonarQube analysis* stage is designed to perform a comprehensive code quality analysis on your code base using SonarQube. The results of this analysis are then sent to a SonarQube server, where they can be visualized, helping developers identify and fix potential issues in the code.

Performing a SonarQube quality gate check

Next is the *Quality Gate* stage. This is a crucial component when integrating Jenkins with SonarQube. Here is the code snippet for the *Quality Gate* stage:

```
stage("Quality Gate") {
    steps {
        timeout(time: 1, unit: 'HOURS') {
            waitForQualityGate abortPipeline: true
        }
    }
}
```

Let's dissect this stage in detail:

- The **stage("Quality Gate") {...}** section defines a Jenkins pipeline stage named *Quality Gate*. The primary purpose of this stage is to wait for SonarQube's quality gate result. A quality gate is a set of conditions the project must meet before it can qualify to be packaged.
- This **steps { . . . }** section within the stage describes a series of sequential tasks that Jenkins will execute as part of the stage.
- The following Jenkins directive, **timeout(time: 1, unit: 'HOURS') { . . . }**, enforces a timeout on the enclosed steps. Here, the timeout is set to 1 hour. If the enclosed steps are not completed within this time frame, Jenkins will abort the stage, marking it as failed.
- The **waitForQualityGate abortPipeline: true** step is a step provided by the SonarQube Jenkins plugin. This command makes the Jenkins job pause and wait for the SonarQube analysis to complete and for the quality gate to be computed. Once the quality gate is available, the job will either continue (if the quality gate passes) or fail (if it doesn't).
- The **abortPipeline: true** parameter specifies that if the quality gate fails (e.g., if the code coverage is below a certain threshold or if there's a critical bug), the Jenkins pipeline should be aborted. This ensures that code that doesn't meet the quality criteria won't progress further in the CI process.

In essence, the *Quality Gate* stage ensures that your code base meets the quality standards set in SonarQube before it progresses further in the Jenkins pipeline. If the code doesn't pass the quality gate, the pipeline is aborted, signaling to the team that there are important quality issues that need to be addressed.

Building and publishing images to Artifactory

Next comes the *Build and Publish Docker Images* stage, aimed at building Docker images for both the frontend and backend components of your application and then publishing those images to a Docker registry:

```
stage('Build and Publish Docker Images') {
    parallel {
        stage('Frontend') {
            steps {
                container(name: 'kaniko', shell: '/busybox/sh') {
                    sh '''#!/busybox/sh
                        VERSION=$(grep '"version":' ./backend/package.json | head -1 | awk -F: '{
print $2 }' | sed 's/[, ]//g')
                        /kaniko/executor --context `pwd` --dockerfile=/backend/dockerfile --insecure
--skip-tls-verify --insecure-registry 4.231.77.212 --skip-tls-verify-registry 4.231.77.212
--destination=4.231.77.212/docker-local/backend:$VERSION-$BUILD_NUMBER --image-name-with-
digest-file=backend-image-file
                        '''
                }
            }
        }
        stage('Backend') {
            steps {
                container(name: 'kaniko', shell: '/busybox/sh') {
                    sh '''#!/busybox/sh
                        VERSION=$(grep '"version":' ./backend/package.json | head -1 | awk -F: '{
print $2 }' | sed 's/[, ]//g')
                        /kaniko/executor --context `pwd` --dockerfile=/frontend/dockerfile --insecure
--skip-tls-verify --insecure-registry 4.231.77.212 --skip-tls-verify-registry 4.231.77.212
--destination=4.231.77.212/docker-local/frontend:$VERSION-$BUILD_NUMBER --image-name-with-
digest-file=frontend-image-file
                        '''
                }
            }
        }
    }
}
```

Let's break it down in detail:

- The **stage('Build and Publish Docker Images')** {...} section defines a Jenkins pipeline stage named **Build and Publish Docker Images**. This stage will run the sub-stages to build the frontend and backend Docker images in parallel to optimize the build process.
- The **parallel {...}** directive allows multiple stages or steps to be executed concurrently. This is especially useful when you have tasks that can run simultaneously, such as building Docker images for different components in our case.
- **stage('Frontend')** {...} defines a sub-stage inside the main *Build and Publish Docker Images* stage, dedicated to handling the frontend component.
- The **container(name: 'kaniko', shell: '/busybox/sh')** {...} step inside this block will be executed in a container labeled **kaniko**, which is specified to use the shell **/busybox/sh**. The **kaniko** executor [2] is a tool for building Docker images without the need for Docker daemon access.
- Within the **Frontend** sub-stage, **VERSION=\$(grep '"version":' ./backend/package.json | head -1 | awk -F: '{ print \$2 }' | sed 's/[,]//g')** extracts the version number from the **package.json** file of the backend. It searches for the **version** key and processes the line to retrieve just the version number.
- Next, the **/kaniko/executor...** command builds and pushes the Docker image. Here's a breakdown of its parameters:

- **--context pwd**: Specifies the current working directory as the build context.
 - **--dockerfile=/backend/dockerfile**: The path to the Dockerfile for the backend.
 - **--insecure...**, **--skip-tls-verify...**, **--skip-tls-verify-registry...**: These flags are related to TLS settings when dealing with insecure/private Docker registries. You don't need these options if your Artifactory server is accessible over HTTPS and equipped with the necessary SSL certificates.
 - **-destination=...**: Specifies the Docker registry and repository where the built image will be pushed. The image is tagged with the version from **package.json** and the Jenkins build number coming from the Jenkins **\$BUILD_NUMBER** environment variable.
 - **--image-name-with-digest-file=backend-image-file**: Specifies a file where the name of the built image with its digest will be saved.
- The **stage('Backend') { ... }** sub-stage is like the **Frontend** sub-stage, but it handles the backend component. The primary difference is the path to the Dockerfile and the image name and tag.

In essence, the *Build and Publish Docker Images* stage concurrently builds Docker images for both the frontend and backend components, using the efficiency of the **kaniko** executor. The images are then pushed to a specified Docker registry. The container spec ensures that the necessary environment and credentials are available for this process.

Publishing build info to Artifactory

The *Publish build info* stage aims to send build information to Artifactory. This build information is attached as metadata to the uploaded Docker images. Here is the code for the *Publish build info* stage:

```
stage('Publish build info') {
    steps {
        rtCreateDockerBuild (
            serverId: 'Default Artifactory Server',
            sourceRepo: 'docker-local',
            kanikoImageFile: "backend-image-file"
        )
        rtCreateDockerBuild (
            serverId: 'Default Artifactory Server',
            sourceRepo: 'docker-local',
            kanikoImageFile: "frontend-image-file"
        )
        rtPublishBuildInfo (
            serverId: 'Default Artifactory Server'
        )
    }
}
```

Let's dissect this stage:

- The **stage('Publish build info') { ... }** section defines a Jenkins pipeline stage named *Publish build info*. Its primary purpose is to push metadata about the build and the associated Docker images to Artifactory. This metadata can include

information about the build environment, dependencies, produced artifacts, and more. It aids in traceability and can be crucial for auditing or troubleshooting purposes.

- The **steps { ... }** section within the stage describes a series of sequential tasks that Jenkins will execute as part of the stage.
- Within the step, we have the **rtCreateDockerBuild()** command, which is provided by the Jenkins Artifactory plugin. It's used to gather information about a Docker build and prepare it for publishing to Artifactory.
- The parameter for **rtCreateDockerBuild()** is **serverId: 'Default Artifactory Server'**, which specifies the Artifactory server's configuration in Jenkins where the build info will be published. The configuration typically includes the server URL, credentials, and other settings.
- Then we have **sourceRepo: 'docker-local'**, which specifies the Artifactory repository that contains the Docker images. This is where the Docker images were pushed in the previous stage.
- Next, **kanikoImageFile: "backend-image-file"** and **kanikoImageFile: "frontend-image-file"** are paths to files that contain the names and digests of the Docker images built by Kaniko in the previous stage. The files were produced by the **--image-name-with-digest-file** parameter of the **Kaniko build** command. The plugin uses these files to gather information about the Docker images. The command is executed twice: once for the backend image and once for the frontend image.
- The **rtPublishBuildInfo** command is also provided by the Jenkins Artifactory plugin. It takes the build information collected in the previous steps and publishes it to the specified Artifactory server.
- The parameter for **rtPublishBuildInfo** is **serverId: 'Default Artifactory Server'**, which specifies the Artifactory server's configuration in Jenkins where the build info will be published.

In essence, the *Publish build info* stage ensures that after building and pushing the Docker images, all relevant metadata about the build and the images is sent to Artifactory. This provides comprehensive traceability and allows teams to understand the full context of each build and its associated artifacts.

Publishing testing results to Jenkins

The **post** section in our Jenkins pipeline specifies actions to be taken after all the stages are complete, regardless of their outcome (whether they've succeeded or failed). This ensures some cleanup or reporting actions are always performed:

```
post {
  always {
    junit '**/test-output/unit-test-report/junit-test-results.xml'
  }
}
```

Let's break down the provided **post** section:

- In the code snippet, the **post { ... }** keyword begins the definition of post-build actions.
- The **always { ... }** directive ensures that the steps inside this block are executed after every run of the pipeline, regardless of the pipeline's success or failure. It's useful for actions such as sending notifications, collecting reports, or cleaning up resources.
- The **junit** step is provided by the Jenkins JUnit plugin. It collects the JUnit XML report, **junit-test-results.xml**, and displays the results in Jenkins, providing a visual representation of the test results.

- `**/test-output/unit-test-report/junit-test-results.xml`: This is a pattern that matches the path to the JUnit XML report. The `**` allows for any directory depth, meaning it will search in all directories and subdirectories for the specified path. This ensures that if your tests produce a JUnit XML report at the given location, Jenkins will pick it up.

The `post` section ensures that after every run of the pipeline, the unit test results are collected and displayed in Jenkins. This provides an immediate visual feedback mechanism, allowing developers to quickly see the test results and any failures that might have occurred. With that, we've completed our overview of the CI pipeline code. Next, we'll establish a multi-branch pipeline job within Jenkins and execute it to witness the CI process in action.

CI in action

Seeing CI in action provides a deeper understanding and appreciation of its intricacies. In this section, we will first learn how to create and configure a Multibranch Job in Jenkins, and then we'll go through a successfully executed Continuous Integration pipeline run. Along the way, we'll delve into the logs, offering a window into the activities of the pipeline. We'll observe how Jenkins seamlessly integrates with tools such as SonarQube and Artifactory, providing comprehensive feedback from unit test results to static code analysis, and how it manages and showcases built Docker images. By the end of this section, you'll have a clear visual representation of what a "green" or successful CI run looks like and how each component contributes to the bigger picture of software quality assurance. So, let's start with creating a Multibranch pipeline job in Jenkins.

Creating a Multibranch Pipeline Job in Jenkins

Creating a Multibranch Pipeline Job in Jenkins allows you to automatically create a pipeline for each branch on your source code repository. This is particularly useful in projects where you might have multiple active branches being worked on simultaneously. Here's a step-by-step guide to creating a Multibranch Pipeline Job named `Hello-World`:

1. Access your Jenkins server by opening your web browser and navigating to your Jenkins instance URL.
2. From the Jenkins dashboard, click on the + **New Item** button in the left-hand side menu.
3. On the resulting page (see *Figure 5.6*), select the type of Jenkins pipeline you wish to create and give it a name, as follows:
 - I. Enter the name `Hello-World` into the **Enter an item name** field.
 - II. Choose **Multibranch Pipeline** from the list of options.
 - III. Click **OK** to proceed.

Enter an item name

» Required field

**Freestyle project**

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**Maven project**

Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

**Pipeline**

Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

**Multi-configuration project**

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

**Folder**

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

**Multibranch Pipeline**

Creates a set of Pipeline projects according to detected branches in one SCM repository.

**Organization Folder**

Creates a set of multibranch project subfolders by scanning for repositories.

Figure 9.1 – Creating a Multibranch Pipeline Job in Jenkins

4. On the Jenkins Multibranch Pipeline **Configuration** page, configure the **Branch Sources** section by scrolling down to it and then clicking on **Add Source** to select your Source Control Management system (e.g., Git, GitHub, Bitbucket). In our case, it's GitHub.
5. Next, fill out the relevant details for GitHub as follows. This is shown in *Figure 9.2*:
 - I. You might need to provide your GitHub credentials. So, using the **Credentials** field, choose the GitHub credentials to interact with repositories (detailed instructions for creating credentials in Jenkins has been provided in the *Using Jenkins Credentials* section in [Chapter 3](#)).
 - II. Next, add the URL of your GitHub repository using the **Repository HTTPS URL** field.
 - III. You can click on the **Validate** button to check the connection.

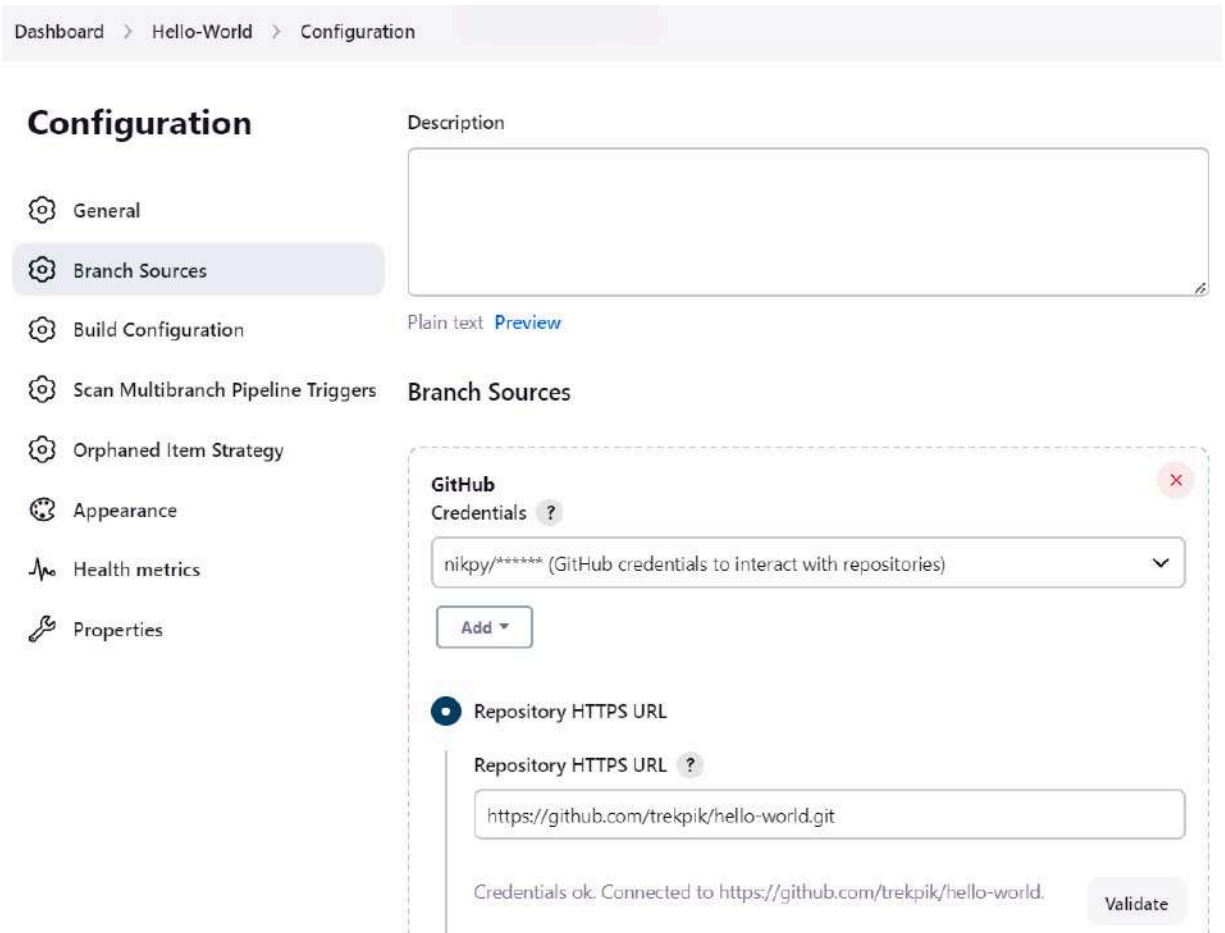


Figure 9.2 – Configuring a Multibranch Pipeline Job in Jenkins

- Next, configure the **Build Configuration** section. By default, Jenkins will look for a file named *Jenkinsfile* in the root of the branch for the build configuration. If your file has a different name or is located elsewhere, you can specify it in the **Build Configuration** section.
- There is a section named **Scan Multibranch Pipeline Triggers**. We don't need to do this. Since we have set up a webhook, Jenkins will be notified immediately of changes, but it's still a good idea to have a periodic scan in case the webhook fails for any reason.
- Once you are done configuring the Jenkins Multibranch Pipeline Job, click the **Save** button at the bottom of the page.

Important! Jenkins will now start the initial scan of the repository and set up pipeline jobs for each branch it discovers. Future changes to branches, as well as new branches, will automatically get picked up (based on the scan trigger configuration). Any time you make changes to a branch, Jenkins will automatically detect those changes and start a build for that branch (assuming you have a valid *Jenkinsfile* in the branch).

NOTE

If you have diligently followed the steps outlined in this chapter, you should encounter no errors. However, if an error does arise, please double-check to ensure that all steps have been correctly executed. If you're still facing issues, don't hesitate

to seek assistance from ChatGPT, which can provide guidance and help troubleshoot any problems you may encounter.

Visualizing the CI pipeline run and test results

In the realm of CI, a successful pipeline run on Jenkins is a rich tapestry of insights, metrics, and artifacts that intertwine various tools. When Jenkins signals a green build, it’s akin to the tip of an iceberg. Beneath that success badge lies a vast ocean of information. Dive into the Jenkins console and you’ll find intricate logs detailing every step of the build, test, and deploy stages. Venture over to SonarQube and you’re greeted with a comprehensive analysis of your code quality, pointing out code smells, vulnerabilities, and technical debt, not to mention the coverage metrics that give a quantifiable measure of how robustly the code has been tested. Meanwhile, Artifactory stands as the guardian of your build artifacts. It not only hosts the resulting binaries but also meticulously manages their metadata, dependencies, and versions. This collective intelligence, sprawling across Jenkins, SonarQube, and Artifactory, is the backbone of modern software delivery, ensuring that every release is not just about pushing code, but about understanding, quality, and traceability.

Let’s start with the Jenkins dashboard. When there’s a successful multibranch pipeline job, users will be greeted with several visual indicators and pieces of information. The Jenkins dashboard elegantly encapsulates the state and details of every job, providing a clear and concise view of the health and history of each CI process, as shown here:

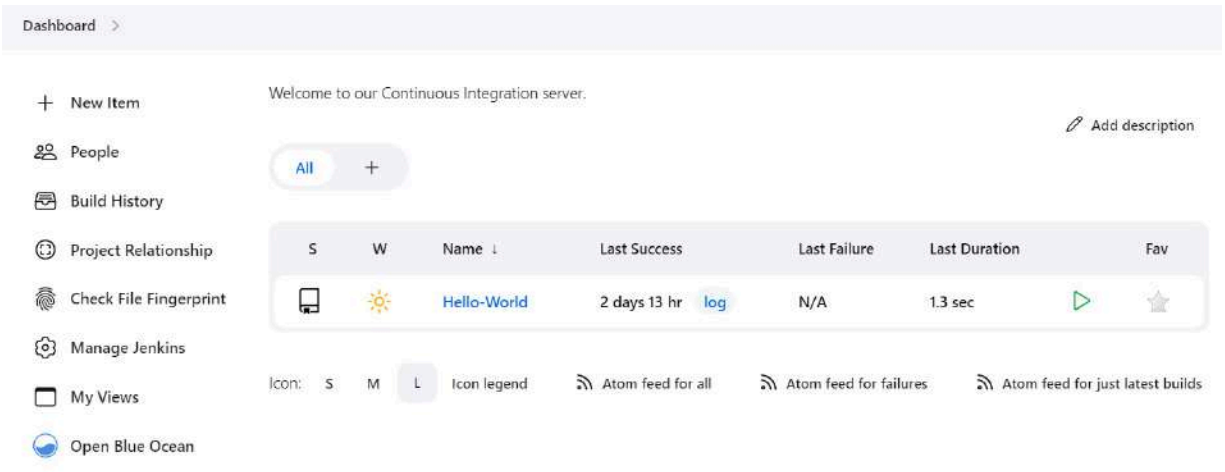


Figure 9.3 – Jenkins items listed on the Jenkins dashboard

Here’s a walk-through:

- **W:** For the *Hello-World* job, given recent successes, users will see a sunny icon, indicating the good health of the job based on recent build trends.
- **Name:** Prominently displayed, this clickable name represents your multibranch pipeline. Hovering over it might reveal more details, such as the description if one was provided during job creation.

- **Open Blue Ocean:** If Jenkins is integrated with Blue Ocean or similar plugins, users can navigate to a more visually engaging representation of the pipeline, highlighting each stage and its respective status. For our *Hello-World* job, this might be a straightforward flow, but the visual aid helps in understanding the sequence of events in the pipeline. We will come to Blue Ocean in a while.

Clicking on the Multibranch Pipeline Job will take you to its page, as shown here:

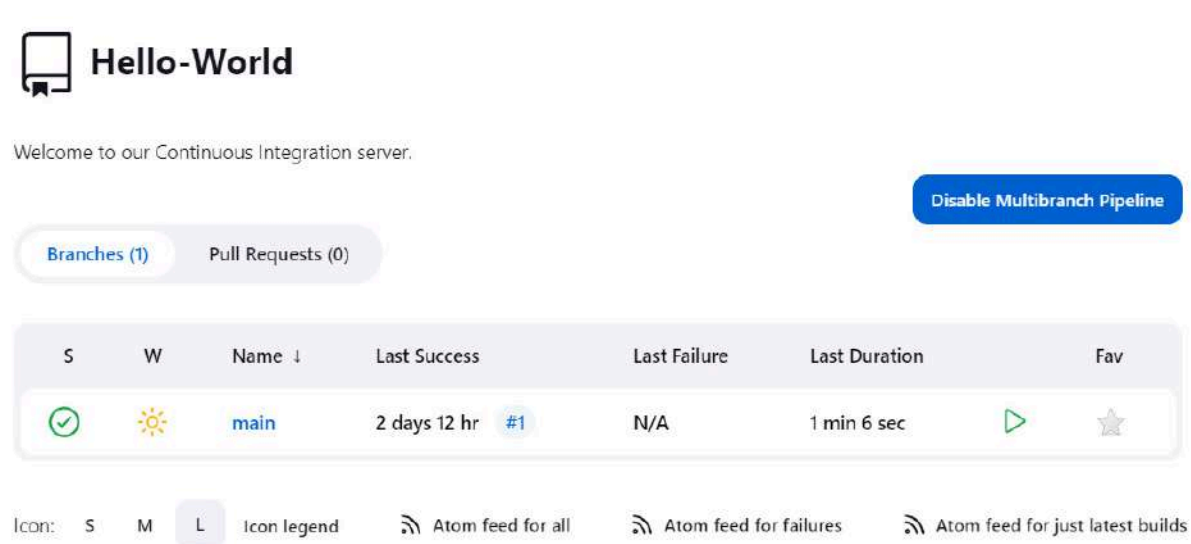


Figure 9.4 – The Multibranch Pipeline Job page

Here's a walk-through of what you might see on a Multibranch Pipeline page:

- **Branches** tab: Lists all the branches from the source code repository that have a *Jenkinsfile*. Each branch is treated as a separate pipeline. If there's a successful run, the corresponding branch will show a green status ball.
- **Pull Requests** tab: Lists all pull requests that have a corresponding *Jenkinsfile* in the source branch, meaning they are set up to be built by Jenkins. Pull requests are a common way to propose code changes, and it's crucial for CI/CD tools such as Jenkins to validate these changes before they are merged. Like the **Branches** tab, each pull request will have a status next to it.

When you click on the **main** branch pipeline of a multibranch pipeline job in Jenkins, especially one that's integrated with GitHub, SonarQube, and Artifactory, you'll be presented with a rich set of information and links related to the pipeline execution for that specific branch. See the following screenshot:

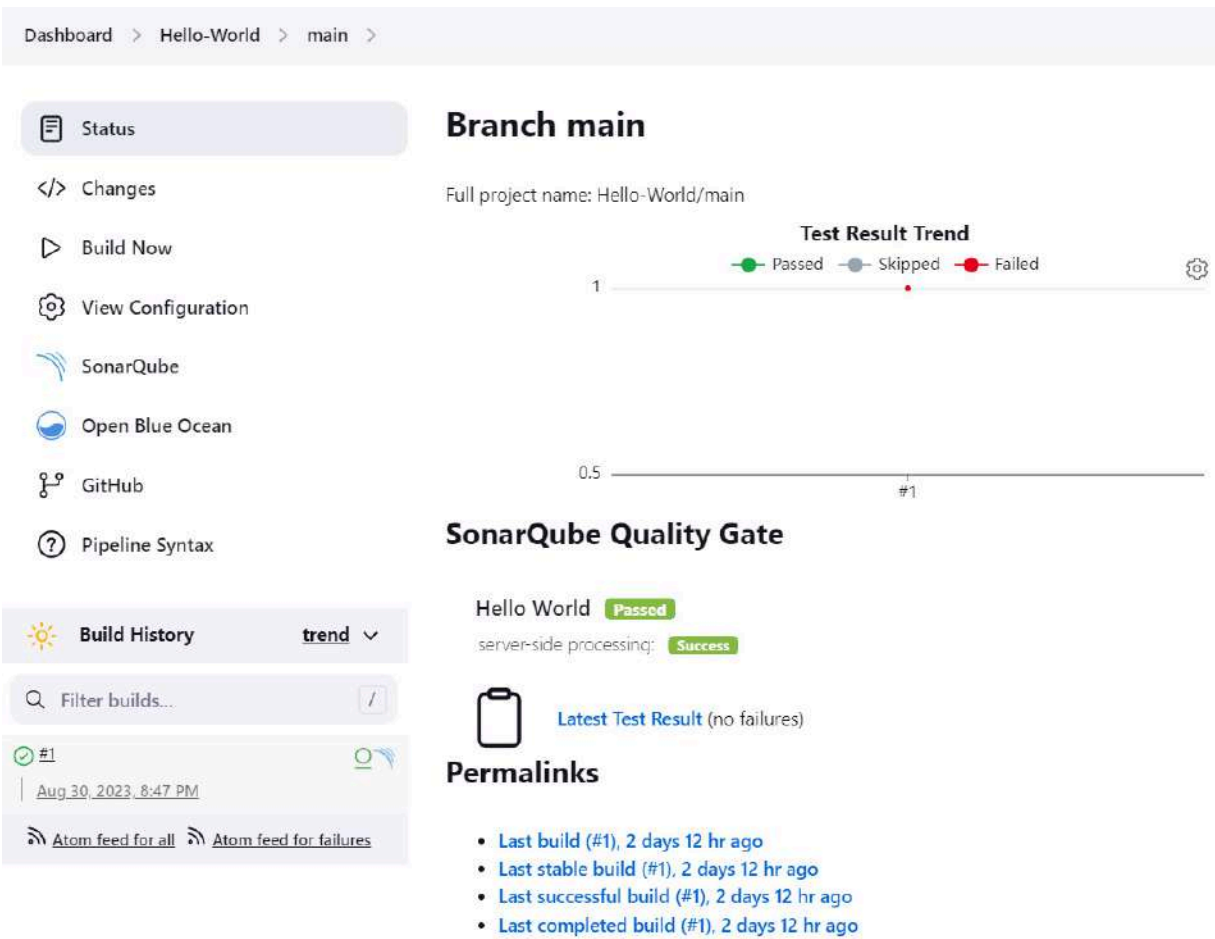


Figure 9.5 – A variety of metrics and links available on the pipeline page

Here’s a step-by-step walk-through of what one might see on the pipeline page of a specific branch:

- **Changes:** This section lists the code changes associated with the latest build. Given the GitHub integration, you might see commit messages, author details, and direct links to the commits on GitHub.
- **Test Result Trend:** This is a graphical representation capturing the evolution of test executions over time. Displayed as a line chart in Jenkins, it plots the number of passed, failed, and skipped tests across consecutive builds. This visualization aids in quickly identifying patterns, ensuring consistent code quality, and highlighting areas needing attention.
- **SonarQube Quality Gate:** SonarQube’s quality gate is a stringent set of conditions determining the health of your code base. Acting as a checkpoint, it assesses various metrics such as code coverage, technical debt, and vulnerabilities. If the code doesn’t meet the predefined criteria, the gate fails, signaling developers to address the issues before progressing further. In our case, it’s a pass.
- **SonarQube:** Given the SonarQube integration, you might find a section or a link leading to the SonarQube dashboard, showing code quality metrics, vulnerabilities, code smells, and code coverage for the **main** branch.
- **GitHub:** Here is a link to your source code repository on GitHub.
- **Build History:** A sidebar displaying a list of recent builds for the **main** branch with their respective statuses. Each entry typically has a build number and clicking on it will lead you to a more detailed view of that specific build. You can also, for each build, see a link to SonarQube analysis and build info on Artifactory.

- **Latest Test Result:** If the pipeline includes unit tests or other testing stages, you'll find a section detailing the test results. This shows the number of tests that have passed, failed, or skipped, as shown here:

Test Result



All Tests

Package	Duration	Fail	(diff)	Skip	(diff)	Pass	(diff)	Total	(diff)
(root)	20 ms	0		0		1	+1	1	+1

Figure 9.6 – Viewing unit testing results in Jenkins

Let's now move on to a Blue Ocean view by clicking on **Open Blue Ocean** in the left-hand side menu. Blue Ocean is a modern, intuitive, and user-friendly interface for Jenkins designed to make CI/CD more approachable to both new and seasoned users. Blue Ocean enhances Jenkins' visualization, emphasizing user-friendliness and clarity. In a multibranch pipeline context, this clarity becomes paramount, given the multiple branches and potential pull requests involved in the project.

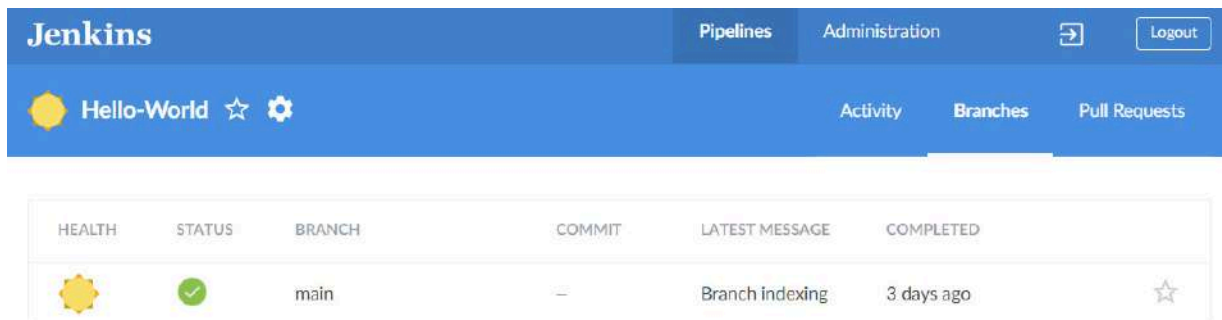


Figure 9.7 – Viewing a multibranch pipeline using the Blue Ocean interface

Here's a dive into the three primary tabs that you can see in the preceding screenshot:

- **Activity:** This gives you a snapshot of recent pipeline runs across all branches and pull requests. Here, you'll see a list of the most recent builds, each with its status, timestamp, and triggering commit message. The status is color-coded, making it easier to discern successful, failed, or unstable runs. By clicking on any entry, users can delve into the specifics of that run, exploring stages, logs, and results.
- **Branches:** This tab presents the health and status of all branches within the repository that have a corresponding **Jenkinsfile**. Like the **Activity** tab, each branch will have a status indicator next to it. A color-coded ball provides a quick status glance, with green/blue for success, red for failure, and yellow for unstable. You can view details such as the last build number, the commit

message related to that build, and how long ago that build ran. Clicking on any branch will take users to the pipeline view for that specific branch, showcasing its build stages, logs, and associated details.

- **Pull Requests:** This tab, as the name suggests, focuses on pipeline runs associated with **pull requests (PRs)** from the source control repository. All PRs with a Jenkinsfile in the source branch are listed here. As with the other tabs, a status indicator provides immediate feedback on the build's outcome for each PR. Essential information, such as the PR's title, source branch, target branch, and a snippet of the last commit message, is displayed. Direct links might also navigate users to the PR on the source control platform, such as GitHub. On selecting a particular PR, users are treated to its dedicated pipeline view, including build stages, results, and other pertinent details.

Clicking on the main branch link shown in *Figure 9.7* will take you to the pipelines overview page, as shown here:

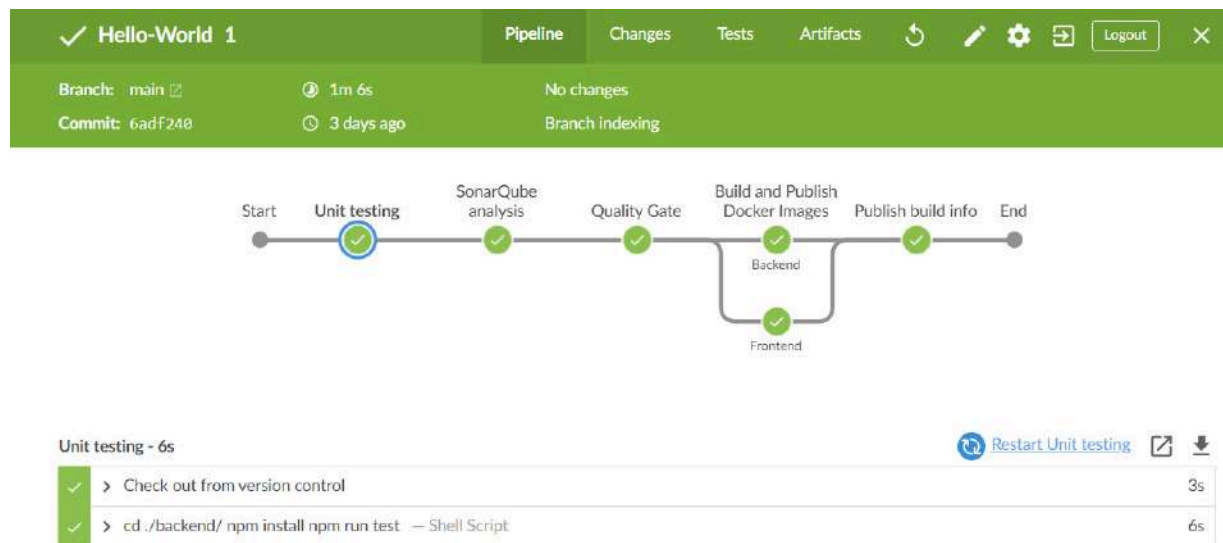


Figure 9.8 – A pipeline run overview in Jenkins Blue Ocean

Here's an explanation of what you may see on the pipeline run page:

- **Pipeline:** As a build progresses, Blue Ocean provides real-time feedback. Each stage is color-coded, allowing you to discern its status at a glance. Blue Ocean's standout feature is its clean, linear representation of the pipeline. Each stage of the pipeline is depicted as a distinct segment or block. The stages flow from left to right, offering a visual journey from the start of the pipeline to its conclusion. Clicking on any stage lets you dive into the console output for that specific segment, making it straightforward to troubleshoot issues or understand what's happening under the hood, as shown here:



Figure 9.9 – Viewing logs for each step of a pipeline stage through the Blue Ocean interface

- **Changes:** The left sidebar typically lists the changes associated with the current build. This includes commit messages, file changes, and author details. Direct links to the repository provide quick navigation to the actual changes in the source control tool, such as GitHub.
- **Tests:** Test results, including passed and failed tests, can be viewed in detail. Similarly, links to build artifacts can be accessed directly, as shown here:

✓ Hello-World 1

Pipeline Changes Tests Artifacts

🔄 ✎ ⚙️ 📄 Logout

Branch: main

🕒 1m 6s

No changes

Commit: 6adf240

🕒 3 days ago

Branch indexing

✓ All tests are passing

Nice one! All 1 tests for this pipeline are passing.

Passed - 1

✓ > Hello World API should return an array of greetings - should return an array of greetings

<1s

Figure 9.10 – Unit test results as viewed through the Blue Ocean interface

- **Artifacts:** If your pipeline generates artifacts, Blue Ocean offers clear visual indications. Uploading artifacts to the Jenkins server is not recommended.
- **Re-run:** Useful utility buttons at the top allow users to re-run pipelines.

Blue Ocean is designed to be accessible on various devices, including tablets and smartphones. This adaptability ensures that developers and CI/CD professionals can monitor and interact with their pipelines on the go.

Visualizing the SCA report and code coverage

A SonarQube analysis report is a comprehensive evaluation of your code base’s health, focusing on code quality, security vulnerabilities, and maintainability. Presented through an intuitive dashboard, the report provides key metrics including technical debt, code smells, and coverage percentage. Detailed insights into potential bugs, duplicated code blocks, and security hotspots allow development teams to pinpoint areas needing improvement. Furthermore, its quality gate status indicates whether the analyzed code meets the predefined criteria. There are multiple ways to get to the SonarQube analysis report. You can either log in to SonarQube and access the *Hello World* project or click on the SonarQube-related links available on the Jenkins pipeline page. Here is the analysis report for our Hello World application:

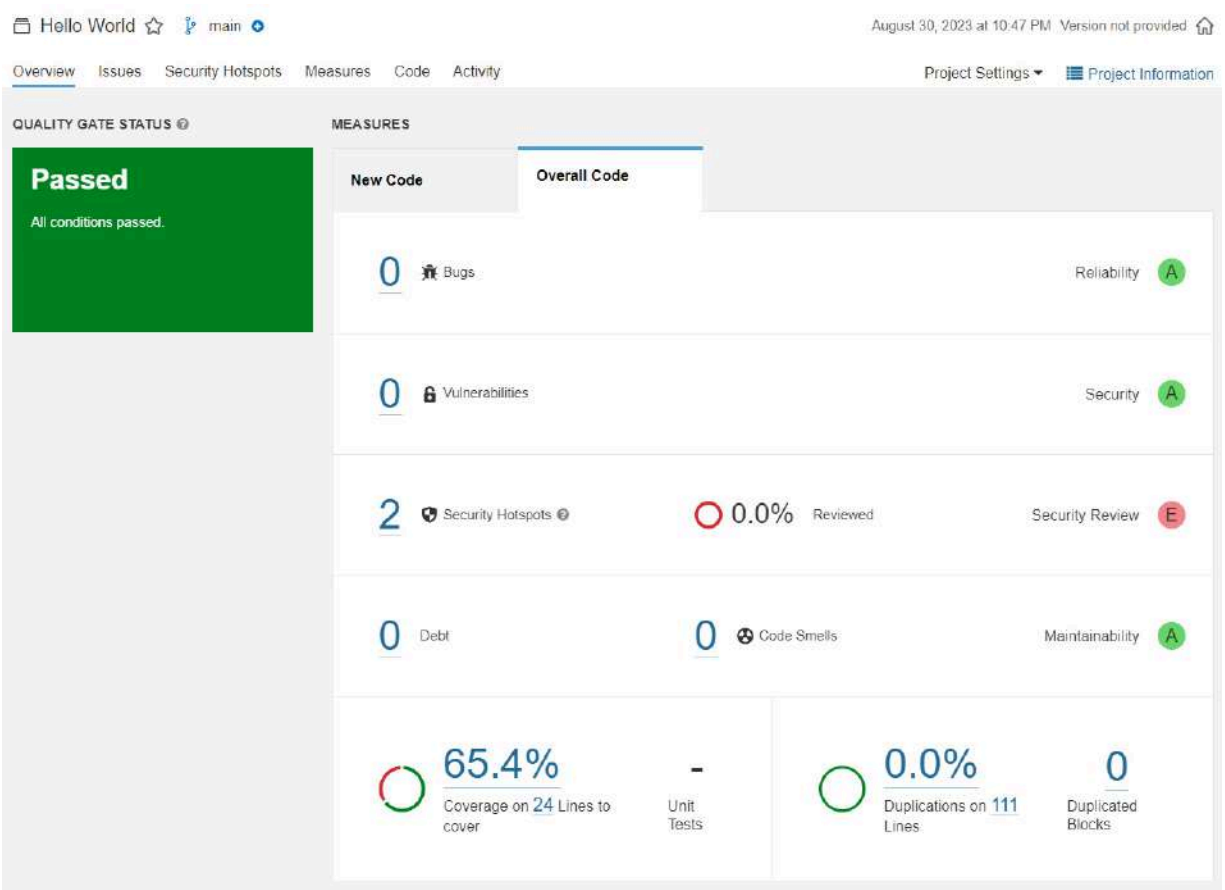


Figure 9.11 – Static code analysis report in SonarQube

This report looks pretty because our source code is extremely small and insignificant. This shows measures that are quantitative indicators providing insight into various aspects of the source code. They form a crucial part of the analysis report and help developers gauge the quality and health of a code base. Here's a deeper look:

- **Bugs:** SonarQube identifies potential bugs in the code that can affect the program's behavior. These are issues that need immediate attention.
- **Vulnerabilities and Security Hotspots:** While vulnerabilities are clear and present dangers that can be exploited, hotspots are suspicious areas that need to be reviewed for potential security threats.
- **Debt:** Represented in time (e.g., 2 days), this approximates the effort needed to fix all maintainability issues. This helps teams understand the potential future cost of quick fixes made today.
- **Code Smells:** These are patterns in the code that may not be problematic now but could lead to bigger issues in the future. They often make the code harder to understand or maintain.
- **Coverage:** This measure, usually presented as a percentage, represents the proportion of the source code that has been tested. Higher coverage typically means that more of the code base has been validated against potential errors.
- **Duplications:** SonarQube can identify blocks of duplicated code, which are often signs of poor maintainability and can lead to potential inconsistencies in the future.
- **Quality Gate Status:** These are sets of rules and conditions. Quality profiles define what should be checked (e.g., coding standards), while quality gates determine the conditions the code must meet to be considered releasable.

Let's look at the code coverage part in detail. Clicking on the code coverage percentage will take you to the coverage overview page:

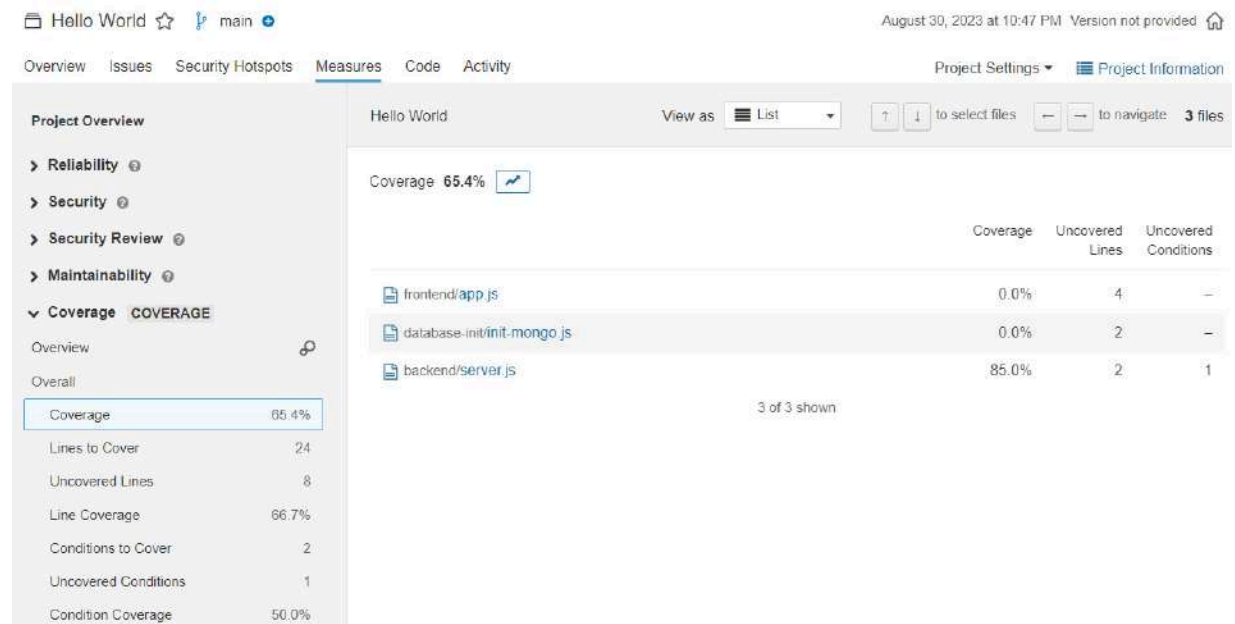


Figure 9.12 – Code coverage report on SonarQube

In the report, we can see that the overall coverage for the entire code base stands at 65.4%. While this is above the halfway mark, it indicates that roughly one-third of the code base hasn't been tested, potentially leaving vulnerabilities or bugs undiscovered. Following is a detailed breakdown:

- **Lines to Cover and Line Coverage:** Out of 24 lines that needed to be covered by tests, 8 remain uncovered, leading to a line coverage of 66.7%. This suggests that while the majority of lines have been tested, there are still several segments of code that haven't been evaluated.
- **Conditions to Cover and Condition Coverage:** For conditional statements (such as **if** and **switch**), there were two conditions to be tested, and one of them wasn't covered. This results in a condition coverage of 50%. **Condition coverage** is crucial as it indicates how thoroughly the logical paths in the code have been evaluated.

On the right-hand side, you can see file-specific details:

- **frontend/app.js:** This file has a coverage of 0.0%, meaning none of its code has been tested. Specifically, four lines remain uncovered and no conditions were reported, suggesting it might be simpler code without conditional logic.
- **database-init/init-mongo.js:** Another file with 0.0% coverage. Two lines in this initialization script are untested. It's vital to test initialization scripts as they often set the stage for subsequent operations.
- **backend/server.js:** In contrast to the previous two files, **server.js** boasts an impressive coverage of 85.0%. However, there are still two lines and one condition that remain untested. Given that this is backend code (typically consisting of crucial business logic), it's reassuring to see high coverage, but the uncovered lines and condition warrant attention.

In conclusion, while the overall code coverage is commendable, there are areas, especially entire files such as **app.js** and **init-mongo.js**, that need more attention. Ensuring comprehensive testing across all files and conditions is crucial for maintaining robust and reliable software.

Visualizing build artifacts

Artifactory, when integrated with a Jenkins CI pipeline, offers a seamless experience for handling Docker images. It provides a centralized, secure repository for storing and managing these images, ensuring version control and traceability. Additionally, Artifactory's efficient metadata management means each Docker image is accompanied by vital information, streamlining troubleshooting and version comparisons.

Begin by logging into the Artifactory server to explore its offerings. Once you're logged in, from the dashboard, navigate to the **Application** tab via the left sidebar. Within this tab, expand the **Artifactory** section. This will reveal four options: **Release Bundles**, **Packages**, **Builds**, and **Artifacts**. Let's explore them, starting with **Packages**.

The **Packages** section showcases distinct entities or software components stored in Artifactory. Think of a package as a container or a grouping mechanism, such as a Docker image, an npm package, or a Maven artifact. It not only holds the actual software artifact but also its associated versions, enabling developers to track and manage different iterations of a component.

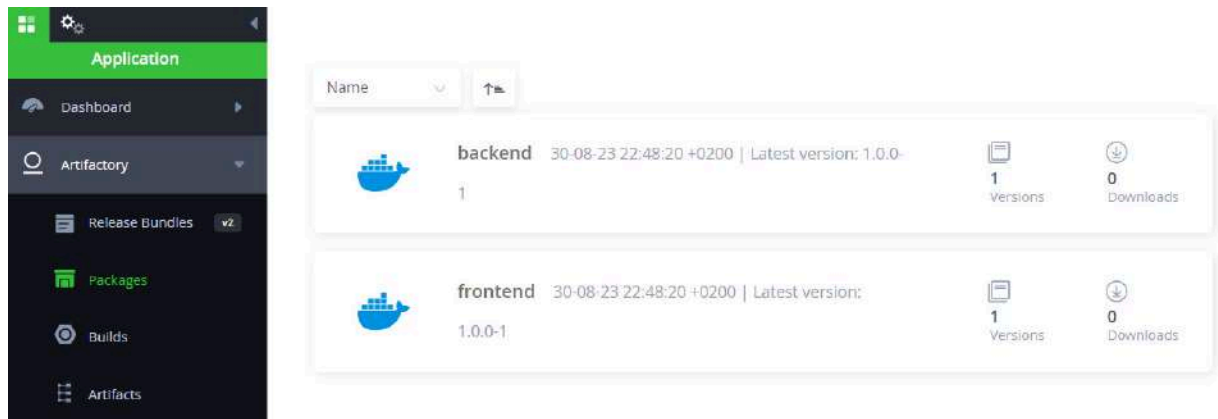


Figure 9.13 – Artifactory's Packages section

Artifactory's **Packages** page is the dedicated space where you can view and manage your repository's artifacts. In the context of Docker images, each package represents a distinct Docker image and its respective versions. Some of the items you will find on the **Packages** page are as follows:

- You'll find entries for both frontend and backend Docker images, signifying successful pushes to Artifactory.
- **Versions:** Each image might have multiple versions or tags associated with it. These versions could correspond to various development stages, features, or fixes. The **Packages** page will list these versions chronologically, allowing easy navigation between old and new image states. Since we have pushed just once, we have only one version.
- **Downloads:** Next to each Docker image and its versions, you'll typically find action buttons or icons. These can allow you to view more details, download the image, delete a particular version, or even promote a version to another repository (from dev to production, for example).
- **Search Packages and Filter:** At the very top of the page, to aid navigation, especially in repositories with numerous packages, the **Packages** page provides search and filter functionalities. You can narrow down the view by typing in the package name, filtering by package type (Docker, in this case), or using other criteria.

Clicking on a specific package or its version opens a detailed view. Here, you'll find extended metadata, such as the Docker layers, image size, creation date, involved environment variables, and any other custom properties or annotations.

The Artifactory **Packages** page, therefore, serves as a central hub for all your artifact management needs, providing transparency, traceability, and control over the Docker images and their lifecycles.

Now let's look at the **Builds** page. Here, Artifactory chronicles the build processes that result in the creation of the packages. By integrating with CI tools such as Jenkins, the **Builds** section captures invaluable metadata about each build, such as the timestamp, environment variables, and dependencies. This ensures that every artifact can be traced back to its origin, providing an avenue for debugging, auditing, and replicating builds.

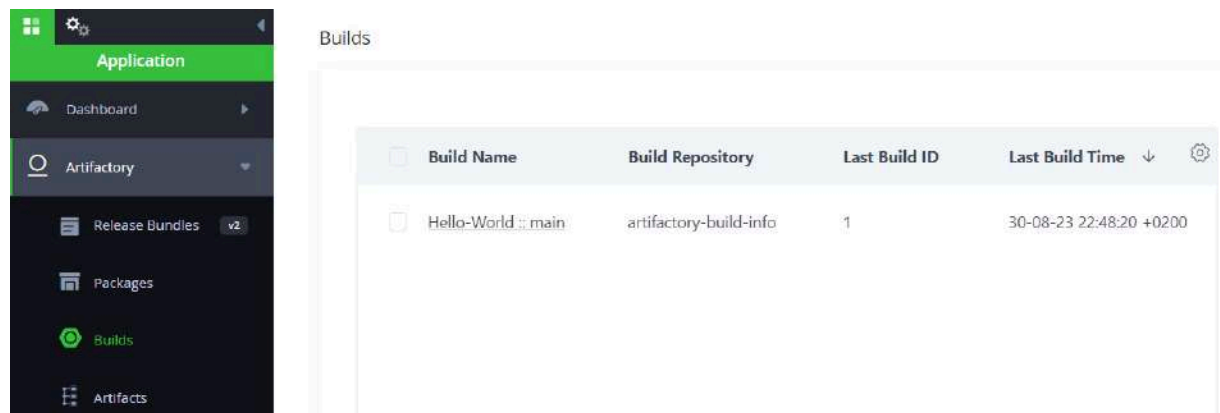


Figure 9.14 – Builds section in Artifactory

Since you've published the build info from Jenkins, Artifactory will associate this metadata with the respective Docker images. It offers traceability, letting you understand the lineage of a Docker image – when it was built, by which Jenkins job, and using what dependencies. By attaching rich metadata to artifacts, development teams can gain a holistic view of the build process. For instance, if an artifact causes issues in a staging environment, the build info can provide immediate insights into what changed, who made the changes, and under what circumstances. This transparency not only accelerates troubleshooting but also enhances collaboration between teams, making the software delivery process more agile and resilient.

Next, let's look at the repository view by clicking on the **Artifacts** section. Delving deeper into the actual files or binaries, the **Artifacts** section is where one finds the raw outputs of build processes. Whether it's a `.jar` file from a Java build, a `.whl` file from a Python package, or a Docker container, this section stores, organizes, and lets you manage these binaries across various repositories, be it `local`, `remote`, or `virtual`.

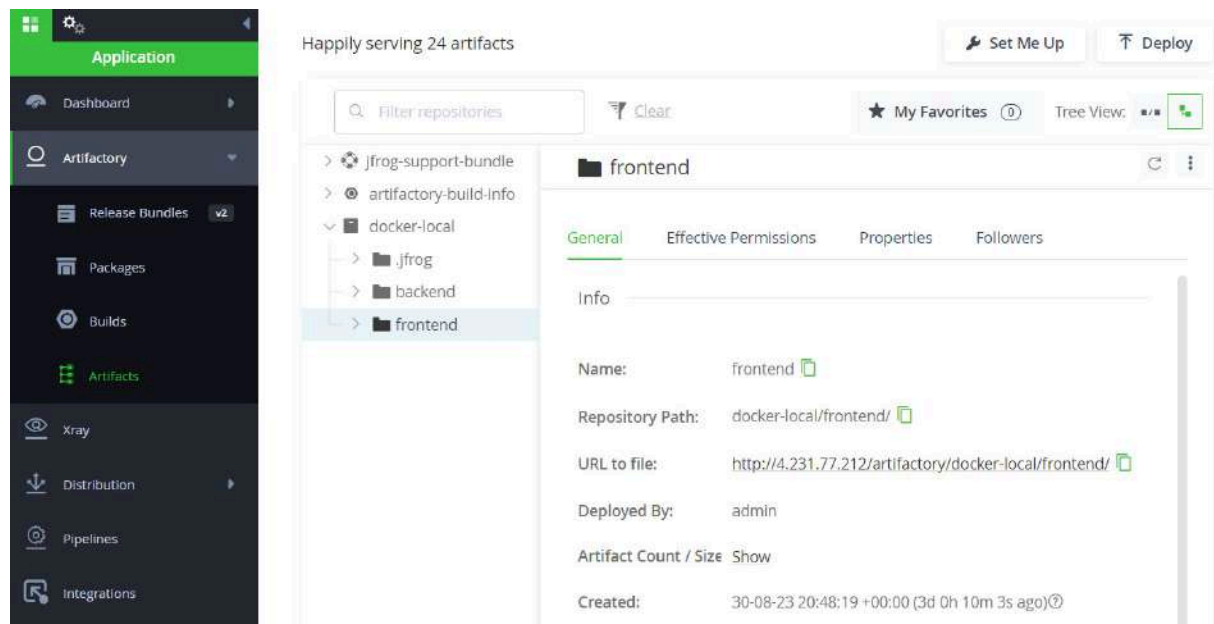


Figure 9.15 – Artifacts section in Artifactory

Let's discuss this metadata in more detail:

- **Name:** The repository is aptly named **frontend**, which suggests it's dedicated to the frontend component of your application. This clear naming convention makes it easier for developers and CI/CD pipelines to target specific components.
- **Repository Path:** The **docker-local/frontend/** path indicates this repository is locally stored within Artifactory and likely holds Docker images, given the **docker-local** designation. This specificity ensures the appropriate segregation of different types of artifacts.
- **URL to file:** The URL is a direct link to access the **frontend** repository. It can be utilized by developers, CI/CD tools, or even deployment scripts to fetch or interact with the artifacts stored in this repository.
- **Deployed By:** The **admin** user seems to have set up or last deployed to this repository. It's crucial to track this information for audit trails and understanding any changes or uploads made to the repository.
- **Artifact Count / Size:** The **Show** placeholder suggests the actual number and size of artifacts that can be viewed on demand. This metric can provide insights into the volume and storage implications of the repository.
- **Created:** The repository was created on **30-08-23 at 20:48:19 +00:00**. This timestamp is essential for understanding the lifecycle of the repository and tracking its age or the timing of major events associated with it.

Such detailed views into repository metrics and properties offer teams clear insights, ensuring efficient management and oversight of artifacts within Artifactory. Together, these sections form the bedrock of Artifactory's functionality, enabling organizations to have a bird's-eye view of their entire software artifact lifecycle, from creation to storage and deployment.

Summary

In this chapter, we navigated the practical aspects of writing CI code — from configuring pipeline agents, running unit tests, static code analyses, and ensuring code quality with SonarQube, to the nuances of building code and publishing artifacts. The role of Kubernetes in the CI process was also highlighted. A significant milestone in our journey was actualizing the CI process. By adding the pipeline code to our source code repository, we were primed to witness CI in action. This culminated in creating a multi-branch pipeline job in Jenkins, offering an insightful look into the mechanics of CI in real time.

Lastly, the visualization aspects enriched our understanding manifold. Using Jenkins' interface, we learned how to visually interpret unit test results, static code analysis reports, code coverage data, and the meticulous details of build artifacts. In essence, this chapter provided the practical know-how of implementing and visualizing a simple CI pipeline using Jenkins. Whether you're a novice or an experienced professional, the insights offered here serve as the cornerstone of building an even more complex, efficient, and scalable CI pipeline.

In the upcoming chapter, we'll evolve our Continuous Integration pipeline into a full-fledged Continuous Deployment pipeline. This will encompass deploying to a staging environment, conducting additional tests, and finally, rolling out to production.

Questions

1. What is the primary function of a multibranch pipeline in Jenkins?
 - A. It allows for parallel execution of jobs across multiple nodes.
 - B. It enables automated build and test processes for each branch in a repository, treating them as separate projects.
 - C. It integrates code from multiple branches into the main branch after successful testing.
 - D. It provides a way to execute different Jenkinsfiles depending on the branch that triggered the build.
2. What is a key feature of Jenkins Blue Ocean?
 - A. It is a machine learning module for predictive analysis in Jenkins.
 - B. It provides a simplified and more user-friendly interface for creating and visualizing Jenkins pipelines.
 - C. It increases the processing speed of Jenkins by optimizing underlying code.
 - D. It is a security add-on for Jenkins to enhance firewall and intrusion detection.
3. What does SonarQube primarily use measures for?
 - A. To assess the performance and speed of the code execution.
 - B. To track and analyze the technical debt of the code base.
 - C. To monitor the runtime behavior and efficiency of applications.

- D. To encrypt code for secure storage in version control systems.
4. Which of the following is NOT a metric used by SonarQube?
- A. Code coverage
 - B. Cyclomatic complexity
 - C. Memory usage
 - D. Duplicated lines density
5. Choose whether this statement is **True** or **False**: The purpose of publishing build information to Artifactory from Jenkins is to track and store the history of deployments and artifacts for auditing and rollback purposes.

Answers

1. Correct answer: **B**

Explanation of incorrect answer choices:

A. Parallel execution of jobs across multiple nodes: While Jenkins can execute jobs in parallel, this is not specific to multibranch pipelines. Multibranch pipelines are more focused on handling different branches rather than parallel execution.

C. Integrating code from multiple branches into the main branch: This is a description of a **Continuous Integration (CI)** process, specifically a task typically done by a version control system, such as merging in Git. Multibranch pipelines facilitate the CI process but don't perform code integration.

D. Executing different Jenkinsfiles depending on the branch: While it's true that multibranch pipelines can execute different Jenkinsfiles based on the branch, this is not their primary function. The main purpose is to automate builds and tests for each branch, which might include using different Jenkinsfiles, but the focus is on branch-specific automation.

2. Correct answer: **B**

Explanation of incorrect answer choices:

A. Machine learning module for predictive analysis: Blue Ocean is not a machine learning tool. It focuses on enhancing the user interface and experience in Jenkins, not on providing predictive analytics capabilities.

C. Increasing processing speed by optimizing underlying code: While performance is important, Blue Ocean's primary aim is not to optimize the processing speed of Jenkins but to improve its user interface and user experience, particularly in visualizing and creating pipelines.

D. Security add-on for enhanced firewall and intrusion detection: Blue Ocean does not serve as a security add-on. Its focus is on the UI/UX aspect of Jenkins, specifically for pipeline creation and visualization, and not on security features such as firewall or intrusion detection.

3. Correct answer: **B**

Explanation of incorrect answer choices:

A. Assessing the performance and speed of code execution: SonarQube's measures are not designed to assess the runtime performance or execution speed of code. Instead, they are focused on code quality aspects such as maintainability, reliability, and security.

C. Monitoring the runtime behavior and efficiency of applications: SonarQube measures do not monitor the runtime behavior or efficiency of applications. It is a static code analysis tool, meaning it analyzes source code at rest, not in execution.

D. Encrypting code for secure storage in version control systems: SonarQube is not involved in encrypting code. Its purpose is to analyze and measure code quality, not to secure code in version control systems.

4. Correct answer: **C**

Explanation:

SonarQube does not measure the memory usage of the application. SonarQube focuses on static code analysis, assessing aspects of code quality such as maintainability, security, and reliability. Memory usage is a dynamic runtime metric, typically assessed during performance testing or monitoring, not static code analysis.

5. Correct answer: **True**

Explanation:

Publishing build information to Artifactory from Jenkins is indeed done to track and store the history of deployments and the artifacts used. This information includes details about the build environment, the dependencies used, the artifacts produced, and other build-related metadata. Having this data in Artifactory allows for efficient tracking of each build's lifecycle, facilitates audits, and aids in rollback scenarios if needed, ensuring a reliable and traceable continuous integration and deployment process.

Further reading

If you wish to know more about some of the concepts that are not covered in depth in this chapter, or are out of the scope of this book but relevant, we are happy to provide you with some references:

- [1] SonarQube (9.9 LTS) analysis parameters: <https://docs.sonarsource.com/sonarqube/9.9/analyzing-source-code/analysis-parameters/>
- [2] Build images in Kubernetes using Kaniko: <https://github.com/GoogleContainerTools/kaniko>

Part 4: Crafting the CD Pipeline

In the final leg of our journey, we enter the high-stakes arena of **Continuous Deployment (CD)**, where our CI pipeline's precision engineering meets the automated prowess of a full-fledged release mechanism. This is the stage at which every earlier calibration, line of code, and integrated tool culminates in a streamlined, self-operating production line that not only builds but also delivers software to the end user without a hitch. Here, we'll look at a code deployment strategy, automated testing, and the smooth transition from staging to production settings. Our plan expands to include cutting-edge GitOps worlds, leveraging tools such as Argo CD to create systems that are as durable as they are responsive.

The inventiveness does not end there. We will investigate AI-powered tools such as GitHub Copilot to make our pipeline scripts smarter, more efficient, and less prone to errors. Jenkins shared libraries provide interchangeable and reusable elements, easing the management of various pipelines. To keep our systems optimal and healthy, we will implement ways to deal with the byproducts of our process, such as eliminating outdated builds. Furthermore, we will strive for security and dependability, not just functionality. We will ensure that our automated release line not only delivers but also protects by integrating tools such as JFrog Xray, finding vulnerabilities early, and keeping our end product secure and sound.

This part has the following chapters:

- [Chapter 10](#), *Planning for Continuous Deployment*
- [Chapter 11](#), *Writing Your First CD Pipeline*
- [Chapter 12](#), *Enhancing Your CI/CD Pipelines*

Planning for Continuous Deployment

In the previous two chapters, we took our initial steps into the world of **Continuous Integration (CI)** by setting up a CI pipeline in Jenkins. By automating the unit testing, **Static Code Analysis (SCA)**, and build phases, we learned how we can significantly reduce the manual work involved in these stages and introduce a reliable way to catch issues early on. While CI handles the integration part of the development life cycle, it's merely one side of the coin. The other side is **Continuous Deployment (CD)**, which automates the delivery of code from the development environment into the production environment.

The essence of CD is to make deployments predictable, safe, and reversible. Automating this process not only minimizes manual error but also speeds up the software delivery process, enabling teams to react more swiftly to market demands. Having said that, in this chapter, we will extend our existing Jenkins CI pipeline to incorporate CD, taking our automation a step further.

As such, we will cover the following main topics in this chapter:

- Understanding Continuous Deployment
- Planning Continuous Deployment

So, let's delve into the realm of Continuous Deployment with Jenkins, enhancing the efficiency, reliability, and speed of our software delivery process.

Technical requirements

Before you begin, make sure that you have a working CI setup and a CI pipeline that pushes Docker images to Artifactory. Additionally, you need the following tools installed on your local development environment:

- An IDE such as Visual Studio Code or anything else you prefer.
- Git.
- The Azure CLI to interact with the Azure cloud.
- kubectl, the Kubernetes command-line tool.
- Helm to deploy workloads on the Kubernetes cluster.
- The Argo CD CLI to configure and interact with your Argo CD server. Refer to the Argo CD documentation to install its CLI (see [1] in the *Further reading* section at the end of the chapter). You can download the Argo CD CLI from <https://github.com/argoproj/argo-cd/releases/latest> for Windows.

For the source code, please refer to this chapter’s folder in the GitHub repository of the book:
https://github.com/PacktPublishing/Learning-Continuous-Integration-with-Jenkins_Third-Edition.

Understanding Continuous Deployment

CD is a software development practice where code changes are automatically built, tested, and deployed to production without manual intervention, provided they pass all the automated tests that are part of your QA strategy. A CD pipeline always includes all stages of a CI pipeline; it’s an extension of CI. While CI deals with automating the building and testing part of the development cycle, CD takes care of the extensive testing and deployment aspects, as depicted in the following diagram.



Figure 10.1 – Continuous Deployment

In a typical CD pipeline, every change that passes all stages of your production pipeline is deployed to production automatically, without any human intervention. This is an excellent way to accelerate the feedback loop with your customers and take pressure off the team as there is no longer a “release day” per se.

There are mainly two strategies to implement a CD pipeline for your product. These strategies are branch-specific and seldom discussed. Let’s look at the one that we will be using in our chapter.

Universal CI with master-only CD

Running CI pipelines for every branch while limiting CD pipelines to the master branch is a common practice in many development workflows. This approach seeks to balance thorough code validation with the caution needed for deployments, especially to production environments.

Some of the advantages of this approach are as follows:

- **Isolation:** Running CI on feature branches allows developers to test changes in isolation from the master branch, reducing the risk of introducing bugs into the main code base.
- **Master stability:** Restricting CD to the master branch helps ensure that only well-reviewed and approved code gets deployed to staging and/or production environments.
- **Resource optimization:** Running full CD pipelines for every branch could consume significant computational resources. Limiting this to the master branch optimizes resource usage.

The following are some design considerations to keep in mind when using this approach:

- **Up-to-date feature branches:** Developers must keep their branches up to date with the master branch to ensure that the CI pipeline is testing the code in an environment that closely mimics the production one. This minimizes the risk of encountering the “it works on my machine” issue.
- **Review process:** Since the master branch is tightly controlled and triggers deployment, the pull request review process should be robust enough to prevent any problematic code from merging into the master branch.

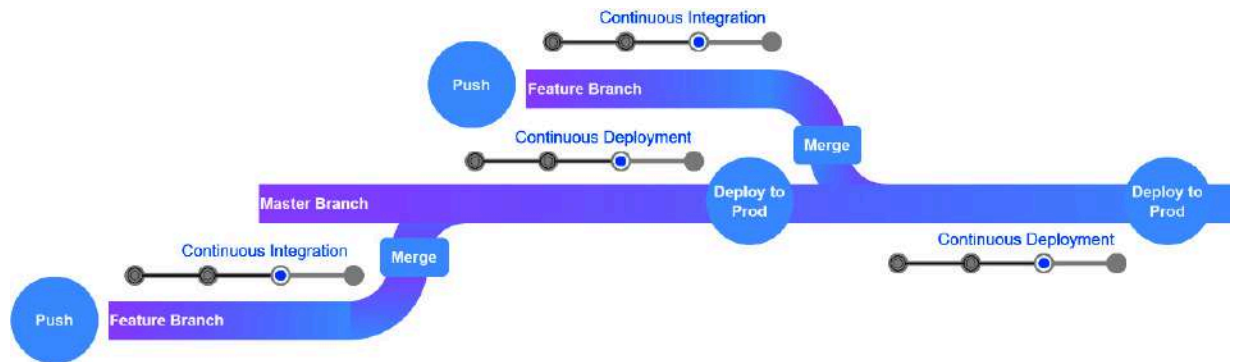


Figure 10.2 – Universal CI with master-only CD strategy

Here is an explanation of the preceding example workflow:

1. **Feature development:** Developers work on feature branches, each of which runs its own CI pipeline upon every push.
2. **CI pipeline:** The merge triggers the CI pipeline, which includes CI steps for building and testing.
3. **Code review:** Once development is complete, a pull request is opened. CI status checks must pass, and code reviews must be approved before merging into the master branch.
4. **Merge to master:** After approval, the code is merged into the master branch.
5. **CD pipeline:** The merge triggers the CD pipeline, which includes CI steps for building and testing, followed by deployment steps to staging, running longer tests, and finally, deployment to production.

By running CD only on the main branch, you can maintain a fast-paced development cycle without sacrificing the stability and reliability of your production environment. We will learn how to implement this *Universal CI with Master-Only CD* strategy in the current chapter. Now, though, let’s look at the other popular but less used strategy.

Universal CI and CD

Running a CD pipeline for every branch is a less common but certainly viable approach, especially in some specific use cases. This strategy comes with its own sets of advantages and considerations.

Some of the advantages of this approach are as follows:

- **Full automation:** This approach provides a fully automated pathway from code commit to deployment (in pre-production), making it faster and easier to get features, bug fixes, and other changes into live environments

- **Real-world testing:** Deploying every branch automatically lets you test changes in an environment that closely mirrors production, which can reveal issues that might not be evident in local or isolated testing
- **Feature flags and A/B testing:** If you're using feature flags or similar techniques, deploying every branch can make it easier to perform A/B tests or roll out features incrementally
- **Fast feedback:** Developers get immediate feedback on how changes perform in a live environment, speeding up the development and review cycle

The following are some design considerations to keep in mind when using this approach:

- **Resource utilization and cost:** Deploying every branch automatically can consume a significant amount of computational and human resources, depending on how many branches you have and how complex your deployment process is. More deployments generally mean more environments, which could lead to higher costs for infrastructure.
- **Complexity:** Managing multiple live versions of an application can become complex, especially when it comes to database schemas, third-party services, and other shared resources.
- **Security risks:** Automatically deploying all branches could expose unfinished or unreviewed code, which might present security risks.
- **Branch naming and organization:** A strict branch-naming convention might be necessary to manage deployments effectively. For example, you might restrict deployments to branches that match certain patterns to mitigate some of the risks and complexities.

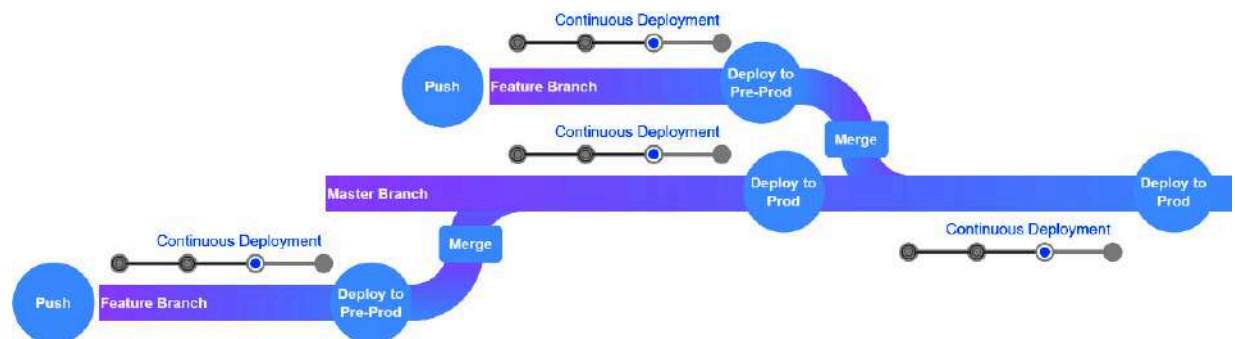


Figure 10.3 – Universal CI and CD strategy

Here is an explanation of the preceding example workflow:

1. **Feature development:** Developers work on feature branches, each of which runs its own CD pipeline upon every push.
2. **CD pipeline:** If the CD steps are successful, the pipeline proceeds to deploy the code to a production-like (pre-production) environment specific to that branch.
3. **Monitoring and feedback:** Post-deployment metrics and feedback are monitored to gauge the impact and status of the changes.
4. **Merge or discard:** Based on feedback and metrics, the branch is either merged into a master branch or discarded.
5. **CD pipeline:** The merge triggers the CD pipeline on the master branch, which finally deploys to production.
6. **Cleanup:** Environments related to discarded or merged branches may be torn down to free up resources.

While a CD pipeline for every branch can accelerate development and offer immediate real-world feedback, it does add complexity and requires rigorous controls to manage effectively.

Understanding GitOps

GitOps is an approach to infrastructure and application management that emphasizes the use of Git repositories as the “single source of truth” for declarative infrastructure and applications. In the GitOps model, the desired state of your system is defined in a Git repository, and changes to this desired state are made through Git pull requests. Automated processes then roll out the changes to the actual system to align it with the desired state defined in Git.

The key advantages of GitOps include the following:

- **Version control:** Since all configurations are stored in a Git repository (GitHub, GitLab, Azure DevOps, etc.), you get all the benefits of version control, such as auditing, rollback capabilities, collaboration, and peer review.
- **Automatic synchronization:** Automated tools such as Argo CD ensure that the real-world state of the system matches the desired state defined in the Git repository. If they diverge, the system self-heals by re-aligning itself to the desired state.
- **Security and compliance:** Git’s native features such as signing and auditing combined with additional tools can add multiple layers of security.
- **Increased productivity:** GitOps simplifies both development and operational tasks, reducing the manual effort needed for provisioning and managing infrastructure and applications.

But why are we discussing GitOps? It’s because, in this chapter, we are going to follow GitOps using Argo CD, a powerful approach for managing deployments of containerized applications to both staging and production environments in a Kubernetes cluster. Here is a basic workflow for GitOps with Argo CD:

1. **Initialization:** Set up a Git repository to hold your application’s Kubernetes manifests. This could include Kubernetes Deployment, Service, and ConfigMap YAML files, or more complex configurations using Helm charts. This repository is different from your source code repository.
2. **Environment branches:** To manage both staging and production, you could have separate branches or folders within your Git repository, each representing one environment.
3. **Argo CD setup:** Install Argo CD in your Kubernetes cluster and configure it to sync with your Git repository. Define two applications in Argo CD, one for each environment (staging and production).
4. **Automated sync:** Configure Argo CD to automatically sync the cluster state with the Git repository’s desired state.
5. **CI/CD integration:** Integrate Argo CD with your existing CI/CD pipeline. When your CI pipeline successfully builds and tests your code, it can update the manifests or Helm charts in the Git repository. Argo CD will detect the changes on the branches or folders of the Git repository and deploy the changes it to its corresponding environment.
6. **Rollbacks and history:** If anything goes wrong, you can easily roll back to a previous version using Git’s version history. Argo CD will automatically sync the cluster to the previous state.
7. **Monitoring and alerting:** Argo CD provides detailed logging and can be configured to send alerts for failed synchronizations, helping you maintain the system’s stability.

Using Argo CD for GitOps in this way allows you to leverage the best practices from both Git and Kubernetes, providing a robust, version-controlled way to manage your application deployments in a

staging and production setup. Now that we have a clear grasp of the basics, let's move forward with planning the implementation of CD.

Planning Continuous Deployment

Let's commence our planning for Continuous Deployment by initially examining a comprehensive CD design. Following this, we will explore the Helm chart of our web application, which is slated for deployment in both staging and production environments. Subsequent steps involve readying the requisite elements on the AKS cluster, coupled with the installation and configuration of Argo CD to facilitate seamless deployment. The creation of applications within Argo CD and the integration with Jenkins will also be executed as part of this comprehensive planning and implementation process. Let's dive in together and get things rolling!

High-level Continuous Deployment design

In the current architecture, Jenkins serves as the primary CD engine. The CD pipeline, orchestrated by Jenkins, automates the building and testing of the application. Upon successful build, Docker images are constructed and securely stored in Artifactory, a robust artifact storage solution, ensuring versioning and traceability of application containers. Here, the CI part ends. For deployment, GitOps principles are applied using Argo CD, a Kubernetes-native deployment tool. Instead of using Kubernetes manifests, a separate GitHub repository is maintained exclusively for a Helm chart of the web application. Helm, acting as the package manager for Kubernetes, simplifies application deployment, scaling, and management. When changes are merged or pushed to this repository, Argo CD detects these alterations and synchronizes the Kubernetes cluster to match the repository's state, ensuring consistent deployments to both staging and production environments. Post-deployment, performance validation is achieved through k6 [2], a modern load testing tool. This ensures that the web application not only functions correctly but also meets specified performance benchmarks. This cohesive blend of tools and practices offers a streamlined, version-controlled, and performance-verified deployment pipeline, maximizing efficiency and reliability for the web application's life cycle.

A NOTE ON TESTING COMPLEXITY

*It's worth acknowledging that Continuous Testing, as part of CD, is a broad and complex field that extends well beyond the scope of this chapter. In a real-world scenario, you'd likely encounter various types of **User Acceptance Testing (UAT)**, multiple layers of performance testing, security testing, and much more to ensure that your application is robust, secure, and capable of handling the demands of production use. However, for the sake of focus and simplicity, this chapter will feature a streamlined approach to testing. We will be using k6 to conduct a single, straightforward performance test. This*

should not be interpreted as an exhaustive or definitive QA strategy but rather as an introduction to incorporating automated performance testing into your Continuous Deployment pipeline.

While this example serves as a solid starting point, it's important to remember that in a real QA/staging environment, you would typically employ a more comprehensive suite of tests to rigorously validate the functionality, performance, and security of your application.

Here's a bird's-eye view of the CD process for the master/main branch:

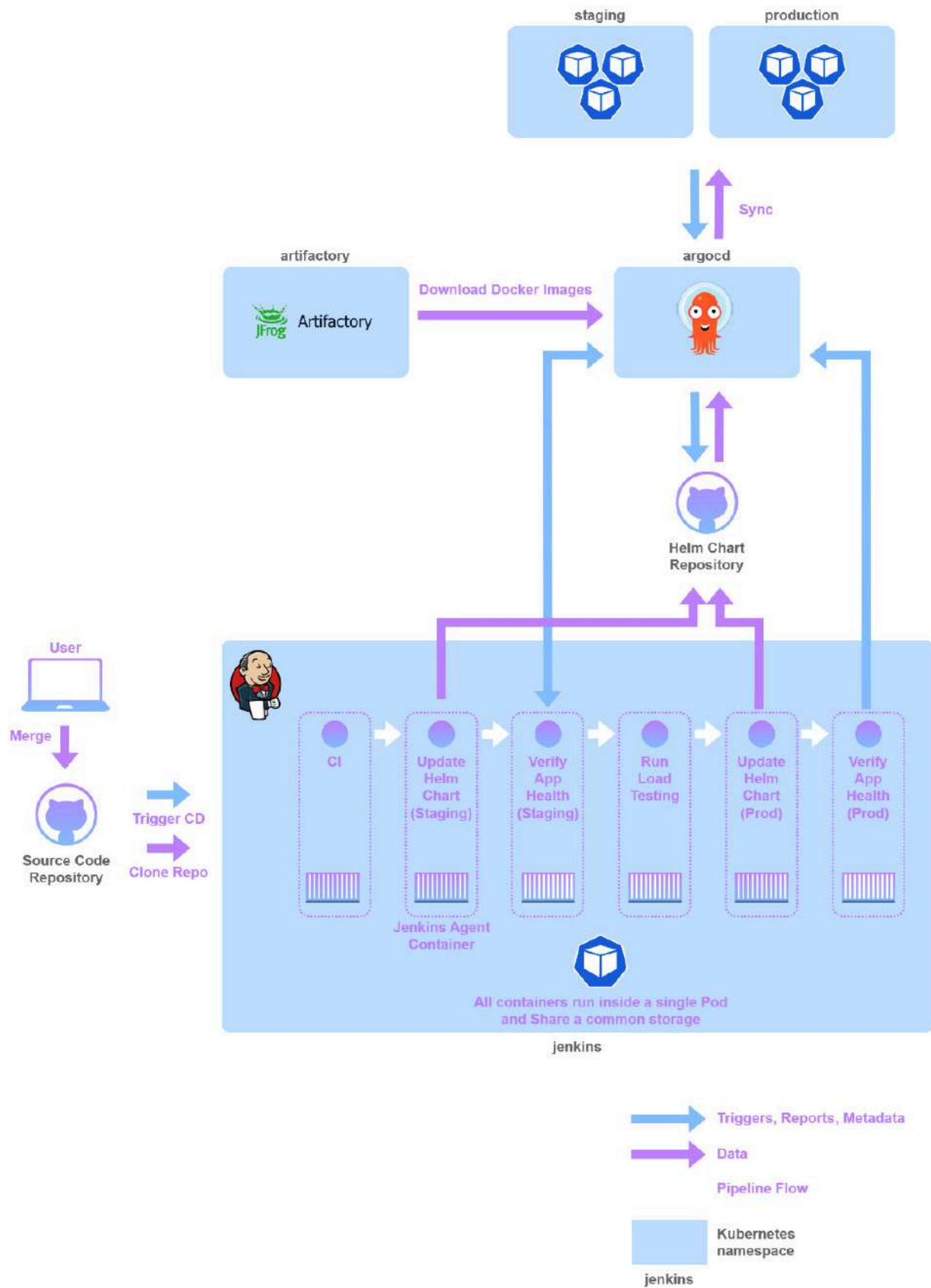


Figure 10.4 – Continuous deployment high-level design

Let us break down the preceding CD workflow:

- **CI on feature branch:** Upon successful CI and peer review, the developer merges their feature branch to the master branch on the source code repository. With webhooks in place, GitHub will trigger a CD pipeline in Jenkins for the master branch and the CD pipeline will follow.
- **CI:** The CD pipeline runs all stages that are part of the CI pipeline.
- **Update Helm chart (staging):** Clone the Helm chart repository, update the staging-specific `values.yaml` file, and push the changes to GitHub.
- **Sync web app (staging):** Argo CD automatically updates the web application running in the staging namespace with the new Docker images available through the Helm chart.
- **Check web app status (staging):** Jenkins, using the Argo CD API commands, checks the sync and health status of the web application running in the staging namespace.
- **Run performance testing:** If the web application is found to be running fine in staging, Jenkins will proceed to run some performance testing using k6 on the web application running in staging. The results of the testing are published inside the build logs.
- **Update Helm chart (production):** If performance testing thresholds pass, Jenkins will again clone the Helm chart repository, this time updating the production-specific `values.yaml` file and pushing the changes to GitHub.
- **Sync web app (production):** Argo CD automatically updates the web application running in the production namespace with the new Docker images available through the Helm chart.
- **Check web app status (production):** Jenkins, using the Argo CD API commands, checks the sync and health status of the web application running in the production namespace.

By designing your CD pipeline in this structured manner, you ensure that the code meets the quality criteria, and any changes can be swiftly and securely propagated through to your environments.

Exploring the Helm chart of our web application

Helm is a package manager for Kubernetes. With Helm, you can define, install, and upgrade even the most complex Kubernetes applications. Helm uses charts, which are packages of preconfigured Kubernetes resources. Let's understand the Helm chart for our web application. The Helm chart can be downloaded from the GitHub repository containing the source code for the chapter. We'll start by examining the Helm chart files in the following sequence:

- **Chart.yaml:** This provides metadata about the Helm chart
- **values-production.yaml and values-staging.yaml:** These files contain the configuration parameters for the Helm chart for respective environments
- **Deployment and service files for both frontend and backend:** This will give us a sense of how the application components are deployed and accessed
- **MongoDB-related files:** We'll look at how the database is set up, given that it's a crucial part of three-tier web applications

Let's start with the `chart.yaml` file to understand the metadata of this Helm chart. Here's what we have in the `chart.yaml` file:

- **name:** `mychart` – This is the name of the Helm chart.
- **version:** `1.0.0` – The chart version is crucial as it enables versioning of your Helm charts, allowing for the streamlined release and tracking of updates. Within our CD pipeline, this version will be automatically updated, aligning with the version specified within the `packages.yaml` file from the source code repository.
- **description:** `A Helm chart for a 3-tier web application` – A brief description of what the chart is about.
- **apiVersion:** `v2` – This indicates the Helm chart API version. `v2` is used for Helm 3, which is the current major version of Helm at the time of writing this book.

Next, we'll look at the two values files, `values-production.yaml` and `values-staging.yaml`, which contain configurations for the production and staging environments, respectively. There's currently no difference in the content of these two files, so we'll discuss the configuration in a general sense. These files contain configuration for the backend, frontend, and the MongoDB service. The following are the configurations that are common to all three:

- **replicaCount:** – Number of pod replicas for the deployment. Set to `1`.
- **repository:** – Docker image repository. The image comes from an Artifactory instance.
- **tag:** – The specific version/tag of the image. We will change this programmatically using the CD pipeline.
- **pullPolicy:** – The policy used to pull the image.
- **type:** – The service type, which is `ClusterIP`, means it's only accessible within the cluster.
- **port:** – The port on which the service runs. For the backend, it's `3000`; for the frontend, it's `80`, and for the database, it's `27017`.
- **resources:**, **nodeSelector:**, **tolerations:**, and **affinity:** – These are configurations related to resource allocation, node selection, tolerations for taints, and affinity/anti-affinity rules, respectively. They are currently empty.

Additionally, the `mongodb` section contains configurations for the persistence storage, outlined as follows:

- **persistence:** – Configuration related to storage persistence
- **enabled:** – Whether persistence is enabled
- **storageSize:** – The size of the storage

Next, let's examine the deployment templates under the `/templates` directory for the frontend and backend components to see how they're deployed and made accessible. The `backend-deployment.yaml` file defines a Kubernetes deployment for the backend component and the `frontend-deployment.yaml` file defines a Kubernetes deployment for the frontend component. Both the deployments are similar in structure and configuration. Here are the key elements:

- **apiVersion:** and **kind:** – Specifies the Kubernetes API version and the type of resource. In this case, it's `Deployment`.

- Under **metadata:**, you have the following:
 - **name:** – The name of the deployment, which can be backend or frontend depending on the service.
 - **labels:** – A key-value pair used for tagging the deployment. This can be useful for querying and organizing resources.
- Under **spec:**, you have the following:
 - **replicas:** – Number of pod replicas. It uses a template placeholder, ``{{ .Values.backend.replicaCount }}``, that will be replaced by the value from the values file.
 - **selector:** – Specifies how to find pods that should belong to this deployment. It uses the **app: backend** or **app: frontend** label for this depending on the deployment file.
 - **template:** – Defines the pod template.
 - **metadata:** – Metadata for the pod, which includes labels.
 - **spec:** – Specifications for the containers running in the pod.
 - **containers:** – List of containers to run in the pod.
 - **name:** – Name of the container.
 - **image:** – Image to use for the container. The repository and tag are filled in using placeholders from the values file.
 - **imagePullPolicy:** – Specifies the image pull policy.
 - **ports:** – List of ports to expose from the container. The backend container exposes port **80**.
 - **imagePullSecrets:** – Specifies the name of the secret to use for pulling images. This is particularly useful when using private image repositories, such as Artifactory in this case.
 - **nodeSelector:**, **tolerations:**, and **affinity:** – These configurations allow for controlling where the pod gets scheduled, managing taints, and setting affinity/anti-affinity rules. The values are dynamically filled from the values file.

Next, let's look at the **backend-service.yaml**, **frontend-service.yaml**, and **mongodb-service.yaml** files to see how the backend, frontend, and mongodb deployments are made accessible. All three files are similar in structure and configuration. These services will provide network access to the backend, frontend, and mongodb pods, respectively. Here's a breakdown of the key elements:

- **apiVersion:** and **kind:** – Specifies the Kubernetes API version and the type of resource. In this case, it's **Service**.
- Under **spec:**, you have:
 - **type:** – The type of service to create. It uses a template placeholder, `{{ .Values.backend.service.type }}`, that will be replaced by the value from the values file (e.g., **ClusterIP**).
 - **ports:** – List of ports to expose from the service.
 - **port:** – The port on which the service will be accessible. The actual value is filled from the values file.
 - **targetPort:** – The port on the pods to which traffic will be directed.

- **selector:** – Specifies how to find pods that the service should direct traffic to. It uses the **app: backend** label for this.

Essentially, these services act as a network interface to the backend, frontend, and mongodb pods, making them accessible within the Kubernetes cluster.

Now, let's move on to the database, MongoDB. We'll begin by examining the **mongodb-configmap.yaml** file to understand any specific configurations for MongoDB. This file defines a Kubernetes ConfigMap that contains an initialization script for MongoDB. Here's a breakdown of its contents:

- **apiVersion:** and **kind:** – Specifies the Kubernetes API version and the type of resource, which is **ConfigMap**.
- Under **metadata:**, you have **name:** – The name of the ConfigMap, which is **mongodb-init-script**.
- Under **data:**, you have **init-mongo.js:** – This is a JavaScript script that will be executed when MongoDB starts up. Its purpose is to select or create the **helloworldapp** database using **db.getSiblingDB('helloworldapp')** and then insert multiple greeting messages in different languages into the **greetings** collection of the **helloworldapp** database.

ConfigMaps are used in Kubernetes to separate configuration data from application code, which makes it easier to manage and reduces the need to rebuild container images when configurations change. In this case, the ConfigMap is used to provide an initialization script for MongoDB.

Lastly, let's examine the **mongodb-statefulset.yaml** file to understand how MongoDB is deployed in a persistent manner, ensuring data is retained across pod restarts. This file defines a Kubernetes StatefulSet for MongoDB. StatefulSets are valuable for applications that require stable, unique network identifiers, stable persistent storage, and ordered, graceful deployment and scaling. Here's an overview of some of the key elements in the file:

- **apiVersion:** and **kind:** – Specifies the Kubernetes API version and the type of resource, which is **StatefulSet**.
- **volumeMounts:** – Specifies where to mount the volumes inside the container. Two mounts are defined: for MongoDB data at **/data/db**, and for the initialization scripts at **/docker-entrypoint-initdb.d/**.
- **volumes:** – Defines volumes to be used in the pod. Here, the initialization scripts from the previously discussed ConfigMap are mounted.
- **volumeClaimTemplates:** – The metadata contains the name of the PVC, which is **mongodb-data**. This name will be used as a prefix for PVCs created for each pod in the StatefulSet.
- Under **spec:**, we have:
 - **accessModes:** – This defines how the volume can be accessed.
 - **ReadWriteOnce** – This means the volume can be mounted as read-write by a single node. This ensures that the MongoDB instance has exclusive read-write access to its data.
 - **requests:** – Here, the amount of storage for the PVC is defined.
 - **Storage:** – The actual storage size is dynamically filled from the values file using the template placeholder, **{{.Values.mongodb.persistence.storageSize }}**.

This should make the Helm chart clear. However, to learn more about Helm charts and their constituents, visit the Chart Guide [3].

With that in mind, I recommend initiating a new repository on GitHub. Once created, upload the Helm chart files we've just reviewed. Make the repository private. This will ensure you have a centralized version-controlled location for your Kubernetes configurations, facilitating easier collaboration and deployment.

Setting up staging and production environments

In this section, we'll embark on four pivotal tasks to harden our infrastructure. First, we'll craft dedicated namespaces and secrets for both staging and production environments, laying a robust foundation for our deployment landscape. Next, the installation of the NGINX ingress controller will be executed, a crucial component for directing traffic to our web application in both environments. The third step involves deploying an ingress file for the web application, streamlining access and routing. Lastly, we will configure insecure registries on **Azure Kubernetes Service (AKS)**, reinforcing the flexibility and security of our deployment process. Join us as we bolster our environments for optimal deployment functionality.

Creating namespaces and secrets

In this section, we delve into the intricacies of setting up distinct environments within a Kubernetes cluster by establishing two separate namespaces: **staging** and **production**. These namespaces function as isolated spaces, allowing for a clear delineation between testing and live environments. To facilitate the seamless deployment of applications, we will integrate the `artifactory-credentials` secret into both namespaces. This essential step grants Kubernetes the required permissions to pull Docker images from Artifactory, ensuring that the correct application versions are deployed in their respective namespaces.

It's worth noting that, in a more complex real-world scenario, the staging and production environments often reside in separate Kubernetes clusters. Such an arrangement offers enhanced security, scalability, and manageability. However, for the sake of clarity and simplicity in this instructional context, we will house both the staging and production environments within a single cluster, using namespaces to demarcate them. This approach provides a foundational understanding while still emphasizing best practices. Having said that, follow the next steps to create two new namespaces and secrets on your existing AKS cluster:

1. Execute the `az login` command in your terminal or command prompt to authenticate with your Azure account:

```
az login
```

2. A browser window will open requesting you to sign in to your Azure account. Once authenticated in the browser, you can close the window. The terminal should then display a list of subscriptions associated with your Azure account.
3. Next, set a default resource group for your Azure CLI session by running the following command in your terminal. Remember to replace **<Your resource group>** with the name of the resource group where you have your AKS instance. Once executed, any subsequent Azure CLI commands that require a resource group will use the specified default unless overridden. Ignore this step if you have only one resource group in your Azure subscription:

```
az configure --defaults group=<Your resource group>
```

4. To establish new namespaces, employ the **kubectl create namespace** command. Execute this command consecutively, substituting the **<namespace name>** placeholder with **staging** and then **production**:

```
kubectl create namespace <namespace name>
```

5. Proceed to use the **kubectl create secret** command to generate two secrets within the **staging** and **production** namespaces we previously established. These secrets will store the authentication details for our Docker registry, which in our context is the Artifactory local repository:

- Substitute **<Artifactory Server IP Address>** with the genuine IP address or URL of your Artifactory server. (Refer to the *Connecting Jenkins to Artifactory* section in [Chapter 3](#) for more info if required.)
- Input the actual password for the **jenkins** user in place of **<Password for user jenkins>**.
- Use the email linked to the **jenkins** user to replace **<Email for user jenkins>**.
- Initially, set **<namespace name>** to **staging** and then to **production**:

```
kubectl create secret \  
  docker-registry artifactory-credentials \  
  --docker-server<Artifactory Server IP Address> \  
  --docker-username=jenkins \  
  --docker-password=<Password for user jenkins> \  
  --docker-email=<Email for user jenkins> \  
  -n <namespace name>
```

With this step completed, our AKS cluster now contains two distinct namespaces: **staging** and **production**. Each namespace holds a secret encompassing credentials for our Artifactory Docker registry, ensuring secure and authorized access.

Installing NGINX Ingress Controllers for the web application

Now, let's proceed to create an NGINX Ingress Controller for our web applications slated for deployment in both the staging and production environments. Please work through the following steps:

1. Execute the following two commands one after the other to add the **ingress-nginx** repository as a source of Helm charts and update the local cache to have the latest information about the available charts from all the added repositories:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx  
helm repo update
```

2. Execute the **helm install** command to deploy the **ingress-nginx** chart with tailored configurations and options. This command will set specific annotations, define the replica count, and associate the NGINX ingress controller with a distinct ingress

class resource and annotation value, as demonstrated in the following snippet. Ensure you run the command twice – initially with **<environment>** set to **staging**, and subsequently to **production**:

```
helm install ingress-nginx-myapp-<environment> \
  ingress-nginx/ingress-nginx \
  -n <environment> \
  --set controller.service.annotations."service\.beta\.kubernetes\.io/azure-load-
balancer-health-probe-request-path"/healthz \
  --set controller.replicaCount=1 \
  --set controller.ingressClassResource.name=myapp-<environment>-nginx \
  --set controller.ingressClassResource.controllerValue=k8s.io/myapp-<environment>-
nginx
```

3. This should create the NGINX Ingress Controller in staging and production namespaces. To check the deployed Helm charts in a specific namespace, you can use the **helm list** command along with the **-n** flag to specify the namespace:

```
helm list --namespace staging
```

This command will show you a list of all Helm releases deployed in the specified namespace. The output includes the name of the release, the namespace, the revision number, the updated date, the status, and the chart name and version.

Deploying Ingress files for the web application

To access our web application in the staging and production environments, we need to expose it using an Ingress resource. Apply the following steps to deploy an Ingress to expose our web application service:

1. First, create a file called **myapp-staging-ingress.yaml** with the following content:

myapp-staging-ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-staging-ingress
  annotations:
spec:
  ingressClassName: myapp-staging-nginx
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend
            port:
              number: 80
```

The preceding code is an Ingress resource definition for the web application to be deployed in the staging namespace. Let's see some of its key elements:

- **ingressClassName: myapp-staging-nginx** specifies the class name of the Ingress controller associated with this Ingress resource as **myapp-staging-nginx**

- **name: frontend** sets the name of the frontend service
- **number: 80** specifies the port configuration for the frontend service by setting the port number to **80**

2. Now, create another file named **myapp-production-ingress.yaml** for our web application to be deployed in the **production** namespace:

myapp-production-ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-production-ingress
  annotations:
spec:
  ingressClassName: myapp-production-nginx
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend
            port:
              number: 80
```

3. Next, save the files and run the **kubectl apply** command to first apply the **myapp-staging-ingress.yaml** manifest file in the **staging** namespace, and then **myapp-production-ingress.yaml** in the **production** namespace:

```
kubectl apply -f ./myapp-staging-ingress.yaml -n staging
kubectl apply -f ./myapp-production-ingress.yaml -n production
```

4. It takes a while for the NGINX Ingress Controller's public IP to associate with the Ingress. Use the Azure dashboard to check the association.

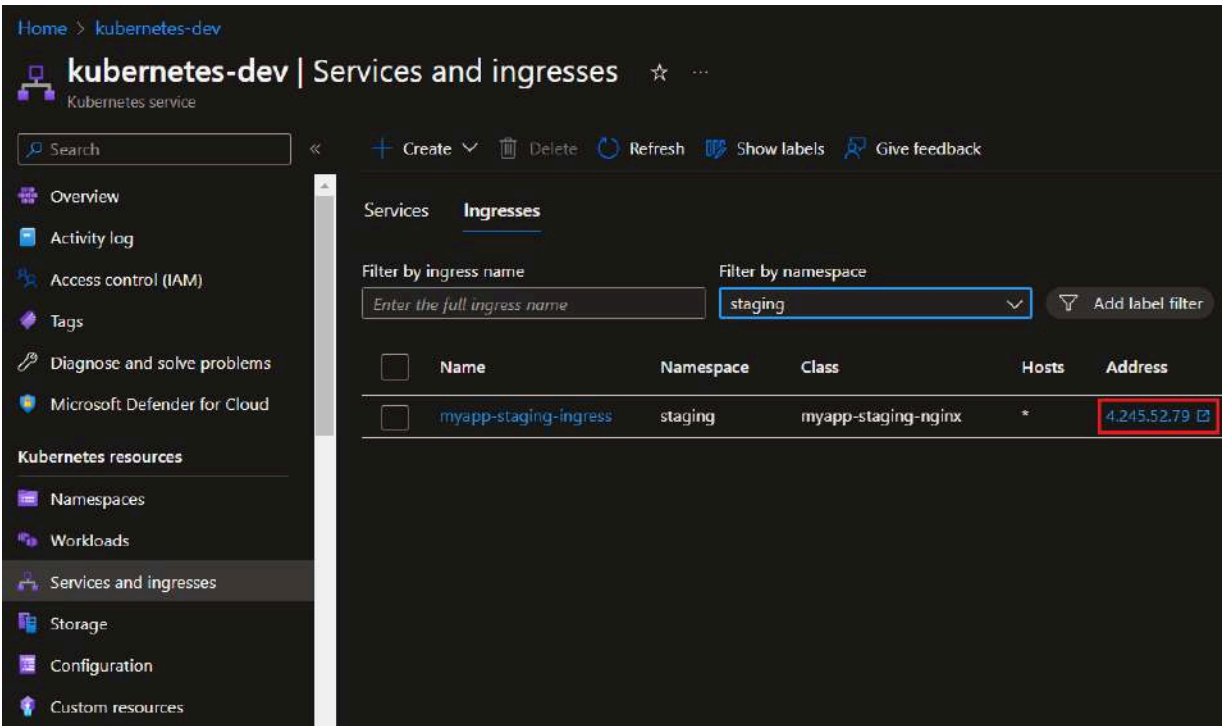


Figure 10.5 – Staging ingress with its public IP address

By adhering to these steps, we gain access to our web applications in both the staging and production environments. Access is available through `http://<Public IP>/`, wherein `<Public IP>` represents the public IP address assigned to the NGINX Ingress Controller of the web application.

Configuring insecure registries

Argo CD automates the deployment of applications within Kubernetes, ensuring that the live environment mirrors the desired state specified in a Git repository (Helm chart). Jenkins, on the other hand, is instrumental in building, testing, and packaging applications. Artifactory serves as a repository manager, storing and cataloging binary artifacts, including Docker images built by Jenkins. For seamless interoperability, it's essential that Argo CD can reliably pull Docker images from Artifactory to deploy them onto Kubernetes.

However, when Artifactory lacks a domain name and SSL certificates, it's perceived as an insecure registry by default configurations. This poses challenges, as secure communications between these tools become hindered. By adjusting configurations and making exceptions for the “insecure” Artifactory registry within Kubernetes nodes, we ensure that Argo CD can fetch and deploy the necessary Docker images. This adjustment, therefore, is not just a technical workaround but a critical step in ensuring that the CD pipeline, spanning Jenkins, Artifactory, and Argo CD, operates without disruptions. Here's a step-by-step guide to address this:

1. To begin, you'd want to establish an interactive shell connection to a specific AKS node. This can be accomplished using the **kubectl debug** command, which initiates a privileged container on your desired node. Remember that you need to run through these steps on every node of your AKS cluster.

2. First, list your nodes to identify the one you want to connect to:

```
kubectl get nodes -o wide
```

3. Then, use the **kubectl debug** command to launch a privileged container on your chosen node:

```
kubectl debug node/<name of the node> -it --image=mcr.microsoft.com/dotnet/runtime-deps:6.0
```

4. Access the node once connected. You'll be placed within a privileged container environment on the node. This provides you with direct access to the node's resources and configurations. To further interact with the node session, use **chroot /host** from within the privileged container.

5. Now we will configure insecure registry settings inside AKS node. To do this, navigate to the **containerd** configuration directory:

```
cd /etc/containerd/
```

6. Here, create a directory structure to hold the **Artifactory** configuration:

```
mkdir -p certs.d/<Artifactory Server IP Address>/
```

7. Within this directory, create the **hosts.toml** file and populate it with the following content:

```
server = https://<Artifactory Server IP Address>
[host."http://<Artifactory Server IP Address>"]
  capabilities = ["pull", "resolve", "push"]
  skip_verify = true
  plain-http = true
```

8. This configuration enables Argo CD to securely pull Docker images from Artifactory, despite the registry being labeled insecure. The **skip_verify** and **plain-http** options specifically bypass SSL verification, ensuring successful image pulls from Artifactory without SSL certificates.

IMPORTANT NOTE

In the context of this chapter, we've opted for certain configurations to maintain simplicity and provide a straightforward learning experience. It's worth emphasizing that in real-world scenarios, particularly in mature production environments, security measures and best practices are paramount. Typically, services such as Artifactory would be outfitted with valid SSL certificates to ensure encrypted and secure communications. By doing so, the challenges we addressed regarding "insecure" registries would be naturally avoided. However, for the sake of this tutorial and to keep the learning curve manageable, we've bypassed the complexities of certificate management. It's essential for readers and implementers to recognize that such shortcuts, while valuable for learning, are not advisable for production deployments. In live, mission-critical environments, always prioritize security by employing valid certificates and adhering to recognized best practices.

We have successfully set up the staging and production environments by creating namespaces, ingress controllers, and secrets for both on our existing AKS cluster on Azure. Now, let's proceed to set up Argo CD.

Installing Argo CD

We now delve into the installation of Argo CD. First, we'll set the stage by installing an NGINX Ingress Controller specifically for Argo CD, ensuring streamlined and secure traffic management. Following this, we transition to the actual installation of Argo CD. Next up is the deployment of an ingress file for Argo CD, further solidifying our setup by enhancing accessibility and routing. The final step in this comprehensive process involves creating a dedicated user on Argo CD, reinforcing security and user management within our Argo CD environment. Let's begin!

Installing an NGINX Ingress Controller for Argo CD

Just like Jenkins, SonarQube, and Artifactory, we will create an NGINX Ingress Controller for Argo CD too. Work through the following steps:

1. Execute the following two commands one after the other to add the **ingress-nginx** repository as a source of Helm charts and update the local cache to have the latest information about the available charts from all the added repositories:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
```

2. Use the **helm install** command to install the **ingress-nginx** chart with specific configurations and options to set annotations, define the replica count, and associate the NGINX Ingress Controller with a specific Ingress Class resource and annotation value, as follows:

```
helm install ingress-nginx-argocd \
  ingress-nginx/ingress-nginx \
  --create-namespace -n argocd \
  --set controller.service.annotations."service\.beta\.kubernetes\.io/azure-load-
balancer-health-probe-request-path"/healthz \
  --set controller.replicaCount=1 \
  --set controller.ingressClassResource.name=argocd-nginx \
  --set controller.ingressClassResource.controllerValue=k8s.io/argocd-nginx
```

3. This should create the NGINX Ingress Controller for Argo CD under the **argocd** namespace. To check the deployed Helm charts in a specific namespace, you can use the **helm list** command along with the **-n** flag to specify the namespace, as follows:

```
helm list --namespace argocd
```

This command will show you a list of all Helm releases deployed in the specified namespace. The output includes the name of the release, the namespace, the revision number, the updated date, the status, and the chart name and version.

Getting started with installing Argo CD

Installing Argo CD is a straightforward process. In the following section, we will seamlessly deploy Argo CD on our AKS cluster utilizing the **kubectl** command. The installation will be confined within the **argocd** namespace, which was previously created during the Ingress-NGINX installation for Argo CD. Deploy Argo CD to your cluster by applying the Argo CD manifest using the following command:

```
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

You should now have an Argo CD server running under the `argocd` namespace on AKS.

Deploying an Ingress file for Argo CD

To access our Argo CD server, we need to expose it using an Ingress resource. Work through the following steps to deploy an Ingress to expose our Argo CD service:

1. First, create a file called `argocd-ingress.yaml` with the following content:

`argocd-ingress.yaml`

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: argocd-ingress
  annotations:
spec:
  ingressClassName: argocd-nginx
  rules:
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: argocd-server
            port:
              number: 80
```

The preceding code is an Ingress resource definition for Argo CD. Let's see some of its key elements:

- `ingressClassName: argocd-nginx` specifies the class name of the Ingress controller associated with this Ingress resource as `argocd-nginx`
- `name: argocd-server` sets the name of the backend service as `argocd-server`
- `number: 80` specifies the port configuration for the backend service by setting the port number to `80`

2. Next, save the file and run the `kubectl apply` command to apply the `argocd-ingress.yaml` manifest file in the `argocd` namespace:

```
kubectl apply -f ./argocd-ingress.yaml -n argocd
```

It takes a while for the NGINX Ingress Controller's public IP to associate with the Argo CD ingress. Use the Azure dashboard to check the association.

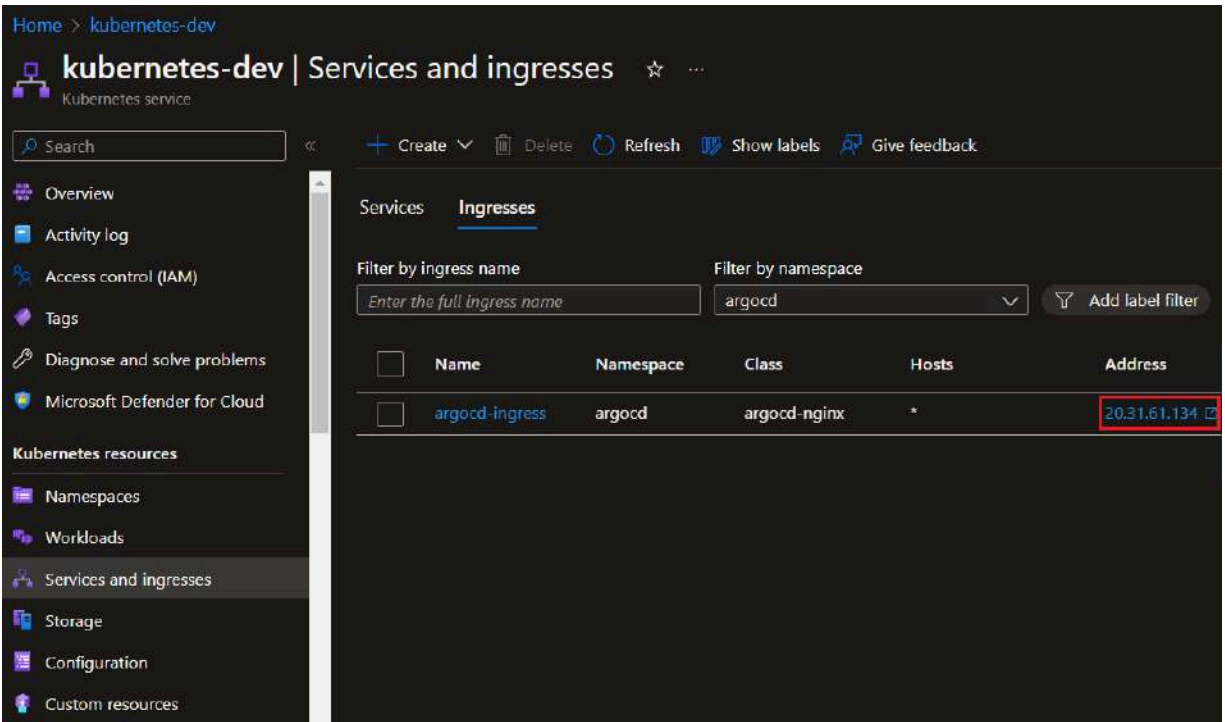


Figure 10.6 – Argo CD ingress with its public IP address

To access Argo CD, visit `http://<Public IP>/`, where `<Public IP>` is the public IP of the Argo CD NGINX Ingress Controller.

Creating a user on Argo CD

Upon installing Argo CD, a default administrator account named `admin` is automatically generated. This account is configured with an initial password, retrievable through the utilization of the Argo CD **Command-Line Interface (CLI)**. However, utilizing the default admin account may prove inadequate for working effectively with Argo CD APIs. For enhanced security and functionality, it is advisable to establish a new administrative user endowed with the requisite privileges. Please proceed with the following steps to achieve this configuration:

1. Run the `argocd admin initial-password` command to get hold of the initial password of the default admin account:

```
argocd admin initial-password -n argocd
```

2. Proceed to log in to Argo CD utilizing the credentials of the admin account. Upon executing the following command, a prompt will request your admin password:

```
argocd login <Argo CD Server IP Address> --username admin --grpc-web-root-path /
```

3. To create a new user, we will update the Argo CD ConfigMap file using the `kubectl edit` command. Running the following command will open the ConfigMap file in Notepad:

```
kubectl edit configmap argocd-cm -n argocd
```

4. Now, add the following lines to the ConfigMap file. Make sure to replace **<username>** with the new username you wish to create. This will create a new username, enabling the user to generate an API key and permitting them to authenticate via both the CLI and GUI. Close the file when you are done making the changes:

```
data:
  accounts.<username>: apiKey, login
```

5. The user gets created immediately. You don't have to restart Argo CD. To verify that the user got created, run the **argocd account list** command. This should list the admin user and the user you just created:

```
argocd account list
```

6. Observe that the admin user lacks the authority to generate API keys. This limitation means the admin account is unable to create access tokens, which are essential for other tools including Jenkins to interact seamlessly with Argo CD APIs.
7. Now, let's set a password for our new user using the **argocd account update-password** command. Remember to replace the **<username>** and **<password>** fields appropriately. While you do so, you will be asked to authenticate the existing logged-in user again, which is **admin** in our case:

```
argocd account update-password --account <username> --new-password <password>
```

8. Next, let us give our new user some privileges to create stuff inside Argo CD. We will do this by updating the RBAC ConfigMap file. Simply run the **kubectl edit** command with the following options and it will open the RBAC ConfigMap file in Notepad:

```
kubectl edit configmap argocd-rbac-cm -n argocd
```

9. Next, add the following lines to the RBAC ConfigMap file. Make sure to replace **<username>** with the new username you just created:

```
data:
  policy.csv: |
    p, role:org-admin, applications, *, */*, allow
    p, role:org-admin, clusters, get, *, allow
    p, role:org-admin, repositories, get, *, allow
    p, role:org-admin, repositories, create, *, allow
    p, role:org-admin, repositories, update, *, allow
    p, role:org-admin, repositories, delete, *, allow
    g, <username>, role:org-admin
  policy.default: role:admin
```

This data section contains the policy information for role-based access control. Here's a breakdown of the code:

- **policy.csv: |** – This line declares a multi-line string that contains policy definitions in CSV format. Each line in the string represents a policy rule.
- **p, role:org-admin, applications, *, */*, allow** – This line is a policy rule. It states that a user with **role:org-admin** is allowed (**allow**) to perform any action (*****) on applications in any project (***/***).
- The subsequent lines grant the **role:org-admin** permissions to get, create, update, and delete repositories, as well as get clusters.
- The ***** symbol is a wildcard that matches any value, providing broad permissions to **role:org-admin**.

- **g, <username>, role:org-admin** – This line is a role assignment. It assigns a specific user (<username>) the role **role:org-admin**. Replace <username> with the actual username of the user you want to assign the role to.
- **policy.default: role:admin** – This line sets the default policy. If a user doesn't match any of the policies defined in **policy.csv**, this default policy will be applied. In this case, it is assigning an admin role as the default, which likely means full access.

10. Once you are done editing the file, close it. The changes take effect immediately. And this is how you create a new user in Argo CD and assign it admin privileges.

Creating applications in Argo CD

In Argo CD, an application represents a set of Kubernetes resources (such as Deployments, Services, and ConfigMaps) and their desired states, as specified in a Git repository. The repository contains the declarative definitions and configuration for the resources, often defined as YAML files. Argo CD continuously monitors the Git repository to ensure that the state of the resources in the Kubernetes cluster matches the state defined in the repository. If a discrepancy is detected, Argo CD can automatically or manually synchronize the cluster's state with the desired state defined in the Git repository, applying any changes as necessary.

Applications in Argo CD are pivotal as they facilitate the GitOps methodology, allowing developers to use Git as the *single source of truth* for both application code and infrastructure. This model enhances traceability, auditability, and ease of rollback in the deployment process. The creation and management of applications in Argo CD can be done using the Argo CD web UI, CLI, or API. These interfaces allow users to define the Git repository, the desired state, and other pertinent settings for the application, ensuring the consistent and reliable deployment and management of Kubernetes resources in different environments.

Creating a personal access token on GitHub

With the personal access token, Argo CD acquires the necessary permissions to clone repositories. To create a personal access token on GitHub with the specific scope (**repo**), work through the following steps:

1. Log in to your GitHub account and click on your profile picture in the top-right corner of the page to select **Settings** from the drop-down menu.
2. In the left sidebar of the **Settings** page, scroll all the way down and click on **Developer settings**, and on the **Developer settings** page, click on **Personal access tokens**.
3. Once on the **Personal access tokens** page, click on the **Generate new token** button..
4. Enter a meaningful description for the token using the **Note** field to help you identify its purpose later. Additionally, set **Expiration** to anything but not **No expiration**.
5. Next, in the **Select scopes** section, check the **repo** box to grant the required permissions to the token.

6. Lastly, scroll down and click on the **Generate token** button.

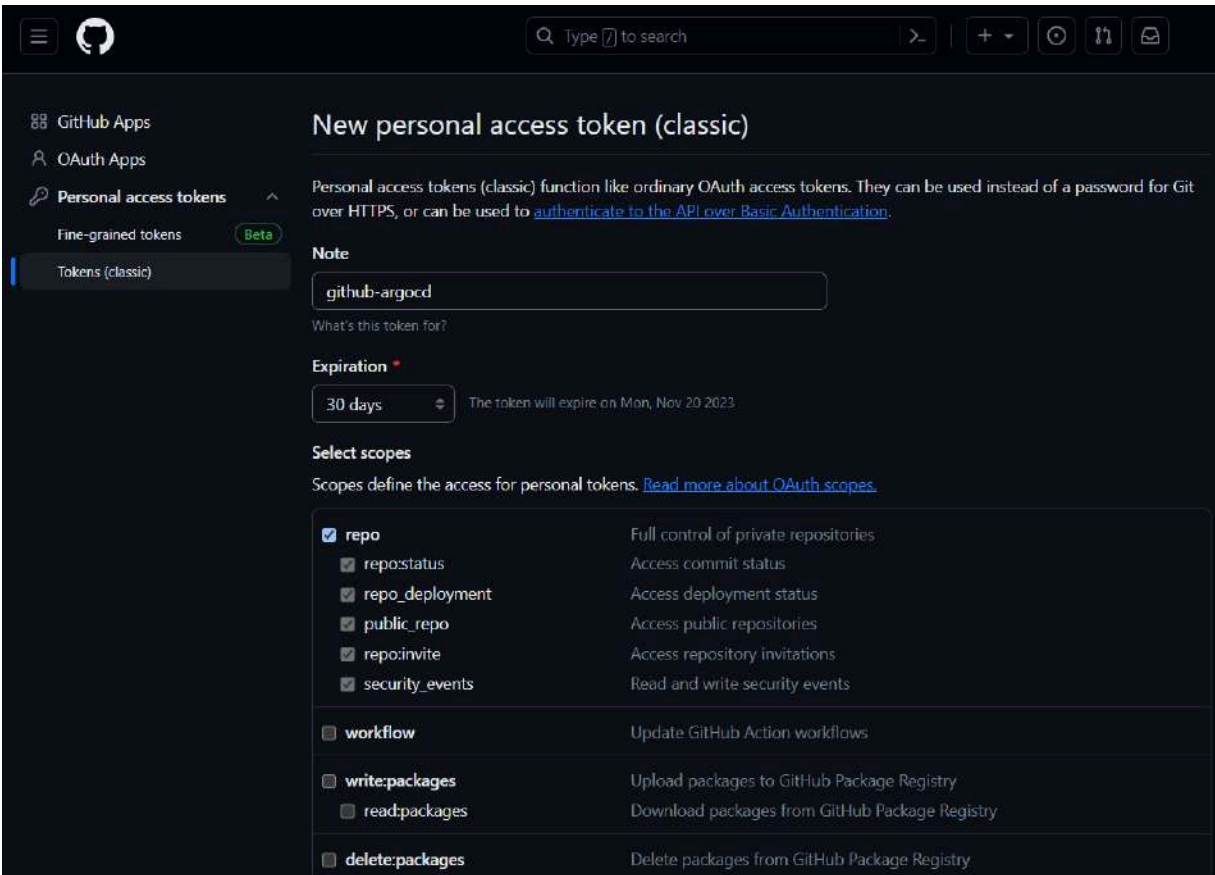


Figure 10.7 – Creating a personal access token on GitHub

After the token is generated, it will be displayed on the screen. Make sure to copy the token, as it will not be shown again. Using this token, we will now connect the GitHub repository containing the Helm chart of our web application to Argo CD.

Connecting a GitHub repository to Argo CD

In the following section, we will connect the GitHub repository containing our Helm chart with our Argo CD server. To do this, we will incorporate the `argocd repo add` command. You may choose to run the following `argocd` command as the default `admin` user or you can switch to the new user we just created inside Argo CD from the previous section:

```
argocd repo add <Repository URL> \  
--username <GitHub Username> \  
--password <GitHub Personal Access Token>
```

In the preceding command, `<Repository URL>` should be replaced with the URL of your GitHub repository. This is the repository that Argo CD will manage, watching for changes to automatically apply to your Kubernetes cluster. `<GitHub Username>` should be replaced with the username for

authenticating to the GitHub repository. `<GitHub Personal Access Token>` should be replaced with your actual GitHub personal access token that we created in the previous section.

Setting up the Argo CD application

In this section, we will use the Argo CD CLI to create an application in Argo CD. This procedure is essential for the deployment of our web application within the staging environment. Post this setup, our focus shifts to Jenkins within the CD pipeline. Jenkins is tasked with the sole responsibility of updating the Helm chart whenever there is a fresh build of Docker images. This streamlined approach eliminates the need for manual intervention in the deployment process, ensuring efficiency and reducing the potential for human error. With the Helm chart updated, Argo CD automatically leaps into action, synchronizing the current state with the desired state defined in the Git repository. This synchronization includes deploying the updated Docker images to the staging environment, ensuring that the most recent version of our web application is always in operation. To achieve this, simply run the following `argocd app create` command:

```
argocd app create hello-world-staging \
  --auto-prune \
  --dest-namespace staging \
  --dest-server https://kubernetes.default.svc \
  -l environment=staging \
  --path . \
  --project default \
  --repo <Repository URL> \
  --revision HEAD \
  --sync-policy automated \
  --values values-staging.yaml
```

The following is a breakdown of the `argocd app create` command, along with explanations for each of the options:

- **argocd app create hello-world-staging** – This command is used to create a new application in Argo CD with the name `hello-world-staging`.
- **--auto-prune** – This option will automatically prune resources that are not in the Git repository, keeping your cluster in sync with the source repository.
- **--dest-namespace staging** – This specifies the destination namespace where the application's resources will be deployed, in this case, the `staging` namespace.
- **--dest-server https://kubernetes.default.svc** – This sets the destination Kubernetes cluster URL where the application's resources should be deployed.
- **-l environment=staging** – This sets a label on the application with a key of `environment` and a value of `staging`. Labels in Argo CD help to categorize your application, facilitating easier management, automation, and resource filtering specific to that environment within your Kubernetes setup. This labeling system is crucial for organizing and differentiating applications across various deployment stages including development, staging, and production.
- **--path .** – This option specifies the path in the Git repository to use for the application's source. The `.` symbol indicates the root of the Git repository.

- **--project default** – This assigns the application to the **default** project in Argo CD.
- **--repo <Repository URL>** – This option specifies the URL of the Git repository containing the application’s source code and manifest files. Replace **<Repository URL>** with the actual URL of your Git repository.
- **--revision HEAD** – This sets the revision of the Git repository to use for the application. **HEAD** refers to the latest commit on the default branch (usually main or master).
- **--sync-policy automated** – This option enables automatic synchronization of the application, ensuring that the cluster state matches the desired state defined in the Git repository.
- **--values values-staging.yaml** – This specifies a values file to use when deploying the application. The **values-staging.yaml** file should contain configuration specific to the **staging** environment.

In summary, this command creates a new Argo CD application named **hello-world-staging** that automatically syncs the **staging** namespace in your Kubernetes cluster with the latest state defined in the specified GitHub repository. The application uses the configuration specified in the **values-staging.yaml** file and is labeled with **environment=staging** for easy identification and management.

Now, let us deploy our web application in a **production** environment. The command is quite like the earlier command used for deploying to the staging environment, with key differences that tailor it for production deployment:

```
argocd app create hello-world-production \
  --auto-prune \
  --dest-namespace production \
  --dest-server https://kubernetes.default.svc \
  -l environment=production \
  --path . \
  --project default \
  --repo <Repository URL> \
  --revision HEAD \
  --sync-policy automated \
  --values values-production.yaml
```

In the preceding **argocd app create** command, here are the key differences:

- **--dest-namespace production** – This flag now targets the **production** namespace, signifying that the resources will be deployed in this namespace, ensuring a separation between staging and production environments.
- **-l environment=production** – This label is set to **environment=production**, providing a clear and quick identification of production resources, aiding in management and potential troubleshooting.
- **--values values-production.yaml** – This command is now utilizing the **values-production.yaml** file. This file likely contains configurations that are specific to the production environment, ensuring the application runs with the correct settings when deployed in production.

Like the staging setup, after the initial setup with this command, Jenkins in the CD pipeline will handle updates to the Helm chart automatically. With each new build of Docker images, Jenkins will update the Helm chart, and Argo CD will thereafter ensure synchronization, deploying the latest

images and configurations in the production environment, keeping the application up to date with minimal manual intervention.

To view the applications on Argo CD dashboard, simply log in to the Argo CD UI using the user credentials and click on the **Applications** section from the left-hand side menu. You should see two applications, one running in the staging environment and the other in production, as shown in the following screenshot.

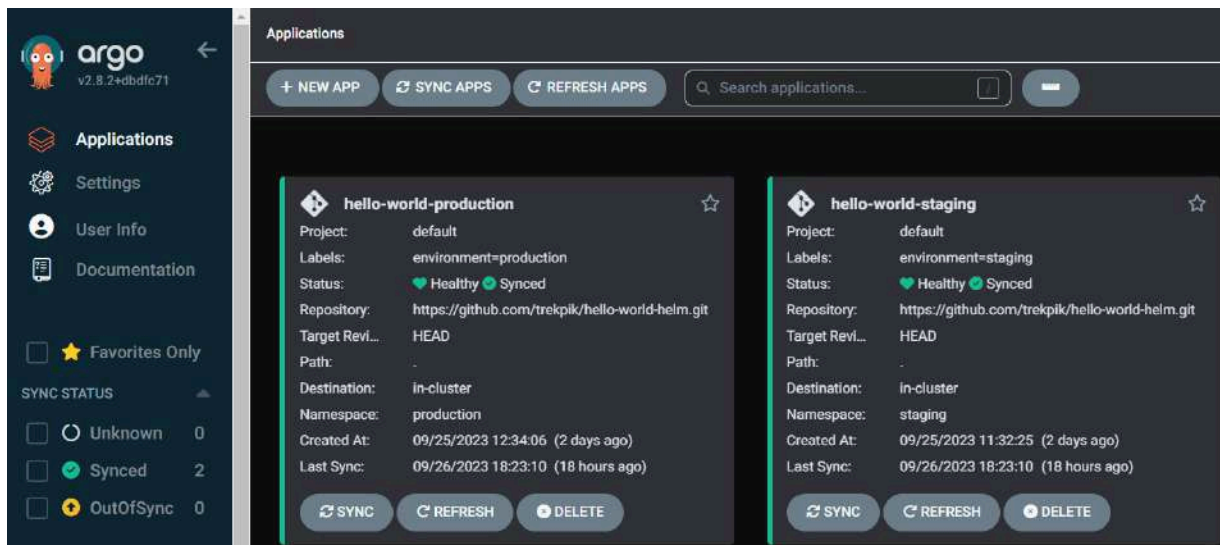


Figure 10.8 – Applications on Argo CD

So far, using the Argo CD CLI, we've instructed Argo CD to deploy and oversee our web application in both the staging and production environments. Let's now delve into the integration of Jenkins and Argo CD.

Integrating Jenkins with Argo CD

In this crucial section, we're going to bridge Jenkins with Argo CD. While there isn't a robust Jenkins plugin available for Argo CD, the interaction between Jenkins and Argo CD is effectively handled through pipelines that seamlessly integrate Argo CD APIs. The two-fold process begins with the creation of an authentication token for the Argo CD user. Following this, we proceed to formulate a secret text credential within Jenkins, allowing Jenkins pipeline code to establish secure and authenticated interactions with Argo CD.

Creating an auth token for the Argo CD user

To generate an access token for a user in Argo CD, work through the following steps. This token will allow you to access the Argo CD API programmatically:

1. First, ensure that you are logged in to the Argo CD CLI.
2. Execute the `argocd account generate-token` command to generate an access token for a specific user account. Replace `<username>` with the name of the user account for which you want to generate the token. Remember, you cannot generate a token for the default admin account in Argo CD, as it is disabled by default:

```
argocd account generate-token \  
  --account <username> \  
  --id argocd-jenkins \  
  --expires-in 90d
```

3. The command will output a token. Make sure to copy and store this token securely, as it provides access to the Argo CD system. For additional security, consider setting an expiration for the token using the `--expires-in` flag while generating the token. In the preceding command, it is set to expire in 90 days.

Creating a secret text credential in Jenkins

By creating a *secret text* credential for the Argo CD user token, you can safely and securely store and manage the token within Jenkins, ensuring that it is readily available to the CD pipeline that requires authentication when interacting with the Argo CD server. Apply the following steps to create a *secret text* credential inside Jenkins:

1. From the Jenkins dashboard, navigate to **Manage Jenkins | Credentials | System | Global credentials (unrestricted)**, and click **+ Add Credentials**.
2. Configure the credential details:
 - Using the **Kind** field, choose **Secret text** from the drop-down list
 - In the **Secret** field, enter the Argo CD user token that we copied earlier
 - Provide an **ID** and **Description** to help identify and describe the credential

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

New credentials

Kind

Secret text

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Secret

.....

ID ?

argocd-jenkins

Description ?

Token to authenticate Argo CD

Create

Figure 10.9 – Creating a Secret text credential inside Jenkins

3. Lastly, click on the **Create** button to save the credential. Once saved, you can use the *secret text* credential inside Jenkins by referencing its ID.

Having successfully navigated through the crucial steps of understanding the high-level CD design, setting up both staging and production namespaces and their NGINX Ingress controllers, diligently installing Argo CD, creating applications in Argo CD for staging and production, and applying the necessary configurations inside Jenkins, we find ourselves well equipped and prepared for the journey ahead. Let's now transition into writing the CD pipeline code in the next chapter.

Summary

In the current chapter, we embarked on a detailed journey to understand and define CD. We began by exploring two strategies for CD: universal CI with master-only CD and the integrated model of universal CI and CD. In the process, we delved into the core concepts of GitOps and its intrinsic connection to CD. Our planning phase focused on the intricacies of high-level CD architecture, examining the Helm chart for our web application and establishing a foundation for both staging and

production environments. This setup entailed creating namespaces and secrets in AKS, configuring insecure registries, and more.

Subsequently, we navigated the installation and configuration of Argo CD and its integration with Jenkins to bolster automation capabilities. Among other skills, we mastered creating an Argo CD user and generating its associated token. Additionally, we honed our ability to seamlessly link a GitHub repository to Argo CD, optimizing application management within the platform.

With the foundation for CD in place, the next chapter will guide us in constructing a CD pipeline in Jenkins. We'll then execute it to observe the testing and deployment of our web application in the staging and production environments respectively.

Questions

1. Choose whether this statement is true or false: *GitOps strictly refers to the use of Git as a single source of truth for declarative infrastructure and applications, where all operational changes are made through pull requests, but does not include automated deployment or monitoring within its scope.*
2. What is the primary difference between **Continuous Delivery (CD)** and **Continuous Deployment** (also referred to as **CD**)?
 - A. Continuous Delivery automates the deployment process to any environment, while Continuous Deployment automates only to production environments
 - B. Continuous Delivery requires manual approval for updates to be released to production, whereas Continuous Deployment automates releases to production without manual intervention
 - C. Continuous Deployment is an extension of Continuous Integration, but Continuous Delivery is not an extension of Continuous Integration
 - D. Continuous Delivery focuses on frequent code integrations, while Continuous Deployment emphasizes automated testing
3. How do Jenkins and Argo CD work together in a CI/CD pipeline, particularly with respect to handling changes in configuration stored in GitHub?
 - A. Jenkins monitors GitHub for configuration changes and triggers Argo CD for deployments to both staging and production environments
 - B. Argo CD monitors GitHub for changes in configuration, automatically deploying these changes to staging and production, while Jenkins handles the build and test phases
 - C. Both Jenkins and Argo CD simultaneously monitor GitHub for changes, but Jenkins handles staging deployments and Argo CD handles production deployments
 - D. Argo CD handles all build, test, and deployment phases based on GitHub changes, while Jenkins is used for post-deployment monitoring
4. When using Helm charts with Argo CD, which file does Argo CD monitor on GitHub to trigger a deployment?
 - A. **Chart.yaml**

B. `Jenkinsfile`

C. `Dockerfile`

D. `values.yaml`

5. Choose whether this statement is **true** or **false**: *Extensive testing is not a crucial part of Continuous Deployment (CD). CD deals only with securely deploying the application in production. In fact, it's the CI process responsibility to ensure that every change deployed to production is stable and does not introduce any regressions or new bugs.*

Answers

1. Correct answer: **False**

Explanation:

This statement is false because while GitOps does primarily focus on using Git as a single source of truth for declarative infrastructure and applications, and emphasizes the importance of changes through pull requests, it also inherently includes automated deployment and monitoring. GitOps is not just about using Git for version control and collaboration; it's also about applying software development practices such as **Continuous Integration** and **Continuous Deployment (CI/CD)** to infrastructure management. This includes automatically applying changes to the infrastructure when commits are made to the repository (automated deployment) and constantly monitoring the state of the infrastructure to ensure it matches the declared state in the repository (monitoring). Therefore, automated deployment and monitoring are integral parts of the GitOps methodology.

2. Correct answer: **B**

Explanation of incorrect answer choices:

A. This option is misleading. Both Continuous Delivery and Continuous Deployment automate the deployment process, but the key difference lies in the level of automation in releasing changes to production.

C. This is incorrect. Both Continuous Delivery and Continuous Deployment are extensions of Continuous Integration.

D. This option confuses the core aspects. Both practices involve frequent code integrations and automated testing. The fundamental difference is in the automatic deployment to production (Continuous Deployment) versus the need for manual release in Continuous Delivery.

3. Correct answer: **C**

Explanation of incorrect answer choices:

A. This choice is incorrect because it's typically Argo CD that monitors the repository for changes in configuration, not Jenkins. Jenkins is focused on the build and test aspects of the

CI/CD pipeline.

B. This is incorrect because it's not common for both tools to monitor GitHub for the same purpose. Argo CD is specifically designed to monitor repositories for changes and apply these to Kubernetes environments, and it doesn't typically divide responsibilities with Jenkins based on staging or production environments.

D. This option is incorrect because Argo CD does not handle the build and test phases; it is focused on deployment. Jenkins is a tool generally used for building and testing in a CI/CD workflow.

4. Correct answer: **D**

Explanation of incorrect answer choices:

A. While this file is a crucial part of a Helm chart, defining metadata about the chart, it is not typically the file that triggers deployments. `chart.yaml` contains information such as the chart version and name, but not the configurable values that determine the specifics of a deployment.

B. This is a script file used by Jenkins to define a Jenkins pipeline. It's not related to Kubernetes deployment specifications and thus not what Argo CD monitors for triggering deployments.

C. This file is used for defining how to build a Docker container image, not for Kubernetes deployment configurations. While important in the build process, Argo CD does not use a Dockerfile to trigger deployments.

5. Correct answer: **False**

Explanation:

Extensive testing is an integral component of **Continuous Deployment (CD)**, playing a vital role in ensuring that all changes are stable, function correctly, and are free from regressions or new bugs. This thorough testing regime typically encompasses a range of tests, including integration, functional, and, where necessary, performance and security tests. The objective is to comprehensively validate changes through automated processes before they are deployed to production, thereby significantly reducing the risk of introducing errors to end users. On the other hand, **Continuous Integration (CI)** primarily focuses on running quick, less extensive tests to provide rapid feedback during the development phase.

Further reading

If you wish to know more about some of the concepts that were not covered in depth in this chapter, or are out of the scope of this book but relevant, we are happy to provide you with some references:

- [1] Installing Argo CD CLI documentation: https://argo-cd.readthedocs.io/en/stable/cli_installation/
- [2] Grafana Labs k6 tool: <https://k6.io/>
- [3] Helm Chart Guide: <https://helm.sh/docs/topics/charts/>

Writing Your First CD Pipeline

In the preceding chapter, we delved into the nuances of setting up a Helm chart, hosted in its own GitHub repository, to optimize and streamline Kubernetes deployments (our web application). This approach ensures more robust maintainability while benefiting from version control. Furthermore, we explored the integration of Argo CD and Jenkins and also established both staging and production environments on AKS. With the foundations for **continuous deployment (CD)** laid out, this chapter will guide you in crafting CD pipeline code. Once executed, you'll witness how to deploy and test our web application – first in staging, then in production.

By the end of this chapter, you'll have a fully operational Jenkins CD pipeline that seamlessly automates the entire spectrum of application processes – from code push to your version control system to its final deployment in a live production environment. This transition signifies your evolution from manual, susceptible deployments to a holistic, automated CD pipeline.

As such, we will cover the following main topics in this chapter:

- Writing CD pipeline code
- CD in action

Technical requirements

Before you begin, make sure that you have the following tools installed in your local development environment:

- An IDE such as Visual Studio Code or anything you wish
- Git

For the source code, please refer to this chapter's folder in this book's GitHub repository:

https://github.com/PacktPublishing/Learning-Continuous-Integration-with-Jenkins_Third-Edition.

Writing CD pipeline code

In this section, we will systematically delve into our CD pipeline code. This includes five new pipeline stages that ensure the streamlined and efficient deployment of our application to both staging and production environments. We'll kick off by updating the Helm chart configurations, carefully ensuring they align with the current deployment specifics. Following this, we'll employ Argo CD to

ascertain the robust synchronization and health status of our deployed application in different environments. Subsequently, performance testing will be conducted to validate the application's robustness and readiness for production deployment. Lastly, ensuring the health and synchronization status in the production environment will mark the culmination of this systematic and structured CD pipeline, ensuring our application is seamlessly and efficiently deployed. Let's look at the CD pipeline code stage by stage.

Updating Jenkins agent settings

In our Jenkins pipeline code, to support the execution of the newly added pipeline stages, we must add two additional containers to our pod template called **git** and **alpine**, as shown here:

```
- name: git
  image: bitnami/git
  command:
    - sleep
  args:
    - 99d
- name: alpine
  image: alpine
  command:
    - sh
  args:
    - -c
    - "while true; do sleep 86400; done"
```

Here is an explanation of the **git** container:

- **name: git:** This sets the name of the container to **git**. This name is used to reference the container elsewhere in the Jenkinsfile, as you will see later in this chapter.
- **image: bitnami/git:** This uses the **git** image from Bitnami, which provides a minified environment for git operations.
- **command: sleep:** This command is used to make the container sleep.
- **args: 99d:** This argument makes the container sleep for 99 days. This is a way to keep the container running for a long time so that it's available to execute Git commands when needed by the Jenkins pipeline. However, it's important to note that this does not imply the pipeline will run for 99 days; this duration is just a precautionary measure to prevent premature container termination.

Here is an explanation of the **alpine** container:

- **name: alpine:** This sets the name of the container to **alpine**.
- **image: alpine:** This uses the **alpine** Linux image, a minimal Docker image based on Alpine Linux with a complete package index and only a small number of installed utilities.
- **command: sh:** This command is used to run a shell within the container.
- **args: -c "while true; do sleep 86400; done":** This argument runs a shell command that enters an infinite loop, making the container sleep for one day (86,400 seconds) at a time. This is another way to keep the container running so that it's available for executing shell commands when needed by the Jenkins pipeline.

In essence, both containers are kept running for a long duration to ensure they are available for use by the Jenkins pipeline as needed, with the `git` container specialized for Git operations and the `alpine` container for general shell operations.

Configuring pipeline options and the environment

The `disableConcurrentBuilds()` directive inside the `options {...}` section is used to prevent concurrent executions of the pipeline. If a new build is triggered while a previous build is still running, the new build will be queued until the previous build is completed. Only use this option in the Jenkinsfile on the main/master branch. Here's why `disableConcurrentBuilds()` is essential in certain scenarios:

- **Prevents overlapping deployments:** If your pipeline is responsible for deploying applications or changes to a shared environment (such as staging or production), concurrent builds could attempt to deploy at the same time. This concurrent deployment can lead to conflicts, errors, and unpredictable behavior in your environment.
- **Database migrations:** If your pipeline includes database migrations or modifications, concurrent builds can cause problems by trying to apply the same changes concurrently, leading to potential data corruption or inconsistent states.
- **Sequential changes:** Ensuring that changes are applied sequentially and in the order they were made is important for consistency and predictability. Disabling concurrent builds helps ensure that each change is applied and tested individually, and in the correct order.
- **Testing:** If automated tests are part of your pipeline, concurrent execution might lead to conflicts and unreliable test results. For instance, two builds might try to access or modify a shared resource or configuration, leading to false test failures or passes. This can also happen during load testing, which typically requires significant system resources (CPU, memory, network bandwidth, and so on), to simulate high traffic and usage scenarios. Concurrent builds can lead to resource contention, where multiple tests compete for limited resources, potentially skewing test results or causing unexpected test failures.
- **Manages system load:** Limiting concurrency helps in managing the load on the Jenkins server and other associated systems (such as test or deployment environments). This limitation prevents the systems from being overwhelmed by too many simultaneous tasks.
- **Troubleshooting:** Disabling concurrent builds makes it easier to troubleshoot issues and failures. When multiple builds run concurrently, it's harder to determine which build caused a specific issue.

The `environment {...}` block in a Jenkinsfile is used to define environment variables that will be accessible to all stages of the Jenkins pipeline:

```
environment {  
    ARTIFACTORY_SERVER = '<Artifactory Server IP Address>'  
    ARGOCD_SERVER = 'http://<ArgoCD Server IP Address>'  
    ARGOCD_APP_NAME_STG = 'hello-world-staging'  
    ARGOCD_APP_NAME_PROD = 'hello-world-production'  
    ARGOCD_TOKEN = credentials('argocd-jenkins')  
}
```

Here's a breakdown:

- **ARGOCD_SERVER**: This variable holds the URL of the Argo CD server. The pipeline will interact with this server to manage application deployments. Make sure you replace it with the Ingress IP of your Argo CD.
- **ARGOCD_APP_NAME_STG**: This variable is assigned the name of the Argo CD application for the staging environment (**hello-world-staging**).
- **ARGOCD_APP_NAME_PROD**: Similarly, this variable holds the name of the Argo CD application for the production environment (**hello-world-production**).
- **ARGOCD_TOKEN**: The `credentials('argocd-jenkins')` function fetches a stored secret from Jenkins (presumably an authentication token for Argo CD). This token is assigned to the **ARGOCD_TOKEN** variable, allowing the pipeline to authenticate with the Argo CD server when making API requests.

This setup keeps the pipeline code clean and maintainable as these variables are defined in one place and can be used throughout the Jenkinsfile, ensuring consistency and ease of updates.

Update Staging Helm Chart Configuration

Our CI pipeline ends with the *Publish build info* stage. After this, we must add a new pipeline stage, titled *Update Staging Helm Chart Configuration*, that executes a series of shell commands within a `git` container to update the `values-staging.yaml` and `Chart.yaml` files in a GitHub repository with new image tags and versions.

Here is a breakdown of what each part of the code does:

- `container('git')`: This specifies that the following steps should be run inside the `git` container.
- `withCredentials(...)`: This part uses the credentials with the `github-repo-jenkins` ID from Jenkins to authenticate against GitHub. It sets the username and password for use in the subsequent commands.
- `curl -L https://github.com/mikefarah/yq/releases/latest/download/yq_linux_amd64 -o /usr/bin/yq && \` and `chmod +x /usr/bin/yq`: These commands download and install `yq`, a command-line YAML processing tool.
- `git clone https://$username:$password@<GitHub Repo URL> && cd <GitHub Repo Name>`: This clones the GitHub repository and changes the current directory to the cloned repository. Make sure you change the GitHub repository URL and GitHub repository name appropriately in the pipeline code.
- `git config user.email "${env.GIT_COMMITTER_EMAIL}"` and `git config user.name "${env.GIT_COMMITTER_NAME}"`: These commands configure the git user email and name for making commits.
- `yq eval ... values-staging.yaml -i`: This command uses `yq` to update the image tags in the `values-staging.yaml` file with the **VERSION** and **BUILD_NUMBER** variables from the environment.
- `yq eval '.version = "${VERSION}"' Chart.yaml -i`: This command uses `yq` to update the version in the `Chart.yaml` file with the **VERSION** variable from the environment.
- `git add .`: This adds all the changes to the staging area.
- `git commit -m "Updated values-staging.yaml and Chart.yaml with new configurations"`: This commits the changes with a descriptive message.
- `git push origin main`: This pushes the committed changes to the `main` branch of the repository.

So, in essence, this Jenkins pipeline stage is used to automatically update the `values-staging.yaml` and `chart.yaml` files in a specific GitHub repository with new configuration values.

Verify Staging Deployment Health

The *Verify Staging Deployment Health* Jenkins pipeline stage is used to check the synchronization and health status of a deployed application in a staging environment using Argo CD's API.

Here is a breakdown of the code:

- **container('alpine')**: This directive specifies that the commands should run inside an Alpine Linux container.
- **apk add --no-cache jq curl**: This command installs **jq** (a lightweight and flexible command-line JSON processor) and **curl** (a command-line tool for getting or sending data, including files, using URL syntax) inside the Alpine container without caching them.
- **DEPLOYMENT_INFO=\$(curl -s -H "Authorization: Bearer \$ARGOCD_TOKEN" "\$ARGOCD_SERVER/api/v1/applications/\$ARGOCD_APP_NAME_STG")**: This executes a **curl** command to make a GET request to the Argo CD API to retrieve information about a specific application (denoted by **\$ARGOCD_APP_NAME_STG**) that's been deployed in the staging environment.
- The **-s** flag tells **curl** to operate in "silent" mode, where it doesn't show progress information or error messages.
- **-H "Authorization: Bearer \$ARGOCD_TOKEN"** includes an authorization header with a bearer token, **\$ARGOCD_TOKEN**, for authenticated access to the Argo CD API.
- The retrieved deployment information is stored in the **DEPLOYMENT_INFO** variable.
- **SYNC_STATUS=\$(echo \$DEPLOYMENT_INFO | jq -r '.status.sync.status')**: This extracts the synchronization status from the **DEPLOYMENT_INFO** JSON response using **jq**.
- The **-r** flag with **jq** outputs raw strings, not JSON-encoded ones.
- The extracted sync status is stored in the **SYNC_STATUS** variable.
- **HEALTH_STATUS=\$(echo \$DEPLOYMENT_INFO | jq -r '.status.health.status')**: Similar to the preceding point, this extracts the health status from the **DEPLOYMENT_INFO** JSON response. The extracted health status is stored in the **HEALTH_STATUS** variable.
- **echo "Sync Status: \$SYNC_STATUS"** and **echo "Health Status: \$HEALTH_STATUS"**: These commands print out the sync and health status to the console for verification.

In essence, this stage fetches and displays the synchronization and health status of an application deployed in the staging environment using Argo CD, ensuring that the application is properly deployed and operating as expected.

Run Performance Testing

The next Jenkins pipeline stage is titled **Run Performance Testing**. This stage is responsible for executing performance tests on the deployed application in the staging environment.

Here is an explanation of each step in the provided script:

- **container('alpine')**: This command specifies that the following commands are to be executed within an Alpine Linux container.
- **curl -L <https://github.com/grafana/k6/releases/download/v0.46.0/k6-v0.46.0-linux-amd64.tar.gz> -o k6.tar.gz**: This command downloads the specified version of k6, a modern load testing tool, from its GitHub releases page. The **-L** flag tells **curl** to follow redirects, and **-o k6.tar.gz** saves the downloaded file as **k6.tar.gz**.
- **tar xzf k6.tar.gz**: This command extracts the downloaded **tar.gz** file, **k6.tar.gz**. The **z** flag is for gzip files, **x** is for extract, and **f** specifies the file to extract.
- **mv k6-v0.46.0-linux-amd64/k6/usr/local/bin/**: This command moves the **k6** binary from the extracted directory to **/usr/local/bin**, making it available for execution from anywhere in the system.
- **chmod +x /usr/local/bin/k6**: This command grants execute permissions to the **k6** binary, allowing it to be run as a program.
- **k6 run performance-tests/performance-test.js**: This command runs the performance test script (**performance-tests/performance-test.js**) using k6. This script contains the performance test definition and configuration for the application.

In summary, this stage automates the process of downloading and installing k6 and then uses it to execute a predefined performance test script against the application deployed in the staging environment. The results of these tests provide insights into the application's performance characteristics under load, helping to identify any performance-related issues before the application is deployed to production.

Let's look at the **performance-test.js** script:

```
import http from 'k6/http';
import { sleep } from 'k6';
export const options = {
  duration: '1m',
  vus: 50,
  thresholds: {
    http_req_duration: ['p(95)<500'], // 95 percent of response times must be below 500ms
  },
};
export default function () {
  http.get('http://<IP Address of Ingress in Staging>');
  sleep(3);
}
```

This k6 script is designed for performing load testing on a web application. It is configured to simulate 50 virtual users sending HTTP GET requests to the **http://<IP Address of Ingress in staging>/** URL for one minute. After each request, the script pauses for three seconds before sending

the next request. Make sure you change this URL to the URL of your web application Ingress that's running in staging.

The script specifies a performance threshold, asserting that 95% of the HTTP request responses should be returned in under 500 milliseconds. This threshold setting helps in asserting the performance expectations and will mark the test as failed if the response times exceed the specified limit.

In essence, this script is a basic load testing tool that checks the responsiveness and performance of the specified web server under a simulated load of 50 concurrent users. To learn more about performance testing using k6, visit *Using k6* (see [1] in the *Further reading* section at the end of this chapter).

Update Production Helm Chart Configuration

The *Update Production Helm Chart Configuration* Jenkins pipeline stage is quite like the earlier stage for updating the staging Helm chart configuration. The significant difference lies in the file it is modifying: `values-production.yaml` instead of `values-staging.yaml`.

This stage performs the following tasks within a `git` container:

- It enters the repository directory
- The `yq` tool is used to update `backend.image.tag` and `frontend.image.tag` in the `values-production.yaml` file with the current `VERSION` and `BUILD_NUMBER` variables
- These changes are then added to the Git staging area, committed with a message, and pushed back to the main branch of the original repository

So, while the earlier stage was used to update the staging configuration, this stage is used to update the production configuration in the `values-production.yaml` file.

Verify Production Deployment Health

The *Verify Production Deployment Health* stage is analogous to the previously explained *Verify Staging Deployment Health* stage, with the key difference residing in the targeted environment: it now checks the health of the production deployment instead of the staging deployment.

There are various aspects of this stage that we must consider:

- It operates within an `alpine` container.
- Information about the application deployed in the production environment is fetched using a `curl` request to the Argo CD API. This information is stored in the `DEPLOYMENT_INFO` variable.

- **SYNC_STATUS** and **HEALTH_STATUS** are extracted from **DEPLOYMENT_INFO** using **jq** to parse the JSON response.
- These statuses are then echoed out to give insights into the synchronization and health of the production deployment.

The overall purpose of this stage is to ensure that the application has been successfully deployed and is running healthily in the production environment.

With this, we can wrap up our detailed exploration into writing the CD pipeline code. Now, it's time to witness the fruit of our labor. In the next section, we will unleash the orchestrated power of our CD setup. You will see each stage of the pipeline, from updating the Helm chart configurations to verifying the deployment's health, seamlessly working together to ensure an automated and reliable deployment process. Let's move on and watch the CD in action!

CD in action

Welcome to this exciting section. We are set to observe the seamless automation and efficiency of the CD pipeline that we meticulously set up in the previous sections. Before we set this well-oiled machine into motion, let's make some vital updates to our GitHub repository containing the source code of our web application. Be careful not to make the following changes to the Helm chart's GitHub repository:

- **Replace Jenkinsfile:** Start by substituting the existing Jenkinsfile in your repository with the newly updated one. This file is now equipped with stages pertinent to CD, ensuring that the entire pipeline is CD-ready.
- **Add a performance test file:** Proceed to create a new folder within your repository named **performance-tests**. Here, deposit the **performance-test.js** file. This script is crucial for assessing the performance of the deployed application, ensuring it operates optimally in the production environment.

With these essential files in place, you're all set to trigger the CD pipeline. Posting these updates to the main branch of the GitHub repository will automatically set the CD pipeline in motion, demonstrating the automation, efficiency, and reliability of the deployment process. Let's dive in and see it unfold live!

Verifying changes on the Helm chart

Verifying that the Helm chart has been updated by Jenkins in the GitHub UI involves checking for the specific commit that Jenkins would have made to update the Helm chart. Here are step-by-step instructions for doing this:

1. Open your web browser and navigate to the GitHub repository where your Helm chart is stored. Make sure you are logged in to GitHub to view private repositories.
2. Go to the main page of the repository. Click on the **Commits** tab, which is generally located next to the **Code** tab just below the repository's name. Here, you will see a list of all the commits made to the repository.

3. Examine the commits to find the one made by Jenkins. Jenkins commits will typically have a specific format or be made by a user that's associated with Jenkins. Look for commits with messages related to Helm chart updates or other identifying information. It should look something like this:

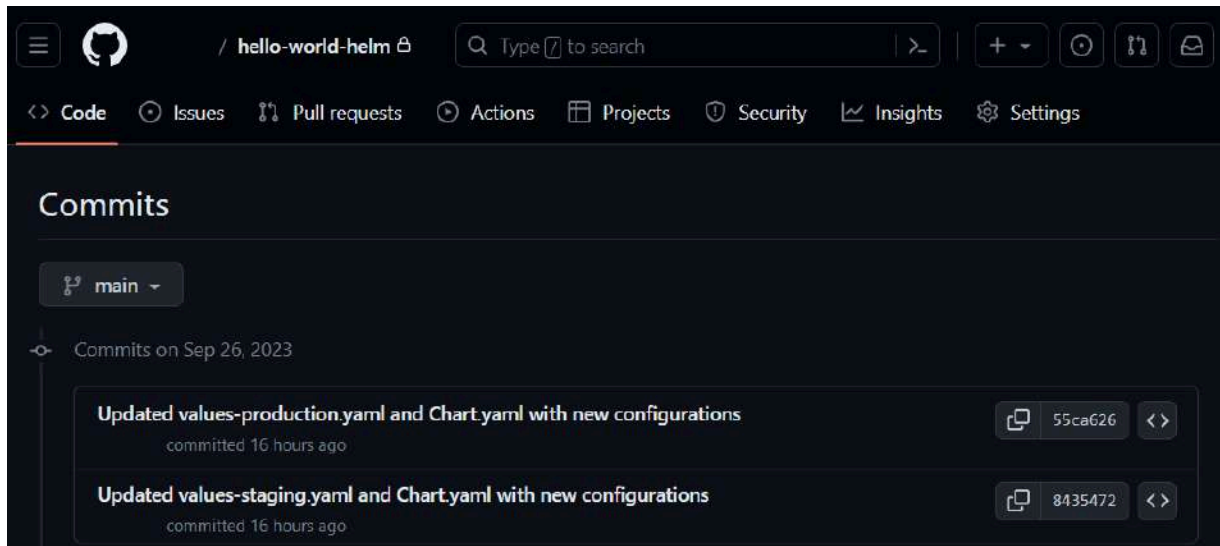


Figure 11.1 – Automated commit on the Helm chart repository by the CD pipeline

4. Click on the commit to view the detailed information provided. Review the changes that have been made to the files in this commit to ensure that Jenkins has made the appropriate updates to the Helm chart.

This step-by-step process should help you verify (in the GitHub UI) that Jenkins has successfully updated the Helm chart.

Verifying the deployment in staging and production

To view the images used by an application in the Argo CD web UI, you can follow these general steps. The exact steps might vary depending on the version of Argo CD you're using and the specific setup of your environment:

1. Open your web browser and navigate to the Argo CD web UI. The URL will be dependent on your specific setup. Log in using your credentials.
2. Once you've logged in, you will be taken to the **Applications** page by default. If not, click on **Applications** in the left sidebar.
3. You will see a list of all your Argo CD applications. Locate and click on the specific application you want to view the images for. In our case, it is either **hello-world-staging** or **hello-world-production**.
4. After clicking on the application, you'll be taken to a detailed view of the application. Here, you can see the status, health, and other information about your application:

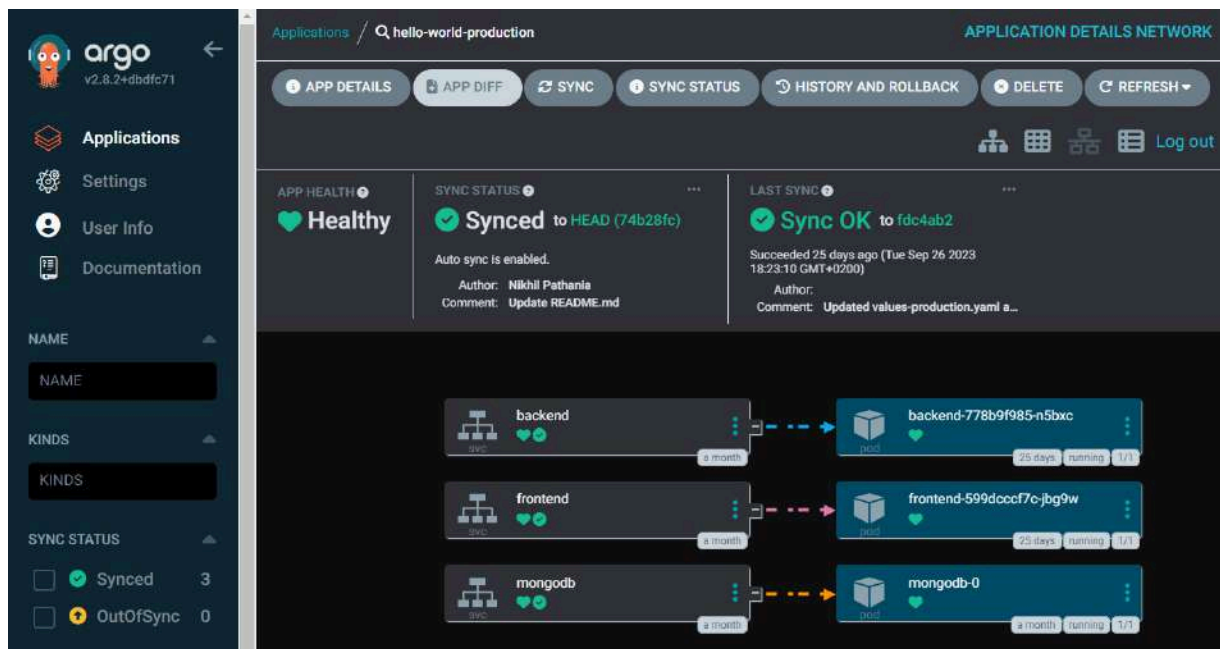


Figure 11.2 – Viewing the application’s health and status in Argo CD

- From here, click on the **APP DETAILS** button at the top; you will end up on the **SUMMARY** page, which will show you a summary of your deployment, including **IMAGES**:

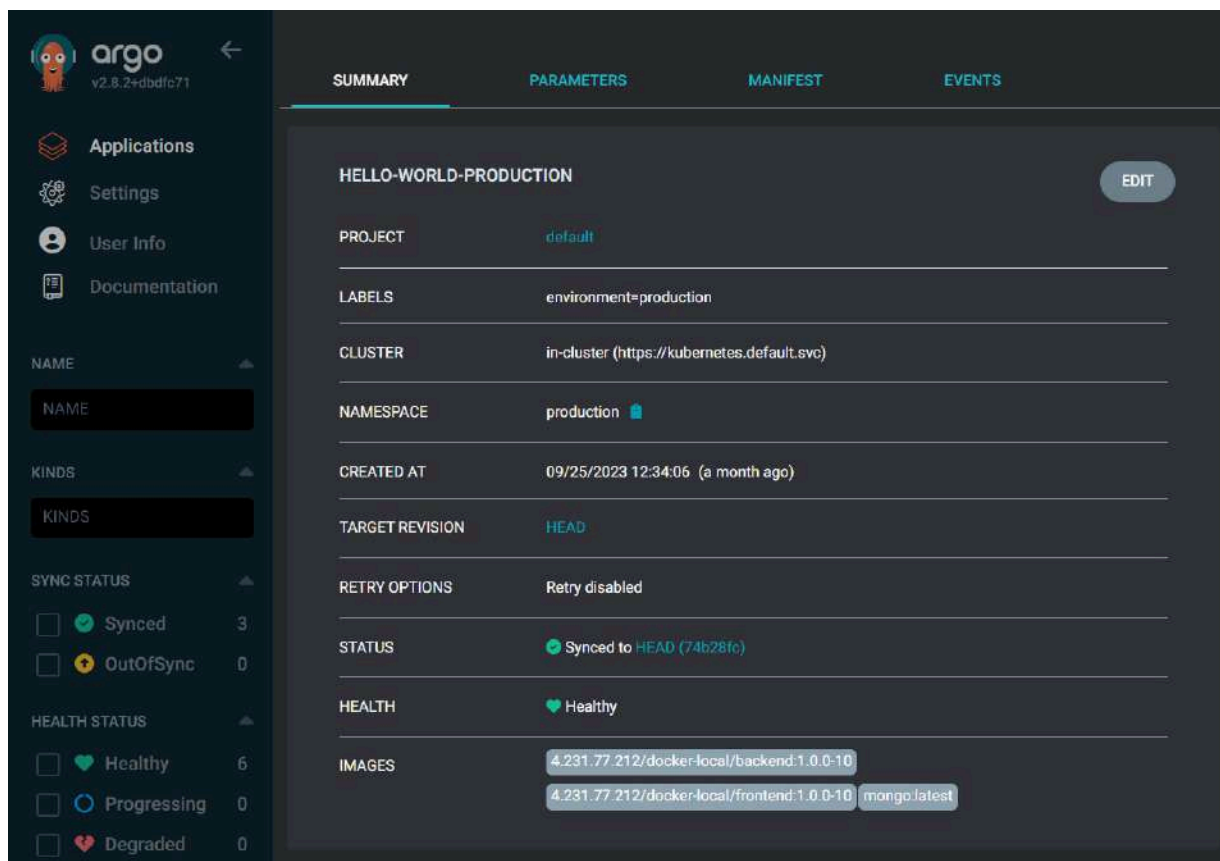


Figure 11.3 – Viewing the deployed image versions in Argo CD

These steps should guide you in viewing the images used by an application in the Argo CD web UI. Ensure the images have the correct expected tags. This information can also be verified by examining the Jenkins pipeline logs, specifically within the *Verify Staging Deployment Health* stage and the *Verify Production Deployment Health* stage.

Visualizing performance testing results

k6, a popular performance testing tool, does not inherently store test result data. It is designed to work with various backend systems where it can send its results for further analysis. In our example, the results are directly displayed in the console. You can access these results by viewing the pipeline logs within the **Run Performance Testing** stage:

```
205 data_received.....: 536 kB 8.9 kB/s
206 data_sent.....: 77 kB 1.3 kB/s
207 http_req_blocked.....: avg=110.51µs min=900ns med=2µs max=9.65ms p(90)=5.51µs p(95)=25.8µs
208 http_req_connecting.....: avg=84.37µs min=0s med=0s max=5.35ms p(90)=0s p(95)=5.9µs
209 ✓ http_req_duration.....: avg=1.45ms min=205.79µs med=1.11ms max=13.68ms p(90)=2.47ms p(95)=3.21ms
210 { expected_response:true }...: avg=1.45ms min=205.79µs med=1.11ms max=13.68ms p(90)=2.47ms p(95)=3.21ms
211 http_req_failed.....: 0.00% ✓ 0 X 1000
212 http_req_receiving.....: avg=63.8µs min=9.29µs med=21.9µs max=8.16ms p(90)=53.72µs p(95)=88.2µs
213 http_req_sending.....: avg=96.4µs min=4.4µs med=7.9µs max=8.53ms p(90)=62.21µs p(95)=144.27µs
214 http_req_tls_handshaking.....: avg=0s min=0s med=0s max=0s p(90)=0s p(95)=0s
215 http_req_waiting.....: avg=1.29ms min=189.69µs med=1.07ms max=9.34ms p(90)=2.33ms p(95)=2.94ms
216 http_reqs.....: 1000 16.653021/s
217 iteration_duration.....: avg=3s min=3s med=3s max=3.01s p(90)=3s p(95)=3s
218 iterations.....: 1000 16.653021/s
219 vus.....: 50 min=50 max=50
220 vus_max.....: 50 min=50 max=50
221
222
223 running (1m00.0s), 00/50 VUs, 1000 complete and 0 interrupted iterations
224 default ✓ [ 100% ] 50 VUs 1m0s
```

Figure 11.4 – Performance testing results as viewed from the pipeline logs

Let's consider a summary of the performance testing results shown here.

The performance test was executed for one minute with a constant **virtual user (VU)** load of 50. During this period, a total of 1,000 HTTP requests were made, with a throughput of approximately 16.65 requests per second. No requests failed, indicating a 100% success rate in responses from the server. Here are some key observations:

- **Data transfer:**
 - The total data received from the server was 536 kB at an average rate of 8.9 kB/s
 - The total data sent to the server was 77 kB at an average rate of 1.3 kB/s
- **Request duration:**
 - The average HTTP request duration was 1.45 ms

- 95% of the requests were processed within 3.21 ms, and the longest request took 13.68 ms
- **Request components:**
 - The average time blocked was very low, at 110.51 μ s, with a maximum of 9.65 ms
 - The average connecting time was 84.37 μ s, peaking at 5.35 ms
 - The receiving time and sending time averaged 63.8 μ s and 96.4 μ s, respectively
 - The waiting time, representing server processing time, averaged 1.29 ms with a max of 9.34 ms
- **Iteration duration:** The average iteration duration was constant at three seconds

In conclusion, the performance testing results display a stable and efficient system that handles 50 VUs efficiently with a satisfactory average request duration and no failed requests. The detailed breakdown of request phases further confirms minimal delays, signaling optimal server performance under the given load conditions.

Summary

In this chapter, we concentrated on crafting the CD pipeline code. We learned how to interact programmatically with GitHub and Argo CD using Jenkins pipeline code and conducted performance testing using k6, a straightforward performance testing tool. In the concluding stages, we engaged in the actual deployment, triggering a CD pipeline. Watching CD unfold, we scrutinized changes in the Helm chart and validated the deployments in both the staging and production environments. This chapter concluded with a hands-on demonstration of visualizing performance testing results, ensuring our proficiency in both the theoretical knowledge and practical skills essential for creating a robust CD pipeline.

In the upcoming chapter, we'll delve into advanced CI/CD pipeline strategies. We'll master the art of designing efficient pipeline templates using Jenkins shared libraries, guaranteeing streamlined and standardized configurations. Additionally, we'll explore how to detect vulnerabilities early with a binary repository manager, bolstering our code base's security, and more.

Questions

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

1. In which of the following scenarios would you use the `disableConcurrentBuilds()` option in Jenkins?
 - A. When running jobs that require exclusive access to certain resources, such as a deployment to a specific environment
 - B. To ensure that the results of jobs, such as load testing, are reliable and not affected by other concurrent job executions
 - C. To maintain a predictable and stable environment for specific jobs where the order of execution is critical

- D. All of the above
2. What is the purpose of the `environment {}` block inside Jenkins pipeline code?
- A. To define environment variables that are accessible by all stages in the pipeline
 - B. To set specific configuration parameters and credentials that the pipeline can use during execution
 - C. To override global environment variables for a specific pipeline run
 - D. All of the above
3. Why is it necessary to check the deployment status and health of an application in a staging environment before running extensive testing in a Jenkins CD pipeline using Argo CD?
- A. To ensure the application's dependencies are correctly configured and operational
 - B. To confirm that the application is successfully deployed and functioning as expected, which is critical for obtaining accurate performance testing results
 - C. To validate that the correct version of the application has been deployed for testing
 - D. All of the above
4. Which of the following method/methods can be used to fetch the health status of an application running on Argo CD programmatically?
- A. Using the Argo CD CLI
 - B. Using the Argo CD API
 - C. Using `kubectl`
 - D. Using the Argo CD UI
5. Choose whether this statement is true or false: The sync status of an application is the same as its health status.

Answers

1. Correct answer: **D**

Explanation:

Because the `disableConcurrentBuilds()` option is beneficial in all these scenarios to avoid conflicts, ensure reliable results, and maintain a predictable execution environment.

2. Correct answer: **D**

Explanation:

Because the `environment {}` block in Jenkins pipeline code is used for setting accessible environment variables across all stages, configuring specific parameters and credentials, and overriding global environment variables for specific pipeline runs.

3. Correct answer: **D**

Explanation:

Checking the deployment status and health of an application in a staging environment before running performance testing in a Jenkins CD pipeline using Argo CD is essential to ensure that dependencies are correctly configured, the application is successfully deployed and functioning as expected, and the correct version of the application is tested, all of which are crucial for obtaining accurate and meaningful performance testing results.

4. Correct answers: **A** and **B**

The Argo CD CLI provides a command, `argocd app status <application-name>`, that can be used to fetch the health status of an application. The health status is reported as either **Healthy** or **Unhealthy**. The Argo CD API does not provide a direct way to fetch the health status of an application. However, the health status can be inferred from the `sync.status` and `health.status` fields of the application resource. Additionally, both methods can be programmed.

Explanation of incorrect answer choices:

C. `kubectl` can be used to get the status of the pods that are part of an application. However, this does not provide a complete picture of the health of the application as it does not take into account the health of the application's resources.

D. The Argo CD UI provides a way to view the health status of an application. However, this is not as flexible as using the CLI or API as it does not allow for filtering or sorting the results. Also, it's a manual process.

5. Correct answer: **False**

Explanation:

The sync status of an application indicates whether the desired state of the application in Git is the same as the actual state of the application in the Kubernetes cluster. The health status of an application indicates whether the application is functioning correctly. For example, an application might be in sync but unhealthy if one of its pods is not running. Conversely, an application might be out of sync but healthy if the desired state of the application in Git is not the same as the actual state of the application in the Kubernetes cluster because it has been updated manually.

Further reading

If you wish to learn more about some of the concepts that are not covered in depth in this chapter, or are outside the scope of this book but relevant, the following is a reference:

- [1] Using k6: <https://k6.io/>

Enhancing Your CI/CD Pipelines

In this final chapter, we delve into a few advanced yet practical techniques to elevate the efficiency and robustness of your **continuous integration (CI)** and **continuous delivery (CD)** processes.

Firstly, we'll introduce GitHub Copilot, a tool that employs artificial intelligence to aid in the development of Jenkins pipeline code. It's not just about automating the code; it's also about making the code smarter, ensuring fewer errors, and increasing development speed.

Next on the list is Jenkins Shared Libraries. As your projects grow, so does the complexity of managing multiple pipelines. Shared Libraries help streamline this by centralizing common code patterns, enabling you to maintain consistent and efficient pipelines across multiple projects with ease.

Builds accumulate quickly in Jenkins, and without proper management, they can congest your system. We'll address strategies to manage and discard old builds, ensuring your CI/CD environment remains optimized and resource-efficient.

But what about scanning our builds for security vulnerabilities? That's where JFrog Xray comes into play. Here, you will be guided on how to seamlessly integrate it with your Jenkins CI pipeline, enabling automated scanning of your builds for vulnerabilities and ensuring that the code you deploy is not just functional but also secure. While this chapter covers these specific advanced topics, remember that the essence of an effective CI/CD pipeline is covered throughout the book.

As such, we will be covering the following topics in the chapter:

- Using GitHub Copilot to write pipeline code
- Using Jenkins Shared Libraries
- Discarding old builds in Jenkins
- Catching security vulnerabilities early in code

Technical requirements

For the current chapter, you need the following tools installed on your local development environment:

- An IDE such as **Visual Studio Code (VS Code)** or anything you choose
- Git

- The Azure CLI, to interact with the Azure cloud
- Helm to deploy workloads on the Kubernetes cluster
- kubectl, the Kubernetes command-line tool

You can download the source code of this chapter from https://github.com/PacktPublishing/Learning-Continuous-Integration-with-Jenkins_Third-Edition.

Using GitHub Copilot to write pipeline code

GitHub Copilot, developed in collaboration between GitHub and OpenAI, is an AI-driven coding assistant designed to streamline and enhance the developer experience. Integrated primarily as a VS Code extension, Copilot is powered by the Codex language model, trained on a multitude of public code repositories from GitHub. This extensive training enables it to provide context-aware code suggestions, spanning from individual lines to entire functions, across multiple programming languages. Its proficiency encompasses generating comments, offering multiple ways to code a function, and assisting in rapid prototyping or when diving into unfamiliar frameworks.

However, it's crucial to approach Copilot as a tool to augment, not replace, the developer's expertise. While its suggestions can be remarkably accurate and time-saving, they aren't infallible. Developers need to critically review the code snippets offered, ensuring both their functional fit and compliance with any licensing requirements, given that Copilot's knowledge originates from a vast array of public repositories.

In essence, GitHub Copilot represents a fusion of AI's potential with coding's intricacies, aiming to elevate productivity, accelerate learning curves, and make coding more intuitive. Yet, its use should be paired with diligence and an understanding of the underlying code logic.

Enabling GitHub Copilot

If your organization uses GitHub, inquire with your IT department about the availability of GitHub Copilot for Business. Otherwise, you'll need to use your personal GitHub account. To activate GitHub Copilot on your individual account, follow these steps:

1. Log in to your GitHub account. In the top-right corner, select your profile picture, and then click on **Settings**.
2. Within the sidebar under the **Code, planning, and automation** category, select **Copilot**.
3. On the GitHub Copilot settings page, choose **Start free trial**.

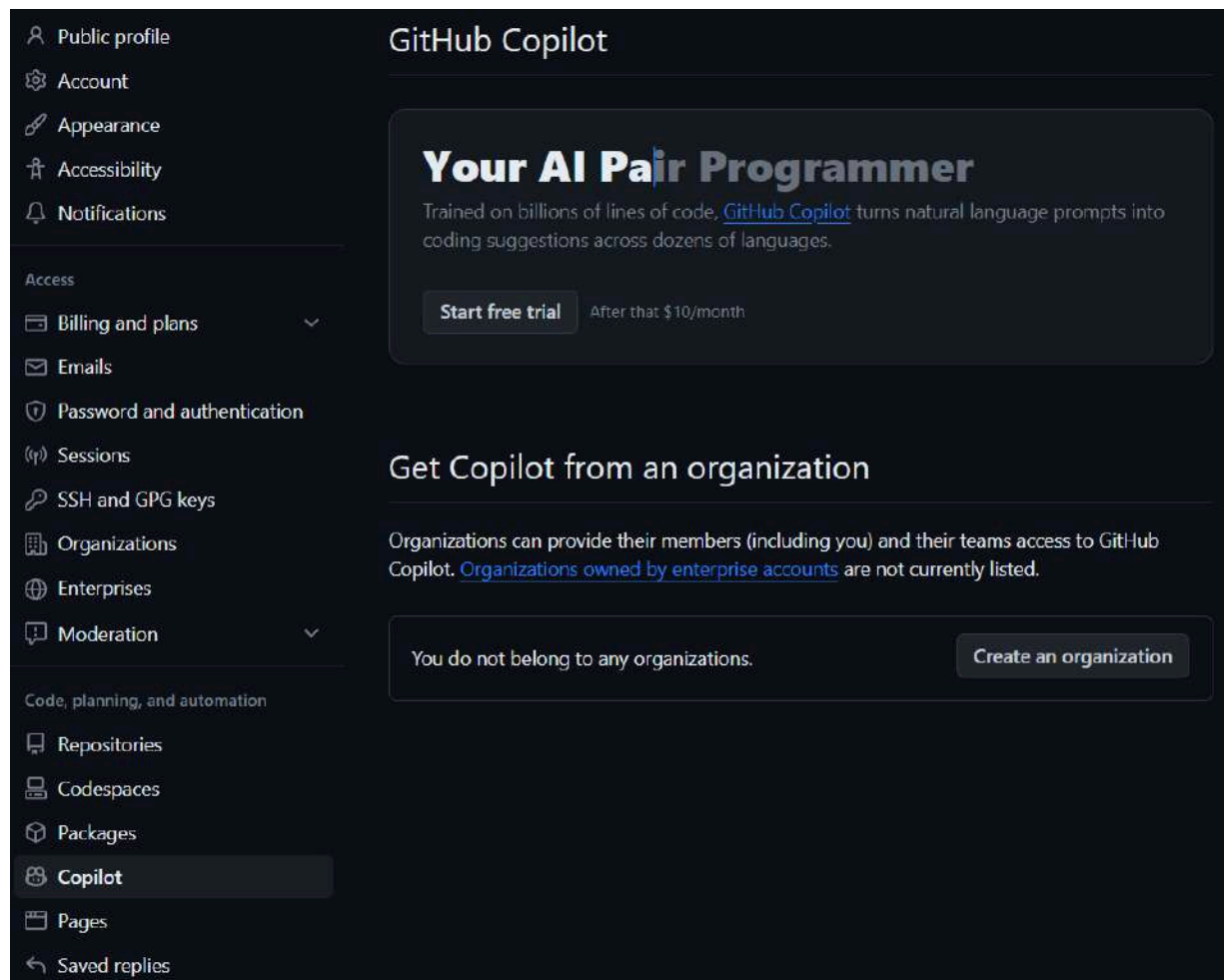


Figure 12.1 – Enabling GitHub Copilot

- Next, decide on a billing frequency – monthly or annually – and click **Get access to GitHub Copilot**. Proceed to verify your payment information, and then hit **Submit**.
- Finalize your choices, and then select **Save and get started**.

IMPORTANT NOTE

As per GitHub, GitHub Copilot is free if you happen to be a verified student, teacher, or maintainer of any popular open source project. For everybody else, you can try GitHub Copilot free for a 30-day trial period, after which you pay a monthly subscription fee.

Using Copilot to write pipeline code

In the following section, we'll learn how to use GitHub Copilot inside the VS Code IDE. Before we dive in, ensure you have the GitHub Copilot extension installed. You can locate it in the **Extensions** area of your VS Code IDE.

Additionally, inside VS Code, if you haven't previously linked the software with your GitHub account, you'll be prompted to do so. Conversely, if you established this connection in the past, GitHub Copilot will authenticate automatically. Consequently, in your web browser, GitHub will request certain permissions essential for the Copilot extension. Affirm these by choosing **Authorize Visual Studio Code**. To wrap up the process, navigate back to VS Code. A dialog box will appear, seeking confirmation of the authentication. Solidify this by clicking on **Open**.

Here's an interactive guide to help you explore using GitHub Copilot inside **VS Code** to write Jenkins pipeline code using Declarative Pipeline syntax:

1. Let's start by creating a new Jenkinsfile. Open VS Code, and click on **File | New File**. Save this file as **Jenkinsfile** to give it the appropriate context.
2. Now, let's begin writing some pipeline code with Copilot.
3. Type **pipeline {**, and note that Copilot automatically suggests a basic structure for the Jenkins pipeline, as shown in the following figure.

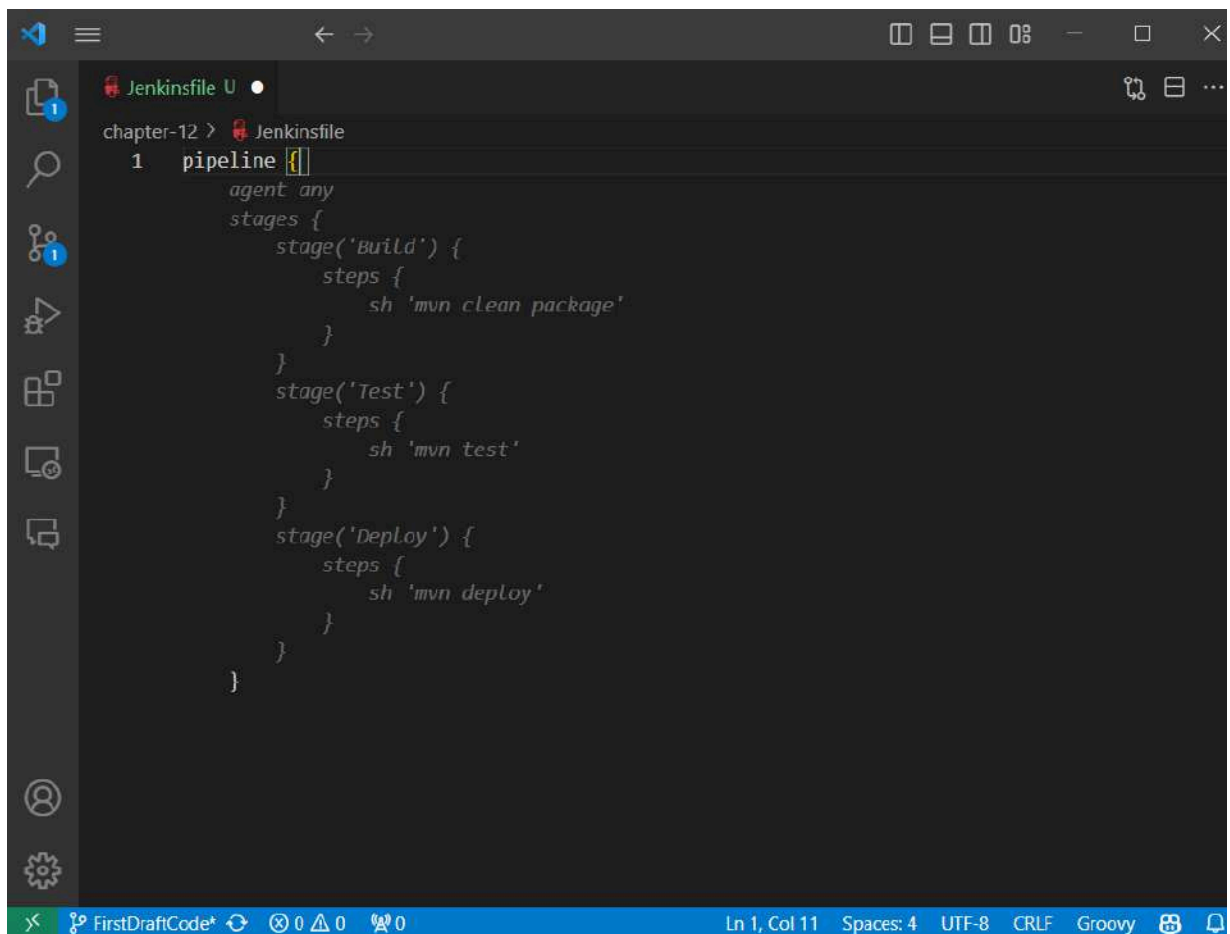
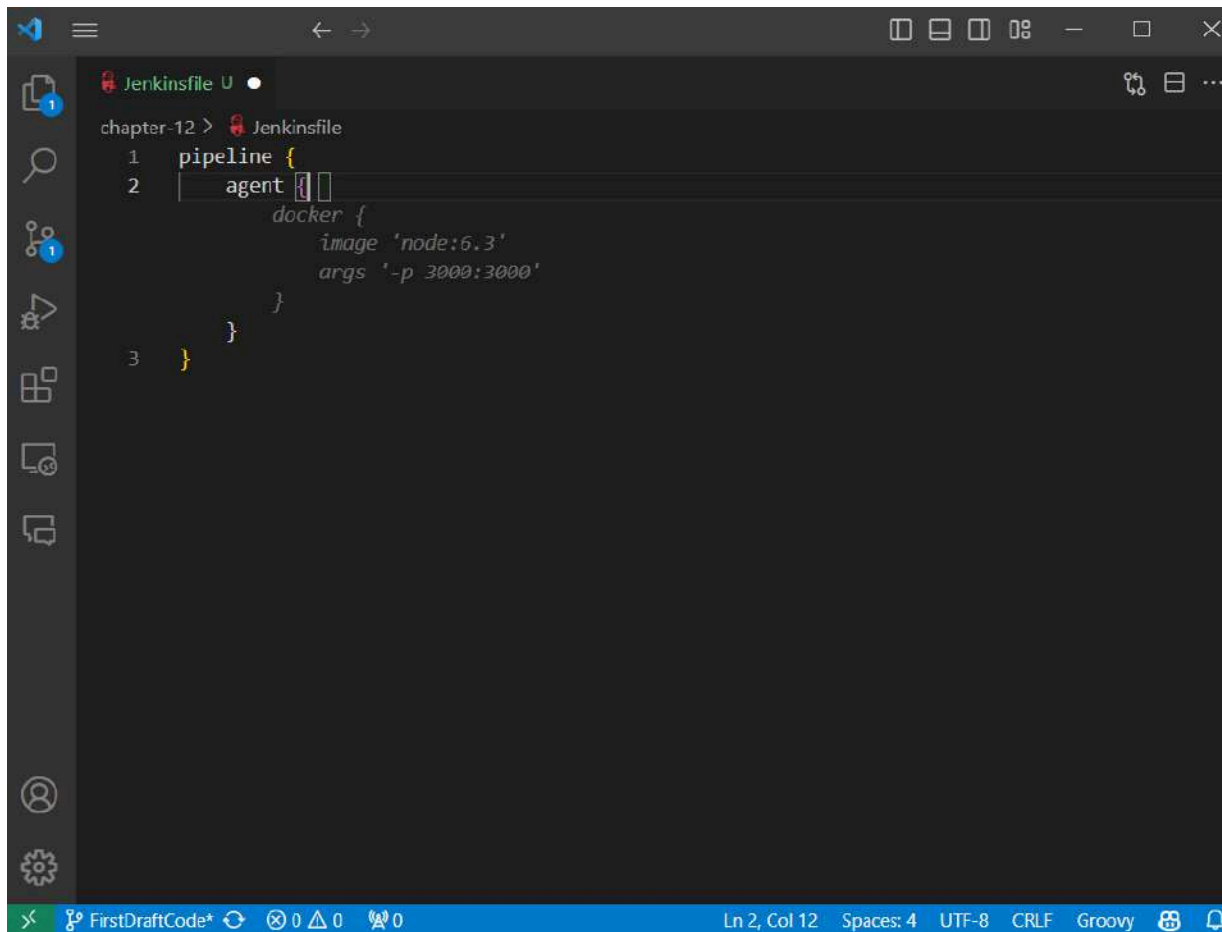


Figure 12.2 – GitHub Copilot in action inside the VS Code IDE

4. Isn't that beautiful? Accept the suggestion if it's what you need, or continue typing to refine the suggestions.

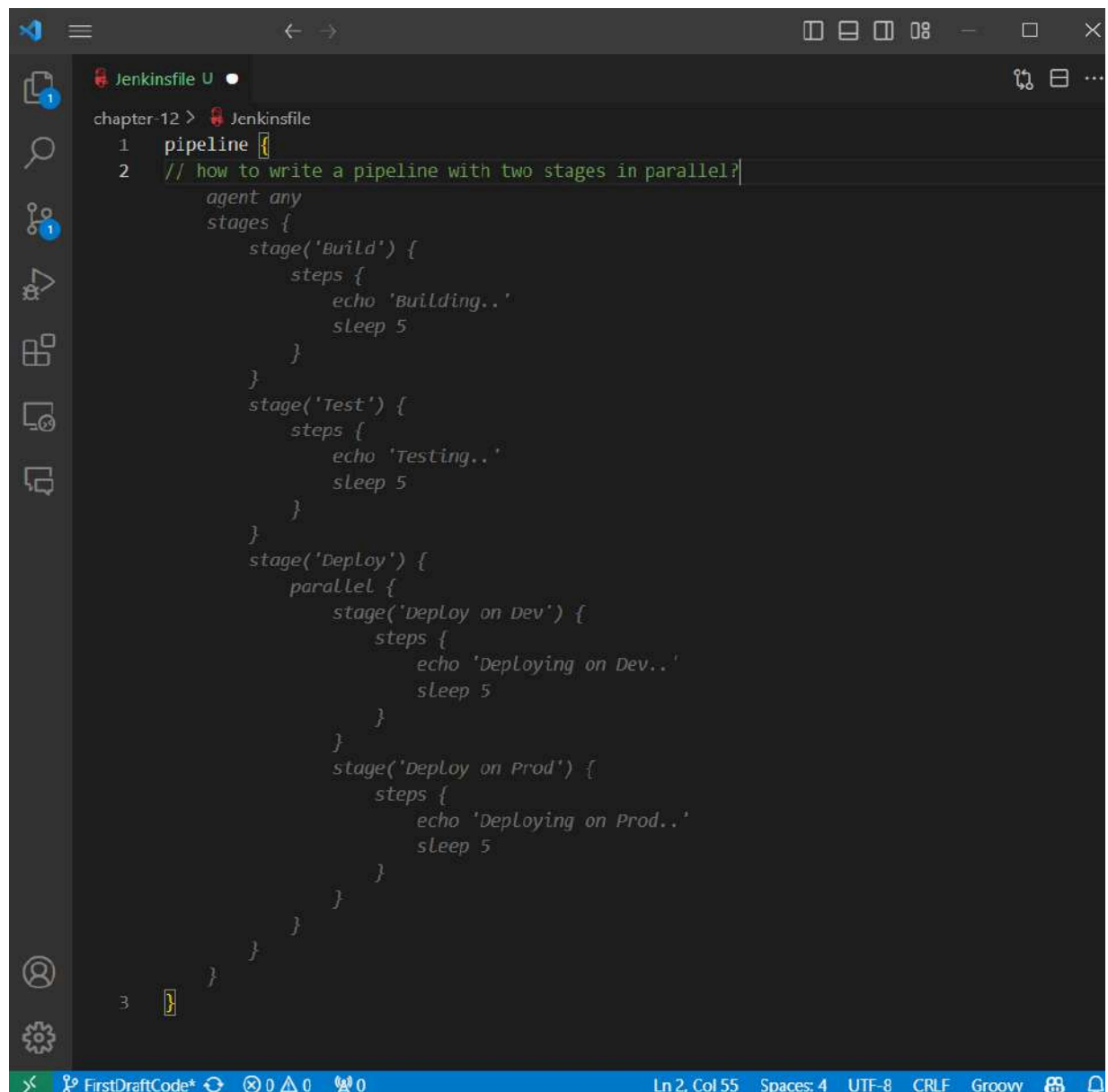
5. In this example, we will ignore the suggestion and instead type **agent {** in the next line, as shown in the following figure:



```
chapter-12 > Jenkinsfile
1 pipeline {
2   agent {
3     docker {
4       image 'node:6.3'
5       args '-p 3000:3000'
6     }
7   }
8 }
```

Figure 12.3 – GitHub Copilot auto-code suggestion

6. Be aware that Copilot may suggest code to use Docker containers as build agents. Click the *Tab* key to accept a suggestion.
7. Next, type **stages {** to begin defining the stages of your pipeline. As you type the names of stages, such as **stage('Build') {**, Copilot might suggest common steps within that stage, such as **steps { sh 'make build' }**.
8. You can continue exploring the Jenkins Declarative Pipeline syntax further using what you learned in [Chapter 6](#).
9. Then, ask Copilot if you're unsure about certain Jenkins syntax or best practices; try typing a comment asking about it. For instance, **// How to write a pipeline with two stages in parallel** might lead Copilot to generate an example for you, as shown in the following figure.



```
chapter-12 > Jenkinsfile
1 pipeline {
2   // how to write a pipeline with two stages in parallel?
   agent any
   stages {
     stage('Build') {
       steps {
         echo 'Building..'
         sleep 5
       }
     }
     stage('Test') {
       steps {
         echo 'Testing..'
         sleep 5
       }
     }
     stage('Deploy') {
       parallel {
         stage('Deploy on Dev') {
           steps {
             echo 'Deploying on Dev..'
             sleep 5
           }
         }
         stage('Deploy on Prod') {
           steps {
             echo 'Deploying on Prod..'
             sleep 5
           }
         }
       }
     }
   }
}
```

The screenshot shows a code editor window titled 'Jenkinsfile U'. The editor displays a Jenkinsfile with a comment on line 2: '// how to write a pipeline with two stages in parallel?'. The code defines a pipeline with an 'agent any' and three stages: 'Build', 'Test', and 'Deploy'. The 'Build' and 'Test' stages are sequential, each with an 'echo' step and a 'sleep 5' step. The 'Deploy' stage uses a 'parallel' block to execute two sub-stages: 'Deploy on Dev' and 'Deploy on Prod', each with its own 'echo' and 'sleep 5' steps. The editor's status bar at the bottom indicates 'Ln 2, Col 55', 'Spaces: 4', 'UTF-8', 'CRLF', and 'Groovy'.

Figure 12.4 – A GitHub Copilot auto-code suggestion using inline comments

While Copilot provides helpful suggestions, always review the generated Jenkinsfile to ensure it meets your specific requirements. Test the Jenkinsfile in a safe environment, such as your feature branch, before merging it with the main. Recall the **describe** | **run** | **troubleshoot** | **refine** method from [Chapter 7](#); that is what you need to do until you get the exact pipeline code you need.

Additionally, GitHub Copilot features a companion tool known as GitHub Copilot Chat. This tool is exclusively available to organizations and enterprises possessing the GitHub Copilot for Business license. GitHub Copilot Chat offers a conversational platform, allowing users to engage with GitHub Copilot within certain IDEs and facilitating real-time answers to coding queries. This interface

eliminates the need to sift through manuals or browse online communities for solutions. As of the time of writing Copilot Chat is compatible with both VS Code and VS. Read more about it on the GitHub docs (see [1] in the *Further reading* section at the end of the chapter).

Remember, GitHub Copilot is a tool to assist and expedite your coding process, but it’s essential to understand and review the suggestions it offers, especially for critical applications such as CI/CD pipelines.

Choosing between Copilot and ChatGPT

GitHub Copilot and ChatGPT, both products of OpenAI’s extensive research in language models, serve distinct yet overlapping purposes in the realm of code assistance. GitHub Copilot, deeply integrated into the developer’s environment via the VS Code IDE, offers real-time, context-aware coding suggestions. Trained specifically on vast amounts of code repositories, Copilot excels in understanding coding patterns, offering entire blocks of code, and even proactively suggesting alternatives or improvements. Its primary aim is to speed up the coding process, reduce the frequency of bugs, and aid developers in navigating unfamiliar frameworks or languages.

In contrast, ChatGPT, while powerful in understanding and generating code, is designed more as a conversational interface. It’s more suited for answering broader queries, explaining concepts, or assisting in brainstorming. While it can generate code snippets, its strength doesn’t lie in real-time code completions or understanding the deeper context of a developer’s ongoing project in the same way Copilot does. Here is a table comparing GitHub Copilot with LLMs, giving you more clarity.

Feature/aspect	GitHub Copilot	Large language models (LLMs)
Primary function	Code completion and generation	Text generation and understanding including code
Use case	Assisting in software development	Diverse, including chat, content creation, problem-solving, and more
Data training	Trained specifically on source code	Trained on a vast array of text data including books, articles, and websites
Language understanding	Focused on programming languages	A broad understanding of natural language

Feature/aspect	GitHub Copilot	Large language models (LLMs)
Customization	Limited to coding contexts	Highly adaptable to various scenarios and contexts
Interaction style	Integrated into development tools	Standalone applications or integrated into various platforms
Output type	Code snippets and solutions	Text across different formats and styles
User base	Primarily software developers	Diverse, including writers, researchers, businesses, and general users
Learning ability	Limited to coding improvements	Continuously learning from diverse text inputs, including coding
Feedback loop	Primarily through code efficiency and accuracy	Broad, including language nuances, factual accuracy, and creative generation

Table 12.1 – A table comparing GitHub Copilot with LLMs

In summary, for coding tasks and real-time assistance within an IDE, GitHub Copilot holds a clear edge, while ChatGPT serves a broader spectrum of conversational and explanatory needs. Now that we’ve explored how GitHub Copilot can assist in writing Jenkins pipeline code, let’s shift our focus to enhancing our CI/CD pipelines further by leveraging Jenkins Shared Libraries for reusable code snippets.

Using Jenkins Shared Libraries

As you journey deeper into the world of Jenkins, you’ll encounter a powerful concept known as **Jenkins Shared Libraries**. Imagine you have multiple Jenkins pipelines across various projects, and many of these pipelines use similar steps or configurations. Instead of replicating these common pieces in every Jenkinsfile, wouldn’t it be more efficient if you could write them once and reuse them across pipelines? That’s precisely where Shared Libraries come into play. They allow you to define and centralize reusable pieces of code, helping streamline pipeline configurations and maintain consistency across projects.

In the real world, large organizations often have standard practices for building, testing, and deploying applications. By encapsulating these practices within Shared Libraries, teams can ensure consistent and efficient pipelines across the board. For example, a company might have a standard way of notifying stakeholders when a deployment is successful. Instead of each team scripting this notification in its own way, a Shared Library can offer a standardized method, ensuring uniformity.

While the intricacies of setting up and using Shared Libraries go beyond a beginner's scope, understanding their purpose and potential benefits can be a springboard to advanced Jenkins mastery. As you continue to evolve your Jenkins expertise, keep Shared Libraries in your toolkit as a solution for scalable, reusable, and efficient CI/CD configurations. Let's learn how to use Shared Libraries with a straightforward example. Once you grasp how it operates, you can further expand and innovate using your own creativity.

Writing a Jenkins Shared Library

Our first step is to create a Jenkins Shared Library hosted in a GitHub repository. The primary reason for using GitHub (or similar version control systems) is that Jenkins requires Shared Libraries to be in a source control repository. We'll create a simple method within the library to send email notifications. Follow these steps:

1. Create a new repository on GitHub named **jenkins-email-lib**. Setting the repository to **public** simplifies the process. However, if you opt for a private repository, you can seamlessly use your existing Jenkins credentials to clone it. Refer to the *Creating a username and password credential inside Jenkins* section from [Chapter 4](#).
2. Within your repo, set up the typical Jenkins Shared Library directory structure, as shown here:

```
jenkins-email-lib
├── vars
│   └── notifyByEmail.groovy
└── src
```

3. Inside the **vars/notifyByEmail.groovy** file, we will write the code for our Jenkins Shared Library. The following is a short snippet of the code. Refer to **notifyByEmail.groovy** from the GitHub repository to view the complete code:

notifyByEmail.groovy (chopped down version)

```
def call(String emailRecipient) {
    switch(currentBuild.result) {
        case 'FAILURE':
            emailxnt (
                subject: "${currentBuild.fullDisplayName} - Build Failed",
                body: "The Jenkins job has failed. Check Jenkins for details. Job: ${env.JOB_URL}",
                to: emailRecipient
            )
            Break
    }
    ...
}
```



```

    case 'SUCCESS':
        emailtext (
            subject: "${currentBuild.fullDisplayName} - Build Successful",
            body: "The Jenkins job completed successfully! You can view the job at:
${env.JOB_URL}",
            to: emailRecipient
        )
        break
    }
}

```

The preceding code defines a method named **call** in a Jenkins Shared Library, which is designed to send an email notification under certain conditions. When this method is invoked, it expects an email recipient's address as an argument. The function checks the result of the current build and sends a tailored email notification for the **FAILURE**, **ABORTED**, and **SUCCESS** states. The email's subject and body are customized based on the build's outcome. The email's subject contains the display name of the build along with its result. The body of the email provides a message, prompting the recipient to check Jenkins for more details, and includes a direct link to the job. The **emailtext** function is a built-in Jenkins function that facilitates sending such email notifications.

4. Commit and push your changes to the GitHub repository.

IMPORTANT NOTE

*It's crucial to note that for this **emailtext** function to work correctly, the email extension's configurations (such as an **Simple Mail Transfer Protocol (SMTP)** server and email content type) must be appropriately set up in the Jenkins global configuration. This setup ensures that Jenkins can send emails when invoked by scripts or pipelines. To learn more about configuring SMTP server details, visit the [Email Extension plugin documentation \[2\]](#)*

Configuring a Shared Library inside Jenkins

To make our library accessible to Jenkins jobs, we need to add it to Jenkins' global configuration.

This step ensures Jenkins knows where to fetch the library from when it's invoked in a pipeline.

Follow these simple steps:

1. Open Jenkins, and navigate to **Manage Jenkins | System**.
2. On the resultant page, scroll all the way down to the **Global Pipeline Libraries** section and click on the **Add** button, as shown in *Figure 12.5*.
3. In the **Name** field, enter **jenkins-email-lib**. This is the name we'll use to reference the library in Jenkinsfiles.
4. In the **Source Code Management** section, choose **Git**, and enter the repository URL for **jenkins-email-lib**.
5. Ensure the **Modern SCM** option is chosen, and select the appropriate credentials if the repository is private. As mentioned before, you can use your existing Jenkins credentials.

Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without “sandbox” restrictions and may use @Grab.

Library

Name ?

jenkins-email-lib

Retrieval method

Modern SCM

Loads a library from an SCM plugin using newer interfaces optimized for this purpose. The recommended option when available.

Source Code Management

Git

Project Repository ?

https://github.com/trekpik/jenkins-email-lib.git

Credentials ?

nikpy/***** (GitHub credentials to interact with repositories)

Add

Figure 12.5 – Configuring Shared Library settings in Jenkins

6. Finally, click on **Apply** and then the **Save** button to save the configuration.

Using the Shared Library inside your pipeline

Now that our Shared Library is set up and configured in Jenkins, we can call its functionality within any Jenkins pipeline. In this example, we’ll illustrate how to invoke the email notification method when a pipeline job fails or gets aborted. Follow these steps:

1. To keep things simple, we will create a Jenkins pipeline job. To do this, click on + **New Item** on the Jenkins dashboard.
2. Give your pipeline a nice name, using the **Enter an item name** field, and choose **Pipeline** from the available options.
3. For simplicity’s sake, we won’t import any Jenkinsfile from a source control repository. Instead, directly paste the following pipeline code into the **Script** section of your pipeline:

```
@Library('jenkins-email-lib') _
pipeline {
    agent any
    stages {
        stage('Example Stage') {
            steps {
                sh 'exit 1'
            }
        }
    }
}
```

```

    }
  }
  post {
    always {
      notifyByEmail 'recipient@example.com'
    }
  }
}

```

4. The preceding code integrates a Jenkins Shared Library named **jenkins-email-lib** into a Jenkins pipeline. The pipeline contains **Example Stage, which** intentionally fails. Regardless of the outcome of stages within the pipeline, an email notification is sent to **recipient@example.com** using the **notifyByEmail** function from the Shared Library. This ensures that whenever the pipeline runs and completes, whether successfully or due to a failure, the specified recipient receives an email notification about it. Refer to **Jenkinsfile** from the GitHub repository to download the pipeline code. Also, in the preceding Jenkins pipeline code, the underscore (**_**) following **@Library('jenkins-email-lib')** is not a typo. It is a necessary element that acts as a script delimiter, allowing the pipeline to correctly interpret and integrate the specified library.
5. Save and run the pipeline. If the pipeline succeeds, fails, or is aborted, the **notifyByEmail** step from the Shared Library will send an email to the specified recipient.

This is a basic example of how Jenkins Shared Libraries can be structured, configured, and invoked. You can read more about the different ways to call a Jenkins Shared Library using the Jenkins documentation [3]. Once you're familiar with this process, you can expand the library with more sophisticated functionalities tailored to your needs.

While Jenkins Shared Libraries are a powerful way to reuse code across multiple pipelines, it's also important to maintain a tidy system; next, we'll cover a brief but crucial topic on how to discard old builds in Jenkins to ensure your CI/CD environment remains efficient and manageable.

Discarding old builds in Jenkins

As more and more projects start to do CI/CD, Jenkins accumulates a large number of builds, each with its logs, artifacts, and other associated data. If not managed properly, this can lead to excessive disk space usage, making the Jenkins server sluggish or even causing disk outages. Hence, discarding old builds becomes an essential housekeeping task.

In the following section, we'll delve into methods for cleaning up older builds in Jenkins. But before we do, let's touch on the key reasons behind the necessity of discarding outdated builds. The following are a few important reasons:

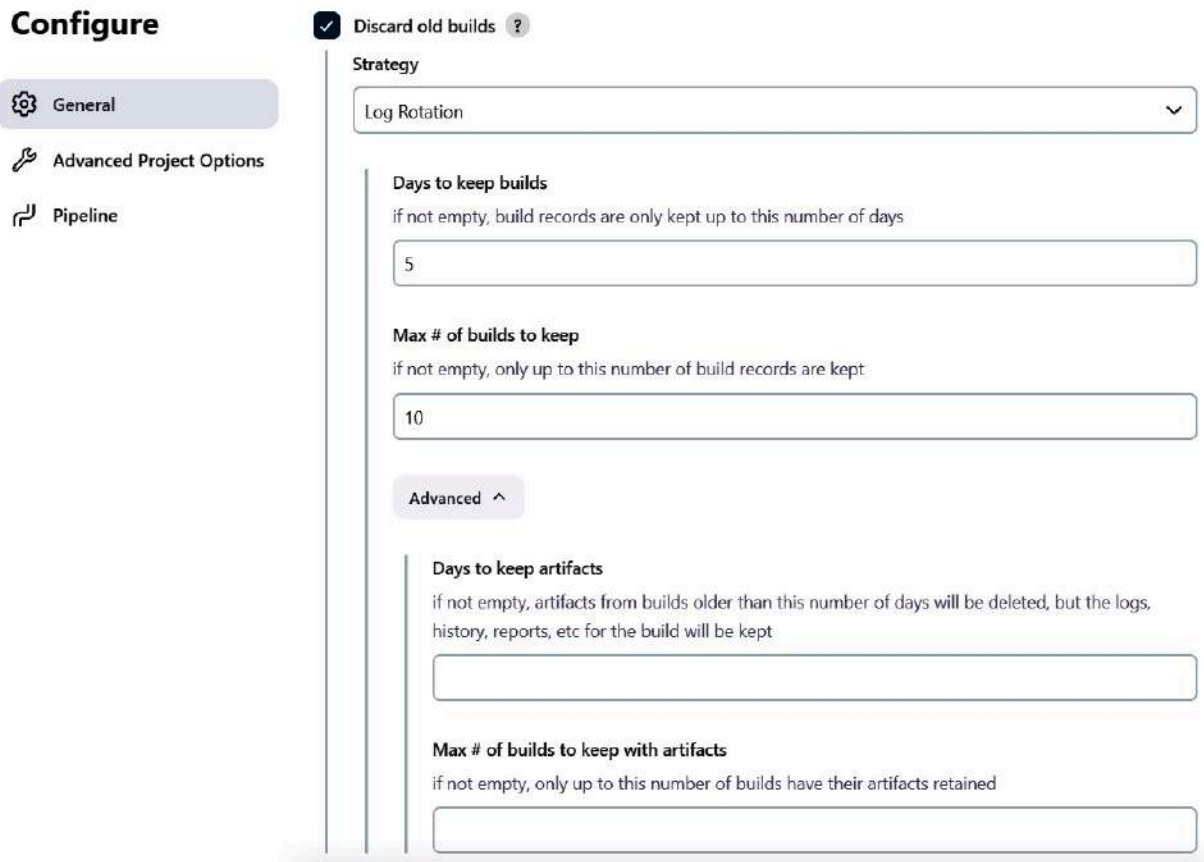
- **Disk space conservation:** Jenkins stores build logs, artifacts, and other metadata for every build. Over time, these accumulate and consume a significant amount of disk space.
- **Performance:** Large numbers of old builds can slow down the Jenkins UI, making it challenging to navigate and retrieve necessary information.
- **Relevance:** As code evolves, older builds often lose their relevance. Teams typically aren't interested in the results of a build from several months ago.

Now, let's look into the Jenkins tool by examining various methods available to discard older builds. These techniques will prove essential in maintaining a clean and efficient CI/CD environment, ensuring that only relevant and recent builds occupy valuable server space:

- **Log rotation in a declarative pipeline:** When using a declarative Jenkins pipeline, you can use the `options { }` directive with the `buildDiscarder` property. The `logRotator` method allows you to specify conditions for discarding old builds. In the following example, Jenkins will retain the last 10 builds and discard any older ones:

```
pipeline {
  agent any
  options {
    buildDiscarder(logRotator(numToKeepStr: '10'))
  }
  stages {
    // your stages here
  }
}
```

- **Via the Jenkins UI:** If you prefer not to set the discard policy within the pipeline code, you can do it via the Jenkins job configuration page. Under the **Build Environment** section, you'll find the **Discard old builds** option, which provides a UI to configure the log rotation strategy.



Configure

- General
- Advanced Project Options
- Pipeline

☒ **Discard old builds** ?

Strategy

Log Rotation

Days to keep builds
if not empty, build records are only kept up to this number of days

5

Max # of builds to keep
if not empty, only up to this number of build records are kept

10

Advanced ^

Days to keep artifacts
if not empty, artifacts from builds older than this number of days will be deleted, but the logs, history, reports, etc for the build will be kept

Max # of builds to keep with artifacts
if not empty, only up to this number of builds have their artifacts retained

Figure 12.6 – The Discard old builds options inside the Jenkins pipeline UI

- **Groovy script:** The methods we discussed earlier are preventative in nature, designed to stop the accumulation of outdated builds from the outset. However, if you already find yourself managing hundreds of Jenkins pipelines, each with numerous builds, the

following option serves as a remedial measure, tailored for such situations. The following code is a Groovy script, **deleteBuilds.groovy**, that, when executed in the Jenkins script console, will retain only the last 10 builds for all jobs (pipelines). Refer to the GitHub repository to download the Groovy script:

deleteBuilds.groovy

```
Jenkins.instance.getAllItems(Job.class).each { job ->
    def builds = job.getBuilds()
    if (builds.size() > 10) {
        builds.subList(10, builds.size()).each { it.delete() }
    }
}
```

To use the preceding script, navigate to your Jenkins dashboard, and click on **Manage Jenkins** on the left sidebar. On the resultant page, look for **Script Console** under the **Tools and Actions** section. The script console in Jenkins is a powerful feature that allows administrators and users with appropriate permissions to execute Groovy scripts directly within the Jenkins environment, enabling them to perform various administrative tasks, troubleshoot issues, and manipulate Jenkins configurations programmatically. *Figure 12.7* illustrates the **Script Console** page. Copy the preceding script into the script input box and click **Run**, as shown in the following figure.

Dashboard > Manage Jenkins > Script Console

Script Console

Type in an arbitrary [Groovy script](#) and execute it on the server. Useful for trouble-shooting and diagnostics. Use the 'println' command to see the output (if you use `System.out`, it will go to the server's stdout, which is harder to see.) Example:

```
println(Jenkins.instance.pluginManager.plugins)
```

All the classes from all the plugins are visible. `jenkins.*`, `jenkins.model.*`, `hudson.*`, and `hudson.model.*` are pre-imported.

```
1 Jenkins.instance.getAllItems(Job.class).each { job ->
2     def builds = job.getBuilds()
3     if (builds.size() > 10) {
4         builds.subList(10, builds.size()).each { it.delete() }
5     }
6 }
```

Run

Figure 12.7 – Using Script Console inside Jenkins

IMPORTANT NOTE

Before running the script, make sure to back up Jenkins, just in case there's an unintended consequence. The script operates on all jobs, not just pipelines, so it would affect freestyle projects as well. Use scripts such as these cautiously, especially in production environments.

Consistently discarding old builds ensures that Jenkins remains performant and that storage is used efficiently. Whether through pipeline code or the Jenkins UI, setting a build discard policy is a best practice that all Jenkins administrators and users should adopt. Additionally, the specific discard strategy should be tailored to the team's needs, considering factors such as storage availability, the frequency of builds, and the retention requirements of build artifacts and logs.

Having discussed how to keep your Jenkins environment clean by discarding old builds, let's now turn our attention to proactively safeguarding your CI process by catching security vulnerabilities early, with the integration of JFrog Xray.

Catching security vulnerabilities early in code

JFrog Xray is a cutting-edge software component analysis tool, designed to help organizations enhance their software security and ensure regulatory compliance. In the era of CI/CD, where code changes are frequently integrated and delivered, ensuring the security of these deliveries becomes paramount. JFrog Xray seamlessly fits into this paradigm by providing continuous security scanning for software artifacts, including Docker images.

In the context of Jenkins, the integration of JFrog Xray elevates the build and deployment process. With Jenkins being the automation engine and JFrog Artifactory acting as the universal artifact repository, incorporating Xray into the workflow allows every build pushed through Jenkins to be automatically scanned for vulnerabilities. This means that before a Docker image is stored in Artifactory, it can be rigorously scrutinized for any known security risks, ensuring that only secure, compliant images are stored and ready for deployment. The advantage of this integrated approach is twofold. First, it provides an automated security gate, ensuring that any vulnerabilities are caught early in the CI pipeline, thereby reducing the risk of deploying insecure software. Second, having vulnerability scanning as part of the CI process ensures that security is a continuous effort and not an afterthought.

In the following section, we will learn to enhance our CI pipeline to conduct vulnerability scans on every code commit using JFrog Xray.

Installing JFrog Xray

In the following section, we'll see how to set up JFrog Xray on our **Azure Kubernetes Service (AKS)** using its Helm chart [4] in its own namespace. It's a simple process! Before we start, make sure you have the following:

- A working Artifactory server for Xray to connect to
- A JFrog Xray trial license, which you should've received with the Artifactory license (check your email)

With those ready, let's dive into the installation steps!

1. First, execute the **az login** command in your terminal or command prompt to authenticate with your Azure account:

```
az login
```

2. A browser window will open, requesting you to sign in to your Azure account. Once authenticated in the browser, you can close the window. The terminal should then display a list of subscriptions associated with your Azure account.
3. Now, set a default resource group for your Azure CLI session by running the following command in your terminal. Remember to replace **<Your resource group>** with the name of the resource group where you have your AKS. Once executed, any subsequent Azure CLI commands that require a resource group will use the specified default one unless overridden:

```
az configure --defaults group=<Your resource group>
```

Ignore this step if you have only one resource group in your Azure subscription.

4. To isolate our JFrog Xray server, we'll deploy it in a dedicated namespace on the AKS cluster. Use the **kubectl create namespace** command, as illustrated here, which will establish a new namespace called **xray**:

```
kubectl create namespace xray
```

5. Before deploying JFrog Xray with its Helm chart, it's essential to have the **join key** on hand. This key is pivotal for secure communication between JFrog services – that is, in clustered configurations or when integrating with other JFrog products, such as Xray itself, the join key acts as a shared secret, ensuring the safe connection of various JFrog services and nodes.
6. To locate the join key, sign in to your Artifactory server. From the dashboard, click on **Administration | User Management | Settings**.
7. On the resultant **Security Configuration** page, scroll all the way down to the **Connection details** section. Input your Artifactory admin password into the **Current Password** field, and click on **Unlock** to reveal the join key.

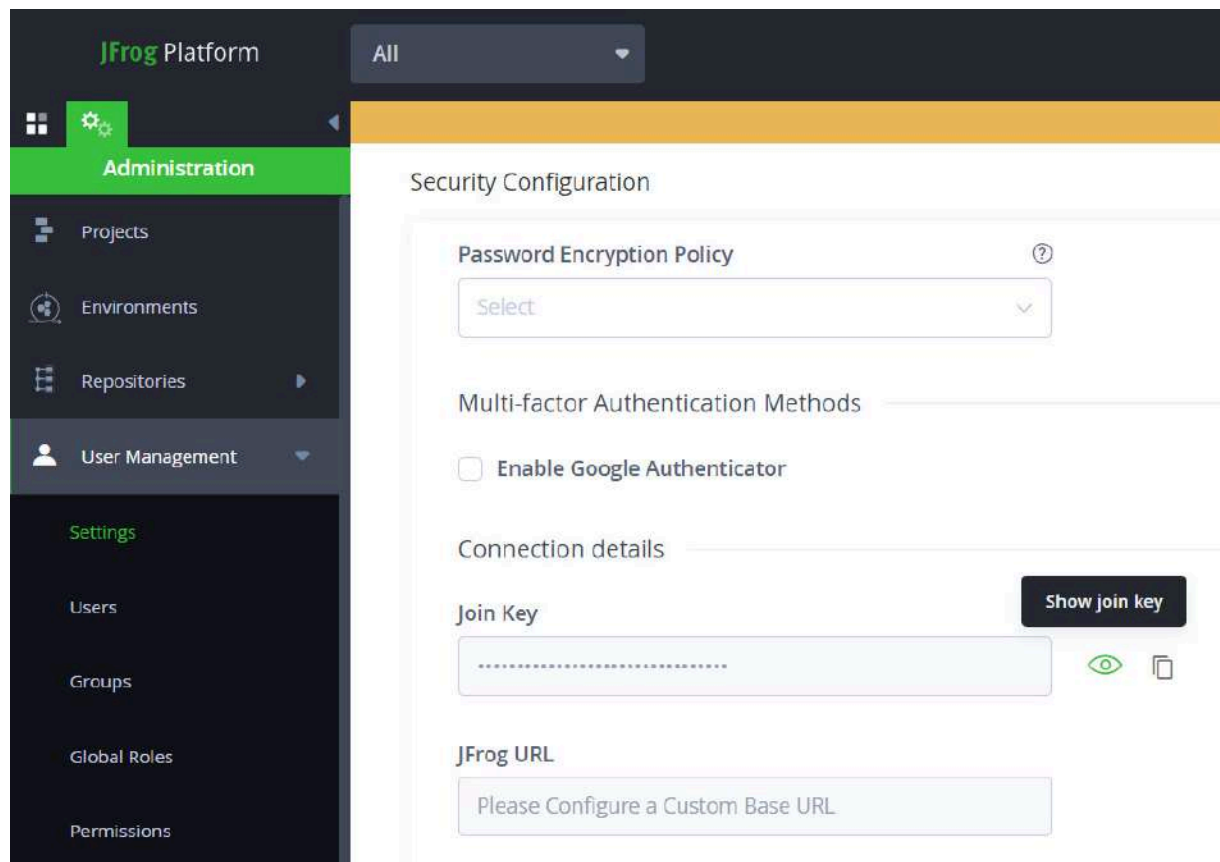


Figure 12.8 – Fetching the join key from the JFrog Platform

8. With the join key in place, run the following commands in sequence to register the JFrog Helm repository with your Helm client. If you have previously executed the `helm repo add jfrog https://charts.jfrog.io` command to install JFrog Artifactory, you can skip this step while setting up JFrog Xray. However, it's important to run `helm repo update` to ensure your Helm repository is up to date before proceeding with the installation of JFrog Xray:

```
helm repo add jfrog https://charts.jfrog.io
helm repo update
```

9. Now, run the `helm upgrade --install xray` command to install JFrog Xray, using the `jfrog/xray` Helm chart:

```
helm upgrade --install xray \
  --set xray.joinKey=<Your Join Key> \
  --set xray.jfrogUrl=<Artifactory Server IP Address> \
  --set router.serviceRegistry.insecure=true \
  --namespace xray jfrog/xray
```

Here's an explanation of the key elements in the preceding command:

- A. `--set xray.joinKey` allows you to specify the join key, which plays a pivotal role in securely connecting JFrog services. Make sure to replace `<Your Join Key>` with your join key.
- B. `--set xray.jfrogUrl` defines the URL or IP address of the Artifactory server that Xray will communicate with. Replace `<Artifactory Server IP Address>` accordingly.

C. `--set router.serviceRegistry.insecure=true` is a configuration flag, indicating that the service registry for the router will operate in an insecure mode. Since our Artifactory server operates without SSL certificates and uses an HTTP URL, we need this configuration. However, if your Artifactory server is protected with SSL certificates, you can omit this option.

D. Finally, `--namespace xray` determines that the installation will occur in the Xray namespace of the Kubernetes cluster. With this, we've successfully completed the installation of JFrog Xray using its Helm chart.

10. One last step is to add your JFrog Xray license. To do that, ensure you have a valid JFrog Xray license key on hand before performing these steps.
11. Next, log in to the JFrog Platform using an administrative account, and click on the **Administration** tab (represented by a gear icon in the top navigation bar).
12. Then, under **License Details**, click on **Activate Xray Trial License**. You will see a section titled **Install New License**. Here, paste your provided JFrog Xray license key into the textbox, and click on the **Save** button.
13. After clicking **Save**, log out and then sign back in for the changes to take effect. Your JFrog Xray instance should now be licensed and fully operational.

This marks a significant step in enhancing our existing CI/CD infrastructure, offering us advanced capabilities to scan and monitor binaries for vulnerabilities. As we utilize JFrog Xray, it's essential to note that there isn't a separate dashboard for Xray. To access its features, simply log in to your Artifactory server, which is the JFrog Platform dashboard. This unified approach ensures seamless integration and navigation between JFrog services.

Scanning for vulnerabilities through a CI pipeline

The JFrog Xray tool is vast and packed with an array of powerful features. Given the breadth and depth of its capabilities, it's challenging to delve into every aspect within the confines of this chapter. Having said that, our focus here will be specific to the CI pipeline. We aim to introduce a dedicated stage for scanning the Docker images we've built, leveraging JFrog Xray's capability. So, let's dive right in and start crafting this new CI pipeline stage:

1. Open your **Hello World** source code project using VS Code.
2. Once inside, create a new branch named **TestingJfrogXray**.
3. Now, with caution, open the Jenkinsfile and remove all stages following the `stage('Publish build info') {}` directive. We do not want to run the pipeline stages related to CD, as our sole focus is to test the JFrog Xray scanning stage.
4. Next, introduce a fresh stage named `stage('Scan Build with Xray') {}`. The content of this new stage should be as follows:

```
stage('Scan Build with Xray') {
  steps {
    xrayScan (
      serverId: 'Default Artifactory Server',
      failBuild: false
    )
  }
}
```

```
}
}
```

This stage will leverage JFrog Xray to scan your Docker images for potential vulnerabilities.

Within this stage, we have the following:

- A. **xrayScan()** is a Jenkins pipeline step provided by the JFrog Jenkins plugin. This step invokes JFrog Xray to scan artifacts or the build itself, depending on its configuration.
- B. **serverId: 'Default Artifactory Server'** specifies the Artifactory server that is connected to the Xray instance. This Artifactory server has been predefined in Jenkins, and by referring to it, the pipeline knows where to send artifacts or builds for scanning.
- C. **failBuild: false** is a configuration setting determining the pipeline's behavior if vulnerabilities are detected. If set to **true**, the Jenkins pipeline would fail upon detecting vulnerabilities; in this case, it's set to **false**, meaning the pipeline will continue executing even if vulnerabilities are detected.

5. Commit and push your modifications, which will automatically initiate the pipeline execution in Jenkins for the **TestingJfrogXray** branch. Monitor the process in Blue Ocean to see its progress.

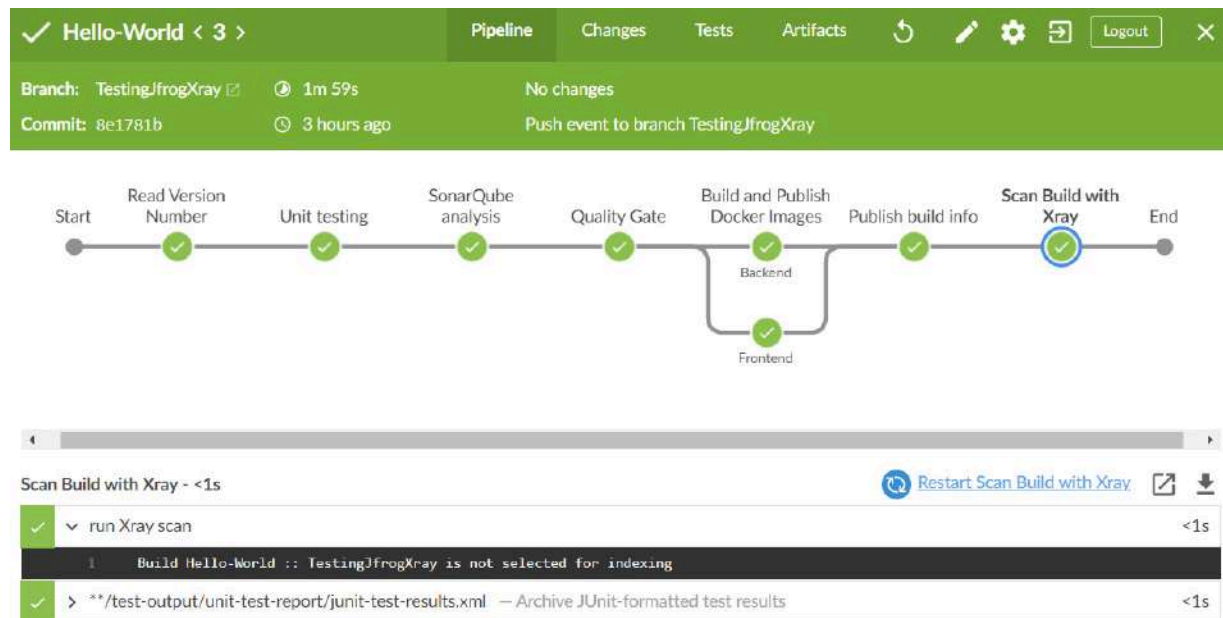


Figure 12.9 – Checking the logs for the stage Scan Build with Xray

6. When the pipeline is finished running, look at the build logs of the **Scan Build with Xray** stage. You should see the following message – **Build Hello-World :: TestingJfrogXray is not selected for indexing**. What does that mean? It typically indicates that while the build was scanned by JFrog Xray, the associated artifacts from that specific build haven't been indexed in Artifactory. So, let's do that.

Adding the Jenkins build to the scans list

For Xray to do its magic, we need to specify which builds it should keep an eye on. This is done by adding specific builds to Xray's scan list. In this section, we will do exactly that. We will walk

through the steps to add the **TestingJfrogXray** Jenkins build to Xray's scan list, ensuring that it is automatically scanned by JFrog Xray. Let's dive in:

1. Begin by logging in to your JFrog Platform instance (the Artifactory server). This is where you have Artifactory and Xray integrated.
2. Once logged in, click on the **Application** tab on the left panel and navigate to the **Xray | Scans List** section, as shown in the following figure.

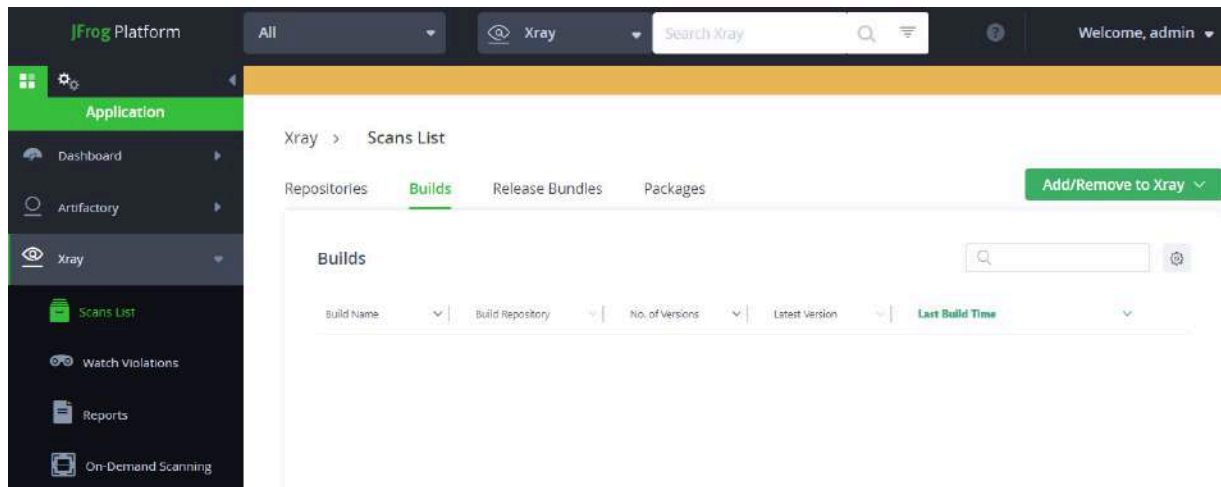


Figure 12.10 – Scans List inside JFrog Xray

3. To add a build for scanning, click on the **Add/Remove to Xray** button and select **Build**.
4. A dialog box or a new window will appear, prompting you to select a build to add to the scan list. Typically, you'll see a list of builds from Artifactory that aren't currently being scanned by Xray. From the displayed list, select the **Hello-World :: TestingJfrogXray** build. You need to actually drag it from left to right, as shown in the following figure.

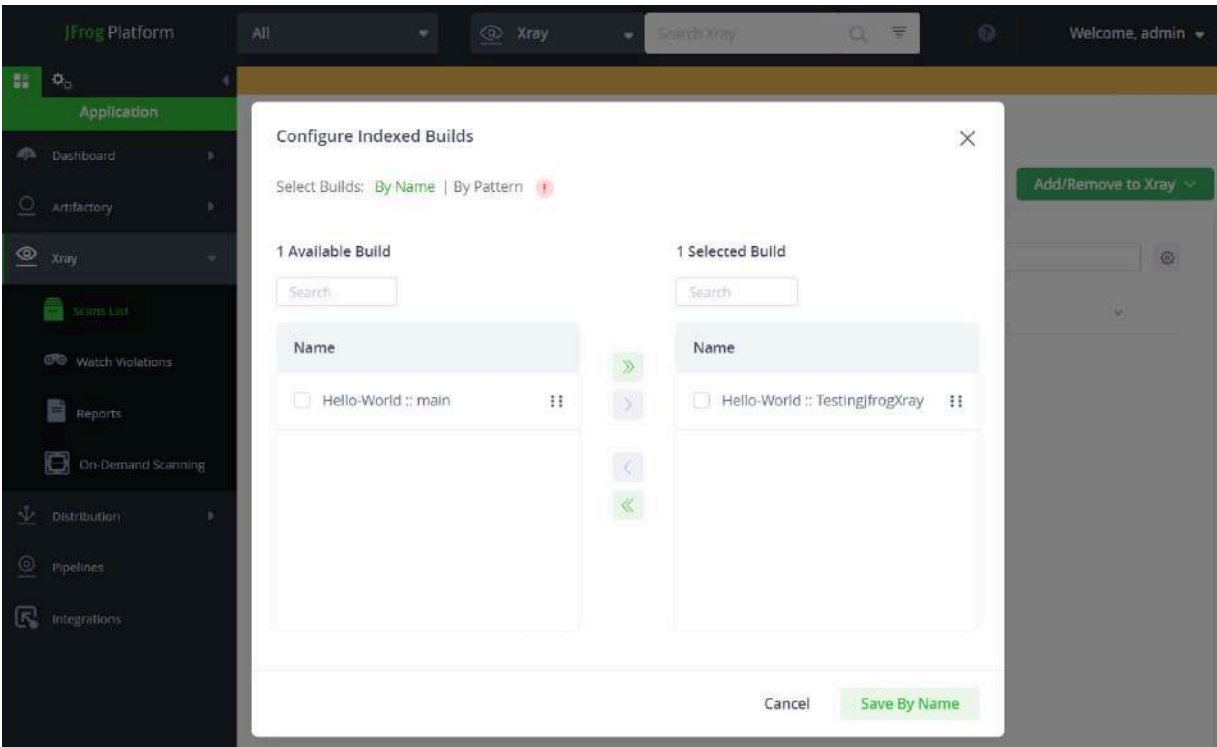


Figure 12.11 – Adding the feature branch build to the scans list

5. Once you've selected the desired build, click on the **Save by Name** button.

By following these steps, you'll have successfully added the **TestingJfrogXray** build to the scan list in JFrog Xray and set up the necessary configurations. Now, let's run our pipeline again to see the scan results.

Visualizing the vulnerability report using Xray

JFrog Xray provides a comprehensive vulnerability report for builds, enabling teams to gain deep insights into potential security risks associated with their artifacts. Once you've integrated Xray with your build system (Jenkins) and set it to scan your builds, accessing the vulnerability report is straightforward. Follow these steps:

1. Rerun the Jenkins pipeline for the **TestingJfrogXray** branch by clicking on the rerun button (marked in red in the following figure).

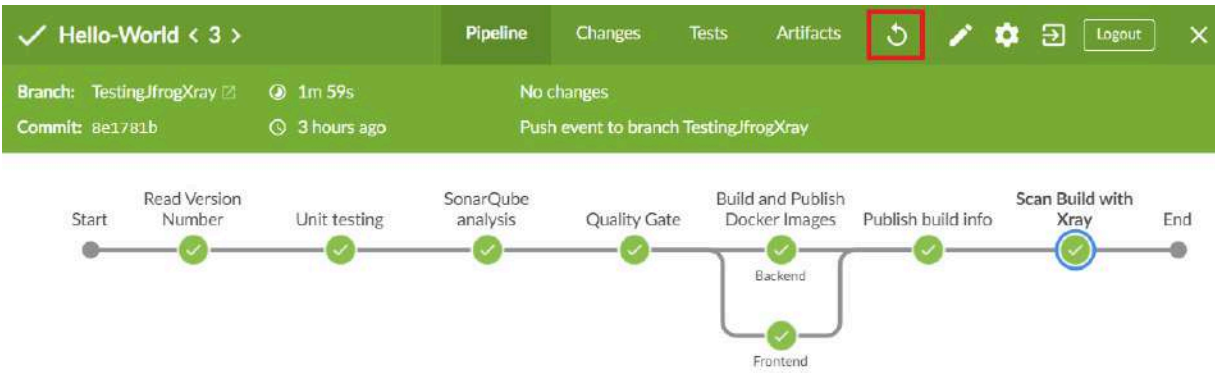


Figure 12.12 – Rerunning the Jenkins pipeline

- When the pipeline is done running, log in to the JFrog Platform (Artifactory server). From the dashboard, click on the **Application** tab and navigate to **Artifactory** | **Builds**. You would be presented with a list of build projects that have been published to Artifactory. This section is essentially a detailed view of all the builds that have been stored and managed by Artifactory. Here's a screenshot of what you'd typically see:

The screenshot shows the JFrog Platform interface. The left sidebar contains navigation options: Application, Dashboard, Artifactory, Release Bundles, Packages, Builds, and Artifacts. The main content area is titled 'Builds' and displays a table of build projects.

Build Name	Build Repository	Last Build ID
Hello-World :: TestingJfrogXray	artifactory-build-info	4
Hello-World :: main	artifactory-build-info	10

Figure 12.13 – Builds listed on Artifactory

- The preceding **Builds** section in Artifactory serves as a central place for teams to track, manage, and gain insights into their build artifacts, ensuring traceability, reproducibility, and security in their CI/CD processes.
- Now, click on the **Hello-World :: TestingJfrogXray** build to list all its pipeline runs. You will immediately see all the Jenkins pipeline runs for the **TestingJfrogXray** branch. The platform smartly highlights any issues encountered during the build process and contrasts the differences between this build and its predecessor. Note the Xray status for the latest build that says **Scanned - No Issues**, as shown in the following figure.

The screenshot shows the JFrog Platform interface. On the left is a sidebar with navigation options: Application, Dashboard, Artifactory, Release Bundles (v2), Packages, Builds, and Artifacts. The main content area is titled 'Builds > Hello-World :: TestingJfrogXray'. It contains a table with the following data:

Build ID	CI Server	Xray status
4	http://20.103.130.176/job/Hello-World/job/TestingJfrogXray/4/	Scanned - No Issues
3	http://20.103.130.176/job/Hello-World/job/TestingJfrogXray/3/	Not Scanned

Figure 12.14 – The Xray scan result per the Jenkins build

- Let's go a level deeper by clicking on the latest build ID. At a glance, you'll see the build's name, its version or number, the timestamp of its creation, the initiator, its duration, details about the CI tool used, and potentially, a link to the build result on the CI server. As you dive deeper, the platform unravels a list of published modules, each with its own set of associated artifacts and dependencies. The context of the build is further enriched by the captured environment variables, which offer insights into the build's conditions and context. If you've integrated JFrog Xray, which we did, you'll also get a concise summary of any security vulnerabilities or license compliance issues tied to this build. This detailed vulnerability breakdown includes severity ratings, affected artifacts, and even potential remediation steps, as shown in the following figure.

The screenshot shows the JFrog Platform interface with the 'Builds' tab selected. The breadcrumb path is 'Builds > Hello-World :: TestingJfrogXray > 4'. The main content area displays details for build ID 4:

- Build Name:** Hello-World :: TestingJfrogXray
- Started:** 19-10-23 16:23:48 +0200 | **Duration:** 2.0 minutes
- Pipeline:** Jenkins/2.414.1
- Build Agent:** admin
- Principal:** admin
- Artifactory Principal:** admin

Below the build details, there are tabs for 'Published Modules', 'Environment', 'Xray Data', 'Issues', 'Diff', 'Release History', and 'Effectiv'. The 'Xray Data' tab is active, showing a summary of '0 Policy Violations' and a message 'No Violations were found'.

Figure 12.15 – A detailed Xray scanning report for a build

In our example, luckily, we do not have any security violations.

The platform doesn't stop at just providing information; it empowers users with actions. You might find options to promote the build, distribute its artifacts, or even discard it. Additionally, for those

keen on the technical intricacies, there's an insight into how dependencies are declared – be it through a Maven **Project Object Model (POM)**, a Gradle script, or other formats. Plus, there's a transparent view of how these dependencies were resolved and fetched during the build process. You can learn more by visiting the JFrog Xray documentation [5]. In essence, this detailed overview ensures that teams have a comprehensive, transparent, and traceable understanding of each build, a boon for both troubleshooting and auditing processes.

Summary

In this chapter, we embarked on a deep dive into refining and fortifying our CI/CD pipelines, ensuring they're not only functional but also secure. We began our exploration with GitHub Copilot, illustrating how artificial intelligence can significantly boost the development process. With Copilot at our side, we learned how to generate Jenkins pipeline code more efficiently, reducing errors and accelerating our coding endeavors. Compared to ChatGPT, GitHub Copilot is decidedly more tailored for developers when writing Jenkins pipeline code.

Next, we learned about Jenkins Shared Libraries. Through a hands-on example, we discerned the value of centralizing common code patterns. This crucial tool enabled us to maintain consistent pipelines across various projects, ensuring the scalability and manageability of our CI/CD processes. Then, we learned about a small but significant topic – optimizing our Jenkins setup through strategies to periodically discard old builds, conserve resources, and ensure our Jenkins environment remains stable.

Last, but certainly not least, we tackled the topic of security one more time with JFrog Xray. We learned how to set it up and weave it into our Jenkins pipeline, automating vulnerability scans for our builds. This ensured that each build, besides being functional, adhered to the highest security standards, protecting our deployments from potential threats.

In summary, this chapter equipped us with a few advanced tools and methodologies to boost our CI/CD pipelines, ensuring that our pipelines are robust, secure, and ready for the challenges of modern software development.

As we close this chapter – and indeed, the book – remember, this is not the end; it is the beginning. The world of technology is ever-evolving, and as you turn the last page of this book, a new chapter in your professional life awaits. Ahead lies the uncharted territory of continuous learning and innovation. I encourage you to keep experimenting, to keep learning, and to share your knowledge and experience with the community.

May this book serve not only as a foundation but also as a springboard into the vast ocean of opportunities that DevOps and automation offer. Good luck on your journey ahead, and may you build, deploy, and release with confidence and expertise. Here's to the countless pipelines you'll perfect and the innovations you'll bring to life.

Questions

1. Choose whether this statement is true or false: *GitHub Copilot and LLMs such as GPT-4 serve the same primary purpose and function in the same way.*
2. Which of the following statements is wrong about a Jenkins Shared Library?
 - A. Jenkins Shared Libraries are typically stored and version-controlled in a source code repository, such as Git.
 - B. Shared Libraries in Jenkins can be defined at a global level, making them accessible across all Jenkins pipelines, or they can be restricted to specific projects for local scope.
 - C. The core purpose of Shared Libraries is to encapsulate common code, steps, and configurations, such as standardized build steps, deployment procedures, or utility functions.
 - D. The Shared Libraries in Jenkins can be written in various programming languages.
3. Which of the following statements is right about discarding old builds in Jenkins?
 - A. It allows disk space conservation.
 - B. It improves Jenkins performance.
 - C. The `logRotator` method allows you to specify conditions for discarding old builds.
 - D. All of the preceding.
4. What does vulnerability scanning in the context of software development typically involve?
 - A. Monitoring network traffic to prevent unauthorized access to the system
 - B. Scanning the code base and dependencies for known security vulnerabilities and flaws
 - C. Checking the software's compliance with performance benchmarks
 - D. Verifying the user authentication and authorization mechanisms in the application
5. Choose whether this statement is true or false: *In a Jenkins CI pipeline, using the Artifactory plugin, it is possible to configure the pipeline to fail if the vulnerability scan reports a failure.*

Answers

1. Correct answer: **False**

Explanation: GitHub Copilot and LLMs such as GPT-4 have distinct primary purposes and functions. GitHub Copilot is specifically designed to assist in software development by providing code suggestions and completions, focusing primarily on programming languages. In contrast,

LLMs such as GPT-4 have a broader scope, capable of generating and understanding natural language text across various contexts and applications, not limited to coding.

2. Correct answer: **D**

Explanation of incorrect answer choices: The shared libraries in Jenkins are primarily written in Groovy, a dynamic, object-oriented programming language that is closely integrated with the Java platform. Groovy provides a flexible and powerful way to write scripts that enhance and extend the capabilities of Jenkins pipelines.

3. Correct answer: **D**

Explanation of correct answer choice: Discarding old builds in Jenkins not only allows for disk space conservation but also improves Jenkins performance. The `logRotator` method is a configurable option that lets you specify conditions for discarding old builds, further enhancing control over the build history and resource utilization.

4. Correct answer: **B**

Explanation of incorrect answer choices:

A. Monitoring network traffic is more related to network security and intrusion detection, not directly to vulnerability scanning within a software code base.

C. Checking compliance with performance benchmarks is a part of performance testing, not vulnerability scanning. Vulnerability scans focus on security aspects rather than performance metrics.

D. While verifying user authentication and authorization mechanisms is important for security, it's more related to application security testing. Vulnerability scanning typically focuses on identifying known vulnerabilities in the code and dependencies, not on testing specific security features of the application.

5. Correct answer: **True**

Explanation: When using the Artifactory plugin in a Jenkins CI pipeline, you can configure the pipeline to fail if a vulnerability scan does not pass. This is a common practice in CI/CD environments to ensure that newly introduced code does not contain known vulnerabilities before it is merged or deployed. By setting up the pipeline to fail upon a failed vulnerability scan, teams can enforce stringent security standards and prevent the progression of vulnerable code through the development life cycle.

Further reading

If you wish to learn more about some of the concepts that are not covered in depth in this chapter, or are out of the scope of this book but relevant, we are happy to provide you with some references:

- [1] GitHub Copilot Chat: <https://docs.github.com/en/copilot/github-copilot-chat/about-github-copilot-chat>
- [2] Email Extension plugin documentation: <https://plugins.jenkins.io/email-ext/>
- [3] Extending with Shared Libraries: <https://www.jenkins.io/doc/book/pipeline/shared-libraries/>
- [4] JFrog Xray Helm charts: <https://github.com/jfrog/charts/tree/master/stable/xray>
- [5] JFrog Xray documentation: <https://jfrog.com/help/r/jfrog-security-documentation/jfrog-xray>

Index

As this ebook edition doesn't have fixed pagination, the page numbers below are hyperlinked for reference only, based on the printed edition of this book.

A

Active Directory (AD) [85](#)

agent node entity, master-agent model [142](#)

Antifactory Query Language (AQL) [16](#)

app.js file [226](#)

application programming interface (APIs) [16](#)

applications, Argo CD

 creating [301](#)

 GitHub repository, connecting [303](#)

 personal access token, creating on GitHub [302](#), [303](#)

 setting up [303-306](#)

Argo CD

 applications, creating [301](#)

 Command-Line Interface (CLI) [299](#)

 Ingress file, deploying [298](#), [299](#)

 installing [296](#), [297](#)

 Jenkins, integrating with [306](#)

 NGINX Ingress Controller, installing for [296](#), [297](#)

 user, creating [299-301](#)

Artifactory [109](#), [236](#)

 Ingress file, deploying for [129-131](#)

 Jenkins, connecting to [127](#)

Jenkins plugin, installing for [131](#), [132](#)

Nginx Ingress controller, deploying for [127](#), [128](#)

user, creating [132-134](#)

using, as Docker registry [237](#), [238](#)

Artifactory plugin

configuring [135-137](#)

Artifactory server

deploying, with Helm chart [128](#), [129](#)

artificial intelligence (AI) [200](#)

Azure

on-demand build agents, creating [147](#)

Azure AD

Jenkins plugin, installing [93](#), [94](#)

Jenkins, registering on [86-93](#)

settings, configuring in Jenkins [95](#), [96](#)

Azure Backup [79](#)

Azure Kubernetes Service (AKS) [47](#), [119](#), [289](#), [345](#)

Kubernetes Secret, creating on [238](#), [239](#)

Azure Managed Disks [79](#)

Azure service principal [148](#)

creating [148-150](#)

credential, creating inside Jenkins [150-152](#)

Azure VM Agents plugin. See Jenkins Azure VM Agents plugin

Azure VM Scale Set (VMSS) [59](#)

B

backend code, Hello World web application [223](#)

Dockerfile [225](#), [226](#)

package.json file [224](#), [225](#)

server.js file [223](#), [224](#)

backend layer, web application architecture

API endpoint [222](#)

Docker containerization [222](#)

MongoDB integration [222](#)

Node.js server [222](#)

bill of materials (BOM) [17](#)

binary repository manager

accounting builds [15](#)

build promotion [16](#)

employing [15](#)

software composition analysis (SCA) [17](#)

software dependency management [17](#)

branching [10](#)

build farm [20](#)

C

Cascading Style Sheets (CSS) file [221](#)

CD in action [321](#)

changes, verifying on Helm chart [322](#)

deployment, verifying in staging and production [323](#), [324](#)

Jenkinsfile, replacing [321](#)

performance test file, adding [321](#)

performance testing results, visualizing [324-326](#)

CD pipeline code

Jenkins agent settings, updating [314](#), [315](#)

performance testing, running [318-320](#)

pipeline options and environment, configuring [315](#), [316](#)

Production Deployment Health, verifying [320](#), [321](#)

production Helm chart's configuration, updating [320](#)

Staging Deployment Health, verifying [317](#), [318](#)

Staging Helm chart's configuration, updating [316](#), [317](#)

writing [314](#)

change failure rate (CFR) [23](#)

built-in quality [23](#)

early defect detection [23](#)

ChatGPT [201](#)

guidelines, for effective interaction [202](#)

limitations [214](#), [215](#)

used, for constructing pipeline code [210-214](#)

ChatGPT, guidelines for effective interaction [202](#), [203](#)

code, refining [209](#), [210](#)

pipeline code, running [206](#), [207](#)

pipeline code, troubleshooting [207-209](#)

pipeline requirements description [203](#), [204](#)

CI in action [254](#)

build artifacts, visualizing [268-271](#)

CI pipeline run and test results, visualizing [257-265](#)

Multibranch Pipeline Job, creating in Jenkins [254-257](#)

SCA report and code coverage, visualizing [265-267](#)

CI pipeline code

Build and Publish Docker images stage [250-252](#)

constructing, ChatGPT used [210-214](#)

Jenkins agent settings, configuring [244-246](#)

Publish build info stage [252](#), [253](#)

SonarQube quality gate check, performing [249](#), [250](#)

static code analysis, performing [247-249](#)

testing results, publishing to Jenkins [253](#), [254](#)

unit testing, performing [246](#), [247](#)

writing [244](#)

communication, web application architecture [222](#)

complexity

managing, with nested stages [184](#), [185](#)

Continuous Deployment (CD) [277](#), [278](#)

Universal CI and CD [280](#), [281](#)

Universal CI with master-only CD [278](#), [279](#)

Continuous Deployment planning [283](#)

Helm chart of web application, exploring [285-289](#)

high-level Continuous Deployment design [283](#)

staging and production environments, setting up [289](#)

workflow [285](#)

continuous integration (CI) [3](#), [21](#), [109](#)

binary repository manager, employing [15](#)

code commit, verifying for build errors [12](#)

code coverage [13](#)

cost-effective system [25](#)

defining [3](#), [4](#)

highly secure application [23](#)

improved software quality [23](#)

key principles [5](#)

parallelizing, through distributed builds [19-21](#)

practicing [9](#)

process [5](#)

software project [220](#)

static code analysis [13](#)

time to market (TTM) [22](#)

unit testing [12](#)

version control system [9](#)

continuous security

DevSecOps [24](#)

software composition analysis [25](#)

static code analysis [25](#)

third-party dependencies, scanning [24](#)

cost-effective system [25](#)

automation, using pipelines [25](#)

configuration as code [26](#), [27](#)

failures [26](#)

Cross-Origin Resource Sharing (CORS) [224](#)

cross-site request forgery (CSRF) [85](#)

crumb [99](#)

CSRF protection

in Jenkins [98](#)

D

database code, Hello World web application [229](#)

database layer, web application architecture

Docker containerization [222](#)

initialization script [222](#)

MongoDB [222](#)

Declarative Directive Generator [188-191](#)

Declarative Pipeline syntax [166-169](#)

- built-in steps [169](#)

- components [170](#)

- declarative structure [169](#)

- directives [176](#)

- post-build actions [169](#)

- sections [170](#)

- simplified stages [169](#)

directives, Declarative Pipeline syntax [176](#)

- environment [176](#)

- input [182](#), [183](#)

- options [177](#), [178](#)

- parameters [178-180](#)

- tools [181](#), [182](#)

- triggers [180](#)

- when [183](#), [184](#)

distributed build farm [142](#)

- methods, for building [143](#), [144](#)

Dockerfile [225-229](#)

Docker registry

- Artifactory, using as [237](#), [238](#)

domain-specific language (DSL) [19](#), [166](#)

F

feature flags [12](#)

folder/pipeline-level credentials [102](#)

Freestyle Job [19](#)

frontend code, Hello World web application [226](#)

app.js file [226](#)

Dockerfile [228](#), [229](#)

index.html file [227](#)

styles.css file [227](#)

frontend layer, web application architecture

Docker containerization [221](#)

dynamic data retrieval [221](#)

simple UI [221](#)

styling [221](#)

G

Generative Pre-trained Transformer (GPT) [201](#)

GitFlow Workflow [10](#)

Gitflow Workflow-based development [4](#)

GitHub [109](#)

Jenkins plugin, installing for [111](#)

personal access token, creating on [112](#), [113](#)

repository, creating on [230](#), [231](#)

GitHub Copilot [330](#)

enabling [330](#), [331](#)

using, to write pipeline code [330-335](#)

versus ChatGPT [335](#), [336](#)

GitHub Copilot Chat [335](#)

GitHub plugin

build status and reporting [111](#)

configuring [116](#), [117](#)

integration with GitHub API [111](#)

pull request integration [110](#)

webhook support [110](#)

GitHub project repository

webhook, creating on [233](#), [234](#)

GitOps [9](#), [281](#)

advantages [282](#)

workflow [282](#)

global analysis token

creating. on SonarQube [123](#), [124](#)

global credentials [102](#)

Google Kubernetes Engine (GKE) [222](#)

graphical user interface (GUI) [16](#)

Groovy script [343](#)

H

Hello World web application

backend code [223](#)

database code [229](#)

frontend code [226](#)

source code [223](#)

Helm chart

Artifactory server, deploying with [128](#), [129](#)

SonarQube Server, deploying with [119](#), [120](#)

high-level CI pipeline design [231-233](#)

source code considerations [231](#)

I

index.html file [227](#)

Ingress file

deploying, for Artifactory [129-131](#)

deploying, for SonarQube [120-122](#)

integration [4](#)

J

Java Development Kit (JDK) [39](#)

Java Runtime Environment (JRE) [39](#)

Java Virtual Machine (JVM) [37](#)

Jenkins

Azure AD settings, configuring [95](#), [96](#)

connecting, to Artifactory [127](#)

connecting, to SonarQube [117](#)

connecting, to version control system (VCS) [110](#)

CSRF protection [98](#)

old builds, discarding [341-344](#)

registering, on Azure AD [86-93](#)

secret text credential, creating inside [113-125](#)

username and password credential, creating [114-135](#)

user permissions, configuring [96-98](#)

Jenkins and Argo CD integration [306](#)

auth token, creating for Argo CD user [306](#), [307](#)

secret text credential, creating in Jenkins [307](#), [308](#)

Jenkins Azure VM Agents plugin

configuring [152-159](#)

installing [148](#)

managed disks [156](#)

unmanaged disks [156](#)

Jenkins Configuration as Code (JCasC) [45](#), [46](#), [94](#), [98](#)

inner workings [46](#), [47](#)

secrets, securing [104](#), [105](#)

working [46](#)

Jenkins Credentials

creation, avoiding at user level [103](#)

folder/pipeline-level credentials [102](#)

global credentials [102](#)

scope [102](#)

types [100](#), [101](#)

user-level credentials [103](#)

using [100](#)

Jenkins deployment, on managed Kubernetes cluster [47](#)

architectural design [48](#), [49](#)

Ingress file, deploying [56](#), [57](#)

Jenkins Helm chart, using [54](#), [55](#)

Jenkins instance, accessing [58](#)

local development environment prerequisites [49](#)

managed Kubernetes cluster, creating [49-52](#)

Nginx Ingress controller, deploying [52-54](#)

preparatory steps [49](#)

Jenkins deployment scenarios evaluation [40](#)

Jenkins, running as container [41](#)

Jenkins, running on bare metal [43](#), [44](#)

Jenkins, running on VM [42](#)

Jenkins serverless, running [41](#)

Jenkins deployment, to cloud instance [59](#)

architectural design [59](#), [60](#)

configuration code preparation [61](#), [62](#)

helper scripts preparation [63](#), [64](#)

Jenkins instance, accessing [71](#), [72](#)

local development environment prerequisites [61](#)

preparatory steps [60](#)

Terraform code preparation [65-70](#)

Terraform code, running [70](#)

Jenkins distributed build architecture [142](#)

distributed build farm [143](#), [144](#)

master-agent model [142](#), [143](#)

Jenkins pipeline code [166](#)

Declarative Pipeline syntax [168](#), [169](#)

Scripted Pipeline syntax [167](#), [168](#)

Jenkins plugin

installing, for Artifactory [131](#), [132](#)

installing, for Azure AD [93](#), [94](#)

installing, for GitHub [111](#)

installing, for SonarQube [122](#), [123](#)

Jenkins server architecture

agent nodes [36](#)

credentials [36](#)

Global Configuration [36](#)

Global Tools [36](#)

master node [36](#)

pipelines [36](#)

plugins [36](#)

Jenkins setup deployment [47](#)

- on managed Kubernetes cluster [47](#)

- to cloud instance [59](#)

Jenkins setup maintenance

- backup strategies, implementing for build data [78](#), [79](#)

- issues, monitoring [78](#)

- Jenkins plugins, upgrading [76](#)

- Jenkins plugins, upgrading with Jenkins Helm chart [76](#)

- Jenkins plugins, upgrading with shell script [77](#)

- Jenkins updates, monitoring [73](#)

- Jenkins upgrade, executing [75](#)

- Jenkins upgrade, with Jenkins Helm chart [75](#)

- Jenkins upgrade, with shell script [76](#)

- performing [72](#)

- structured approach, following [72](#)

- users, informing [73](#), [74](#)

Jenkins setup plan [34](#)

- deployment scenarios, evaluating [40](#)

- Jenkins Configuration as Code (JCasC) [45](#)

- Jenkins server architecture [34-36](#)

- system requirements [37](#)

Jenkins Shared Library [336](#)

- configuring [339](#)

- using [336](#)

- using, in pipeline [340](#), [341](#)

- writing [337](#), [338](#)

Jenkins system requirements

JVM configurations [39](#), [40](#)

network connectivity [39](#)

optimal hardware requirements [37](#), [38](#)

JFrog Artifactory [15](#)

JFrog Xray [17](#), [345](#)

installing [345-348](#)

Jenkins build, adding to scans list [350](#), [351](#)

vulnerabilities, scanning through CI pipeline [348](#), [349](#)

vulnerability report, visualizing [351-354](#)

K

k6 [324](#)

key principles, CI

automation [6](#)

code change, validating [6](#)

continuous feedback [7](#), [8](#)

fail fast [6](#)

pulling frequently [5](#)

Kubernetes

on-demand build agents, creating [144](#)

plugin configuration [145-147](#)

Kubernetes Secret

creating, on AKS [238](#), [239](#)

L

language models [200](#), [201](#)

M

master-agent model [142](#)

agents [142](#)

benefits [143](#)

master [142](#)

Maven Project Object Model (POM) [354](#)

metrics [13](#)

N

nested stages [185](#)

parallel nested stages, working with [186](#), [187](#)

sequential nested stages, working with [185](#), [186](#)

used, for managing complexity [184](#), [185](#)

Nginx Ingress controller

deploying, for Artifactory [127](#), [128](#)

deploying, for SonarQube [117-119](#)

O

Object Relational Mapping (ORM) [222](#)

on-demand build agents

creating, on Azure [147](#)

creating, on Kubernetes [144](#)

OpenAI [201](#)

orchestration, with CI/CD tool [17](#)

automation pipelines [19](#)

pipeline visualization and notification [19](#)

seamless and broader integration [17](#)

outdated builds

discarding, in Jenkins [341-344](#)

P

package.json file [224](#), [225](#)

parallel development [10](#)

parallel nested stages

 working with [186](#), [187](#)

personal access token

 creating, on GitHub [112](#), [113](#)

Pipeline as Code [166](#)

pipeline code

 vocabulary, building [166](#)

 writing, with GitHub Copilot [330-335](#)

project

 setting up, in SonarQube [235](#), [236](#)

property set [16](#)

pull requests (PRs) [262](#)

Q

quality gates [14](#)

quality metrics [13](#)

quality profiles and rules [14](#)

R

repository

 creating, on GitHub [230](#), [231](#)

S

Scripted Pipeline syntax [166](#), [167](#)

 domain-specific language (DSL) [167](#), [168](#)

 imperative programming [167](#)

 versus, Declarative Pipeline syntax [168](#)

secrets

 securing, in JCasC [104](#), [105](#)

secret text credential

 creating, inside Jenkins [113-125](#)

sections, Declarative Pipeline syntax

 agent [170](#), [171](#)

 pipeline [170](#)

 post [174](#), [175](#)

 stages [171](#), [172](#)

 steps [172-174](#)

security vulnerabilities

 catching early, in code [345](#)

sequential nested stages

 working with [185](#), [186](#)

server.js file [223](#), [224](#)

Shared Libraries [19](#)

Simple Mail Transfer Protocol (SMTP) [338](#)

Snippet Generator [188](#), [191-193](#)

SonarQube [13](#), [109](#)

 global analysis token, creating on [123](#), [124](#)

 Ingress file, deploying for [120-122](#)

 Jenkins, connecting to [117](#)

 Jenkins plugin, installing for [122](#), [123](#)

 Nginx Ingress controller, deploying for [117-119](#)

 project, setting up [235](#), [236](#)

SonarQube plugin

 configuring [125](#), [126](#)

SonarQube Server

- deploying, with Helm chart [119](#), [120](#)

staging and production environments

- Ingress files, deploying for web application [292-294](#)

- insecure registries, configuring [294-296](#)

- namespaces, creating [290](#), [291](#)

- NGINX Ingress Controllers, installing for web application [291](#), [292](#)

- secrets, creating [290](#), [291](#)

- setting up [289](#)

static code analysis [13](#)

- quality gates [14](#)

- quality metrics [13](#)

- quality profiles and rules [14](#)

- styles.css file [227](#)

T

- time to market (TTM) [22](#)

- tolls, avoiding [22](#)

- velocity [22](#)

- volume and variety [22](#)

- transformer architecture [201](#)

- trunk-based branching [11](#)

- trunk-based development [4](#)

U

- unit testing, CI

- performing [12](#)

- Universal CI and CD [280](#)

- advantages [280](#)

- design considerations [280](#)

- example workflow [281](#)

Universal CI with master-only CD

- advantages [279](#)

- design considerations [279](#)

- example workflow [279](#), [280](#)

user

- creating, in Artifactory [132-134](#)

User Acceptance Testing (UAT) [283](#)

user authentication

- configuring [86](#)

user-level credentials [103](#)

username and password credential

- creating, inside Jenkins [114-135](#)

user permissions

- configuring [86](#)

- configuring, in Jenkins [96-98](#)

V

version control system, CI [9](#)

- GitFlow Workflow [10](#)

- parallel development [10](#)

- trunk-based workflow [11](#)

version control system (VCS) [109](#)

- Jenkins, connecting to [110](#)

virtual hard disks (VHDs) [156](#)

virtual machines (VMs) [36](#), [143](#)

virtual user (VU) [325](#)

visual pipeline editor [167](#)

Visual Studio Code (VS Code) [332](#)

vocabulary

- building, for pipeline code [166](#)

W

web application architecture [220](#), [221](#)

- backend layer [222](#)

- communication [222](#)

- database layer [222](#)

- frontend layer [221](#)

- setup and deployment [222](#)

webhook

- creating, on GitHub project repository [233](#), [234](#)

white-box testing [13](#)



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

<packt>



1ST EDITION

The Linux DevOps Handbook

Customize and scale your Linux distributions
to accelerate your DevOps workflow



DAMIAN WOJSŁAW | GRZEGORZ ADAMOWICZ

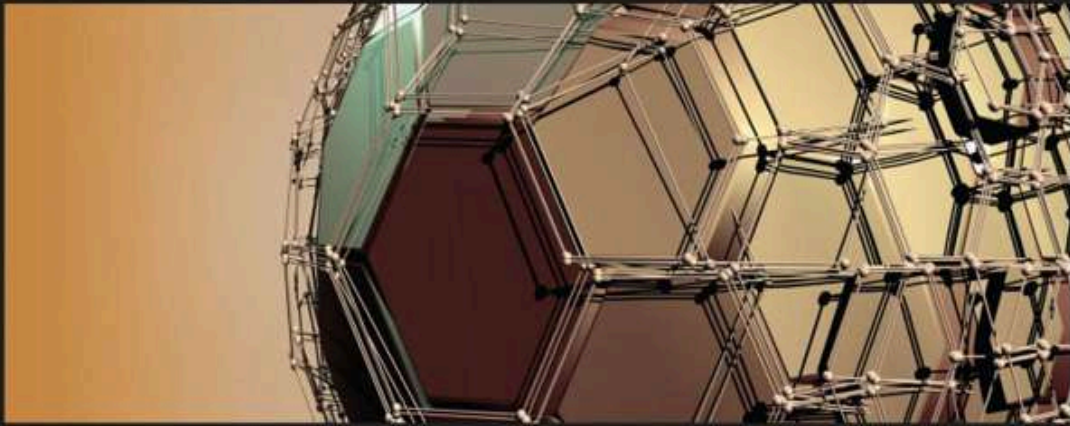
The Linux DevOps Handbook

Damian Wojsław, Grzegorz Adamowicz

ISBN: 978-1-80324-566-9

- Understand how to manage infrastructure using Infrastructure as Code (IaC) tools such as Terraform and Atlantis
- Automate repetitive tasks using Ansible and Bash scripting
- Set up logging and monitoring solutions to maintain and troubleshoot your infrastructure
- Identify and understand how to avoid common DevOps pitfalls
- Automate tasks and streamline workflows using Linux and shell scripting
- Optimize DevOps workflows using Docker

<packt>



1ST EDITION

Containers for Developers

Handbook

A practical guide to developing and delivering
applications using software containers



FRANCISCO JAVIER RAMÍREZ UREA

Containers for Developers Handbook

Francisco Javier Ramrez Urea

ISBN: 978-1-80512-798-7

- Find out how to build microservices-based applications using containers
- Deploy your processes within containers using Docker features
- Orchestrate multi-component applications on standalone servers
- Deploy applications cluster-wide in container orchestrators
- Solve common deployment problems such as persistency or app exposure using best practices
- Review your application's health and debug it using open-source tools
- Discover how to orchestrate CI/CD workflows using containers

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Learning Continuous Integration with Jenkins*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835087732>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

<packt>



3RD EDITION

Learning **Continuous Integration with Jenkins**

An end-to-end guide to creating operational, secure,
resilient, and cost-effective CI/CD processes

NIKHIL PATHANIA