

Machine Learning in Production

Master the art of delivering robust Machine Learning solutions with MLOps



Suhas Pote



Machine Learning in Production

*Master the art of delivering robust
Machine Learning solutions with MLOps*

Suhas Pote



www.bpbonline.com

Copyright © 2023 BPB Online

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 BPB Online 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.

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

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-10-1

Dedicated to

My parents:

*I am so grateful for your belief in me, and
for the many sacrifices you have made throughout
my life. Your love and support mean the world to me.*

About the Author

Suhas Pote has over eight years of multidisciplinary experience in data science, playing central roles in numerous projects as a technical leader and data scientist, delivering projects using open-source technologies for big companies, including successful projects in South America, Europe, and the United States. He is experienced in client engagement and working collaboratively with different teams. Currently, he is a process manager at Eclerx and is an accomplished postgraduate, having completed a degree in Data Science, Business Analytics, and Big Data. He holds a Bachelor's degree focused on Electronics and Telecommunication Engineering. In the meantime, he successfully got many certifications in data science and related tools. Furthermore, the author participates as a speaker in Data Science conferences and writes technical articles on machine learning and related topics. He also contributes to technical communities worldwide, such as Stack Overflow.

About the Reviewer

Karan Vijay Singh is a passionate Big Data / ML Engineer with extensive professional experience in the domain of Cloud Cost Optimization. With his optimisation skills, he has saved his current organization 0.5M dollar a year in cost. He also has co-authored a research paper in 2019 in the field of cloud computing which has over 270 citations worldwide.

Karan did his Master's in Mathematics (Computer Science) from University of Waterloo in Canada. He is currently working as a Data Engineer at Lightspeed Commerce in Canada.

Acknowledgement

I want to express my deepest gratitude to my family and friends for their unwavering support and encouragement throughout this book's writing, especially my wife, Dr. Archana and my son, Eshansh.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. It was a long journey of revising this book, with valuable participation and collaboration of reviewers, technical experts, and editors.

I would also like to acknowledge the valuable contributions of my colleagues and co-worker during many years working in the tech industry, who have taught me so much and provided valuable feedback on my work.

Finally, I would like to thank all the readers who have taken an interest in my book and for their support in making it a reality. Your encouragement has been invaluable.

Preface

Productionizing the machine learning model is a complex task that requires a comprehensive understanding of the latest technologies and CI/CD pipeline. MLOps become increasingly popular in the field of Data Science.

This book is designed to provide a comprehensive guide to building and deploying ML applications with MLOps. It covers a wide range of topics, including the basics of Python programming, Git, Machine Learning life cycle, Docker, and advanced concepts such as packaging Python code for ML models, monitoring, model security, Kubernetes, testing using pytest and the use of CI/CD pipeline for building and deploying robust and scalable ML applications on cloud platforms, including Azure, GCP, and AWS.

Throughout the book, you will learn about the MLOps, various tools, and techniques to deploy ML models. You will also learn how to use them to productionize ML models and applications that are efficient, scalable, and easy to maintain. Additionally, you will learn about best practices and design patterns for MLOps.

This book is intended for data scientists, software developers, data engineers, data analysts, and managers who are new to MLOps and want to learn how to productionize ML models. It is also helpful for experienced data scientists and ML engineers who want to expand their knowledge of these technologies and improve their skills in deploying ML models in production.

With this book, you will gain the knowledge and skills to become proficient in the field of MLOps. I hope you will find this book informative and helpful.

Chapter 1: Python 101 – explains the Python fundamental concepts needed for the reader to equip with the prerequisites required for this book, including Python installation, data structures, control statements, loops, functions, and data manipulation using pandas. This is a quick refresher for those who have worked with Python. If you are a beginner, this chapter will cover the most common Python commands needed to build and deploy ML models in production. It allows the reader to learn fundamental concepts related to the Object-Oriented Programming paradigm using Python.

Chapter 2: Git and GitHub Fundamentals – presents a detailed overview of Git workflow, including common Git commands with practical examples. This is

essential content for the entire book, as this chapter covers fundamental aspects of Git and GitHub that influence technical decisions to build the CI/CD pipelines to deploy ML models in the production environment.

Chapter 3: Challenges in ML Model Deployment – covers the various stages of the ML life cycle, including details on the challenges of each stage. It also covers the common challenges in deploying models in the production environment and how MLOps can help to overcome them. Additionally, the chapter discusses different approaches to deploying ML models in production.

Chapter 4: Packaging ML Models – allows the reader to learn fundamental concepts related to modular programming using the Python language, including Python packaging, dependency management, and good practices of software development to develop stable, readable, and extensible code for robust enterprise applications. Furthermore, the chapter explains the virtual environment, testing code using pytest, serializing, and deserializing ML models. It covers packaging ML models, code, and dependencies so that the package can be installed and consumed on another machine or server.

Chapter 5: MLflow-Platform to Manage the ML Life Cycle – gives special attention to streamlining machine learning development, including tracking experiments, packaging code into reproducible runs, and sharing and deploying models. It demonstrates how to train, deploy, and reuse ML models using MLflow through practical examples based on the use case. This chapter explains the role of MLflow in an ML life cycle.

Chapter 6: Docker for ML – shows the basic concepts of Docker and provides practical examples of common Docker commands with a use case for the reader. Learning these commands allows the reader to package ML code with its dependencies and run the application inside Docker containers. This chapter includes practical examples of Docker objects such as Docker images and containers.

Chapter 7: Build ML Web Apps Using API – explains in detail the most commonly used frameworks for building web-based ML apps using the Python language, including FastAPI, Streamlite, and Flask. It also allows the reader to learn the basics of Gunicorn, NGINX, and APIs, including an explanation of REST APIs, and much more.

Chapter 8: Build Native ML Apps – is dedicated to building native ML applications in Python to give the reader more familiarity with converting Python apps into Windows and Android apps and ways to consume them. This chapter covers practical examples of working with Tkinter, kivy, kivyMD, pyinstaller, and buildozer.

Chapter 9: CI/CD for ML – allows the reader to learn and implement the different stages of the CI/CD pipeline using GitHub and Jenkins, including committing, building, testing, and deploying. This chapter explains the GitHub and Jenkins integrations to build an automated CI/CD pipeline for deploying ML apps using Python and Docker.

Chapter 10: Deploying ML Models on Heroku – will guide the reader in configuring and building a CI/CD pipeline using GitHub Actions to deploy a web-based ML app on the Heroku platform. This chapter also explains three methods for deploying the web app on Heroku: Heroku Git, GitHub integration, and Container registry. It covers practical examples for creating workflow (YAML) files for GitHub Actions, including using tox and pytest for testing the code, and an automated Heroku pipeline for faster deployments.

Chapter 11: Deploying ML Models on Microsoft Azure – explains the step-by-step approach to building CI/CD pipeline using GitHub Actions to deploy web-based ML apps to Azure web services. In the other approach, the reader will learn to build a CI/CD pipeline and deploy web-based scalable ML apps to Azure Container Instances (ACI) and Azure Kubernetes Service (AKS) using Azure DevOps and Azure Machine Learning (AML) Service.

Chapter 12: Deploying ML Models on Google Cloud Platform – Shows how to build an automated CI/CD pipeline to deploy ML models on Google Kubernetes Engine (GKE), without the need to integrate any external tool, service, or platform. This chapter allows the reader to learn and implement Kubernetes functionality to run scalable ML apps using Google Kubernetes Engine (GKE).

Chapter 13: Deploying ML Models on Amazon Web Services – presents an overview of various cloud compute services offered by Amazon Web Services (AWS). This chapter allows the reader to learn and implement an automated CI/CD pipeline with Continuous Training (CT) to deploy scalable enterprise ML apps on Amazon Elastic Container Service (ECS). Additionally, it covers the integration of Application Load Balancer (ALB), Amazon Virtual Private Cloud

(Amazon VPC), and security groups into Amazon Elastic Container Service (ECS) for building robust and secure enterprise ML apps.

Chapter 14: Monitoring and Debugging – is dedicated to monitoring ML and operational metrics, including servers, cost of services, drifts in ML, input data, ML models, and much more. This chapter shows the importance and fundamental concepts of monitoring in the ML life cycle. It also covers practical examples using whylogs and WhyLabs for ML model monitoring, and Prometheus and Grafana for operational monitoring.

Chapter 15: Post-Productionizing ML Models – presents a detailed overview of adversarial machine learning, including different types of adversarial attacks and how to mitigate them. It also covers fundamental concepts of A/B testing and the future scope of MLOps in the industry.

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/v1v0nzp>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Machine-Learning-in-Production>. In case there's an update to the code, it will be updated on the existing GitHub repository. We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At www.bpbonline.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 BPB books and eBooks.

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 **business@bpbonline.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 **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights 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.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



Table of Contents

1. Python 101.....	1
Introduction.....	1
Structure.....	1
Objectives.....	2
Install Python	2
<i>On Windows and Mac OS</i>	2
<i>On Linux</i>	2
Install Anaconda	2
Install code editor	3
Hello World!	3
<i>Execution of Python file</i>	3
Data structures	4
<i>Common data structures</i>	4
Control statements and loops	6
Functions.....	8
Object Oriented Programming (OOP).....	9
Numerical Python (NumPy).....	10
Pandas	12
Conclusion	15
Points to remember	15
Multiple choice questions.....	16
<i>Answers</i>	16
Questions	16
Key terms	16
2. Git and GitHub Fundamentals.....	17
Introduction.....	17
Structure.....	17
Objectives.....	18
Git concepts	18

<i>Git + Hub = GitHub</i>	19
Common Git workflow.....	19
Install Git and create a GitHub account.....	20
<i>Linux (Debian/Ubuntu)</i>	20
<i>Git for all platforms</i>	20
<i>GUI clients</i>	20
<i>Create a GitHub account</i>	20
Common Git commands.....	21
<i>Setup</i>	21
<i>New repository</i>	21
<i>Update</i>	21
<i>Changes</i>	21
<i>Revert</i>	22
Let's Git	22
<i>Configuration</i>	22
<i>Initialize the Git repository</i>	22
<i>Check Git status</i>	22
<i>Add a new file</i>	23
Conclusion	26
Points to remember	26
Multiple choice questions	26
<i>Answers</i>	27
3. Challenges in ML Model Deployment	29
Introduction	29
Structure	29
Objectives.....	30
ML life cycle.....	30
<i>Business impact</i>	31
<i>Data collection</i>	31
<i>Data preparation</i>	32
<i>Feature engineering</i>	32
<i>Build and train the model</i>	33

<i>Test and evaluate</i>	34
<i>Model deployment</i>	34
<i>Monitoring and optimization</i>	35
Types of model deployment.....	35
<i>Batch predictions</i>	35
<i>Web service/REST API</i>	36
<i>Mobile and edge devices</i>	36
<i>Real-time</i>	37
Challenges in deploying models in the production environment.....	37
<i>Team coordination</i>	37
<i>Data-related challenges</i>	38
<i>Portability</i>	38
<i>Scalability</i>	38
<i>Robustness</i>	38
<i>Security</i>	39
MLOps.....	39
Benefits of MLOps	40
<i>Efficient management of ML life cycle</i>	41
<i>Reproducibility</i>	41
<i>Automation</i>	41
<i>Tracking and feedback loop</i>	42
Conclusion	42
Points to remember	42
Multiple choice questions.....	43
<i>Answers</i>	43
Questions	43
Key terms	43
4. Packaging ML Models.....	45
Introduction.....	45
Structure.....	45
Objectives.....	46
Virtual environments	46

Requirements file	47
Serializing and de-serializing ML models	48
Testing Python code with pytest	48
<i>pytest fixtures</i>	49
Python packaging and dependency management	49
<i>Modular programming</i>	50
<i>Module</i>	50
<i>Package</i>	50
Developing, building, and deploying ML packages	51
<i>Business problem</i>	52
<i>Data</i>	52
<i>Building the ML model</i>	53
<i>Developing the package</i>	53
Set up environment variables and paths	70
<i>Build the package</i>	71
<i>Install the package</i>	71
<i>Package usage with example</i>	73
Conclusion	74
Points to remember	74
Multiple choice questions	74
<i>Answers</i>	75
Questions	75
Key terms	75
5. MLflow-Platform to Manage the ML Life Cycle	77
Introduction	77
Structure	77
Objectives	78
Introduction to MLflow	78
<i>Set up your environment and install MLflow</i>	79
<i>Miniconda installation</i>	79
<i>MLflow components</i>	82
MLflow tracking	83

<i>Log data into the run</i>	84
MLflow projects	94
MLflow models	97
MLflow registry	100
<i>Set up the MySQL server for MLflow.....</i>	101
<i>Start the MLflow server.....</i>	103
Conclusion	109
Points to remember	109
Multiple choice questions.....	109
<i>Answers</i>	110
Questions	110
6. Docker for ML.....	111
Introduction.....	111
Structure.....	111
Objectives.....	112
Introduction to Docker.....	112
<i>It works on my machine!</i>	112
<i>Long setup.....</i>	112
<i>Setting up your environment and installing Docker</i>	113
<i>Docker installation.....</i>	113
<i>Uninstall old versions</i>	113
<i>Install Docker Engine</i>	114
<i>Docker compose.....</i>	115
Hello World with Docker	116
Docker objects	117
<i>Dockerfile.....</i>	117
<i>Docker image</i>	117
<i>Docker containers.....</i>	118
<i>Docker container networking.....</i>	118
Create a Dockerfile	119
Build a Docker image.....	120
Run a Docker container	120

Dockerize and deploy the ML model	122
Common Docker commands	127
Conclusion	127
Points to remember	127
Multiple choice questions.....	128
<i>Answers</i>	128
Questions	128
7. Build ML Web Apps Using API	129
Introduction.....	129
Structure.....	129
Objectives.....	130
Rest APIs	130
FastAPI	131
Streamlit	139
Flask.....	143
<i>Gunicorn</i>	150
<i>NGINX</i>	150
Conclusion	154
Points to remember	154
Multiple choice questions.....	155
<i>Answers</i>	155
8. Build Native ML Apps	157
Introduction.....	157
Structure.....	157
Objectives.....	158
Introduction to Tkinter	158
<i>Hello World app using Tkinter</i>	159
Build an ML-based app using Tkinter.....	160
<i>Tkinter app</i>	163
Convert Python app into Windows EXE file	167
Build an ML-based app using kivy and kivyMD	170
<i>KivyMD app</i>	173

Convert the Python app into an Android app	179
Conclusion	180
Points to remember	181
Multiple choice questions.....	181
<i>Answers</i>	181
Questions	181
9. CI/CD for ML	183
Introduction.....	183
Structure.....	183
Objectives.....	184
CI/CD pipeline for ML.....	184
Continuous Integration (CI).....	185
Continuous Delivery/Deployment (CD).....	186
Continuous Training (CT)	186
Introduction to Jenkins	187
<i>Installation</i>	187
Build CI/CD pipeline using GitHub, Docker, and Jenkins	189
<i>Develop codebase</i>	189
<i>Create a Personal Access Token (PAT) on GitHub</i>	196
<i>Create a webhook on the GitHub repository</i>	197
Configure Jenkins	199
Create CI/CD pipeline using Jenkins.....	202
<i>Stage 1: 1-GitHub-to-container</i>	203
<i>Stage 2: 2-training</i>	205
<i>Stage 3: 3-testing</i>	207
<i>Stage 4: 4-deployment-status-email</i>	210
Conclusion	217
Points to remember	217
Multiple choice questions.....	218
<i>Answers</i>	218
Questions	218

10. Deploying ML Models on Heroku	219
Introduction.....	219
Structure.....	219
Objectives.....	220
Heroku.....	220
Setting up Heroku	221
Deployment with Heroku Git.....	222
Deployment with GitHub repository integration.....	222
REVIEW APPS.....	223
STAGING.....	223
PRODUCTION	224
<i>Heroku Pipeline flow</i>	224
Deployment with Container Registry.....	225
GitHub Actions	225
Configuration.....	226
CI/CD pipeline using GitHub Actions and Heroku	227
Conclusion	246
Points to remember	246
Multiple choice questions.....	247
<i>Answers</i>	247
Questions	247
11. Deploying ML Models on Microsoft Azure.....	249
Introduction.....	249
Structure.....	249
Objectives.....	250
Azure	250
Set up an Azure account	251
Deployment using GitHub Actions	251
<i>Infrastructure setup</i>	263
<i>Azure Container Registry</i>	263
<i>Azure App Service</i>	267
<i>GitHub Actions</i>	270

<i>Service principal</i>	273
<i>Configure Azure App Service to use GitHub Actions for CD</i>	274
Deployment using Azure DevOps and Azure ML	278
Azure Machine Learning (AML) service.....	279
<i>Workspace</i>	280
<i>Experiments</i>	280
<i>Runs</i>	280
Configure CI pipeline.....	290
Configure CD pipeline.....	292
Conclusion	299
Points to remember	299
Multiple choice questions.....	299
<i>Answers</i>	300
Questions	300
12. Deploying ML Models on Google Cloud Platform.....	301
Introduction.....	301
Structure.....	301
Objectives.....	302
Google Cloud Platform (GCP).....	302
<i>Set up the GCP account</i>	303
Cloud Source Repositories	304
Cloud Build	309
Container Registry	311
Kubernetes	312
Google Kubernetes Engine (GKE).....	313
Deployment using Cloud Shell – Manual Trigger	316
CI/CD pipeline using Cloud Build.....	317
<i>Create a trigger in Cloud Build</i>	318
Conclusion	323
Points to remember	323
Multiple choice questions.....	323
<i>Answers</i>	324

Questions	324
13. Deploying ML Models on Amazon Web Services.....	325
Introduction.....	325
Structure.....	325
Objectives.....	326
Introduction to Amazon Web Services (AWS).....	326
<i>AWS compute services</i>	326
<i>Amazon Elastic Compute Cloud (EC2)</i>	327
<i>Amazon Elastic Container Service (ECS)</i>	327
<i>Amazon Elastic Kubernetes Service (EKS)</i>	327
<i>Amazon Elastic Container Registry (ECR)</i>	327
<i>AWS Fargate</i>	328
<i>AWS Lambda</i>	328
<i>Amazon SageMaker</i>	328
Set up an AWS account	329
<i>AWS CodeCommit</i>	331
<i>Continuous Training</i>	336
<i>Amazon Elastic Container Registry (ECR)</i>	336
<i>Docker Hub rate limit</i>	337
<i>AWS CodeBuild</i>	339
<i>Attach container registry access to CodeBuild's service role</i>	345
<i>Amazon Elastic Container Service (ECS)</i>	346
<i>AWS ECS deployment models</i>	347
<i>EC2 instance</i>	347
<i>Fargate</i>	348
<i>Task definition</i>	349
<i>Running task with the task definition</i>	350
<i>Amazon VPC and subnets</i>	351
<i>Load balancing</i>	352
<i>Target group</i>	353
<i>Security Groups</i>	356
<i>Application Load Balancers (ALB)</i>	358

<i>Service</i>	363
CI/CD pipeline using CodePipeline	369
<i>AWS CodePipeline</i>	371
Monitoring.....	378
Conclusion.....	379
Points to remember	379
Multiple choice questions.....	380
<i>Answers</i>	380
Questions	380
14. Monitoring and Debugging	381
Introduction.....	381
Structure.....	381
Objectives.....	382
Importance of monitoring	382
Fundamentals of ML monitoring	383
Metrics for monitoring your ML system.....	385
Drift in ML.....	386
<i>Types of drift in ML</i>	386
<i>Techniques to detect the drift in ML</i>	388
<i>Addressing the drift in ML</i>	389
Operational monitoring with Prometheus and Grafana.....	390
ML model monitoring with whylogs and WhyLabs.....	402
<i>whylogs</i>	403
<i>Constraints for data quality validation</i>	404
<i>WhyLabs</i>	407
Conclusion	416
Points to remember	416
Multiple choice questions.....	417
<i>Answers</i>	417
Questions	417



15. Post-Productionizing ML Models	419
Introduction	419
Structure	419
Objectives.....	420
Bridging the gap between the ML model and the creation of business value	420
Model security.....	420
<i>Adversarial attack</i>	421
<i>Data poisoning attack</i>	422
<i>Distributed Denial of Service attack (DDoS)</i>	422
<i>Data privacy attack</i>	422
<i>Mitigate the risk of model attacks</i>	423
A/B testing	423
MLOps is the future	424
Conclusion	425
Points to remember	425
Multiple choice questions.....	425
<i>Answers</i>	426
Questions	426
Index	427-434

Introduction

Python 101 guides you with everything from the installation of Python to data manipulation in Pandas DataFrame. In a nutshell, this chapter is designed to equip you with the prerequisites required for this book. It is a quick refresher for those who have worked with Python. And if you are a beginner, this chapter is going to cover the most common Python commands required to build and deploy machine models.

Structure

- This chapter covers the following topics:
- Installation of Python
- Hello World!
- Data structures
- Control statements and loops
- Functions
- OOPs
- NumPy
- Pandas

Objectives

After studying this chapter, you should be able to install Python on your machine, and store and manage data using common data structures in Python. You should be able to use Python's data processing packages for data manipulation, and you should also know how to create classes, methods, and objects.

Install Python

Make sure the Python version you are installing is compatible with the libraries and applications you will work on. Also, maintain the same Python version throughout the project to avoid any errors or exceptions. Here, you will install Python version 3.6.10.

On Windows and Mac OS

Here's the link to install Python 3.6.10:

<https://www.python.org/downloads/release/python-3610/>

On Linux

Ubuntu 16.10 and 17.04:

```
sudo apt update  
sudo apt install python3.6
```

Ubuntu 17.10, 18.04 (Bionic) and onward:

Ubuntu 17.10 and 18.04 already come with Python 3.6 as default. Just run `python3` in the terminal to invoke it.

Install Anaconda

Anaconda Individual Edition contains conda and Anaconda Navigator, as well as Python and hundreds of scientific packages. When you install Anaconda, all of these will also get installed.

<https://docs.anaconda.com/anaconda/install/>

Note: Review the system requirements listed on the site before installing Anaconda Individual Edition.

Install code editor

Any of the following code editors can be chosen; however, the preferred one is Visual Studio Code:

Visual Studio Code (<https://code.visualstudio.com>)

Sublime Text (<https://www.sublimetext.com>)

Notepad++ (<https://notepad-plus-plus.org/downloads/>)

Hello World!

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its simple, easy-to-learn syntax emphasizes readability and therefore, reduces the cost of program maintenance. Python supports modules and packages, which encourages modular programming and code reusability. It is developed under an OSI-approved open-source license, making it freely usable and distributable, even for commercial use. The **Python Package Index (PyPI)** hosts thousands of third-party modules for Python.

Open the terminal and type a `python` to open the Python console.

Now you are in the Python console.

Example: 1.1 Hello world in Python

```
1. print("Hello World")
2. Hello World
```

To come out of the console you can use:

```
1. exit()
```

Execution of Python file

Let's create a file named `hello_world.py` and add the `print("Hello World")` to it.

Now, run the file in the terminal:

```
python hello_world.py
```

You should get the following output:

```
Hello World
```

Data structures

Data structures are nothing but particular ways of storing and managing data in memory so that it can be easily accessed and modified later. Python comes with a comprehensive set of data structures, which play an important role in programming because they are reusable, easily accessible, and manageable.

Common data structures

You will study some of the common data structures in this section.

Array

Arrays are collections of homogeneous items. One can use the same data type in a single array.

Example: 1.2 Array data type

```
1. import array as arr
2.
3. my_array = arr.array("i", (1, 2, 3, 4, 5))
4.
5. print(my_array)
6. array('i', [1, 2, 3, 4, 5])
7.
8. print(my_array[1])
9. 2
```

Dictionary

Dictionaries are defined as comma separated **key:value** pairs enclosed in curly braces.

Example: 1.3 Dictionary data type

```
1. my_dict = {'name': 'Adam', 'emp_id': 3521, 'dept':'Marketing'}
2. print(my_dict['name'])
3. Adam
```

List

It is a collection of heterogeneous items enclosed in square brackets. One can use the same or different data types in a list.

Example: 1.4 List data type

```

1. my_list=['apple', 4, 'banana', 6]
2. print(my_list[0])
3. apple

```

Set

A set is a collection of unique (non-duplicate) elements enclosed in curly braces. A set can take heterogeneous elements.

Example: 1.5 Set data type

```

1. my_set = {3,5,6}
2. print(my_set)
3. {3, 5, 6}
1. my_set = {1, 2.0, (3,5,6), 'Test'}
2. print(my_set)
3. {1, 2.0, 'Test', (3, 5, 6)}

```

String

A string is used to store text data. It can be represented by single quotes ("") or double quotes ("").

Example: 1.6 String data type

```

1. print('The cat was chasing the mouse')
2. The cat was chasing the mouse

```

Tuple

Tuples are collections of elements surrounded by round brackets (optional), and they are immutable, that is, they cannot be changed.

Example: 1.7 Tuple data type

```

1. my_tuple_1 = ('apple', 4, 'banana', 6)
2. print(my_tuple_1[0])
3. apple

```

Tuple can be declared without round brackets, as follows:

```

1. my_tuple_2 = 1, 2.0, (3,5,6), 'Test'
2. print(my_tuple_2[0])
3. 1

```

Control statements and loops

Python has several types of control statements and loops. Let's look at them.

if...else

The **if** statement executes a block of code if the specified condition is true. Whereas, the **else** statement executes a block of code if the specified condition is false. Furthermore, you can use the **elif** (else if) statement to check the new condition if the first condition is false.

Syntax:

if condition1:

 Code to be executed

elif condition2:

 Code to be executed

else:

 Code to be executed

Example: 1.8 If...else control statement

```
1. x = 26
2. y = 17
3.
4. if(x > y):
5.     print('x is greater than y')
6. elif(x == y):
7.     print('x and y are equal')
8. else:
9.     print('y is greater than x')
10.
11.x is greater than y
```

for loop

It is used when you want to iterate over a sequence or iterable such as a string, list, dictionary, or tuple. It will execute a block of code a certain number of times.

Syntax:

```
for val in sequence:
```

Code to be executed

Example: 1.9 For loop

```
1. num = 2
2. for i in range(1, 11):
3.     print(num, 'X', i, '=', num*i)
4.
5. 2 X 1 = 2
6. 2 X 2 = 4
7. 2 X 3 = 6
8. 2 X 4 = 8
9. 2 X 5 = 10
10. 2 X 6 = 12
11. 2 X 7 = 14
12. 2 X 8 = 16
13. 2 X 9 = 18
14. 2 X 10 = 20
```

In the preceding example, the `range()` function generates a sequential set of numbers using start, stop, and step parameters.

while loop

It is used when you want to execute a block of code indefinitely until the specified condition is met. Furthermore, you can use an `else` statement to execute a block of code once when the first condition is false.

Syntax:

```
while condition:
```

Code to be executed

Example: 1.10 While loop

```
1. num = 1
2. while num<= 10:
3.     if num % 2 == 0:
4.         print(num)
5.     num = num + 1
```

- 6.
- 7. 2
- 8. 4
- 9. 6
- 10. 8
- 11. 10

pass statement

In Python, the pass statement acts as a placeholder. If you are planning to add code later in loops, function definitions, class definitions, or if statements, you can write a pass to avoid any errors as it does nothing. It allows developers to write the logic or condition afterward and continue the execution of the remaining code.

Syntax:

if condition:

pass

Example: 1.11 Pass statement

```
1. num = 10
2. if num % 2 == 0:
3.     pass
```

Functions

Developers use functions to perform the same task multiple times. A function is a block of code that can take arguments, perform some operations, and return values. Python enables developers to use predefined or built-in functions, or they can write user-defined functions to perform a specific task.

Example: 1.12 Pre-defined or built-in functions

```
1. print("Hello World")
2. Hello World    # Output
```

Example: 1.13 User-defined functions

```
1. a = 5
2. b = 2
3.
4. # Function definition
```

```

5. def add():
6.     print(a+b)
7.
8. # Calling a function
9. add()
10.
11. # Output

```

12.7

Object Oriented Programming (OOP)

Python is an object-oriented language, so everything in Python is an object. Let's understand its key concepts and their importance:

Class: A class acts like a template for the objects. It is defined using the **class** keyword like the **def** keyword is used while creating a new function. A Python class contains objects and methods, and they can be accessed using the period (.).

Object: An object is an instance of a class. It depicts the structure of the class and contains class variables, instance variables, and methods.

Method: In simple words, it is a function defined while creating a class.

__init__: For the automatic initialization of data members by assigning values, you should use the **__init__** method while creating an instance of a class. It gets called every time you create an object of the class. The usage is equivalent to the constructor in C++ and Java.

Self: It helps us access the methods and attributes of the class. You are free to name it anything; however, as per convention and for readability, it is better to declare it as self.

Example: 1.14 Class definition

```

1. class Multiply:
2.     def mul(self):
3.         result = self.num1 * self.num2
4.         return result
5.
6.     def __init__(self,num1,num2):
7.         self.num1 = num1
8.         self.num2 = num2

```

Class definition

```
9.  
10.      a = Multiply(5,6)  
11.                                         Passing parameters to class  
12.      a.mul()  
13.                                         Calling method of class  
14.      30
```

Note: Follow standard naming conventions for variables, functions, and methods. For example:

- For variables, functions, methods, packages, and modules: `my_variable`
- For classes and exceptions: `MyClass`
- For constants: `MY_CONSTANT`

Numerical Python (NumPy)

In a nutshell, it is an array processing package. NumPy stands for Numerical Python. By default, the data type of the NumPy array is defined by its elements. In NumPy, the dimension of arrays is referred to by rank; for example, a 2-D array means Rank 2 array.

NumPy is popular because of its speed. While working on a large dataset, you will see the difference if you use NumPy.

You can install the NumPy package using pip:

```
pip install numpy
```

Example: 1.15 NumPy array

```
1. import numpy as np  
2. # Rank 0 Array (scaler)  
3. my_narray = np.array(42)  
4. print(my_narray)  
5. 42  
6.  
7. print("Dimension: ", my_narray.ndim)  
8. Dimension: 0  
9.  
10.     print("Shape: ", my_narray.shape)  
11.     Shape: ()  
12.
```

```
13.     print("Size: ", my_narray.size)
14.     Size:  1
1. # Rank 1 Array (vector)
2. my_narray = np.array([4,5,6])
3. print(my_narray)
4. [4 5 6]
5. print("Dimension: ", my_narray.ndim)
6. Dimension:  1
7. print("Shape: ", my_narray.shape)
8. Shape:  (3,)
9. print("Size: ", my_narray.size)
10.    Size:  3

1. # Rank 2 Array (matrix)
2. my_narray = np.array([[1,2,3],[4,5,6]])
3. print(my_narray)
4. [[1 2 3]
5. [4 5 6]]
6. print("Dimension: ", my_narray.ndim)
7. Dimension:  2
8. print("Shape: ", my_narray.shape)
9. Shape:  (2, 3)
10.    print("Size: ", my_narray.size)
11.    Size:  6

1. # Rank 3 Array
2. my_narray = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
3. print(my_narray)
4. [[[ 1  2  3]
5. [ 4  5  6]]
6. [[ 7  8  9]
7. [10 11 12]]]
8. print("Dimension: ", my_narray.ndim)
9. Dimension:  3
10.   print("Shape: ", my_narray.shape)
11.   Shape:  (2, 2, 3)
```

```
12.     print("Size: ", my_ndarray.size)
13.     Size: 12
```

Reshaping array

Suppose you want to change the shape of a `ndarray` (N-dimension array) without losing the data; it can be done using the `reshape()` or `flatten()` method.

Example: 1.16 Reshaping NumPy array

```
1. # Rank 1 Array
2. my_1array = np.array([1,2,3,4,5,6])
3. print(my_1array)
4. [1 2 3 4 5 6]

1. # converting Rank 1 array to Rank 2 array
2. my_2array = my_1array.reshape(2, 3)
3. print(my_2array)
4. [[1 2 3]
5. [4 5 6]]

1. #converting Rank 2 array to Rank 1 array
2. my_1array = my_2array.flatten()
3. print(my_1array)
4. [1 2 3 4 5 6]
```

Note: Here, you can use `my_2array.reshape(6)` instead of `flatten()`.

Pandas

Pandas is an open-source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. A DataFrame is a 2-dimensional labeled data structure with columns of potentially different types.

You can install the Pandas package using pip:

```
pip install pandas
```

Example: 1.17 Pandas DataFrame usage

```
1. import pandas as pd
2. df = pd.DataFrame(data={'fruit': ['apple', 'banana', 'orange',
```

```
    'mango',
3.                               'size': ['large', 'medium', 'small',
    'medium'],
4.                               'quantity': [5,4,8, 6]})

5.

6. print(df)
7.      fruit      size   quantity
8.  0   apple     large        5
9.  1   banana   medium       4
10. 2  orange    small        8
11. 3  mango    medium       6
```

In the following example, `head()` retrieves the top rows of the Pandas DataFrame.

Example: 1.18 Get first n rows of DataFrame

```
1. df.head(2)
2.      fruit      size   quantity
3.  0   apple     large        5
4.  1   banana   medium       4
```

Whereas `tail()` retrieves the bottom rows of the Pandas DataFrame.

Example: 1.19 Get the last n rows of DataFrame

```
1. df.tail(2)
2.      fruit      size   quantity
3.  2  orange    small        8
4.  3  mango    medium       6
```

In the following example, `describe()` shows a quick statistical summary of numerical columns of the DataFrame.

Example: 1.20 Get basic statistical information

```
1. df.describe()
2.      quantity
3. count  4.000000
4. mean   5.750000
5. std    1.707825
6. min    4.000000
```

```
7. 25%    4.750000
8. 50%    5.500000
9. 75%    6.500000
10. max    8.000000

1. df['fruit']
2. 0      apple
3. 1      banana
4. 2      orange
5. 3      mango
6. Name: fruit, dtype: object
```

loc - selection by label

A **loc** gets rows (and/or columns) with specific labels. To get all the rows in which fruit size is medium, **loc** is written as follows:

Example: 1.21 Get the data by location

```
1. df.loc[df['size'] == 'medium']
2.      fruit      size   quantity
3. 1  banana  medium        4
4. 3  mango  medium        6
```

iloc - selection by position

iloc gets rows (and/or columns) at the index's locations. To get the row at index 2, **iloc** is written as follows:

Example: 1.22 Get the data by position

```
1. df.iloc[2]
2.      fruit      size
3. 2  orange  small
4.      quantity     8
5. Name: 2, dtype: object
```

To retrieve the rows from index 2 to 3 and columns 0 to 1, **iloc** is written as follows:

```
1. df.iloc[2:4, 0:2]
2.      fruit      size
3. 2  orange  small
4. 3  mango  medium
```

You can call `groupby()` and pass the size, that is, the column you want the group on, and then use the `sum()` aggregate method.

Example: 1.23 Group the data, then use an aggregate function

```
1. df.groupby(['size'])['quantity'].sum()
2. size
3. large      5
4. medium    10
5. small      8
6. Name: quantity, dtype: int64
```

Save and load CSV files

Example: 1.24 Save and load CSV files

```
1. df.to_csv("fruit.csv")
2. df = pd.read_csv("fruit.csv")
```

Save and load Excel files

Example: 1.25 Save and load Excel files

```
1. df.to_excel("fruit.xlsx", sheet_name="Sheet1")
2. df = pd.read_excel("fruit.xlsx", "Sheet1", index_col=None)
```

For more information, you can refer to the official documentation at <https://pandas.pydata.org/docs/>.

Conclusion

This chapter discussed the essential concepts of the Python language with examples. At the beginning of this chapter, you learned how to install Python in the system. Then, you studied the common data structures and object-oriented programming concepts in Python with examples. Further on, data pre-processing commands were covered using packages like NumPy and Pandas, which will play a major role in developing machine learning models.

In the next chapter, you will learn about git and GitHub with examples.

Points to remember

- Ensure to use the same Python version throughout the machine learning project.

- The `__init__` method in a class is used for auto initialization of data members and methods. However, it is not mandatory.
- Tuples are immutable, that is, they are not changeable, but if they contain elements like a list, then the list is mutable but not the entire tuple.

Multiple choice questions

1. What is the output of `np.array([[1,2,3],[4,5,6]])`?
 - a) Rank 1 array
 - b) Rank 2 array
 - c) Rank 2 array
 - d) All of the above
2. Sets can take _____ elements.
 - a) heterogenous
 - b) homogeneous
 - c) only integer
 - d) only string

Answers

1. b
2. a

Questions

1. What is the difference between a list and a tuple?
2. What is the use of `__init__()`?
3. When should you use a pass statement?

Key terms

Object-Oriented Programming: It refers to a programming paradigm based on the concept of objects, which can contain data and methods.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 2

Git and GitHub

Fundamentals

Introduction

It is difficult to maintain multiple versions of files while working in a team. Git solves this problem by allowing developers to collaborate and share files with other team members. This chapter covers the most needed basic Git and GitHub concepts that form the foundation of this book. It explains Git configuration, forking Git repo, pushing your codebase to GitHub, and the cloning process. In addition, it covers an introduction to GitHub Actions, common Git commands with syntax, and examples. Maintaining multiple versions of files when working in a team can be difficult, but Git is the solution.

Structure

The following topics will be covered in this chapter:

- Git concepts
- Common Git workflow
- Install Git and create a GitHub account
- Common Git commands
- Let's Git

Objectives

After studying this chapter, you should know how to execute Git commands and connect to remote GitHub repositories. Understand and execute Git processes. You should know how to push, fetch, and revert changes to the remote Git repository, and you should be familiar with how to install and perform the basic configuration of Git on your machine or server.

Git concepts

Git is an open-source **Distributed Version Control System (DVCS)**. A version control system allows you to record changes to files over a period.

Git is used to maintain the historical and current versions of source code. In a project, developers have a copy of all versions of the code stored in the central server.

Git allows developers to do the following:

- Track the changes, who made the changes, and when
- Rollback/ restore changes
- Allow multiple developers to coordinate and work on the same files
- Maintain a copy of the files at the remote and local level

The following image depicts Git as a VCS in a team where developers can work simultaneously on the same files and keep track of who made the changes. Here, *F1*, *F2*, and *F3* are file names.

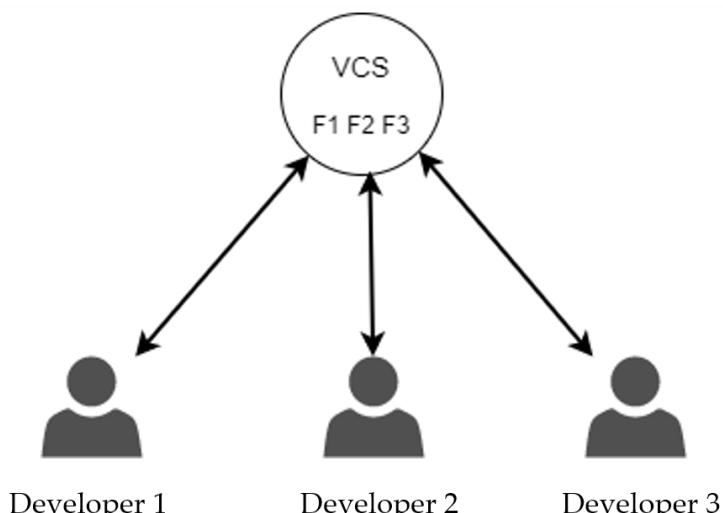


Figure 2.1: Git scenario in team

Git + Hub = GitHub

Git and GitHub are separate entities. Git is a command-line tool, whereas GitHub is a platform for collaboration. You can store files and folders on GitHub and implement changes to existing projects. By creating a separate branch, you can isolate these changes from your existing project files.

GitHub Actions makes it easy to automate all your software workflows with Continuous Integration/Continuous Deployment. You can build, test, and deploy the code right from GitHub.

Common Git workflow

The following figure depicts the general Git workflow. It covers operations like creating or cloning the Git repository, updating the local repository by pulling files from the remote repository, and pushing the local changes to the remote repository.

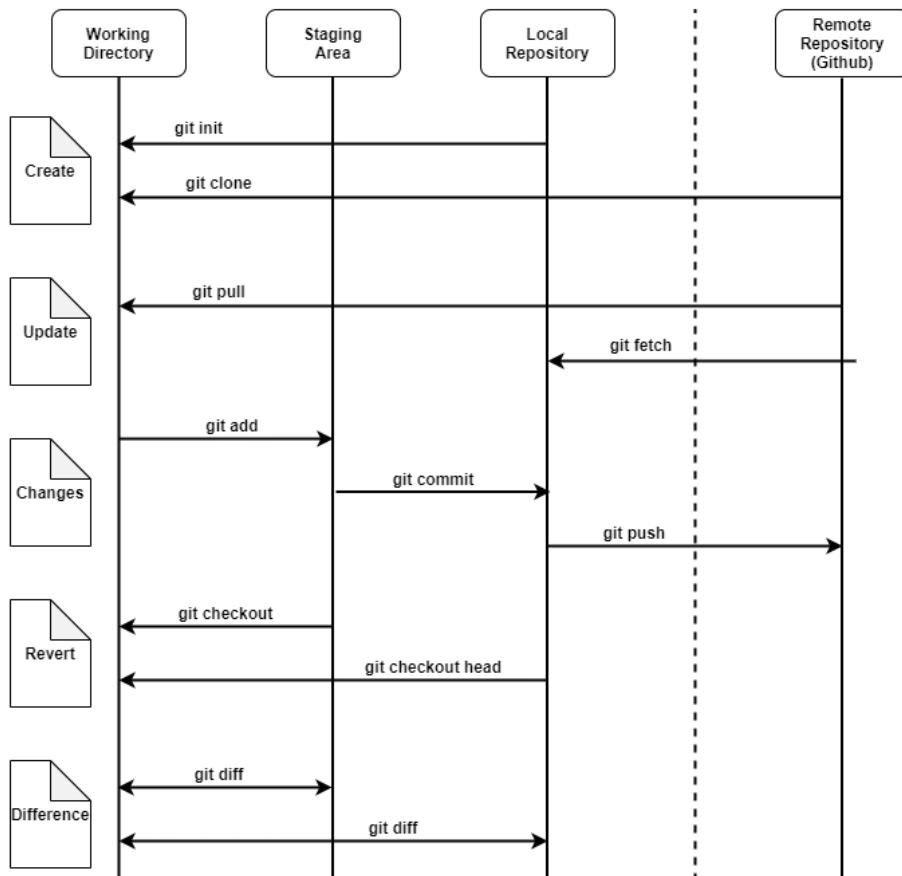


Figure 2.2: Git workflow

Install Git and create a GitHub account

First off, you will need to install Git and then create an account on GitHub. To install Git on your machine, follow the given instructions.

Linux (Debian/Ubuntu)

In Ubuntu, open the terminal and install Git using the following commands:

```
$ sudo apt-get update  
$ sudo apt-get install git
```

Git for all platforms

Download Git for your machine from the official website:

<https://git-scm.com/downloads>

GUI clients

Git comes with built-in GUI tools; however, you can explore other third-party GUI tools at <https://git-scm.com/downloads/guis>.

Click on the download link for your operating system and then follow the installation steps.

After installing it, start your terminal and type **git --version** to verify the Git installation.

If everything goes well, you will see the Git installed successfully.

Create a GitHub account

If you are new to GitHub, then can join GitHub at <https://github.com/join>.

If you're an existing user, sign in to your account.

Create a new repository by clicking on the plus sign + (top-right corner):

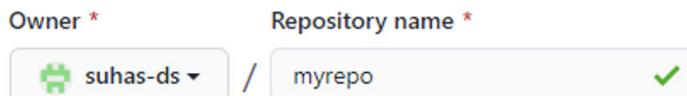


Figure 2.3: Creating a new repository on GitHub

Hit the **Create repository** button. Complete the required fields, and your Git repo will be created.

You can download and install the GitHub desktop from <https://desktop.github.com/>.

Common Git commands

Some of the common Git commands are discussed in this section.

Setup

Set a name that is identifiable for credit when reviewing the version history:

```
git config --global user.name "[firstname lastname]"
```

Set an email address that will be associated with each history marker:

```
git config --global user.email "[valid-email-id]"
```

New repository

Initialize an existing directory as a Git repository:

```
git init
```

Retrieve an entire repository from a hosted location via URL:

```
git clone [url]
```

Update

Fetch and merge any commits from the remote branch:

```
git pull
```

Fetch all the branches from the remote Git repo:

```
git fetch [alias]
```

Changes

View modified files in the working directory staged for your next commit:

```
git status
```

Add a file to your next commit (stage):

```
git add [file]
```

Commit your staged content as a new commit snapshot:

```
git commit -m "[descriptive message]"
```

Transfer local branch commits to the remote repository branch:

```
git push [alias] [branch]
```

Revert

View all the commits in the current branch's history:

```
git log
```

Switch to another branch:

```
git checkout ['branch_name']
```

Let's Git

Create a new directory *code* and switch to the *code* directory:

```
suhas@test:~/code$ sudo chmod -R 777 /home/suhas/code
```

Configuration

Provide the username and the user's email:

```
suhas@test:~/code$ git config --global user.name "Suhas"
```

```
suhas@test:~/code$ git config --global user.email "suhasp.ds@gmail.com"
```

Note: Always read the output of commands to know what happened in the background.

Initialize the Git repository

```
suhas@test:~/code$ git init
```

```
Initialized empty Git repository in /home/suhas/code/.git/
```

Check Git status

```
suhas@test:~/code$ git status
```

```
On branch master
```

```
No commits yet
```

```
nothing to commit (create/copy files and use "git add" to track)
```

Add a new file

A new file `hello.py` has been added using the `touch` command:

```
suhas@test:~/code$ touch hello.py
```

You can notice the change in the output of the `git status`:

```
suhas@test:~/code$ git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hello.py
nothing added to commit but untracked files present (use "git add" to track)
```

Add the `hello.py` file to staging:

```
suhas@test:~/code$ git add hello.py
```

Again, check the output of the Git status:

```
suhas@test:~/code$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   hello.py
```

Commit with the message `New file`:

```
suhas@test:~/code$ git commit -m "New file"
[master (root-commit) bd49a5e] New file
  1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100755 hello.py
```

Check the output of Git status:

```
suhas@test:~/code$ git status
On branch master
nothing to commit, working tree clean
```

Update the `hello.py` file using the `nano` command:

```
suhas@test:~/code$ sudo nano hello.py
```

Now, add the `print ("Hello Word!")` to the `hello.py` file.

You can see the changes in the output of the `git status`:

```
suhas@test:~/code$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working
     directory)

        modified:   hello.py
no changes added to commit (use "git add" and/or "git commit -a")
```

Add the `hello.py` file to staging:

```
suhas@test:~/code$ git add hello.py
```

Commit along with the message `added print text`:

```
suhas@test:~/code$ git commit -m "added print text"
[master af7cfec] added print text
 1 file changed, 1 insertion(+)
```

View the logs using the `git log` command:

```
suhas@test:~/code$ git log
commit af7cfec53ff6dc49126d24aedb20a065c54ef4a0 (HEAD -> master)
Author: "Suhas Pote" <"suhas.pote@gmail.com">
Date:  Sun Jul 5 21:34:15 2020 +0530

    added print text
commit bd49a5e5280de35811848a80299d900e6e6509ce
Author: "Suhas Pote" <"suhas.pote@gmail.com">
Date:  Sun Jul 5 21:26:12 2020 +0530

    New file
```

Git identifies each commit uniquely using the SHA1 hash function, based on the contents of the committed files. So, each commit is identified with a 40-character-long hexadecimal string.

```
suhas@test:~/code$ git checkout af7cfec53ff6dc49126d24aedb20a065c54ef4a0
Note: checking out 'bd49a5e5280de35811848a80299d900e6e6509ce'.
```

You are in a 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again.

Example:

```
git checkout -b <new-branch-name>
HEAD is now at bd49a5e New file
```

If you notice, the *hello.py* file is backed to the previous state, that is, a blank file. It is like a time machine!

Now, let's get back to the latest version of *hello.py*.

```
suhas@test:~/code$ git reset --hard
af7cfab53ff6dc49126d24aedb20a065c54ef4a0
HEAD is now at af7cfab added print text
```

It's time to push the files to the GitHub repo.

```
suhas@test:~/code$ git remote add origin https://github.com/suhas-ds/
myrepo.git

suhas@test:~/code$ git push -u origin master
Username for 'https://github.com': suhas-ds
Password for 'https://suhas-ds@github.com':
Counting objects: 6, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 477 bytes | 477.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/suhas-ds/myrepo
 * [new branch]      master -> master
Branch 'master' is set up to track remote branch 'master' from 'origin'.
```

Note: Here, the origin acts as an alias or label for the URL, and the master is a branch name.

To pull the latest version of the file execute the following command.

```
suhas@test:~/code$ git pull origin master
From https://github.com/suhas-ds/myrepo
 * branch            master       -> FETCH_HEAD
Already up to date.
```

If you want to load files from another GitHub repository, or if you are working on another system and want to load files from your GitHub repository, you can achieve this by cloning it.

```
suhas@test:~/myrepo$ git clone https://github.com/suhas-ds/myrepo.git
Cloning into 'myrepo'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 0), reused 6 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
```

You can practice frequently used Git commands with more files and Git repositories.

Conclusion

In this chapter, you explored Git and GitHub, and you learned the basic Git commands, such as **git commit** and **git push**. Then, you installed and configured Git, and you pushed and cloned files from a remote GitHub repository. To use the GitHub platform, it is essential to know the Git workflow.

In the next chapter, you will learn about the challenges faced while deploying Machine Learning models in production and understand how to overcome them.

Points to remember

- Git identifies each commit uniquely using the SHA1 hash function, based on the contents of the committed files.
- Git is a command-line tool, whereas GitHub is a platform for collaboration.
- Git is used to maintain the historical and current versions of source code.

Multiple choice questions

1. The _____ command shows all commits in the current branch's history.
 - a) `git checkout`
 - b) `git push [alias] [branch]`
 - c) `git log`
 - d) `git pull`

2. Which of the following statements does not apply to git?
 - a) Tracks who created/made what changes and when
 - b) Rollback/restore changes
 - c) Allow multiple developers to coordinate and work on the same files
 - d) Does not maintain a copy of the files at a remote and local repository

Answers

1. c
2. d

Questions

1. What is Git? What is GitHub?
2. What is the command to upload changes to the remote repository?
3. How do you roll back/revert the changes?
4. How do you initialize a new repository?
5. What is the use of the Git status command?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 3

Challenges in ML Model Deployment

Introduction

A study found that 87% of Data Science and Machine Learning projects never make it into production. In this chapter, you will study common problems you may face while deploying machine learning models.

Deploying ML models is entirely based on your end goals, such as frequency of predictions, latency, number of users, single or batch predictions, and accessibility.

Structure

- This chapter covers the following topics:
- ML life cycle
- Types of model deployment
- Challenges while deploying models
- MLOps
- Benefits of MLOps

Objectives

This chapter will cover the various approaches to deploy ML models in production. After studying this chapter, you should be able to use MLOps to overcome challenges in the manual deployment of ML models. You will also study the different phases of the ML life cycle in this chapter.

ML life cycle

The machine learning life cycle is a periodic process. It starts with the business problem, and the last stage is monitoring and optimization. However, it is not so straightforward. For instance, if there is a modification in the business requirement, you may have to execute all the stages of the ML life cycle again. In some scenarios, you may have to go back to the previous stages of the ML life cycle to fulfill the criteria of that specific stage. For example, if you get lower accuracy of the model than the threshold, you need to revisit the previous stages to improve the accuracy. It can be done by adding new features, optimizing model parameters, and so on.

The following figure shows the different stages of the ML life cycle.

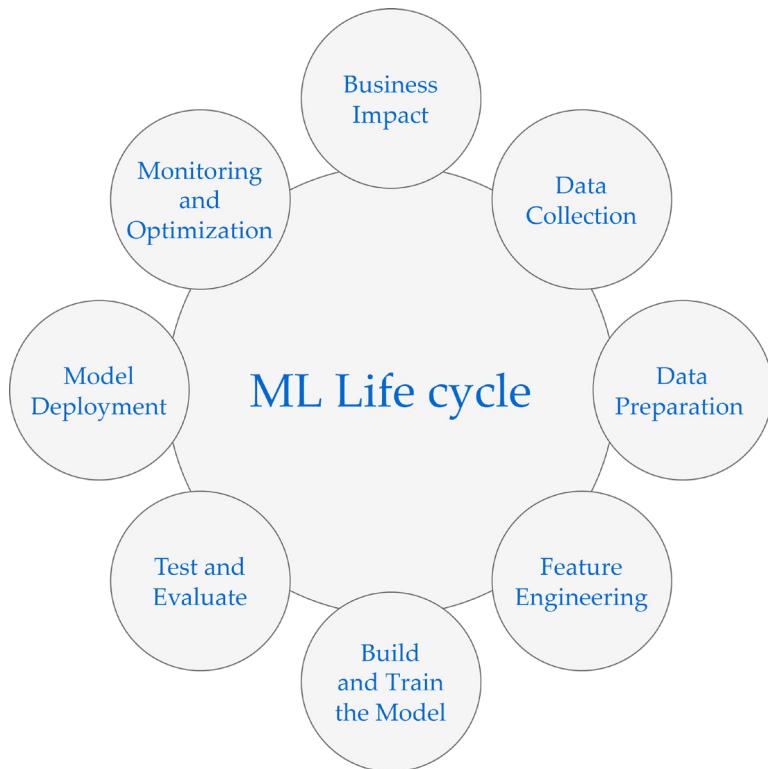


Figure 3.1: ML life cycle

Business impact

The first and most important stage is defining an idea and its impact on the project. Starting a project without considering the business impact, complexities, expected results, and time required may lead to delayed delivery, repetitive work, and poor resource utilization.

The business impact could be anything, like an increase in revenue, a decrease in expenses, or reducing human errors. One needs to understand the business pain points and try to assess the possibilities of having an ML solution that will solve multiple business problems. In some scenarios, getting quick results is more important.

Data collection

Data collection is the process of gathering data from one or multiple sources. Although it looks easy, it is not. Before collecting the data, you need to answer the following questions:

- What data needs to be collected?
- What are the different sources of the data?
- What is the type of data?
- What is the size of the data?

In the real world, you may have to collect data from different sources, like relational databases, NoSQL databases, the web, and so on. To avoid any glitches or delays, you must build the pipeline for data collection.

For better results, you can combine the available data with external data, such as social media sites, weather data, and public data.

Here are a few ways to collect data:

- E-surveys
- Authorized web scraping tool
- Click data
- Social media platforms
- Website tracking
- Subscription / registration data
- Image data - CCTV / camera

- Voice data - customer care

Set up a meeting with clients / stakeholders and domain experts to get to know the data. They will help you understand the data so that you can decide which data needs to be collected for model building.

Challenges:

- Data is scattered in different locations
- No uniformity in the data
- Combining data from different sources
- 3Vs: volume, velocity, variety
- Data storage

Data preparation

The data collected usually is in a raw format. One needs to process it so that it can be used for further analysis and model development. This process of cleaning, restructuring, and standardization is known as data preparation.

Around 70%-80% of the time goes into this stage of the ML project. This is a tedious task, but it is unavoidable. Data reduction technique is used when you have a large amount of data to be processed.

Data preparation aims to transform the raw data into a format so that EDA, that is, Exploratory Data Analysis, can be performed efficiently to gain insights.

Challenges:

- Missing values
- Outliers
- Disparate data format
- Data standardization
- Noise in the data

Feature engineering

In this stage, you prepare the input data that can be fed to the model, which makes it easier for the machine learning model by deriving meaningful features, data transformations, and so on.

Suppose you are combining the features to create a new one that could help ML models to understand the data better and identify hidden patterns. For instance, you mix sugar, water, and lemon juice to make lemonade rather than consuming them separately.

Here are a few feature engineering techniques:

- Label encoding
- Combining features to create a new one
- One hot encoding
- Imputation
- Scaling
- Removing unwanted features
- Log transformation

Challenges:

- Lack of domain knowledge
- Creating new features from the existing set of features
- Selecting useful features from a set of features

Build and train the model

ML model building requires the previous stages to be completed successfully. First, you should have the training, test, and validation sets ready (for supervised algorithms). After a baseline model is created, it can be compared with new models.

This process involves the development of multiple models to see which one is more efficient and gives better results. You must consider the computing resource requirements for this stage.

Once the final model is built on the training data, the next step is to check its performance against the unseen, that is, the test data.

Challenges:

- Model complexity
- Computational power
- Identifying a suitable model
- Model training time

Test and evaluate

Here, you will build the test cases and check how the model is performing against the new data. Pre-production or pre-deployment activities are done here. Performance results are analyzed and, if required, you have to go back to the previous stages to fix the issue.

After passing this stage, you can push the ML model into the production phase.

Challenges:

- Insufficient test data
- Reiterating the process until the output fulfills the requirements
- Identifying the platform to evaluate model performance on real data
- Deciding which test to use
- Logging and analyzing test results

Model deployment

This refers to exposing the trained ML models to real-world users. Your model is performing well on test and validation sets, but if another system or users cannot utilize it, then it does not meet its purpose.

For instance, you have built a model to predict customers who are likely to churn, but it is not done until you deploy a model that will start delivering the predictions on real data.

Model deployment is crucial because you have to consider the following factors:

- Number of times predictions to be delivered
- Latency of the predictions
- ML system architecture
- ML model deployment and maintenance cost
- Complexity of infrastructure
- This will be covered in detail in the following chapters.

Challenges:

- Portability issues
- Scalability issues
- Data-related challenges
- Security threats

Monitoring and optimization

Last but not least, monitoring and optimization is the stage where observing and tracking are required. You will have to check the model performance as it degrades over time.

If the model metrics, such as accuracy, go below the predefined threshold, then it needs to be tracked, and a model needs to be retrained, either automatically or manually. Similarly, input data needs to be monitored because it may happen that the input data schema does not match or it contains missing values.

Apart from this, the infrastructure metrics, such as RAM, free space, and system issues, need to be tracked.

It is good to maintain the log records of metrics, intermediate outputs, warnings, and errors.

Challenges:

- Data drift
- Deciding the threshold value for different metrics
- Anomalies
- Finalizing model evaluation metrics that need to be tracked

Types of model deployment

There are various ways to deploy ML models into production; however, there is no generic way to do it. This section will walk you through popular ways to deploy ML models.

Batch predictions

This is the simplest method. Here, the ML model is trained on static data to make predictions, which are saved in the database, such as MS-SQL, and can be integrated into existing applications or accessed by the business intelligence team.

Generally, ML model artifacts are used for making predictions as it saves time. Model artifact needs to be updated on new data for better predictions.

This method is well-suited for small organizations and beginners. You can schedule the cron job to make predictions after certain time intervals.

Pros:

- Affordable
- Less complex

- Easy to implement

Cons:

- Moderate latency
- Not suitable for ML-centric organizations

Web service/REST API

Web service/REST API is the popular method for deploying models. Unlike batch predictions, it does not process a bunch of records; it processes a single record at a time. In near real-time, it takes the parameters from users or existing applications and makes predictions.

It can take inputs as well as return the outputs in JSON format. JSON is a popular and compatible format that makes it easy for software or website developers to integrate it into existing applications.

When an event gets triggered, REST API passes the input parameters to the ML model and returns the predictions.

Pros:

- Easy to integrate
- Flexible
- Economical (pay-as-you-go plan)
- Near real-time predictions

Cons:

- Scalability issues
- Prone to security threats

Mobile and edge devices

When there are situations such as actions/ decisions that need to be taken immediately or there is no internet connectivity, the ML model needs to be deployed on these devices.

Edge devices include sensors, smartwatches, and cameras installed on robots.

This type of deployment is different from the preceding methods. In this, input data may not go to remote servers for making predictions. There are cloud service providers, such as Microsoft Azure, that offer the required infrastructure for this. **Tiny Machine Learning (TinyML)** is another such alternative capable of performing on-device sensor data analytics at an extremely low power.

ML models deployed on mobile devices are useful. Developing ML-based android / IOS applications, such as voice assistant or camera image-based attendance, are use cases of ML models deployed on mobiles.

Pros:

- Low power requirement
- Cost-effective
- Smaller sizes

Cons:

- Limited hardware resources
- A complex and tedious process

Real-time

- Here, analysis is to be done on streaming data. In this approach, the user passes input data for prediction and expects (near) real-time prediction. This is quite a challenging approach compared to the ones you studied previously.
- You can decrease the latency of the predictions by small-sized models, caching predictions, No-SQL databases, and so on.

Pros:

- Very low or no latency
- Can work with streaming data

Cons:

- Complex architecture requirements
- Computationally expensive

Challenges in deploying models in the production environment

Deploying machine models in production is crucial. While deploying ML models, technical challenges are not the only ones. The following are the common challenges you may come across.

Team coordination

If you are working in a small organization or a small data science team, you might need to play different roles and will be responsible for end-to-end model development.

However, in large organizations, there are separate teams for different stages.

These teams interact with each other to carry out the entire process. Most of the time, the production or the succeeding team is not aware of the preceding stages. Lack of coordination and miscommunication can lead to the failure of the ML life cycle.

Data-related challenges

Usually, data scientists develop models on a limited amount of data in the experimentation phase, but they might face challenges while deploying models in the production environment as large-scale data can impact the model performance.

New data keeps on changing its behavior, but the data preparation/pre-processing stage may not be able to handle these new issues. For instance, string records are present in a numeric column. At times, re-running the experiments by fixing the issues is required to get reliable results.

Portability

Portability issues arise when moving the codebase from your local machine to the server and vice versa. For instance, suppose you developed a model and want to move the code to the server to rerun the model. However, you may face issues, like the library not being compatible with the current OS version, and the line of code working with Python's x version and not with the currently installed version.

In such cases, creating a virtual environment and re-installing the packages are required.

Scalability

Suppose you deployed models using web service and 100x more users are trying to access the same API. REST API may stop responding or the latency may increase.

Managing ML models on a small scale is relatively easy compared to large organizations, where multiple models are being served on a scale. Deploying models on a scale is still a new thing for many organizations.

Robustness

In a nutshell, the robustness of ML models is measured by the reliable output delivered despite the variations in the input data. In the real world, delivering 100% accuracy is nearly impossible. So, one can say that prediction error on unseen data should be close to training error.

User behavior can be unpredictable. For instance, suppose a model requires a string value, but the user provides an alphanumeric value. In such scenarios, the model output might not be reliable.

Security

ML systems are prone to security breaches. They may be forced to deliver false predictions by deliberately providing poisonous data or adding noise to the data. The models perform better when trained on new data. However, a person or a group of people can deliberately pass the poisonous data to the model, intending to change the model predictions.

Data security is at risk as the model keeps training on new data. While training the model, attackers can steal sensitive information.

MLOps

MLOps combines machine learning processes and best practices of DevOps to deliver consistent output with automated pipelines and management. MLOps has been an emerging space in the industry for the last few years.

The following figure shows the increasing popularity of MLOps over time.

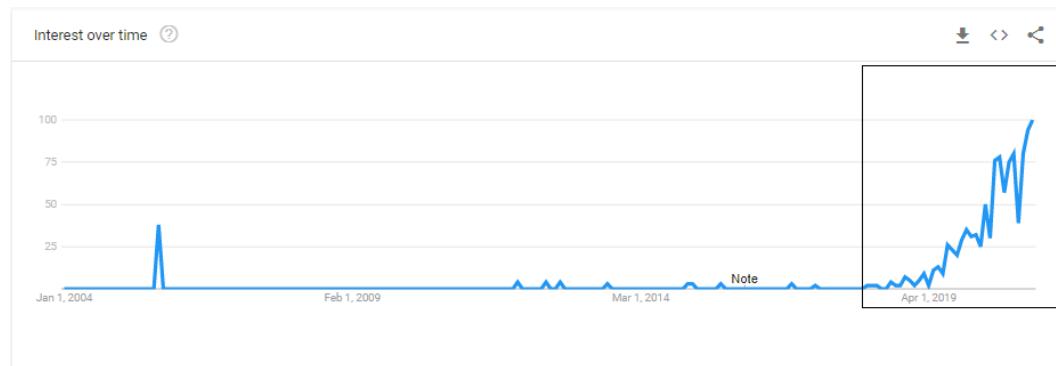


Figure 3.2: The rise of MLOps. Source: Google trends

MLOps bridges the gap between machine learning experiments and model deployment in the production environment. Nowadays, many data scientists are forced to execute the MLOps processes manually.

The MLOps process involves multiple professionals, and data scientists play a critical role. Subject matter experts understand and collect the requirements from the client. Then, data engineers collect the data from multiple sources and execute the ETL jobs. Once this is done, data scientists build the models and the DevOps

team builds the CI/CD pipeline and monitors it. Finally, feedback is sent to the data scientist or the concerned team for validation.

MLOps streamline and automate this process to speed up delivery and build efficient products/services.

MLOps is a combination of three disciplines as shown in the following figure

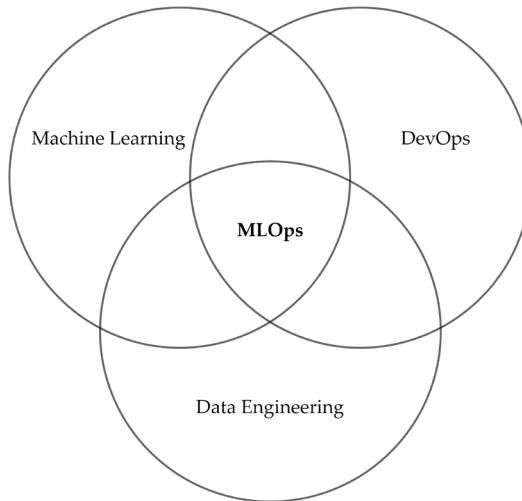


Figure 3.3: Machine learning operations

MLOps is different from DevOps because the code is usually static in the latter, but that's not the case in MLOps.

In MLOps, the model keeps training on new data, so a new model version gets generated recurrently. If it meets the requirements, it can be pushed into the production environment. This is why MLOps requires **Continuous Training (CT)** along with **Continuous Integration (CI)** and **Continuous Delivery/Deployment (CD)**.

In DevOps, developers write the code as per the requirements and then release it to the production environment, but in the case of machine learning, developers first need to collect the data and clean it. They write code for model building and then build the ML model. Finally, they release it into the production environment.

Benefits of MLOps

MLOps processes not only speed up the ML journey from experiments to production, but also reduce the number of errors. MLOps automates Machine Learning and Deep Learning model deployment in a production environment. Moreover, it reduces dependency on other teams by streamlining the processes.

Efficient management of ML life cycle

MLOps takes care of the ML life cycle by following the fundamental principles of machine learning projects. It is based on agile methodology. An automated CI/CD pipeline helps speed up the process of retraining models on new data, testing before the deployment, monitoring, and feedback loops. The current stage depends on the output of the preceding stages, so there are fewer chances of issues in the deployment process.

If the amount of traffic or number of requests increase for a deployed model, it increases the required resources; similarly, the required resources decrease when the amount of traffic or number of requests reduces.

There are many platforms (like GitHub Actions) available in the market that help you set up MLOps workflow.

Reproducibility

Developer: It works on my machine.

Manager: Then, we are going to ship your machine to the client.

- Reproducibility plays a crucial role in the ML life cycle. While transferring the code files to another machine or server, reproducibility reduces debugging time. MLOps works on **DRY (Don't Repeat Yourself)** principles and allows you to get consistent output.
- Given the same input, the replicated workflow should produce an identical output. For this, developers use container orchestration tools like Docker so that it will create and set up the same environment with dependencies in another machine or server for consistent output.

Automation

Mostly, developers deploy models several times before the final deployment to ensure that everything is in place and the output is as expected. Without automation, this would be a time-consuming and tedious process. Automation increases productivity, as you are less likely to test, deploy, scale, and monitor ML models manually.

At every stage, certain rules and conditions are implemented, and the model moves to the next stage only when these conditions are met. This reduces the active involvement of other teams every time you are planning to deploy it in production.

There is a low or no delayed deployment with testing since one inter-team dependency is reduced.

After making the required changes in the code, speedy deployment is done in an automated fashion (if it passes the test cases and satisfies the conditions). This increases the overall productivity of the team.

Tracking and feedback loop

Tracking model performance, metrics, test results, and output becomes easy if you set up MLOps workflow properly. The model's performance may degrade over time, hence one may need to retrain the model on new data.

Thanks to tracking and feedback loops, it sends alerts about the model's performance, metrics, and so on.

For instance, if the feedback loop sends the information that model accuracy has dropped below 68%, then the model needs to be retrained. Again, it will check the model's performance. If it is above the threshold level, it will pass on to the next stage; else, the model needs to be recalibrated using the latest data.

Conclusion

In this chapter, you studied the key challenges faced while deploying ML models in production. At the beginning of this chapter, you learned how the ML life cycle works, and you understood its different stages and their challenges. Next, you got exposure to different types of model serving techniques and looked at their pros and cons. Then, you analyzed the challenges you may face while deploying ML models in production. Finally, you learned the benefits of MLOps.

In the next chapter, you will learn to develop, build and install custom python packages for ML models using a use case.

Points to remember

- MLOps bridges the gap between the Machine Learning experiments stage and its model deployment in the production environment.
- Model deployment strategy depends on business requirements, users, and applications.
- MLOps combines machine learning processes and best practices of DevOps to deliver consistent output with automated pipelines and management.

Multiple choice questions

1. What are the different ways of collecting data for an ML problem?
 - a) E-surveys
 - b) Web scraping
 - c) Click data
 - d) All the above
2. If you provide a similar input as the original workflow to the replicated workflow, it should produce an identical output known as:
 - a) Portability
 - b) Reproducibility
 - c) Scalability
 - d) Label encoding

Answers

1. d
2. b

Questions

1. What are the different ways of deploying ML models?
2. How do MLOps differ from DevOps?
3. What are the challenges faced while deploying ML models?
4. What are the benefits of MLOps?

Key terms

- **Container:** A container is a standard unit of software that packages up the code and all its dependencies so that the application runs quickly and reliably despite the computing environment.
- **Continuous Integration (CI):** It is an automated process to build, test, and execute different pieces of code.
- **Continuous Delivery/Deployment (CD):** It is an automated process of frequently building tested code and deploying it to production.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 4

Packaging ML Models

Introduction

In this chapter, you will learn how to modularize Python code and build ML packages that can be installed and consumed on another machine or server. Modular programs simplify code debugging, reusability, and maintainability. You will start by creating a virtual environment and then data pre-processing, followed by building the ML model and finally, creating test cases for the package.

Structure

This chapter covers the following topics:

- Virtual environment
- Requirements file
- Serializing and de-serializing ML models
- Testing Python code using pytest
- Python packaging and dependency management
- Developing, building, and deploying ML packages
- Set up environment variables and paths

Objectives

After completing this chapter, you should be able to create a virtual environment and install the required dependencies in it. You should also be able to package the modules and dependencies required to build an ML model that can be installed on local machines or remote servers. Additionally, you should be able to save the trained ML model by pickling it. This chapter will also help you learn to modularize code and build test cases using pytest to check integrity and functionality.

Virtual environments

While working on multiple projects, **ProjectA** requires **PackageY_version2.6**, whereas **ProjectB** requires **PackageY_version2.8**. In such scenarios, you can't keep both versions of the same package globally.

The virtual environment is the solution. It allows you to isolate dependencies for each project. You can create the virtual environment anywhere and install the required packages in it.

Without the virtual environment, packages are installed globally at the default Python location:

```
/home/suhas/.local/lib/python3.6/site-packages
```

With the virtual environment, packages are installed inside the virtual environment's Python location:

```
/home/suhas/code/packages/venv_package/lib/python3.6/site-packages
```

Installing different versions of the same package in different virtual environments for different projects is possible. You can have five packages in **venv1** and nine packages in **venv2**.

The following figure best depicts the scenario wherein three different projects have separate virtual environments with different versions of Python and packages installed in them.

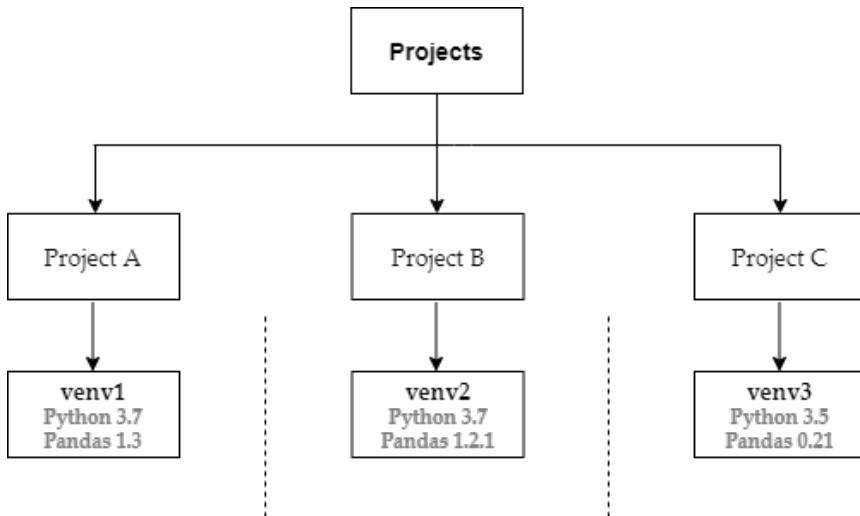


Figure 4.1: Virtual environment

Virtual environment installation:

```
pip install virtualenv
```

Creating a virtual environment:

```
virtualenv venv_package
```

Activating a virtual environment:

```
source venv_package/bin/activate  
(venv_package) suhas@suhasVM:~/code/packages/prediction_model$
```

Deactivating a virtual environment:

```
deactivate
```

To see the list of packages installed in the virtual environment:

```
pip list
```

Or

```
pip freeze
```

Requirements file

The requirements file holds the list of packages that can be installed using pip.

To create the requirements file:

```
pip freeze > requirements.txt
```

To install the list of packages from the requirements file, you can use the following command:

```
pip install -r requirements.txt
```

Where `-r` refers to `--requirement`.

It instructs pip to install all the packages from the given requirements file.

Serializing and de-serializing ML models

Serializing is a process through which a Python object hierarchy is converted into a byte stream, whereas deserialize is the inverse operation, that is, a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. In Python, serialization and deserialization refer to pickling and unpickling, respectively.

Here, `joblib.dump()` and `joblib.load()` will be used as a replacement for a `pickle.dump()` and `pickle.load` respectively, to work efficiently on arbitrary Python objects containing large data, large NumPy arrays in particular, as shown here.

Import the `joblib` library:

```
1. import joblib
```

Then create an object to be persisted:

```
2. joblib.dump(ML_model_object, filename)
```

An object can be reloaded:

```
3. joblib.load(filename)
```

Note: If you are switching between Python versions, you may need to save a different joblib dump for each Python version.

Testing Python code with pytest

Testing your code assures you that it is giving the expected results and its functions are working bug-free. pytest tool allows you to build and run tests with ease. It comes with simple syntax and several features.

`pytest` can be installed using `pip`:

```
pip install pytest
```

To run the **pytest**, simply switch to the package directory and run the following command:

```
pytest
```

It searches for *test_*.py* or **_test.py* files.

The **pytest** output can be any of the following:

- A dot (.) means that the test has passed.
- An F means that the test has failed.
- An E means that the test raised an unexpected exception.

```
pytest -v
```

-v or **--verbose** flag allows you to see whether individual tests are passing or failing.

pytest fixtures

pytest fixtures are basic functions, and they run before the test functions are executed. pytest fixtures are useful when you are running multiple test cases with the same function return value.

It can be declared by the **@pytest.fixture** marker. The following is an example:

```
1. @pytest.fixture
2. def xyz_func():
3.     return "ABC"
```

When pytest runs a test case, it looks at the parameters in that test function's signature and then searches for fixtures that have the same names as those parameters. Once pytest finds them, it runs those fixtures, captures what they returned (if any), and passes those objects into the test functions as arguments.

You can configure **pytest** using the *pytest.ini* file.

Python packaging and dependency management

Suppose you have created the final ML model in a Python notebook. This Python notebook holds all the steps right from loading the data to predicting the test data.

However, this notebook should not be used in the production environment for the following reasons:

- Difficult to debug
- Require changes at multiple locations
- Lots of dependencies
- No modularity in the code
- Conflict of variables and functions
- Duplicate code snippets

Modular programming

Python is a modular programming language. Modular programming is a design approach in which code gets divided into separate files, such that each file contains everything necessary to execute a defined piece of logic and can return the expected output when imported by other files that will act as input for them. These separate files are called modules.

Module

A module is a Python file that can hold classes, functions, and variables. For instance, `load.py` is a module, and its name is **load**.

Package

A package contains one or more (relevant) modules, such that they are interlinked with each other. A package can contain a subpackage that holds the modules. It uses the inbuilt file hierarchy of directories for ease of access. A directory with subdirectories can be called a package if it contains the `__init__.py` file.

Packages will be installed in the production environment as part of the deployment, so before you package anything, you'll want to have answers to the following deployment questions:

- Who are your app's users? Will your app be installed by other developers doing software development, operations people in a data center, or a less software-savvy group?
- Is your app intended to run on servers, desktops, mobile clients (phones, tablets, and so on), or embedded in dedicated devices?
- Is your app installed individually, or in large deployment batches?

The following figure depicts the sample structure of the Python package, where **f1**, **f2**, **f3**, **f4**, and **f5** are modules of the package.

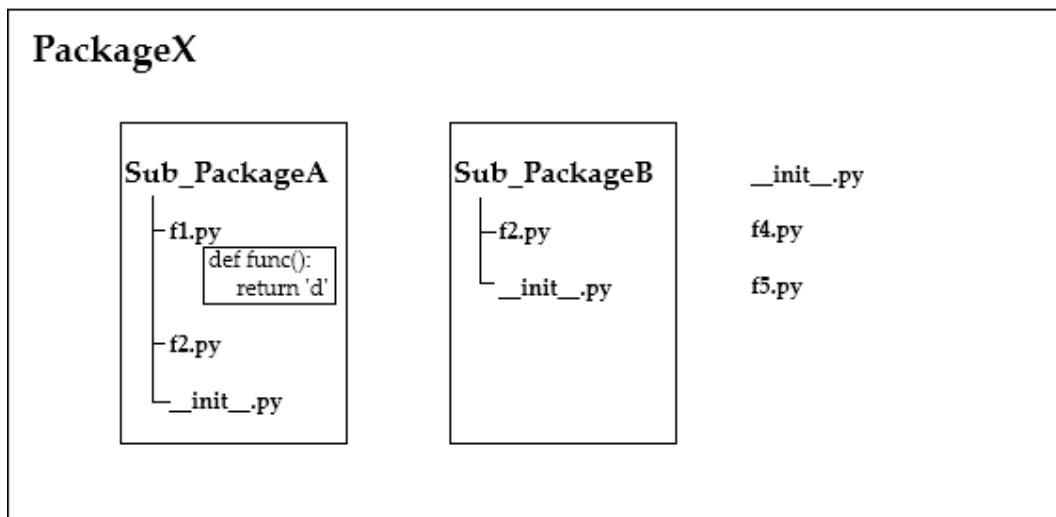


Figure 4.2: Python package structure

Import module syntax:

```
import <module_name>
import <module_name> as <alt_name>
```

Import **f4** module:

1. from My_Package import f4
2. from My_Package import f4 as load

Import **f1** module to use **x()**:

1. from My_Package.Sub_PackageA import f1
2. f1.func()

Or you can use the following method to import the module:

1. from My_Package.Sub_PackageA.f1 import func
2. func()

Developing, building, and deploying ML packages

In this section, you will study a use case that will begin by defining the business problem and end with installing and consuming an ML package. You will learn to

develop and build a custom Python package for ML models. The custom package is portable and can be reused for the given project. You will be able to install a custom package like any other Python package and make predictions on the new data.

Business problem

A company wants to automate the loan eligibility process based on customer details provided by filling out online application forms. It is a classification problem where you must predict whether a loan would be approved.

Data

The data corresponds to a set of financial transactions associated with individuals. The data has been standardized, de-trended, and anonymized. Refer to the following table for details of the data columns:

Variable	Description
Loan_ID	Unique Loan ID
Gender	Male / Female
Married	Applicant's marital status(Y/N)
Dependents	Number of dependents
Education	Applicant's Education (Graduate/Under Graduate)
Self_Employed	Self-employed (Y/N)
ApplicantIncome	Applicant's income
CoapplicantIncome	Co-applicant's income
LoanAmount	Loan amounts in thousands
Loan_Amount_Term	Term of the loan in months
Credit_History	Credit history meets guidelines (Y/N)
Property_Area	Urban/ Semi Urban/ Rural
Loan_Status	Loan approved (Y/N)

Table 4.1: Data description

Building the ML model

At this point, you may build and compare different ML algorithms using a Jupyter notebook.

Identify the model that is performing better than the others and finalize the best parameters (and hyperparameters) for the model. In this experimentation stage, you should be ready with information like the variables needed to transform, missing value imputation, the list of numerical variables, and the data path.

When you are done with the experimentation stage, you can start building the package. The purpose of packaging is reusability, portability, fewer errors, and automation of machine learning processes.

Start by creating a separate directory for building the package. Maintain separate modules (Python files) for different stages and operations, for instance, separate modules for pre-processing of the data. This will modularize the code and make debugging the code easy.

Build the test cases that will run the code files and verify the integrity of the files and the expected output.

This chapter does not focus on optimizing ML models, so you can further improve the model performance by fine-tuning it. This package would be limited to the current project, that is, you should not use the same package for different projects.

Developing the package

Following is the directory structure of the package:

```
prediction_model
├── MANIFEST.in
└── prediction_model
    ├── config
    │   ├── config.py
    │   └── __init__.py
    ├── datasets
    │   ├── __init__.py
    │   ├── test.csv
    │   └── train.csv
    ├── __init__.py
    ├── pipeline.py
    └── predict.py
```

```
|   ├── processing
|   |   ├── data_management.py
|   |   ├── __init__.py
|   |   └── preprocessors.py
|   ├── trained_models
|   |   ├── classification_v1.pkl
|   |   └── __init__.py
|   ├── train_pipeline.py
|   └── VERSION
├── README.md
├── requirements.txt
└── setup.py
└── tests
    ├── pytest.ini
    └── test_predict.py
```

__init__.py

The `__init__.py` module is usually empty. However, it facilitates importing other Python modules.

This file indicates that the directory should be treated as a package.

MANIFEST.in

A `MANIFEST.in` file consists of commands, one per line, instructing `setuptools` to add or remove a set of files from the `sdist` (source distribution). In Python, distribution refers to the set of files that allows packaging, building, and distributing the modules. `sdist` contains archives of files, such as source files, data files, and `setup.py` file, in a compressed tar file (`.tar.gz`) on Unix.

The following table lists the commands and the descriptions. However, this chapter will only cover a few of them.

You can update `MANIFEST.in` as per the project requirements. Refer to the following table for general commands being used in the manifest file.

Command	Description
<code>include pat1 pat2 ...</code>	Add all files matching any of the listed patterns.
<code>exclude pat1 pat2 ...</code>	Remove all files matching any of the listed patterns.

recursive-include dir-pattern pat1 pat2 ...	Add all files under directories matching the dir-pattern that matches any of the listed patterns.
recursive-exclude dir-pattern pat1 pat2 ...	Remove all files under directories matching the dir-pattern that matches any of the listed patterns.
global-include pat1 pat2 ...	Add all files anywhere in the source tree matching any of the listed patterns.
global-exclude pat1 pat2 ...	Remove all files anywhere in the source tree matching any of the listed patterns.
graft dir-pattern	Add all files under directories matching the dir-pattern .
prune dir-pattern	Remove all files under directories matching the dir-pattern .

Table 4.2: Commands for the manifest file

Let's add the commands to the manifest file to include or exclude files:

```

1. include *.txt
2. include *.md
3. include *.cfg
4. include *.pkl
5. recursive-include ./prediction_model/*
6.
7. include prediction_model/datasets/train.csv
8. include prediction_model/datasets/test.csv
9. include prediction_model/trained_models/*.pkl
10. include prediction_model/VERSION
11. include ./requirements.txt
12. exclude *.log
13. recursive-exclude * __pycache__
14. recursive-exclude * *.py[co]

```

config.py

A configuration module contains constant variables, the path to the directories, and initial settings. Variables and functions can be accessed by importing this module into other modules. For instance, the **TARGET** variable holds the dependent column name **Loan_Status**.

Here, specify the path for the package's root directory, data directory, train file name, test file name, features, and path of other files and directories.

```
1. #Import Libraries
2. import pathlib
3. import os
4. import prediction_model
5.
6. PACKAGE_ROOT = pathlib.Path(prediction_model.__file__).resolve().parent
7.
8. DATAPATH=os.path.join(PACKAGE_ROOT, 'datasets')
9. SAVED_MODEL_PATH=os.path.join(PACKAGE_ROOT, 'trained_models')
10.
11. TRAIN_FILE='train.csv'
12. TEST_FILE='test.csv'
13.
14. TARGET='Loan_Status'
15.
16. #Features to keep
17. FEATURES=['Gender','Married','Dependents',
18.             'Education','Self_Employed','ApplicantIncome',
19.             'CoapplicantIncome','LoanAmount','Loan_Amount_Term',
20.             'Credit_History','Property_Area'] # Final feature to
keep in data
21.
22. NUMERICAL_FEATURES=['ApplicantIncome', 'LoanAmount', 'Loan_
Amount_Term'] d
23.
24. CATEGORICAL_FEATURES=['Gender','Married','Dependents',
25.                         'Education','Self_Employed','Credit_
History',
26.                         'Property_Area'] #Categorical
27.
28. FEATURES_TO_ENCODE=['Gender','Married','Dependents',
29.                       'Education','Self_Employed','Credit_History',
```

```
30.          'Property_Area' ] #Features to Encode  
31.  
32. TEMPORAL_FEATURES=['ApplicantIncome']  
33. TEMPORAL_ADDITION='CoapplicantIncome'  
34. LOG_FEATURES=['ApplicantIncome', 'LoanAmount'] #Features for Log Transformation  
35. DROP_FEATURES=['CoapplicantIncome'] #Features to Drop
```

data_management.py

This module contains functions required for loading the data, saving serialized ML model, and loading deserialized ML model using joblib.

```
1. #Import Libraries  
2. import os  
3. import pandas as pd  
4. import joblib  
5.  
6. #Import other files/modules  
7. from prediction_model.config import config  
8.  
9. def load_dataset(file_name):  
10.     """Read Data"""  
11.     file_path = os.path.join(config.DATAPATH,file_name)  
12.     _data = pd.read_csv(file_path)  
13.     return _data  
14.  
15. def save_pipeline(pipeline_to_save):  
16.     """Store Output Of Pipeline  
17.     Exporting pickle file of trained Model """  
18.     save_file_name = 'classification_v1.pkl'  
19.     save_path = os.path.join(config.SAVED_MODEL_PATH, save_file_name)
```

```
20.     joblib.dump(pipeline_to_save, save_path)
21.     print("Saved Pipeline : ", save_file_name)
22.
23. def load_pipeline(pipeline_to_load):
24.     """Importing pickle file of trained Model"""
25.     save_path = os.path.join(config.SAVED_MODEL_PATH, pipeline_to_
load)
26.     trained_model = joblib.load(save_path)
27.     return trained_model
```

preprocessors.py

This module holds all the fit and transform functions required by the sklearn pipeline:

```
1. #Import Libraries
2. import pandas as pd
3. import numpy as np
4. from sklearn.base import BaseEstimator, TransformerMixin
5. from sklearn.preprocessing import LabelEncoder
6.
7. #Import other files/modules
8. from prediction_model.config import config
```

The mean of the feature is used for imputing missing numerical values. However, you can use other missing value imputation techniques, such as stochastic regression imputation, interpolation, and cold deck imputation.

```
1. #Numerical Imputer
2. class NumericalImputer(BaseEstimator, TransformerMixin):
3.     """Numerical Data Missing Value Imputer"""
4.     def __init__(self, variables=None):
5.         self.variables = variables
6.
7.     def fit(self, X,y=None):
8.         self.imputer_dict_={}
```

```
9.         for feature in self.variables:
10.             self.imputer_dict_[feature] = X[feature].mean()
11.         return self
12.
13.     def transform(self,X):
14.         X=X.copy()
15.         for feature in self.variables:
16.             X[feature].fillna(self.imputer_dict_[feature],inplace=True)
17.         return X
```

The most frequent value of the feature is used for imputing missing categorical values, but you can also use other missing value imputation techniques, such as missing indicator imputation, **Multivariate Imputation by Chained Equations (MICE)** algorithm, and systematic random sampling imputation.

```
1. #Categorical Imputer
2. class CategoricalImputer(BaseEstimator,TransformerMixin):
3.     """Categorical Data Missing Value Imputer"""
4.     def __init__(self, variables=None):
5.         self.variables = variables
6.
7.     def fit(self, X,y=None):
8.         self.imputer_dict_={}
9.         for feature in self.variables:
10.             self.imputer_dict_[feature] = X[feature].mode()[0]
11.         return self
12.
13.     def transform(self, X):
14.         X=X.copy()
15.         for feature in self.variables:
16.             X[feature].fillna(self.imputer_dict_[feature],inplace=True)
17.         return X
```

Post that, categorical features are encoded for model training.

```
1. #Categorical Encoder
2. class CategoricalEncoder(BaseEstimator,TransformerMixin):
3.     """Categorical Data Encoder"""
4.     def __init__(self, variables=None):
5.         self.variables=variables
6.
7.     def fit(self, X,y):
8.         self.encoder_dict_ = {}
9.         for var in self.variables:
10.             t = X[var].value_counts().sort_values(ascending=True).
index
11.             self.encoder_dict_[var] = {k:i for i,k in enumerate(t,0)}
12.         return self
13.
14.     def transform(self,X):
15.         X=X.copy()
16.         #This part assumes that the encoder does not introduce a NANS
17.         #In that case, a check needs to be done and the code
should break
18.         for feature in self.variables:
19.             X[feature] = X[feature].map(self.encoder_dict_
[feature])
20.         return X
```

The following code snippet is used for creating new features:

```
1. #Temporal Variables
2. class TemporalVariableEstimator(BaseEstimator,TransformerMixin):
3.     """Feature Engineering"""
4.     def __init__(self, variables=None, reference_variable = None):
```

```
5.         self.variables=variables
6.         self.reference_variable = reference_variable
7.
8.     def fit(self, X,y=None):
9.         #No need to put anything, needed for Sklearn Pipeline
10.        return self
11.
12.    def transform(self, X):
13.        X=X.copy()
14.        for var in self.variables:
15.            X[var] = X[var]+X[self.reference_variable]
16.        return X
```

Apply log transformation on variables for making patterns in the data more interpretable.

```
1. # Log Transformations
2. class LogTransformation(BaseEstimator, TransformerMixin):
3.     """Transforming variables using Log Transformations"""
4.     def __init__(self, variables=None):
5.         self.variables = variables
6.
7.     def fit(self, X,y):
8.         return self
9.
10.    #Need to check in advance if the features are <= 0
11.    #If yes, needs to be transformed properly (E.g., np.log1p(X[var]))
12.    def transform(self,X):
13.        X=X.copy()
14.        for var in self.variables:
15.            X[var] = np.log(X[var])
16.        return X
```

The following code snippet is used for dropping features that are insignificant to the model:

```
1. # Drop Features  
2. class DropFeatures(BaseEstimator, TransformerMixin):  
3.     """Dropping Features Which Are Less Significant"""  
4.     def __init__(self, variables_to_drop=None):  
5.         self.variables_to_drop = variables_to_drop  
6.  
7.     def fit(self, X,y=None):  
8.         return self  
9.  
10.    def transform(self, X):  
11.        X=X.copy()  
12.        X=X.drop(self.variables_to_drop, axis=1)  
13.        return X
```

pipeline.py

In this module, sklearn-pipeline is used. The model can be deployed without a sklearn-pipeline, but it is recommended to build a sklearn-pipeline. The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters.

```
1. #Import Libraries  
2. from sklearn.pipeline import Pipeline  
3. from sklearn.preprocessing import MinMaxScaler  
4. from sklearn.linear_model import LogisticRegression  
5.  
6. #Import other files/modules  
7. from prediction_model.config import config  
8. import prediction_model.processing.preprocessors as pp  
9.
```

```
10. loan_pipe=Pipeline([
11.     ('Numerical Imputer',pp.NumericalImputer(variables=config.
    NUMERICAL_FEATURES)),
12.     ('Categorical Imputer',pp.CategoricalImputer(variables=config.
    CATEGORICAL_FEATURES)),
13.     ('Temporal Features',
    pp.TemporalVariableEstimator(variables=config.TEMPORAL_FEATURES,
        reference_variable=config.TEMPORAL_ADDITION)),
14.     ('Categorical Encoder',
    pp.CategoricalEncoder(variables=config.FEATURES_TO_ENCODE)),
15.     ('Log Transform', pp.LogTransformation(variables=config.LOG_
    FEATURES)),
16.     ('Drop Features', pp.DropFeatures(variables_to_drop=config.
    DROP_FEATURES)),
17.     ('Scaler Transform', MinMaxScaler())),
18.     ('Linear Model', LogisticRegression(random_state=1))
20. ])
```

predict.py

A *predict.py* module loads the saved ML model (.pkl) and makes predictions on the new data.

```
1. #Import Libraries
2. import pandas as pd
3. import numpy as np
4. import joblib
5.
6. #Import other files/modules
7. from prediction_model.config import config
8. from prediction_model.processing.data_management import load_
pipeline
9.
10. pipeline_file_name = 'classification_v1.pkl'
```

```
11.  
12. _loan_pipe = load_pipeline(pipeline_file_name)  
13.  
14. def make_prediction(input_data):  
15.     """Predicting the output"""  
16.  
17.     # Read Data  
18.     data = pd.DataFrame(input_data)  
19.  
20.     # prediction  
21.     prediction = _loan_pipe.predict(data[config.FEATURES])  
22.     output = np.where(prediction==1, 'Y', 'N').tolist()  
23.     results = {'prediction': output}  
24.     return results
```

requirements.txt

The following libraries are included in this file:

```
1. # Model Building Requirements  
2. joblib==0.16.0  
3. numpy==1.19.0  
4. pandas==1.0.5  
5. scikit-learn==0.23.1  
6. scipy==1.5.1  
7. sklearn==0.0  
8.  
9. # testing requirements  
10. pytest<5.0.0,>=4.6.6  
11.  
12. # packaging
```

```
13. setuptools==40.6.3
```

```
14. wheel==0.32.3
```

classification_v1.pkl

This is a pickled file from the trained model.

train_pipeline.py

This module loads the training data and passes it to the pipeline, then saves the pickle file of the model to the local directory.

```
1. #Import Libraries
```

```
2. import pandas as pd
```

```
3. import numpy as np
```

```
4.
```

```
5. #Import other files/modules
```

```
6. from prediction_model.config import config
```

```
7. from prediction_model.processing.data_management import load_
dataset, save_pipeline
```

```
8. import prediction_model.processing.preprocessors as pp
```

```
9. import prediction_model.pipeline as pl
```

```
10. from prediction_model.predict import make_prediction
```

```
11.
```

```
12. def run_training():
```

```
13.     """Train the model"""
```

```
14.     #Read Data
```

```
15.     train = load_dataset(config.TRAIN_FILE)
```

```
16.
```

```
17.     #separating Loan_status in y
```

```
18.     y = train[config.TARGET].map({'N':0 , 'Y':1})
```

```
19.     pl.loan_pipe.fit(train[config.FEATURES],y)
```

```
20.     save_pipeline(pipeline_to_save=pl.loan_pipe)
```

```
21. if __name__=='__main__':
22.     run_training()
```

setup.py

To configure and install packages from the source directory, create a *setup.py* file. It is specific to the package. PIP will use the *setup.py* file to install packages. Go to the directory where the *setup.py* file is located and install the packages using the **pip install .** (period) command.

```
1. import io
2. import os
3. from pathlib import Path
4.
5. from setuptools import find_packages, setup
6.
7. # Package meta-data
8. NAME = 'prediction_model'
9. DESCRIPTION = 'Train and deploy prediction model.'
10. URL = 'https://github.com/suhas-ds/prediction_model'
11. EMAIL = 'suhasp.ds@gmail.com'
12. AUTHOR = 'Suhas Pote'
13. REQUIRES_PYTHON = '3.6'
14.
15. here = os.path.abspath(os.path.dirname(__file__))
16.
17. # What packages are required for this module to be executed?
18. def list_reqs(fname='requirements.txt'):
19.     with io.open(os.path.join(here, fname), encoding='utf-8') as fd:
20.         return fd.read().splitlines()
21. try:
```

```
22.     with io.open(os.path.join(here, 'README.md'),
encoding='utf-8') as f:
23.         long_description = '\n' + f.read()
24. except FileNotFoundError:
25.     long_description = DESCRIPTION
26.
27. # Load the package's __version__.py module as a dictionary.
28. ROOT_DIR = Path(__file__).resolve().parent
29. PACKAGE_DIR = ROOT_DIR / NAME
30. about = {}
31. with open(PACKAGE_DIR / 'VERSION') as f:
32.     _version = f.read().strip()
33.     about['__version__'] = _version
34.
35. setup(
36.     name=NAME,
37.     version=about['__version__'],
38.     description=DESCRIPTION,
39.     long_description=long_description,
40.     long_description_content_type='text/markdown',
41.     author=AUTHOR,
42.     author_email=EMAIL,
43.     python_requires=REQUIRES_PYTHON,
44.     url=URL,
45.     packages=find_packages(exclude=('tests',)),
46.     package_data={'prediction_model': ['VERSION']},
47.     install_requires=list_reqs(),
48.     extras_require={},
```

```
49.     include_package_data=True,
50.     license='MIT',
51.     classifiers=[
52.         'License :: OSI Approved :: MIT License',
53.         'Programming Language :: Python',
54.         'Programming Language :: Python :: 3',
55.         'Programming Language :: Python :: 3.6',
56.         'Programming Language :: Python :: Implementation :: CPython',
57.         'Programming Language :: Python :: Implementation :: PyPy'
58.     ]
59. )
```

VERSION

This file holds the version of the package. Here, a **major.minor.micro** scheme is used.

```
1. 0.1.0
```

pytest.ini

To disable warnings while running pytest, you have to configure the *pytest.ini* file.

```
1. [pytest]
2. addopts = -p no:warnings
```

test_predict.py

It fetches a single record from the validation data and verifies the output using **assert** statements.

It validates the following checks:

- The output is not null.
- The output data type is **str**.
- The output is **Y** for given data (fixed).

These checks are run using pytest.

```
1. #Import Libraries
2. import pytest
3. #Import files/modules
4. from prediction_model.config import config
5. from prediction_model.processing.data_management import load_
dataset
6. from prediction_model.predict import make_prediction
7.
8. @pytest.fixture
9. def single_prediction():
10.     ''' This function will predict the result for a single record'''
11.     test_data = load_dataset(file_name=config.TEST_FILE)
12.     single_test = test_data[0:1]
13.     result = make_prediction(single_test)
14.     return result
15.
16. #Test Prediction
17. def test_single_prediction_not_none(single_prediction):
18.     ''' This function will check if the result of prediction is not
None'''
19.     assert single_prediction is not None
20. def test_single_prediction_dtype(single_prediction):
21.     ''' This function will check if the data type of the result of the
prediction is str i.e. string '''
22.     assert isinstance(single_prediction.get('prediction')[0], str)
23. def test_single_prediction_output(single_prediction):
24.     ''' This function will check if the result of the prediction is Y '''
25.     assert single_prediction.get('prediction')[0] == 'Y'
```

Set the environment variable **PYTHONPATH** and go to the *package* directory, and then run the following command:

```
pytest -v
```

If everything goes well, you should get an output as follows:

```
===== test session starts =====
platform linux -- Python 3.6.9, pytest-4.6.11, py-1.10.0, pluggy-0.13.1 -- /home/suhas/code/venv_package1/bin/python
cachedir: .pytest_cache
rootdir: /home/suhas/packages/prediction_model
collected 3 items

tests/test_predict.py::test_single_prediction_not_none PASSED [ 33%]
tests/test_predict.py::test_single_prediction_dtype PASSED [ 66%]
tests/test_predict.py::test_single_prediction_output PASSED [100%]

===== 3 passed in 0.87 seconds =====
```

Figure 4.3: pytest output

sdist

Python's sdist are compressed archives (**.tar.gz** files) containing one or more packages or modules.

This creates a *dist* directory containing a compressed archive of the package (for example, **<PACKAGE_NAME>-<VERSION>.tar.gz** in Linux).

wheel

This is the binary distribution or bdist, and it supports Windows, Mac, and Linux.

Wheels will speed up the installation if you have compiled code extensions, as the build step is not required. A wheel distribution is a built distribution for the current platform. The installable wheel will be created under the *dist* directory, and a *build* directory will also be created with the built code.

Set up environment variables and paths

You may need to add the path to the environment variables. It allows you to import modules and functions:

Open the *.bashrc* file using terminal

```
sudo nano ~/.bashrc
```

Add the path to the package directory. Here's an example:

```
PYTHONPATH=/home/suhas/code/packages/prediction_model:$PYTHONPATH
export PYTHONPATH
```

Then run `source ~/.bashrc`.

Reopen the terminal and test using the following:

```
echo $PYTHONPATH
```

Build the package

1. Go to the project directory and install dependencies:

```
pip install -r requirements.txt
```

2. Create a pickle file:

```
python prediction_model/train_pipeline.py
```

3. Creating a source distribution and wheel:

```
python setup.py sdist bdist_wheel
```

Install the package

Go to the project directory where the `setup.py` file is located and install this project with the `pip` command:

To install the package in editable or developer mode:

```
pip install -e .
```

- `.` refers to the current directory.
- `-e` refers to --editable mode.

Normal installation of the package:

```
pip install .
```

- `.` refers to the current directory.

You can push the entire package to the GitHub repository.

To install it from the GitHub repository.

With git:

```
pip install git+https://github.com/suhas-ds/prediction_model.git
```

Without git:

```
pip install https://github.com/suhas-ds/prediction_model/tarball/master
```

Or:

```
pip install https://github.com/suhas-ds/prediction_model/zipball/master
```

Or:

```
pip install https://github.com/suhas-ds/prediction_model/archive/master.zip
```

After building the package, your directory structure should look like this:

`prediction_model`

```
|── build  
|   ├── bdist.linux-x86_64  
|   └── lib  
|       ├── prediction_model  
└── dist  
    ├── prediction_model-0.1.0-py3-none-any.whl  
    └── prediction_model-0.1.0.tar.gz  
└── MANIFEST.in  
└── prediction_model  
    ├── config  
    |   ├── config.py  
    |   └── __init__.py  
    ├── datasets  
    |   ├── __init__.py  
    |   ├── test.csv  
    |   ├── train.csv  
    |   └── __init__.py  
    ├── pipeline.py  
    ├── predict.py  
    └── processing  
        ├── data_management.py  
        └── __init__.py  
    └── preprocessors.py  
    └── trained_models  
        └── classification_v1.pkl
```

```
|   |   |— __init__.py
|   |— train_pipeline.py
|   |— VERSION
|— prediction_model.egg-info
|   |— dependency_links.txt
|   |— PKG-INFO
|   |— requires.txt
|   |— SOURCES.txt
|   |— top_level.txt
|— README.md
|— requirements.txt
|— setup.py
|— tests
|   |— pytest.ini
|   |— test_predict.py
```

Package usage with example

Start a Python console (in a virtual environment):

```
(venv_package) suhas@ds:~/code/packages$ python
```

1. Python 3.6.9
2. [GCC 8.4.0] on linux
3. Type "help", "copyright", "credits" or "license" for more information.

Import the `prediction_model` package and make the predictions:

1. `import prediction_model`
2. `from prediction_model import train_pipeline`
3. `from prediction_model.predict import make_prediction`
4. `import pandas as pd`
- 5.
6. `train_pipeline.run_training() # Save the pickle object of the trained model`

```
Saved Pipeline : classification_v1.pkl # Output  
7. test_data = pd.read_csv("/home/suhas/code/test.csv") # Load the data  
8. result = make_prediction(test_data[0:1])#Make prediction on the first row  
9. print(result)  
{'prediction': ['Y']}
```

You should get the output as Y (that is, 'Yes') for the given set of input data.

Conclusion

This chapter discussed the importance of modular programming in the production environment; virtual environments play a crucial role when you are working on multiple projects. After that, you explored the steps to create and activate a virtual environment. You can list the packages to be installed inside the virtual environment in the *requirements.txt* file so that all the required packages can be installed in one go. Writing test cases boosts confidence about the package, functions, and overall integrity. Packages are portable and reusable, and they can be easily debugged.

In the next chapter, you will learn to leverage ML flow for tracking the ML experiments and saving the serialized models.

Points to remember

- The Python package holds multiple modules, and each module contains functions, classes, and variables.
- The *MANIFEST.in* file contains the list of files to be included and excluded.
- The *requirements.txt* files hold the list of packages to be installed using pip.
- Python refers to serialization and deserialization by the terms pickling and unpickling, respectively.
- The *__Init__.py* file indicates that the directory should be treated as a package.

Multiple choice questions

1. Identify a Python module.

- a) *load.py*
- b) *requirements.txt*
- c) *transform.ccm*
- d) *MANIFEST.in*

2. Python fixtures can be declared by which of the following?
 - a) fixture
 - b) pytest.fixture
 - c) @pytest.fixture
 - d) fixture.pytest

Answers

1. a
2. c

Questions

1. What is a Python module?
2. How can you interpret the output of the pytest command?
3. How can you install Python packages in editable mode?

Key terms

- **PYTHONPATH:** It augments the default search path for modules. The PYTHONPATH variable contains a list of directories whose modules are to be accessed in the Python environment.
- **Joblib:** Joblib is a set of tools to provide lightweight pipelining in Python. It is used to persist the model for future use, the following in particular:
 - Transparent disk-caching of functions and lazy re-evaluation (memorize pattern).
 - Simple parallel computing.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 5

MLflow-Platform to Manage the ML Life Cycle

Introduction

The Machine Learning life cycle involves many challenges. For instance, data scientists need to try different models containing multiple parameters and hyperparameters. They need to keep track of the model that is performing well and its parameters. Next, they need to save the serialized model for reusability. This chapter explains the role of MLflow in an ML life cycle. MLflow is a platform for streamlining machine learning development, including tracking experiments, packaging code into reproducible runs, and sharing and deploying models. It can manage a complete ML life cycle.

Structure

This chapter discusses the following topics:

- Introduction to MLflow
- MLflow tracking
- MLflow projects
- MLflow models
- MLflow model registry

Objectives

After studying this chapter, you should be able to train, reuse and deploy ML models using MLflow. You should also be able to track model evaluation metrics and model parameters, pickle the trained models, and compare two model results on MLflow UI. MLflow supports many downstream tools when it comes to deployment. You should be able to pick any iteration (by run ID) that is outperforming others and consume it through Python code, terminal, or postman.

Introduction to MLflow

MLflow is an open-source platform for managing the end-to-end machine learning life cycle.

MLflow allows data scientists to run as many experiments as they want before deploying the model into production; however, it keeps track of model evaluation metrics, such as RMSE and AUC. It also tracks the hyperparameters used while building the model. It enables you to save the trained model along with its best hyperparameters. Finally, it allows you to deploy an ML model into a production server or cloud. You can even keep track of the models being used in staging and production so that other team members can be aware of this information.

MLflow is library-agnostic, that is, you can use any popular ML library with it. Moreover, you can use any popular programming language for it as MLflow functions can be accessed via REST API and **Command Line Interface (CLI)**.

Integrating MLflow with your existing code is quite easy as it requires minimal changes. If you are working on a local system, it will automatically create a `mlrun` directory, wherein it stores the output, artifacts, and metadata. It creates a separate directory for each run. However, you can specify the path to create the `mlrun` directory. MLflow allows you to store information of each run into databases, such as MySQL or PostgreSQL.

MLflow is more useful in the following scenarios:

- **Comparing different models:** MLflow offers a UI that allows users to compare different models. You can compare Random Forest vs Logistic regression side by side, along with their model metric and parameters used. MLflow supports a wide range of model frameworks.
- **Cyclic model deployment:** In production, it is required to push a new version of the model after every data change, when new requirements come up, or after building a model better than the current one. In these scenarios, MLflow helps keep track of the models that are in staging (pre-production) and models that are in production with versions and brief descriptions.

- **Multiple dependencies:** If you are working on different projects or different frameworks, each of them will have a different set of dependencies. MLflow helps you to maintain dependencies along with your model.
- **Working with a large data science team:** MLflow stores the model metrics, parameters, time created, versions, users, and so on. This information is accessible to other team members working on the same projects. They can track all the metadata using MLflow UI or SQL table (if you are storing it in the database).

Set up your environment and install MLflow

In this section, you will install miniconda and then install `mlflow` in the conda environment. However, if you already have anaconda or miniconda installed, you can create a conda environment and install `mlflow` using PIP.

Miniconda installation

If you have anaconda installed on your machine, then you can skip the next step, that is, installing and setting up Miniconda.

You can download Miniconda from the following link:

<https://docs.conda.io/en/latest/miniconda.html#linux-installers>

Or you can simply execute the following command in the terminal (for Linux):

```
wget https://repo.anaconda.com/miniconda/Miniconda3-py37_4.10.3-Linux-x86_64.sh
```

Run the installation:

```
bash Miniconda3-latest-Linux-x86_64.sh
```

Read and follow the prompts on the installer screens. Restart the terminal when the miniconda installation is completed.

In the terminal, you will see that the base conda environment is auto-activated. You can disable the auto-activation of the base environment by running the following command in the terminal:

```
conda config --set auto_activate_base false
```

Verify the conda installation using the following command:

```
conda list
```

Run the following command to update conda (optional):

```
conda update conda
```

Let's create a virtual environment:

```
conda create -n venv python=3.7
```

Where:

- **-n** refers to the name of the virtual environment.
- **venv** is the name of the virtual environment.

The preceding command will create a virtual environment with Python version 3.7.

Once the conda environment is created, it needs to be activated by running the following command:

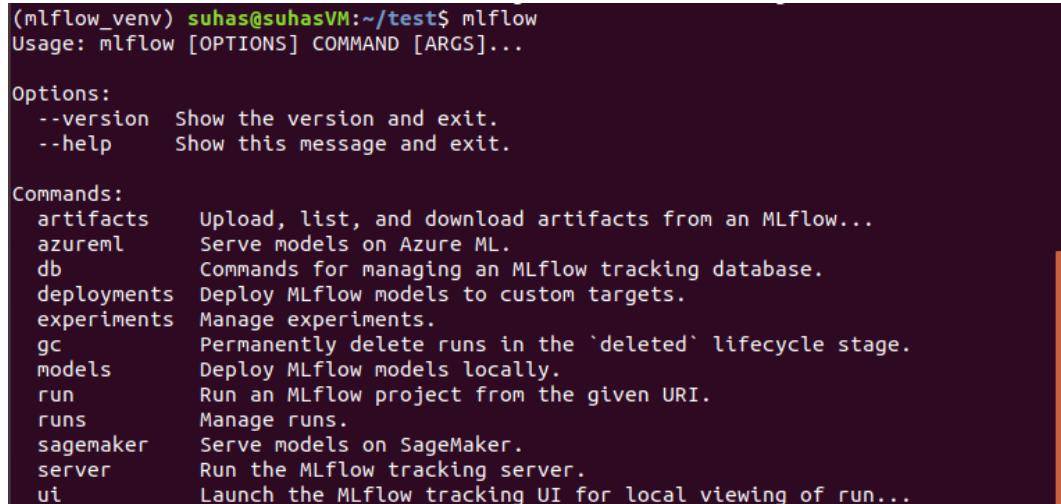
```
conda activate venv
```

Finally, install MLflow using **pip**:

```
pip install mlflow
```

Note: By default, the MLflow project uses conda for installing dependencies; however, you can proceed without conda by using the **-no-conda** option, for instance, **mlflow run . -no-conda**.

After installing MLflow, type **mlflow** in the terminal and hit enter to check its usage, options, and commands. The following figure shows the usage and options of **mlflow**:



```
(mlflow_venv) suhas@suhasVM:~/test$ mlflow
Usage: mlflow [OPTIONS] COMMAND [ARGS]...

Options:
  --version  Show the version and exit.
  --help      Show this message and exit.

Commands:
  artifacts   Upload, list, and download artifacts from an MLflow...
  azureml    Serve models on Azure ML.
  db          Commands for managing an MLflow tracking database.
  deployments Deploy MLflow models to custom targets.
  experiments Manage experiments.
  gc          Permanently delete runs in the `deleted` lifecycle stage.
  models     Deploy MLflow models locally.
  run        Run an MLflow project from the given URI.
  runs       Manage runs.
  sagemaker   Serve models on SageMaker.
  server     Run the MLflow tracking server.
  ui         Launch the MLflow tracking UI for local viewing of run...
```

Figure 5.1: MLflow usage, options, and commands

To open the web UI of MLflow, run the following command in the terminal (refer to figure 5.2):

```
mlflow ui
```

```
(mlflow_venv) suhas@suhasVM:~/test$ mlflow ui
[2021-09-23 20:02:36 +0530] [23435] [INFO] Starting gunicorn 20.1.0
[2021-09-23 20:02:36 +0530] [23435] [INFO] Listening at: http://127.0.0.1:5000 (23435)
[2021-09-23 20:02:36 +0530] [23435] [INFO] Using worker: sync
[2021-09-23 20:02:36 +0530] [23438] [INFO] Booting worker with pid: 23438
```

Figure 5.2: Starting MLflow UI

Open the browser; you should see a web UI of MLflow, as shown in the following figure:

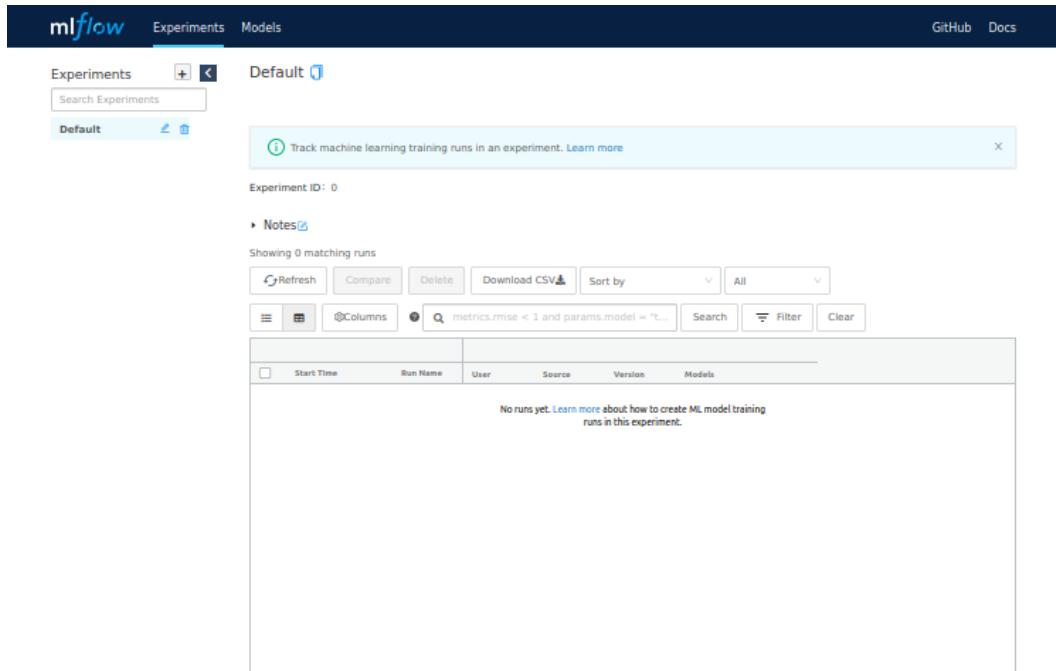


Figure 5.3: MLflow UI

You can execute a series of experiments and capture multiple runs of the experiments. Each experiment can contain multiple runs. You can also capture notes for the experiments. Apart from this, there are many customization options available, such as sorting by the columns, showing or hiding the columns, and changing the view of the table.

Note: Suppose you interrupted the running MLflow UI service and rerun the `mlflow ui` command; in that case, you may get the error as shown in the following figure:

```
(mlflow_venv) suhas@suhasVM:~/test$ mlflow ui
[2021-09-24 22:58:02 +0530] [23736] [INFO] Starting gunicorn 20.1.0
[2021-09-24 22:58:02 +0530] [23736] [ERROR] Connection in use: ('127.0.0.1', 5000)
[2021-09-24 22:58:02 +0530] [23736] [ERROR] Retrying in 1 second.
[2021-09-24 22:58:03 +0530] [23736] [ERROR] Connection in use: ('127.0.0.1', 5000)
[2021-09-24 22:58:03 +0530] [23736] [ERROR] Retrying in 1 second.
[2021-09-24 22:58:04 +0530] [23736] [ERROR] Connection in use: ('127.0.0.1', 5000)
[2021-09-24 22:58:04 +0530] [23736] [ERROR] Retrying in 1 second.
[2021-09-24 22:58:05 +0530] [23736] [ERROR] Connection in use: ('127.0.0.1', 5000)
[2021-09-24 22:58:05 +0530] [23736] [ERROR] Retrying in 1 second.
[2021-09-24 22:58:06 +0530] [23736] [ERROR] Connection in use: ('127.0.0.1', 5000)
[2021-09-24 22:58:06 +0530] [23736] [ERROR] Retrying in 1 second.
[2021-09-24 22:58:07 +0530] [23736] [ERROR] Can't connect to ('127.0.0.1', 5000)
Running the mlflow server failed. Please see the logs above for details.
```

Figure 5.4: MLflow UI error

It is because the address port is in use; so, you have to release it first, and then you can rerun the `mlflow ui`. It can be done using the following command:

```
sudo fuser -k 5000/tcp
```

MLflow components

MLflow is categorized into four components:

- MLflow tracking
- MLflow projects
- MLflow models
- MLflow registry

These components are devised such that they can work together seamlessly; however, you are free to use individual components as you need. For example, you are free to serve a model using MLflow without using a tracking component.

The following figure shows the MLflow's components and their major role:

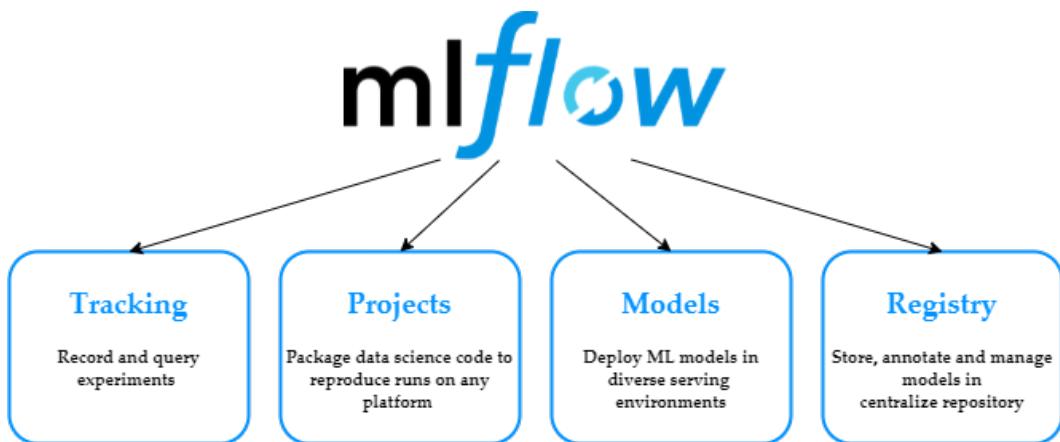


Figure 5.5: MLflow components

This chapter will cover each component's functionality with examples.

MLflow tracking

MLflow tracking is an API and UI for logging parameters, code versions, metrics, and artifacts when running your machine learning code and for visualizing the results.

MLflow captures the following information in the form of runs, where each run means executing a block of code:

- **Start and end time:** It records the start and end times of an experiment.
- **Source:** It can be the name of the file to launch the run or the MLproject name.
- **Parameters:** They contain the data in key-value pairs. These are nothing but the model input parameters you want to capture, such as the number of trees used in a random forest algorithm. For instance, `n_estimators` is key, and its value is 100. You need to call MLflow's `log_param()` to store the parameters.
- **Metrics:** A metric is used to measure the performance of the model, such as the accuracy of the model. It holds the data in a key-value pair; however, the value should be numeric only. You need to call MLflow's `log_metric()` to store the metric.

- **Artifacts:** When you want to store a file or object (such as a pickle file of the trained model), then the function of the artifacts comes to the rescue. You can store a serialized trained model, plot, or CSV file using this function, and it can be called using `log_artifacts()`.

First, you have to store the file or object in the local directory, and from there, you can save the file or object by providing the path of that directory.

Log data into the run

`mlflow.set_tracking_uri()`

It connects to MLflow tracking **Uniform Resource Identifier (URI)**. By default, tracking URI is set to the `mlruns` directory; however, you can set it to a remote server (HTTP/HTTPS), local directory path, or a database like MySQL.

```
1. import mlflow  
2. mlflow.set_tracking_uri('http://localhost:5000')
```

`mlflow.tracking.get_tracking_uri()`

This function will return the current tracking URI of MLflow.

`mlflow.create_experiment()`

It will create a new experiment. You can capture the runs by providing the experiment ID while executing `mlflow.start_run`. The experiment name should be unique.

```
1. exp_id = mlflow.create_experiment("Loan_Prediction")
```

`mlflow.set_experiment()`

This method activates the experiment so that the runs will be captured under the provided experiment. A new experiment is created in case the mentioned experiment does not exist. By default, the experiment is set to 'Default', as shown in figure 5.3.

`mlflow.start_run()`

It starts a new run or returns the currently active run. You can pass the run name, run ID, and experiment's name under the current run that needs to be tracked.

```
1. # For single iteration  
2. run = mlflow.start_run()  
3.  
4. # For multiple iterations  
5. with mlflow.start_run(run_name="test_ololo") as run:
```

`mlflow.end_run()`

It ends the currently active run (if any).

`mlflow.log_param()`

It logs a single key-value parameter in the currently active run. The key and value are both strings. Use `mlflow.log_params()` to log multiple parameters at once.

```
1. n_estimators = 100  
2. mlflow.log_param("n_estimators:", n_estimators)
```

`mlflow.log_metric()`

This MLflow's function will track the model metrics, such as the model's accuracy. To track multiple metrics, use `mlflow.log_metrics()`.

```
1. accuracy = 0.8  
2. mlflow.log_metric("accuracy", accuracy)
```

`mlflow.set_tag()`

It stores the data in a key-value pair. In this, you can set labels for the identification or any specific metric you want to track. To set multiple tags, use `mlflow.set_tags()`.

```
1. import mlflow  
2. with mlflow.start_run():  
3.     mlflow.set_tag("model_version", "0.1.0")
```

`mlflow.log_artifact()`

This function will log or store the files or objects in the *artifacts* directory; however, you would need to store it in the local directory first, and then it can pull the files or objects.

```
1. import pandas as pd  
2. import mlflow  
3.  
4. dir_name = 'data_dir'  
5. file_name = 'data_dir/cust_sale.csv'  
6. data = pd.DataFrame({'Cust_id': [461, 462, 463], 'Sales': [2631, 8462, 4837]})  
7. data.to_csv(file_name, index=False)  
8. mlflow.log_artifacts(dir_name)
```

`mlflow.get_artifact_uri()`

It will return the path to the artifact's root directory, where the artifacts are stored.

```
1. import mlflow  
2. mlflow.get_artifact_uri()  
3. './mlruns/0/be1cd88ebd704e9ab7629fd364747e1e/artifacts'
```

Let's consider the scenario of loan prediction, where the objective is to predict whether a customer is eligible for a loan. First, import the required libraries.

```
1. # Importing required packages  
2. import pandas as pd  
3. import numpy as np  
4.  
5. from sklearn.linear_model import LogisticRegression  
6. from sklearn.ensemble import RandomForestClassifier  
7. from sklearn.tree import DecisionTreeClassifier  
8.  
9. from matplotlib import pyplot as plt  
10.  
11. from sklearn import preprocessing  
12. from sklearn.model_selection import train_test_split, GridSearchCV  
13. from sklearn import metrics  
14.  
15. import mlflow
```

The next step is to load the datasets and capture numerical and categorical column names in separate variables:

```
1. # Reading the data  
2. data = pd.read_csv("loan_dataset.csv")  
3. num_col = data.select_dtypes(include=['int64','float64']).columns.  
tolist()  
4. cat_col = data.select_dtypes(include=['object']).columns.tolist()  
5. cat_col.remove('Loan_Status')  
6. cat_col.remove('Loan_ID')
```

Treat missing values for categorical and numerical columns:

```

1. # Creating a list of categorical and numerical variables
2. for col in cat_col:
3.     data[col].fillna(data[col].mode()[0], inplace=True)
4.
5. for col in num_col:
6.     data[col].fillna(data[col].median(), inplace=True)

```

Cap extreme values to the 5th and 95th percentile for numerical data.

```

1. # Clipping extreme values
2. data[num_col] = data[num_col].apply(lambda x: x.clip(*x.quantile([0.05, 0.95])))

```

Create a new feature named **TotalIncome**, which is the sum of the applicant's income and the co-applicant's income.

```

1. # creating a new feature as Total Income
2. data['LoanAmount'] = np.log(data['LoanAmount']).copy()
3. data['TotalIncome'] = data['ApplicantIncome'] + data['CoapplicantIncome']
4. data['TotalIncome'] = np.log(data['TotalIncome']).copy()

```

Drop the applicant income and co-applicant income columns.

```

1. # Dropping ApplicantIncome and CoapplicantIncome
2. data = data.drop(['ApplicantIncome', 'CoapplicantIncome'], axis=1)

```

Convert categorical columns to numeric columns using a label encoding technique.

```

1. # Label encoding categorical variables
2. for col in cat_col:
3.     le = preprocessing.LabelEncoder()
4.     data[col] = le.fit_transform(data[col])
5.
6. data['Loan_Status'] = le.fit_transform(data['Loan_Status'])

```

Split the data into train and test (70:30).

```

1. # Train test split
2. X = data.drop(['Loan_Status', 'Loan_ID'], 1)

```

```
3. y = data.Loan_Status  
4.  
5. SEED = 1  
6.  
7. X_train, X_test, y_train, y_test = train_test_split(X,y, test_size  
=0.3, random_state = SEED)
```

Build a logistic regression model using a grid search cross-validation approach.

```
1. #_____Logistic Regression_____#  
2.  
3. lr = LogisticRegression(random_state=SEED)  
4. lr_param_grid = {  
5.     'C': [100, 10, 1.0, 0.1, 0.01],  
6.     'penalty': ['l1','l2'],  
7.     'solver':['liblinear']  
8. }  
9.  
10. lr_gs = GridSearchCV(  
11.         estimator=lr,  
12.         param_grid=lr_param_grid,  
13.         cv=5,  
14.         n_jobs=-1,  
15.         scoring='accuracy',  
16.         verbose=0  
17.     )  
18. lr_model = lr_gs.fit(X_train, y_train)
```

Build a decision tree model using a grid search cross-validation approach.

```
1. #_____Decision Tree_____#  
2.  
3. dt = DecisionTreeClassifier(  
4.     random_state=SEED  
5. )  
6.
```

```
7. dt_param_grid = {  
8.     "max_depth": [3, 5, 7, 9, 11, 13],  
9.     'criterion' : ["gini", "entropy"],  
10. }  
11.  
12. dt_gs = GridSearchCV(  
13.     estimator=dt,  
14.     param_grid=dt_param_grid,  
15.     cv=5,  
16.     n_jobs=-1,  
17.     scoring='accuracy',  
18.     verbose=0  
19. )  
20. dt_model = dt_gs.fit(X_train, y_train)
```

Build a random forest model using a grid search cross-validation approach.

```
1. # _____ Random Forest _____ #  
2.  
3. rf = RandomForestClassifier(random_state=SEED)  
4. rf_param_grid = {  
5.     'n_estimators': [400, 700],  
6.     'max_depth': [15,20,25],  
7.     'criterion' : ["gini", "entropy"],  
8.     'max_leaf_nodes': [50, 100]  
9. }  
10.  
11. rf_gs = GridSearchCV(  
12.     estimator=rf,  
13.     param_grid=rf_param_grid,  
14.     cv=5,  
15.     n_jobs=-1,  
16.     scoring='accuracy',  
17.     verbose=0
```

```
18.      )
19. rf_model = rf_gs.fit(X_train, y_train)
```

Create a function for evaluating the model's metrics.

```
1. # Model evaluation metrics
2. def model_metrics(actual, pred):
3.     accuracy = metrics.accuracy_score(y_test, pred)
4.     f1 = metrics.f1_score(actual, pred, pos_label=1)
5.     fpr, tpr, thresholds1 = metrics.roc_curve(y_test, pred)
6.     auc = metrics.auc(fpr, tpr)
7.     plt.figure(figsize=(8,8))
8.     # plot auc
9.     plt.plot(fpr, tpr, color='blue', label='ROC curve area = %0.2f' %auc)
10.    plt.plot([0,1],[0,1], 'r--')
11.    plt.xlim([-0.1, 1.1])
12.    plt.ylim([-0.1, 1.1])
13.    plt.xlabel('False Positive Rate', size=14)
14.    plt.ylabel('True Positive Rate', size=14)
15.    plt.legend(loc='lower right')
16.
17.    # Save plot
18.    plt.savefig("plots/ROC_curve.png")
19.
20.    # Close plot
21.    plt.close()
22.
23.    return(accuracy, f1, auc)
```

Create a function for capturing information like the model parameters and model metrics using MLflow.

```
1. # MLflow's Logging functions
2. def mlflow_logs(model, X, y, name):
```

```
3.  
4.     with mlflow.start_run(run_name = name) as run:  
5.  
6.         # Run id  
7.         run_id = run.info.run_id  
8.         mlflow.set_tag("run_id", run_id)  
9.  
10.        # Make predictions  
11.        pred = model.predict(X)  
12.  
13.        # Generate performance metrics  
14.        (accuracy, f1, auc) = model_metrics(y, pred)  
15.  
16.        # Logging best parameters  
17.        mlflow.log_params(model.best_params_)  
18.  
19.        # Logging model metric  
20.        mlflow.log_metric("Mean cv score", model.best_score_)  
21.        mlflow.log_metric("Accuracy", accuracy)  
22.        mlflow.log_metric("f1-score", f1)  
23.        mlflow.log_metric("AUC", auc)  
24.  
25.        # Logging artifacts and model  
26.        mlflow.log_artifact("plots/ROC_curve.png")  
27.        mlflow.sklearn.log_model(model, name)  
28.  
29.        mlflow.end_run()
```

Predict on test data and capture the model metrics and parameters using the preceding function.

1. # Make predictions using ML models
- 2.

```

3. mlflow_logs(dt_model, X_test, y_test, "DecisionTreeClassifier")
4. mlflow_logs(lr_model, X_test, y_test, "LogisticRegression")
5. mlflow_logs(rf_model, X_test, y_test, "RandomForestClassifier")

```

Activate the virtual environment:

```
conda activate venv
```

The following figure shows the activation and deactivation commands after successfully creating a conda environment.

```

# To activate this environment, use
#
#     $ conda activate venv
#
# To deactivate an active environment, use
#
#     $ conda deactivate

suhas@ds:~/mlb/mlflow$ conda

```

Figure 5.6: Conda activation command

Run the *train.py* file, which will capture the parameters and metrics and store the artifacts at a given location.

Start the MLflow UI and then run the *train.py* file, as shown in the following figure.

```

(venv) suhas@ds:~/mlb/mlflow$ mlflow ui &
[2] 10041
(venv) suhas@ds:~/mlb/mlflow$ [2021-10-01 21:57:19 +0530] [10047] [INFO] Starting gunicorn 20.1.0
[2021-10-01 21:57:19 +0530] [10047] [INFO] Listening at: http://127.0.0.1:5000 (10047)
[2021-10-01 21:57:19 +0530] [10047] [INFO] Using worker: sync
[2021-10-01 21:57:19 +0530] [10050] [INFO] Booting worker with pid: 10050
(venv) suhas@ds:~/mlb/mlflow$ python train.py

```

Figure 5.7: MLflow UI command for keeping it running in the background

```
python train.py
```

This does not print anything, so one can check the output on MLflow's UI.

For demonstration, three different classifiers are implemented, namely, logistic regression, decision tree, and random forest, to compare their performances.

The following figure shows that model metrics, parameters, model names, and run ID have been captured. Overall, the logistic regression's accuracy is 78.9%, which is slightly better than the others.

Start Time	Run Name	User	Source	AUC	Accuracy	Mean cv score	F1-score	C	criterion	max_depth	run_id
1 minute ago	RandomForestClassifier	suhas	train.py	0.685	0.778	0.823	0.853	-	entropy	15	bc049e099f1f447183e3df658c8ad18
1 minute ago	LogisticRegression	suhas	train.py	0.689	0.789	0.818	0.862	100	-	-	62d27d96e68842da9b0de782ede2...
1 minute ago	DecisionTreeClassifier	suhas	train.py	0.652	0.757	0.811	0.841	-	gini	3	a67794e63444115b42d4cb8f342d...
15 minutes ago	RandomForestClassifier	suhas	train.py	0.685	0.778	0.823	0.853	-	entropy	15	ba50bf3ff3e44487965bd3ab0b27546
15 minutes ago	LogisticRegression	suhas	train.py	0.689	0.789	0.818	0.862	100	-	-	5a390faee3c54869e2764687e203...
15 minutes ago	DecisionTreeClassifier	suhas	train.py	0.652	0.757	0.811	0.841	-	gini	3	d765e56b187d4b18a270d66c649c...

Figure 5.8: MLflow UI for train.py output

Capture MLflow logs under the `Loan_prediction` experiment instead of the `Default` experiment.

```

1. # Make predictions using ML models
2. mlflow.set_experiment("Loan_prediction")
3. mlflow_logs(dt_model, X_test, y_test, "DecisionTreeClassifier")
4. mlflow_logs(lr_model, X_test, y_test, "LogisticRegression")
5. mlflow_logs(rf_model, X_test, y_test, "RandomForestClassifier")

```

By default, MLflow will capture the information under the experiment name `Default`. However, you can specify the experiment name in the command itself, as follows:

```
python train.py -experiment-name Loan_prediction
```

This command won't print any output in the terminal as there are no print statements in the `train.py` file.

Now, the information is getting captured under the **Loan_prediction** experiment, as shown in the following figure:

Start Time	Run Name	User	Source	AUC	Accuracy	Mean cv score	F1-score	C	criterion	max_depth	run_id
33 seconds ago	RandomForestClassifier	suhas	train.py	0.685	0.778	0.823	0.853	-	entropy	15	f3fb8dab34988913dab7e50b2835f
35 seconds ago	LogisticRegression	suhas	train.py	0.689	0.789	0.818	0.862	100	-	-	7c9eaaad681451097110d9205da504b
37 seconds ago	DecisionTreeClassifier	suhas	train.py	0.652	0.757	0.811	0.841	-	gini	3	526da0101a3b43cb21c56d38fa4e66

Figure 5.9: MLflow UI for a new experiment

You can deactivate the virtual environment using the following command:

```
conda deactivate
```

MLflow projects

Once you are done with the experimentation phase, your next step would be packaging all the code as a project with its dependencies. Let's say you want to shift the codebase and dependencies to the server or to another machine; MLflow will do the job for you.

MLflow allows you to package the codebase and its dependencies to make it reproducible and reusable. MLflow projects provide API and CLI capabilities that will help you integrate your model in MLOps.

You can run the MLflow project directly from the remote git repository (provided it should contain all the necessary files); alternatively, you can run it from the local CLI.

Following are the fields of the MLproject file:

- **Name:** It is the name of the project, and it can be any text.
- **Environment:** This is the environment that will be used at the time of execution of the entry point command. This will contain dependencies/packages required by the entry point or MLflow project.

- **Entry point:** The entry point section holds the command to be executed inside the MLflow project environment. This command can take arguments; it is a mandatory field and cannot be left blank.
- **Parameters:** This section holds one or more arguments that will be used by the entry point commands, but it is optional.

Create a *conda.yaml* file:

```
1. name: Loan_prediction  
2.  
3. channels:  
4.   - defaults  
5.  
6. dependencies:  
7.   - python=3.7  
8.   - pip  
9.   - pip:  
10.    - mlflow  
11.    - numpy==1.19.5  
12.    - pandas==1.1.5  
13.    - matplotlib==3.3.4  
14.    - scikit-learn==0.24.2
```

Then, create an *MLproject* file:

```
1. name: Loan_prediction  
2.  
3. conda_env: conda.yaml  
4.  
5. entry_points:  
6.   main:  
7.     command: "python train.py"
```

Switch to the directory where the *MLproject* file and the conda environment are present. Locate the YAML file and run:

```
mlflow run .
```

You can create and set the experiment name using the CLI, as shown in the following figure.

```
mlflow.set_experiment("Loan_prediction")
(venv) suhas@ds:~/mlb/mlflow$ mlflow experiments create --experiment-name Loan_prediction
Created experiment 'Loan_prediction' with id 1
```

Figure 5.10: Creating a new experiment using the command

```
mlflow run . --experiment-name Loan_prediction
```

The following figure shows the output of the preceding command:

```
(venv) suhas@ds:~/mlb/mlflow$ mlflow run . --experiment-name Loan_prediction
2021/11/20 00:18:14 INFO mlflow.projects.utils: === Created directory /tmp/tmpkv82_lo6 for downloading remote URIs passed to arguments of type 'path' ==
=
2021/11/20 00:18:14 INFO mlflow.projects.backend.local: === Running command
'source /home/suhas/miniconda3/bin/.../etc/profile.d/conda.sh && conda activate mlflow-b5055c7e7fba84943972ce6c0cd08f32c893b5ce 1>&2 && python train.py'
in run with ID '40c8762545204d6ca12e726d091876cd' ===
2021/11/20 00:19:23 INFO mlflow.projects: === Run (ID '40c8762545204d6ca12e726d091876cd') succeeded ==
```

Figure 5.11: Output of ML project's run command

Run the following command in your terminal to run MLproject from the GitHub repository:

```
mlflow run https://github.com/suhas-ds/mlflow_loan_prediction
--experiment-name Loan_prediction
```

The following figure shows the output of the preceding command:

```
(venv) suhas@ds:~/mlb/mlflow$ mlflow run https://github.com/suhas-ds/mlflow_loan_prediction --experiment-name Loan_prediction
2021/11/20 00:08:47 INFO mlflow.projects.utils: === Fetching project from https://github.com/suhas-ds/mlflow_loan_prediction into /tmp/tmpsre4t5yp ===
Username for 'https://github.com': suhas-ds
Password for 'https://suhas-ds@github.com':
2021/11/20 00:09:20 INFO mlflow.projects.utils: === Created directory /tmp/tmpnb2xu1sb for downloading remote URIs passed to arguments of type 'path' ==
=
2021/11/20 00:09:20 INFO mlflow.projects.backend.local: === Running command
'source /home/suhas/miniconda3/bin/.../etc/profile.d/conda.sh && conda activate mlflow-b5055c7e7fba84943972ce6c0cd08f32c893b5ce 1>&2 && python train.py' in run with ID '2315c9aba4eb4e43836d83c19355d33e' ===
2021/11/20 00:10:32 INFO mlflow.projects: === Run (ID '2315c9aba4eb4e43836d83c19355d33e') succeeded ==
```

Figure 5.12: Running ML project from GitHub

You can see the new experiment **Loan_prediction** created and the information captured in it.

When you open the details of the **LogisticRegression** model, you can see Git Commit, as this was run directly from GitHub.

The following figure shows the output for logistic regression after running the MLflow project from the GitHub repository.

Date: 2021-10-02 22:36:30
User: suhas
Source: train.py
Duration: 1.3s
Git Commit: d48e0ae091e241740151613ebbd8731de256eae6b9
Status: FINISHED

Name	Value
C	100
penalty	l1
solver	liblinear

Name	Value
AUC	0.689
Accuracy	0.789
Mean cv score	0.818
f1-score	0.862

Figure 5.13: Logistic regression results

MLflow models

The MLflow models module lets you package the model in different ways, such as python function, Scikit-learn (sklearn), and Spark MLlib (spark). This flexibility helps you to connect associated downstream tools effortlessly.

When you log the model using `mlflow.sklearn.log_model(model, name)`, a model directory gets created, and it stores the files and metadata associated with the models. You will see the following directory structure:

```
LogisticRegression/
├── conda.yaml
├── MLmodel
├── model.pkl
└── requirements.txt
```

The following figure shows the MLmodel details for the logistic regression model:

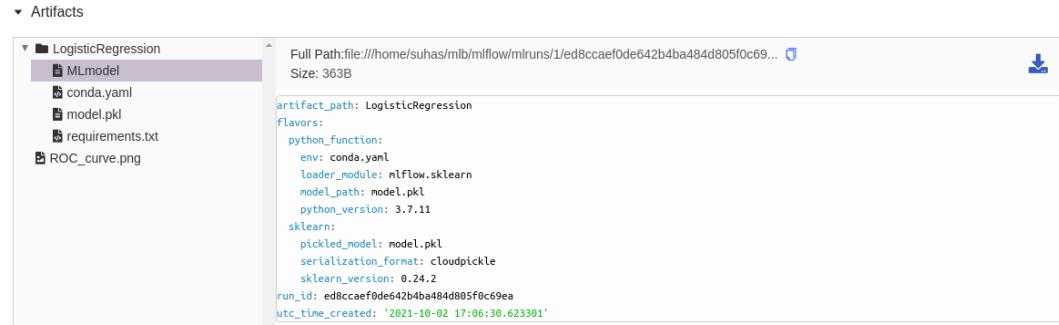


Figure 5.14: MLmodel

Now, let's predict by following the instruction under **Predict on a Pandas DataFrame**, as shown in the following figure:

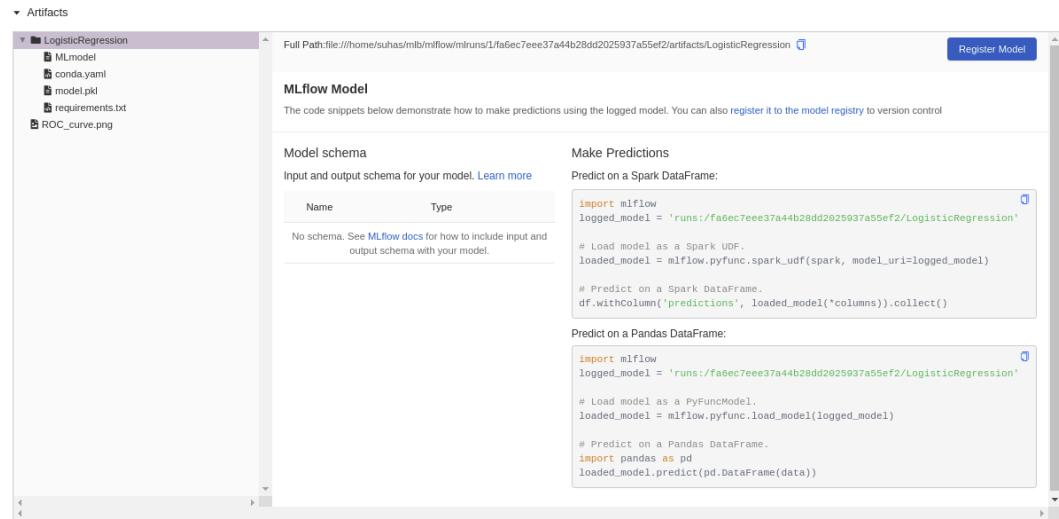


Figure 5.15: MLflow model

Now open the Python console by typing **python** in the terminal. Here, the aim is to create a pandas DataFrame and pass it to the **predict** function. You can load the DataFrame from the local directory.

In the following figure, a pandas DataFrame is used for prediction using MLflow's model.

```
>>> import mlflow
>>> logged_model = 'runs:/ed8ccae0de642b4ba484d805f0c69ea/LogisticRegression'
>>> loaded_model = mlflow.pyfunc.load_model(logged_model)
>>> import pandas as pd
>>> # Predict on a Pandas DataFrame
>>> loaded_model.predict(pd.DataFrame([[1.0,0.0,0.0,0.0,0.0,4.85203026,360.0,1.0,2.0,8.67402599]]))
array([1])
>>> █
```

Figure 5.16: Making predictions on pandas DataFrame using MLflow model

Now, deploy a local REST server to serve the predictions using the MLmodel.

By default, the server runs on port **5000**. If the port is already in use, you can use the **--port** or **-p** option to provide a different port.

For instance, `mlflow models serve -m runs: /<RUN_ID>/model --port 1234`.

To deploy to the server, run the following command:

```
mlflow models serve -m /home/suhas/mlb/mlflow/
mlruns/0cc5b1a8724f4e818b4e92776ca21b73/0/artifacts/LogisticRegression/
-p 1234
```

In the preceding command, replace the model path with the model path used in your machine and provide a port **1234** to run on.

In the following figure, the MLflow model generated using logistic regression is deployed:

```
(venv) suhas@ds:~/mlb/mlflow$ mlflow models serve -m /home/suhas/mlb/mlflow/mlruns/0/0cc5b
1a8724f4e818b4e92776ca21b73/artifacts/LogisticRegression/ -p 1234
2021/11/19 21:55:12 INFO mlflow.models.cli: Selected backend for flavor 'python_function'
2021/11/19 21:55:16 INFO mlflow.pyfunc.backend: === Running command 'source /home/suhas/mi
niconda3/bin/..../etc/profile.d/conda.sh && conda activate mlflow-c9f596e179ab5335a829b6f7eb
ebb0275583927d 1>&2 && gunicorn --timeout=60 -b 127.0.0.1:1234 -w 1 ${GUNICORN_CMD_ARGS} -
-m mlflow.pyfunc.scoring_server.wsgi:app'
[2021-11-19 21:55:17 +0530] [5695] [INFO] Starting gunicorn 20.1.0
[2021-11-19 21:55:17 +0530] [5695] [INFO] Listening at: http://127.0.0.1:1234 (5695)
[2021-11-19 21:55:17 +0530] [5695] [INFO] Using worker: sync
[2021-11-19 21:55:17 +0530] [5705] [INFO] Booting worker with pid: 5705

[2021-11-19 21:56:17 +0530] [5695] [INFO] Handling signal: winch
```

Figure 5.17: Deploying MLflow model

The preceding figure illustrates that the server is listening at <http://127.0.0.1:1234>.

Call the REST API using the following curl command:

```
curl -X POST -H "Content-Type:application/json; format=pandas-split" --data '{"columns":["Gender","Married","Depen-
dents","Education","Self_Employed","LoanAmount","Loan_Amount_
Term","Credit_History","Property_Area","TotalIncome"],"da-
ta":[[1.0,0.0,0.0,0.0,0.0,4.85203026,360.0,1.0,2.0,8.67402599]]}' 
http://127.0.0.1:1234/invocations
```

The following figure depicts calling the REST API of the deployed model using the curl command:

```
(venv) suhas@ds:~/mlb/mlflow$ curl -X POST -H "Content-Type:application/json; format=pandas-split" --data '{"columns":["Gender","Married","Dependents","Education","Self_Employed","LoanAmount","Loan_Amount_Term","Credit_History","Property_Area","TotalIncome"],"data":[[1.0,0.0,0.0,0.0,0.0,4.85203026,360.0,1.0,2.0,8.67402599]]}' http://127.0.0.1:1234/invocations
[1](venv) suhas@ds:~/mlb/mlflow$ █
```

Figure 5.18: Curl command output

Optionally, you can call the REST API of the deployed model using the curl command, as shown in the following figure:

```
(venv) ~/mlb/mlflow $ curl http://127.0.0.1:1234/invocations -H "Content-Type:application/json" -d '{
> "columns":["Gender", "Married", "Dependents", "Education", "Self_Employed", "LoanAmount",
> "Loan_Amount_Term", "Credit_History", "Property_Area", "TotalIncome"],
> "data":[[1.0,0.0,0.0,0.0,0.0,4.85203026,360.0,1.0,2.0,8.67402599]]}'
```

Figure 5.19: Curl command output

The following figure shows the output of a different set of input data:

```
suhas@suhasVM:~$ curl -X POST -H "Content-Type:application/json; format=pandas-split" --data '{"columns":["Gender","Married","Dependents","Education","Self_Employed","LoanAmount","Loan_Amount_Term","Credit_History","Property_Area","TotalIncome"],"data":[[1.,1.,0.,1.,0.,4.55387689,360.,1.,2.,8.25556865]]}' http://127.0.0.1:1234/invocations
[1]suhas@suhasVM:~$ curl -X POST -H "Content-Type:application/json; format=pandas-split" --data '{"columns":["Gender","Married","Dependents","Education","Self_Employed","LoanAmount","Loan_Amount_Term","Credit_History","Property_Area","TotalIncome"],"data":[[1.,1.,3.,0.,0.,5.06,360.,0.,1.,8.62]]}' http://127.0.0.1:1234/invocations
[0]suhas@suhasVM:~$ suhas@suhasVM:~$ █
```

Figure 5.20: Curl command output

The server should respond with an output similar to [0] or [1].

Here, [0] is labeled as No, and [1] is labeled as Yes.

MLflow registry

MLflow registry is a platform for storing and managing ML models through UI and a set of APIs.

It keeps track of the model lineage, different versions, and transitions of the models from one state to another, like from staging to production. Every authorized team member can track all the preceding information.

To explore this component, the database needs to be connected to MLflow. MLflow components explored earlier can be connected to a database for storing the information.

Set up MLflow's tracking URI using the following command:

```
export MLFLOW_TRACKING_URI=http://localhost:5000
```

Set up the MySQL server for MLflow

You can install MySQL server if it is not installed already. For this, you can also refer to the official document at <https://dev.mysql.com/doc/mysql-apt-repo-quick-guide/en/>.

First, create **mlflow_user** in MySQL using the following command:

```
mysql -u mlflow_user
CREATE USER 'mlflow_user'@'localhost' IDENTIFIED BY 'mlflow'
GRANT ALL ON db_mlflow.* TO 'mlflow_user'@'localhost';
FLUSH PRIVILEGES;
```

Enter the password when prompted.

Create and select the database:

```
CREATE DATABASE IF NOT EXISTS db_mlflow;
use db_mlflow;
```

To know which user(s) have the access to **db_mlflow** database and its privileges, you can execute the following command:

```
SELECT * FROM mysql.db WHERE Db = 'db_mlflow'\G;
```

The following figure shows the privileges of `mlflow_user` associated with the `db_mlflow` database:

```
mysql> SELECT * FROM mysql.db WHERE Db = 'db_mlflow'\G;
***** 1. row ****
      Host: localhost
      Db: db_mlflow
    User: mlflow_user
Select_priv: Y
Insert_priv: Y
Update_priv: Y
Delete_priv: Y
Create_priv: Y
Drop_priv: Y
Grant_priv: N
References_priv: Y
Index_priv: Y
Alter_priv: Y
Create_tmp_table_priv: Y
Lock_tables_priv: Y
Create_view_priv: Y
Show_view_priv: Y
Create_routine_priv: Y
Alter_routine_priv: Y
Execute_priv: Y
Event_priv: Y
Trigger_priv: Y
1 row in set (0.03 sec)
```

Figure 5.21: Permissions of MLflow_user

Install the MySQLdb module:

```
sudo apt-get install python3-mysqldb
```

Install MySQL client for Python:

```
pip install mysqlclient
```

Here are the concepts and key features of the model registry:

- **Registered model:** Once the model is registered using MLflow's UI or API, the model is considered a registered model. MLflow's model registry captures model versions and keeps track of the model's stages (for example, production, and staging) and other metadata.
- **Model version:** MLflow's model registry maintains the version of each model after registering it. For instance, if you saved a model name with a classification model, then it would be assigned to version 1 by default. However, after saving that model with the same name again, it will be saved as version 2.
- **Model stage:** For each model version, you can assign different stages, like staging, production, and archived. However, you cannot assign two stages to the same version of the model.

- **Annotations and descriptions:** You can add comments, short descriptions, and annotations for models. Your team members will come to know about the model through the descriptions you add.

Start the MLflow server

All the required steps to start the MLflow server with MySQL as a database have been completed.

To start the MLflow server, you can use the following command:

```
mlflow server --host 0.0.0.0 --port 5000 --backend-store-uri mysql://
mlflow_user:mlflow@localhost/db_mlflow --default-artifact-root $PWD/mlruns
```

The syntax to start the MLflow server is `mlflow server <args>`. In the preceding command, MLflow uses the backend store as MySQL server (provide MLflow username and database name) and the default artifacts store as the `mlruns` directory.

You can run the `MLproject` and pass the experiment name as a parameter (optional).

```
mlflow run . --experiment-name 'Loan_prediction'
```

You can see, in the MLflow UI, that the version column shows alphanumeric values. The model's metrics parameters are displayed under the `Loan_prediction` experiment.

The following figure shows the output of the `Loan_prediction` experiment using MySQL as a backend store:

The screenshot shows the MLflow UI interface. At the top, there is a dark header bar with the 'mlflow' logo, 'Experiments', 'Models', 'GitHub', and 'Docs' buttons. Below the header, the main content area has a title 'Loan_prediction' with a search icon. On the left, a sidebar lists 'Experiments' (Default and Loan_prediction selected), 'Notes' (with a note about tracking machine learning training runs), and a search/filter bar. The main panel displays a table of experiment runs. The table has columns: Start Time, Run Name, Source, Version, AUC, Accuracy, Mean cv score, C, criterion, max_depth, and run_id. There are 6 matching runs listed:

				Metrics >	Parameters >		Tags				
	Start Time	Run Name	Source	Version	AUC	Accuracy	Mean cv score	C	criterion	max_depth	run_id
<input type="checkbox"/>	25 seconds	RandomFo...	train.py	a1f8bf	0.685	0.778	0.823	-	entropy	15	ba376c69b
<input type="checkbox"/>	28 seconds	LogisticRe...	train.py	a1f8bf	0.689	0.789	0.818	100	-	-	71a375138
<input type="checkbox"/>	30 seconds	DecisionTr...	train.py	a1f8bf	0.652	0.757	0.811	-	gini	3	37857831e
<input type="checkbox"/>	16 minutes	RandomFo...	train.py	a1f8bf	0.685	0.778	0.823	-	entropy	15	60a96cd9a
<input type="checkbox"/>	16 minutes	LogisticRe...	train.py	a1f8bf	0.689	0.789	0.818	100	-	-	93a700225

Figure 5.22: Output of ML project on MLflow UI

Using the **show tables;** command, you can see that new tables have been created, such as metrics and experiments. Refer to *Figure 5.23* to see the output of the command:

```
mysql> show tables;
+-----+
| Tables_in_db_mlflow |
+-----+
| alembic_version
| experiment_tags
| experiments
| latest_metrics
| metrics
| model_version_tags
| model_versions
| params
| registered_model_tags
| registered_models
| runs
| tags
+-----+
12 rows in set (0.00 sec)
```

Figure 5.23: MLflow tables in MySQL database

Let's check the experiments table using the command shown in *Figure 5.24*. As of now, **Default** and **Loan_prediction** are the two experiments being displayed.

```
mysql> select * from experiments;
+-----+-----+-----+-----+
| experiment_id | name      | artifact_location | lifecycle_stage |
+-----+-----+-----+-----+
|          0    | Default   | /home/suhas/mlb/mlflow/mlruns/0 | active        |
|          1    | Loan_prediction | /home/suhas/mlb/mlflow/mlruns/1 | active        |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Figure 5.24: MLflow's list of experiments is MySQL database

If you click on the **Models** tab on the top, you should see the following figure. As the model is not registered, the table does not contain any model information.

Figure 5.25: Initial UI of MLflow models

Now, go to the run ID on the **Experiment** tab and scroll down; then, click on the model directory in the UI.

The **Register Model** button will appear, as shown in the following figure:



Figure 5.26: Register Model button

Click on the **Register Model** button, and a pop-up window will appear, as shown in the following figure. In the current case, the **Create New Model** option is selected from the drop-down menu, along with a model name; however, you can give any human-readable name to it. Finally, click on the **Register** button.

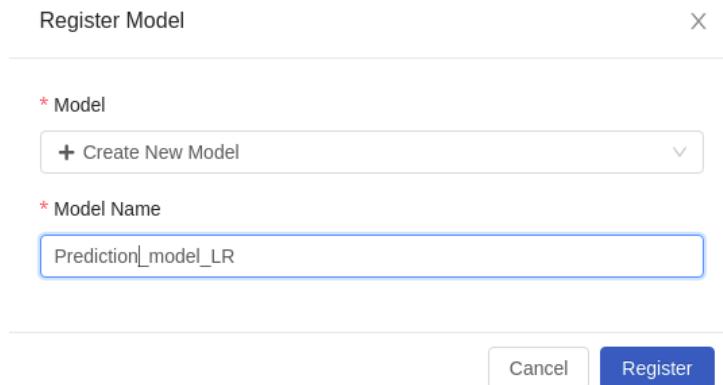


Figure 5.27: Register model window

As shown in the following figure, the **Register Model** button will change to the model's name (with hyperlink).



Figure 5.28: Registered model

Now, you should see the registered model on the **Models** tab. By default, it will label it as version 1. Since a state has not been assigned, you should see – (dash) in the staging and production columns.

The following figure shows the list of registered models:

Name	Latest Version	Stage	Last Modified	Tags
Prediction_model_LR	Version 1	Staging	2021-10-03 23:29:17	Classifier.Logistic Regression

Figure 5.29: Registered model in MLflow Models UI

We can change the model's state using MLflow UI or terminal. Here, the state is being changed to staging using MLflow's UI.

As shown in the following figure, the model's state is being changed from **None** to **Staging**:

Figure 5.30: Registered model's state transition

In the following figure, you can see that it is showing **Version 1** in the staging column of the registered model.

Name	Latest Version	Stage	Last Modified	Tags
Prediction_model_LR	Version 1	Staging	2021-10-03 23:32:29	Classifier:Logistic Regression

Figure 5.31: Registered model's state transitioned to staging

The registered model's information can be found in the **registered_models** table in MySQL, as shown in the following figure:

```
mysql> select * from registered_models;
+-----+-----+-----+-----+
| name | creation_time | last_updated_time | description |
+-----+-----+-----+-----+
| Prediction_model_LR | 1633282534977 | 1633282535089 |          |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Figure 5.32: List of registered models in MySQL table

When you are working in a team, you can add descriptions and tags to the registered model. It helps other team members to learn more about the model.

Now, add descriptions and tags for **Prediction_Model_LR**.

Name	Value	Actions
Classifier	Logistic Regression	

Version	Registered at	Created by	Stage	Description
Version 1	2021-10-03 23:05:35		None	

Figure 5.33: Registered model's tags and description

You have learned how to register models in MLflow, and now you are going to learn how to serve the model to make predictions on the given data.

Deploy the model from MLflow's registry using the following command:

```
mlflow models serve -m "models:/Prediction_model_LR/Staging"
```

```
1. import mlflow.pyfunc
2.
3. model_name = "Prediction_model_LR"
4. stage = 'Staging'
5. model=mlflow.pyfunc.load_model(model_uri=f"models:{model_name}/{stage}")
6.
7. model.predict([[1.,1.,0.,1.,0.,4.55387689,360.,1.,2.,8.25556865]])
8. array([1])
```

Note: You can call the model's REST API using postman, as shown in the following figure.

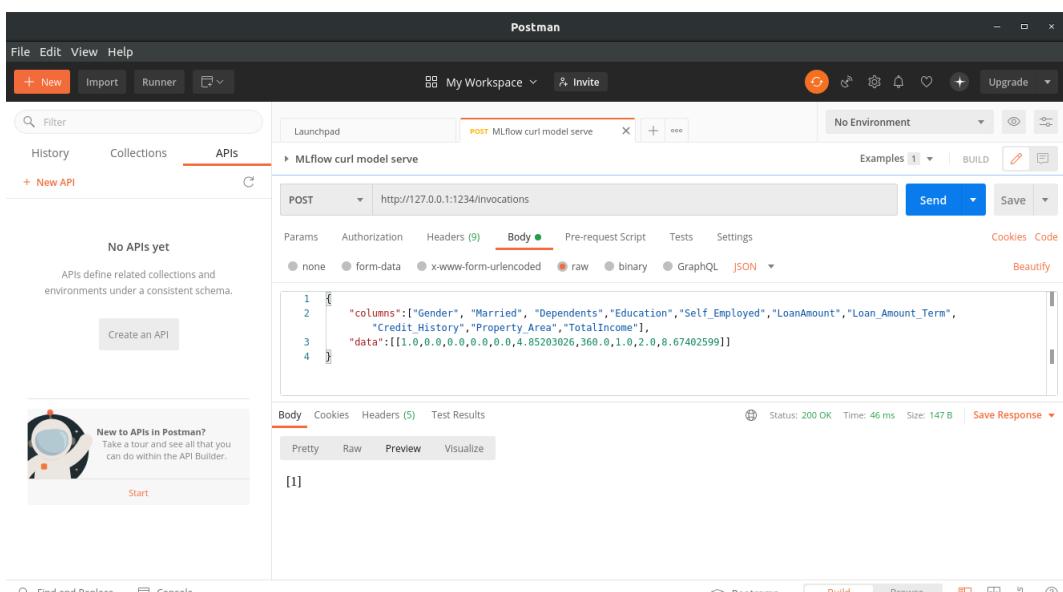


Figure 5.34: Calling REST API of the deployed model using postman

You can get postman from the following link:

<https://www.postman.com/downloads/>

There are various ways to consume the REST API of deployed models. You can also use this REST API in other applications.

Conclusion

This chapter explained the importance of MLflow in the ML life cycle and the production environment. You explored the role, functionality, and usage of MLflow, with examples of four components of MLflow. Then, you learned how MLflow helps data science developers at various stages of the ML life cycle. MLflow tracking allows one to choose the best model by keeping track of model metrics, hyperparameters used, and other useful information. The MLflow project component helps you to manage dependencies, and it can be run from the GitHub repository. The MLflow registry acts as a central location for registered models and changing the states, such as staging and production.

In the next chapter, you will learn how to use a docker for portability in transferring ML projects from one machine to another or to the server.

Points to remember

- MLflow helps you from the experimentation stage to the deployment stage of an ML project.
- Except for the model registry, all the components can be used without being integrated with a database like MYSQL; however, it is a good practice to integrate them with a database like MYSQL.
- By default, the MLflow project uses conda for installing dependencies; however, you can proceed without conda by using the **-no-conda** option.
- Each MLflow component can be accessed separately; however, you can connect them to create the flow.

Multiple choice questions

1. Which one of the following is not a MLflow component?

- a) MLflow tracking
- b) MLflow develop
- c) MLflow models
- d) MLflow registry

2. In the MLflow registry, the model state from staging to production can be changed using which of the following?
 - a) MLflow command
 - b) MLflow UI
 - c) Both a and b
 - d) Mode state cannot be changed from staging to production

Answers

1. b
2. c

Questions

1. What is MLflow?
2. What is the role of the *conda.yaml* file?
3. What is the command to serve the model?

CHAPTER 6

Docker for

ML

Introduction

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can be run on the same machine and share the OS kernel. Each running container is considered an isolated process in user space.

Docker automates the repetitive and time-consuming configuration task, saving both time and effort for the developer. Docker is a one-stop solution that includes UI, APIs, CLIs, and last but not least, security. Docker runs on Linux, Windows, and Mac OS.

Structure

This chapter discusses the following topics:

- Role of Docker in Machine Learning
- Hello World with Docker
- Create a Dockerfile
- Build Docker image
- Run Docker container
- Dockerize and deploy the ML model

Objectives

After studying this chapter, you should be able to build and deploy ML models into production as a container or orchestration service using Docker. You should be able to create a Dockerfile, which contains all the commands to be executed. Using Dockerfile, you should be able to build and run Docker images. You should also be able to run container service in the background using a detached mode. Finally, you can deploy an ML model using Docker.

Introduction to Docker

Docker is a containerization platform to package applications and their dependencies in the form of a container. It ensures that all the required libraries and dependencies are wrapped in an isolated environment to run the application smoothly in the development, test, and production environments. Docker is popular among developers as it is lightweight, fast, portable, secure, and more efficient than virtual machines. A containerized application will start running as soon as you run the Docker container.

Docker has its own Docker registry, called Docker hub. Docker hub allows developers to store and distribute container images over the internet. An image tag enables developers to differentiate images. A Docker registry has public and private repositories. A developer can store a container image on the Docker hub using the push command and retrieve one using the pull command.

It works on my machine!

Many a time, the code works perfectly on your machine but throws an error when you run it on another machine. This happens with developers and data scientists as well. The reason could be anything from a different OS to a different release of an OS, different python versions, or dependency issues. So, when they face this issue, they might end up spending a lot of time fixing it. With Docker, you should not encounter these issues, as Docker packages require files, configuration, and commands for seamless flow.

Long setup

The traditional method of deploying in production environments takes a lot of time, as it needs to move the necessary files, install dependencies, configure, and save the output manually. Many a time, developers need to set up the environment first. It is more time-consuming when you have to repeat this process for different stages of the project, such as development, pre-production, and production. Manual deployment scripts are difficult to manage.

With Docker, you can put all the required files in the directory and write down the configuration, OS version, and commands to be executed sequentially in a Dockerfile. You can also connect the two Docker containers with the same network. Additionally, you can use the same Dockerfile for development, pre-production, and production.

Docker ensures reproducibility, portability, easy deployment, granular updates, lightness, and simplicity.

Setting up your environment and installing Docker

You can refer to the instructions at <https://docs.docker.com/installation> to install the latest Docker-maintained package on your preferred operating system and take a look at the official Docker engine installation guide at <https://docs.docker.com/engine/install/ubuntu/>.

Docker installation

Here are the general steps you can follow to install Docker on your Ubuntu machine. The Docker engine installation requires one of the following 64-bit Ubuntu versions:

- Ubuntu Kinetic 22.10
- Ubuntu Jammy 22.04 (LTS)
- Ubuntu Focal 20.04 (LTS)
- Ubuntu Bionic 18.04 (LTS)

Uninstall old versions

First off, uninstall old versions of Docker (if any):

```
sudo apt-get remove docker docker-engine docker.io containerd runc
```

Set up the repository:

```
$ sudo apt-get update
$ sudo apt-get install \
  ca-certificates \
  curl \
  gnupg \
  lsb-release
```

Add Docker's official GPG key.

The **GNU Privacy Guard (GPG or GnuPG)** is a command-line tool that enables the implementation of public-key encryption and verification services. GPG is commonly used in Linux to sign files digitally, which assures the authenticity of software project files.

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

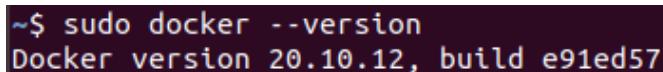
Use the following command to set up the repository:

```
$ echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/
  docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.
  list > /dev/null
```

Install Docker Engine

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
sudo docker --version
```

The following figure shows the current Docker version:



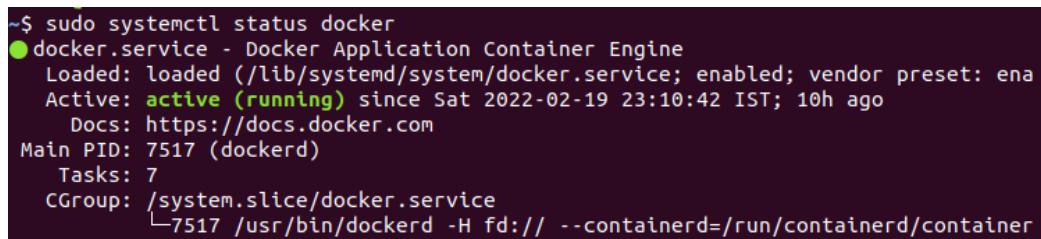
```
~$ sudo docker --version
Docker version 20.10.12, build e91ed57
```

Figure 6.1: Docker version

Run the following command in the terminal to check the status of the Docker service:

```
sudo systemctl status docker
```

The following figure shows the output of the preceding command:



```
~$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
  Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: ena
  Active: active (running) since Sat 2022-02-19 23:10:42 IST; 10h ago
    Docs: https://docs.docker.com
   Main PID: 7517 (dockerd)
      Tasks: 7
     CGroup: /system.slice/docker.service
             └─7517 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/container
```

Figure 6.2: Docker status check

Docker compose

Docker compose enables developers to configure and run more than one container. It reads the configuration from the `docker-compose.yml` file. A single `docker-compose up` command can start the services and run the multi-container applications. On the other hand, you can destroy all of this using the `docker-compose down` command. You can also remove the volumes by adding the `--volumes` flag.

You can install the Docker compose using the following command:

```
sudo curl -L "https://github.com/docker/compose/releases/
download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/
bin/docker-compose
```

The following figure shows the Docker compose installation:

```
~$ sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
[sudo] password for suhas:
      % Total    % Received % Xferd  Average Speed   Time     Time     Time  Current
                                     Dload  Upload   Total   Spent   Left  Speed
 100  664  100  664    0      0  1181       0  --::--  --::--  --::--  1179
 100 12.1M  100 12.1M    0      0  3003k       0  0:00:04  0:00:04  --::-- 4689k
```

Figure 6.3: Docker compose installation

Authorize `docker-compose` to execute files:

```
sudo chmod +x /usr/local/bin/docker-compose
```

Finally, verify the installation:

```
docker-compose --version
```

The following figure shows the current Docker compose version:

```
~$ docker-compose --version
docker-compose version 1.29.2, build 5becea4c
```

Figure 6.4: Docker compose version

The installation provides the following:

- Docker Engine
- Docker CLI client
- Docker Compose

Hello World with Docker

This is a sample Docker image to test whether Docker is working properly. By running the following command, you will create the first Docker container:

```
sudo docker run hello-world
```

The following figure shows the execution of the preceding command:

```
~$ sudo docker run hello-world
[sudo] password for suhas:
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:97a379f4f88575512824f3b352bc03cd75e239179eea0fecc38e597b2209f49a
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Figure 6.5: Docker image - hello-world

It also generates the following output, which tells you about what happened behind the scenes:

To generate this message, Docker took the following steps:

1. **The Docker client contacted the Docker daemon.**
2. **The Docker daemon pulled the "hello-world" image from the Docker Hub. (amd64)**
3. **The Docker daemon created a new container from that image, which runs the executable that produces the output you are currently reading.**
4. **The Docker daemon streamed that output to the Docker client, which sent it to your terminal.**

You executed the `docker run` command, followed by the image name. The required image was not there in the system, so rather than stopping further execution, it pulled the image from the Docker hub.

The following figure shows the Docker structure in a system. As you can see, Docker can have multiple containers running in an isolated environment while sharing the same infrastructure.

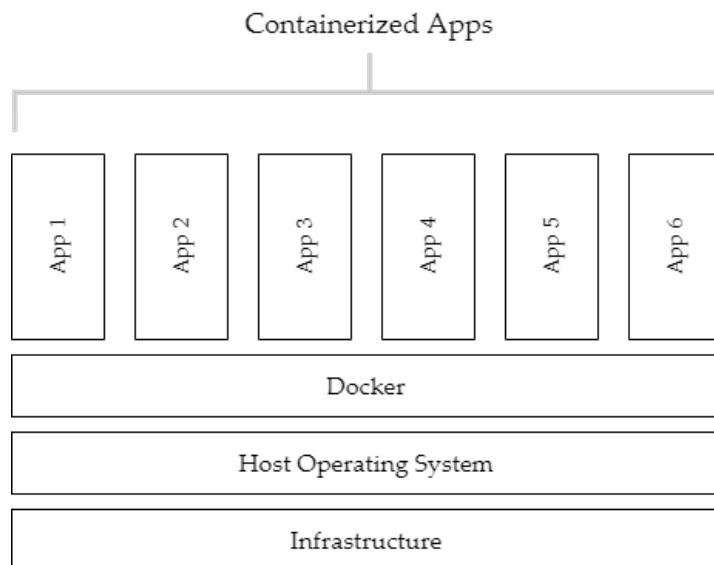


Figure 6.6: Docker stack

Docker objects

While working with Docker, you can create and use Docker objects like images, containers, volumes, and networks. In this chapter, you are going to learn about some of these objects.

Dockerfile

Dockerfile can be considered as a set of commands or instructions that enables developers to build Docker images. These commands or instructions get executed sequentially. It is a plain text document with no extension.

Docker image

To create a Docker container, a Docker image needs to be created. It stores all the code and dependencies required to run the application and acts as a template to run a container instance. A Docker image can be uploaded on the Docker hub, from where it can be pulled to the server or system to run the container.

To view the list of available Docker images, execute the following command in the terminal:

```
docker images
```

Docker containers

An instance of a container is created when you run the Docker image. You can use the same Docker image to run as many containers as you want. It is an important component of the Docker ecosystem. Docker runs containers in an isolated environment.

An additional layer called a container layer, gets automatically created on top of the existing image layers when the developer runs a container. A Docker container has its own read and write layer, which allows developers to make changes that are specific to that container. Suppose you are running three containers using the same Docker image, and you install another version of the python package inside a running container. This will not affect the existing version of the python package in other containers. Docker manages the data within the Docker container using Docker Volumes.

All the files that you created in an image or a container are part and parcel of the Union file system. However, the data volume is part of the Docker host file system, and it is simply mounted inside the container.

It is initialized when the container is created. By default, it is not deleted when the container is stopped. Data volumes can be shared across containers too and can be mounted in read-only mode as well.

The command to check all the running containers is **docker ps**.

The command to check all the running and stopped containers is **docker ps -a**.

Detached mode

To run Docker in the background, run the container in detached mode. You can use the **-d** flag to run the container in detached mode.

Docker container networking

Typically, a Docker host comprises multiple Docker containers. Docker containers also need to interact and collaborate with local as well as remote ones to come out with distributed applications. The bridge network is the default network interface that Docker Engine assigns to a container.

docker network ls

This command will show you the list of networks and their scope. The output contains the following headers:

NETWORK ID, NAME, DRIVER, SCOPE

To check the network details of any Docker container, use the following command, followed by the network ID:

```
docker network inspect <NETWORK ID>
```

Port mapping

The **-p** flag is used to map container ports to host ports. Consider this example:

```
docker run -p <myport>:<containerport> nginx
```

Note: To observe the output, use docker logs [container_id].

Create a Dockerfile

Dockerfile follows a simple and easy-to-understand structure, that is, # comment, followed by an instruction and an argument. It is a standard practice to write instructions in the uppercase to differentiate between instructions and arguments.

FROM

- The **FROM** instruction sets the base image for subsequent instructions.
- A valid Dockerfile must have a FROM instruction.
- **FROM** can occur multiple times in the Dockerfile.

CMD

- **CMD** defines a default command to execute when a container is being created.
- **CMD** does not get executed while building an image.
- Can be overridden at runtime.

RUN

It executes a command in a new layer on top of the current image and commits the results.

COPY

The **COPY** instruction copies new files or directories from **<src>** and adds them to the file system of the container at the path **<dest>**.

ENTRYPOINT

This helps you to configure the container as an executable; it is similar to **CMD**. There can be at max one instruction for **ENTRYPOINT**; if more than one is specified, only the last one will be honored.

WORKDIR <path>

This sets the working directory for the **RUN**, **CMD**, and **ENTRYPOINT** instructions that follow it.

EXPOSE

This exposes the network ports on the container, on which it will listen at runtime.

ENV

This will set the environment variables **<key>** to **<value>** within the container. When a container is running from the resulting image, it will pass and persist all the information to the application running inside the container.

Build a Docker image

Using the **docker build** command, users can automate a docker build that executes several command-line instructions in succession.

The **docker build** command builds an image from a Dockerfile and a context:

```
docker build -t ImageName:TagName dir
```

Where:

- **-t**: Image tag
- **ImageName**: The name you want to give to your image
- **TagName**: The tag you want to give to your image
- **dir**: The directory where the Dockerfile is present

For the current directory, simply use . (period):

```
sudo docker build -t myimage:v1 .
```

Check the newly created image using the **docker images** command.

The next step is to build the container from the newly created image.

Note: To check all the commands run against that image, execute the following command **docker history [Image_id]**

Run a Docker container

A Docker container is a runtime instance of a Docker image. The **docker run** command can run the Docker image as follows:

```
docker run --name test -it myimage:v1
```

Where:

- **-it**: It is used to mention that you want to run the container in interactive mode.
- **--name**: It is used to give a name to the container.
- **myimage**: It is the image name that is to be run.
- **v1**: It is the tag of the image.

The **docker inspect [container_id]** command will populate the complete information of the container in JSON format.

The **docker top [container_id]** command will show top-level processes within a container.

The following figure shows the lifecycle and flow of the Docker container:

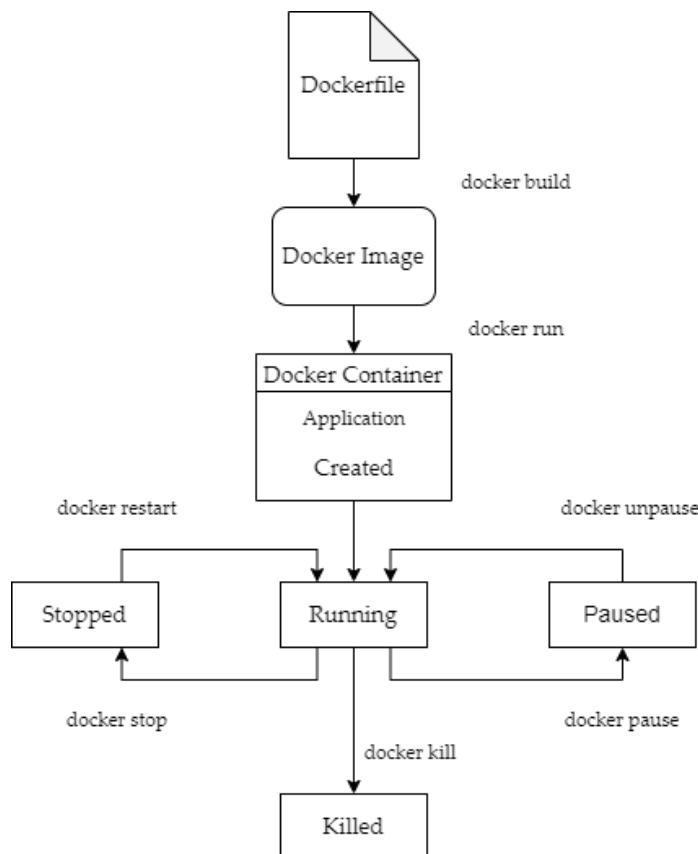


Figure 6.7: Docker container lifecycle

Dockerize and deploy the ML model

Let's consider the scenario of loan prediction, where you need to predict whether a customer's loan will be approved.

```
1. # Importing the required packages  
2. import pandas as pd  
3. import numpy as np  
4. import warnings  
5. warnings.filterwarnings('ignore')  
6. from sklearn.linear_model import LogisticRegression  
7.  
8. from sklearn import preprocessing  
9. from sklearn.model_selection import train_test_split  
10. from sklearn import metrics  
11. import joblib
```

After that, load the data sets and capture numerical and categorical column names in separate variables for the next step:

```
1. # Loading the data  
2. data = pd.read_csv("loan_dataset.csv")  
3. # Missing value treatment (if found)  
4. num_col = data.select_dtypes(include=['int64','float64']).columns.tolist()  
5. cat_col = data.select_dtypes(include=['object']).columns.tolist()  
6. cat_col.remove('Loan_Status')
```

Handle missing values for categorical and numerical columns:

```
1. for col in cat_col:  
2.     data[col].fillna(data[col].mode()[0], inplace=True)  
3.  
4. for col in num_col:  
5.     data[col].fillna(data[col].median(), inplace=True)
```

Clip extreme values for numerical data:

```
1. # Outlier treatment (if found)
2. data[num_col] = data[num_col].apply(
3.     lambda x: x.clip(*x.quantile([0.05, 0.95])))
```

Create a new feature as **TotalIncome**, which is the sum of applicant income and co-applicant income:

```
1. # Creating a new variable
2. data['TotalIncome'] = data['ApplicantIncome'] +
   data['CoapplicantIncome']
3. data = data.drop(['ApplicantIncome', 'CoapplicantIncome'], axis=1)
```

Convert categorical to numeric columns using the label encoding technique:

```
1. cat_col.remove('Loan_ID')
2.
3. # Encoding categorical features
4. for col in cat_col:
5.     le = preprocessing.LabelEncoder()
6.     data[col] = le.fit_transform(data[col])
7.
8. data['Loan_Status'] = le.fit_transform(data['Loan_Status'])
9.
10. # Model building
11. X = data.drop(['Loan_Status', 'Loan_ID'], 1)
12. y = data.Loan_Status
13. features = X.columns.tolist()
14.
15. model = LogisticRegression(solver='lbfgs', max_iter=1000, random_
state=1)
16. model.fit(X, y)
17.
18. joblib.dump(model, 'LR_model.pkl')
```

Now, create a menu-driven prediction inside the while loop, which will take the user input and make the predictions:

```
1. # Menu driven
2.
3. print("Type 'exit' to terminate.....\n")
4. print('''Gender: Female = 0, Male=1
5. Married: No = 0, Yes = 1
6. Education: Graduate = 0 , Under-graduate = 1
7. Self_Employed: No = 0, Yes = 1
8. Property_Area: Urban = 2, Semiurban = 1, Rural = 0
9. Loan_Status: No = 0, Yes = 1\n''')
10.
11.print('''Pass the data in following sequence separated by comma
12.Gender, Married, Dependents,Education,Self_
Employed,LoanAmount,Loan_Amount_Term,Credit_History,Property_
Area,TotalIncome\n''')
13.
14.# model = joblib.load('LR_model.pkl')
15.
16.while True:
17.    user_data=input("Enter your data: ")
18.
19.    if(user_data=="exit"):
20.        break
21.
22.    data = list(map(float, user_data.split(',')))
23.
24.    # exception handling
25.    if(len(data)<10):
26.        print("Incomplete data provided!!")
27.    else:
```

```
28.  
29.      # predicting the value  
30.      predicted_value=model.predict([data])  
31.      print("/_____/")  
32.      if (predicted_value[0]):  
33.          print("\tCongratulations! your loan approval request is processed")  
34.      else:  
35.          print("\tSorry! your loan approval request is rejected")  
36.      print("/_____/")
```

Next, create a *requirements.txt* file of dependencies.

```
1. numpy==1.19.5  
2. pandas==1.1.5  
3. scikit-learn==0.24.2  
4. joblib==1.0.1
```

Now, create a *Dockerfile* for the preceding application with dependencies.

```
1. # STEP 1: Install base image. Optimized for Python  
2. FROM python:3.7-slim-buster  
3.  
4. # STEP 2: Upgrading pip  
5. RUN pip install --upgrade pip  
6.  
7. # STEP 3: Copying all the files to the app directory  
8. COPY . /app  
9.  
10.# STEP 4: Set the working directory to the previously added app  
directory  
11.WORKDIR /app  
12.  
13.# STEP 5: Giving permissions to python file  
14.RUN chmod +x train.py
```

```

15.

16. # STEP 6: Install required python dependencies from the requirements
file

17. RUN pip install -r requirements.txt

18.

19. # STEP 7: Run the train.py file

20. ENTRYPOINT ["python"]

21.

22. CMD ["train.py"]

```

When you have all the files at one place, you can start building the Docker image using the **docker build** command, as shown in the following figure:

```

~/docker $ 
  sudo docker build -t loan-pred:v1 .
[sudo] password for suhas:
Sending build context to Docker daemon 45.57kB
Step 1/8 : FROM python:3.7-slim-buster
3.7-slim-buster: Pulling from library/python
15115158dd02: Pull complete
4d445d10bda3: Pull complete
8f0ddad46ba4: Pull complete
7203f3e1b87f: Pull complete
3ccf9777e1da: Pull complete
Digest: sha256:27d7470d6f2c317558d53197778816a3f7e973b36d5afe34228b90c1130d232b
Status: Downloaded newer image for python:3.7-slim-buster
    --> 6b35aa4d7178

```

Figure 6.8: Docker image build started

This may take a few minutes to complete. After completion, you should see the last message, as shown in the following figure:

```

Successfully built 3eef29e04342
Successfully tagged loan-pred:v1
~/docker $

```

Figure 6.9: Docker image built

The next step is to run the image to start the Docker container instance using the **docker run** command. Finally, it's time to make the prediction. Add a brief description before taking the input from the user and, in this case, input data needs to pass in the sequence given in the description.

Execute the **docker run** command with the **-it** flag to run the image in an interactive mode, as shown in the following figure:

```

~/docker $ 
  sudo docker run -it loan-pred:v1
Type 'exit' to terminate.....

Gender: Female = 0, Male=1
Married: No = 0, Yes = 1
Education: Graduate = 0 , Under-graduate = 1
Self_Employed: No = 0, Yes = 1
Property_Area: Urban = 2, Semiurban = 1, Rural = 0
Loan_Status: No = 0, Yes = 1

Pass the data in following sequence seperated by comma
Gender, Married, Dependents,Education,Self_Employed,LoanAmount,Loan_Amount_Term,
Credit_History,Property_Area,TotalIncome

Enter your data: 1.0,0.0,0.0,0.0,0.0,273,360.0,1.0,2.0,5429
/_____
               Congratulations! your loan approval request is processed
/_____
Enter your data: exit
~/docker $

```

Figure 6.10: Docker container - ML model prediction

To come out of the application, the user can pass **exit**; this will break the **while** loop.

Common Docker commands

Just run the **docker** command in the terminal, and you will get the list of commonly used Docker commands. You can also refer to the exhaustive list of Docker CLI commands at <https://docs.docker.com/engine/reference/commandline/docker/>.

Conclusion

A Docker container runs in an isolated environment. You learned the importance of the Docker framework in the ML lifecycle and production environment. In this chapter, you explored the role, functionality, and usage of Docker components, with examples. You also learned how to create a Dockerfile and build and run an image in a container. You learned to create and deploy an ML model using Docker in interactive mode.

The next chapter begins with REST APIs and explains how to use API for deploying ML models. Then, it will cover web frameworks, and finally, you will learn to build a UI for ML model API.

Points to remember

- Docker is a containerization platform for packaging applications and their dependencies in the form of a container.

- Docker ensures reproducibility, portability, easy deployment, granular updates, lightness, and simplicity.
- Docker manages data within the Docker container using Docker Volumes.
- An instance of a container gets created when you run the Docker image.

Multiple choice questions

1. **What is the command to check the history of the Docker image?**
 - a) docker history [Image_id]
 - b) docker all commands [Image_id]
 - c) docker hist [Image_id]
 - d) history [Image_id]
2. **Which one of the following sentences is incorrect?**
 - a) Docker host comprises multiple Docker containers.
 - b) A Docker container is a runtime instance of a Docker image.
 - c) The command to check running containers is `docker ps`.
 - d) A Docker container instance cannot be created using a Docker image.

Answers

1. a
2. d

Questions

1. What is Docker?
2. What is the role of *Dockerfile*?
3. What is the command to build the Docker image from *Dockerfile*?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Build ML Web Apps Using API

Introduction

When developing a web app in Python, you are very likely to use a framework for it. A framework is a library that eases the life of a developer while building scalable, standard, and production-ready web applications.

This chapter will begin with REST APIs and how to use APIs for deploying ML models. Moving on, it will cover different web frameworks and finally, illustrate the steps to build a UI for ML model API. Toward the end of this part, you should know how to deploy an ML web app.

Structure

In this chapter, the following topics will be discussed:

- REST APIs
- FastAPI
- Streamlit
- Flask
- Build ML Web App

Objectives

After studying this chapter, you should be able to build and deploy ML-based web apps. You should also be able to create an API for your ML model and call it by passing the parameters. Additionally, this chapter will teach you to create web applications using FastAPI, Streamlit, and Flask frameworks. It will cover integrating NGINX and Gunicorn with Flask to create automated ML-based web applications, and you should be comfortable developing web apps in any framework out of the three most commonly used frameworks: FastAPI, Streamlit, and Flask.

REST APIs

REST is an acronym for **R**epresentational **S**tate **T**ransfer. REST is an architectural style mainly created to guide the development and design of the architecture for the World Wide Web (WWW). In simple words, a web service or web API following REST architecture is a REST API.

REST is a pattern to make APIs that can be used to access resources like images, videos, text, JSON, and XML hosted on the server. RESTful API provides a common platform for communicating between applications built in different programming languages. Refer to *Figure 7.1*:

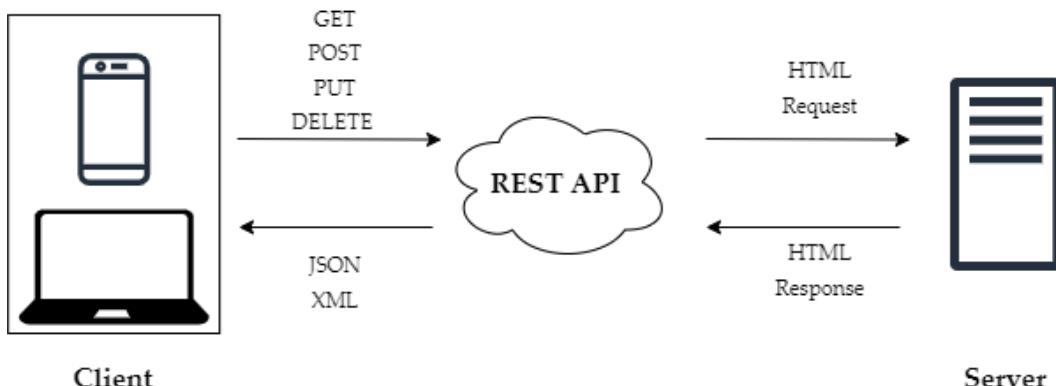


Figure 7.1: REST API

The interesting thing about the API is that the client does not need to know the internal operations performed at the server's end and vice versa. REST API treats any data requested/processed by the user as a resource; it can be text, image, video, and so on.

REST API is stateless; it means that the client should provide all the parameters in the request every time the API is called. The server will not store previous parameters passed with the request by the client.

FastAPI

FastAPI is a web framework for developing RESTful APIs in Python. FastAPI is a lightweight (compared to Django), easy-to-install, easy-to-code yet high-performing framework. It enables the development of REST API with a minimal code requirement. FastAPI comes with built-in standard and interactive documentation. Once you develop and run the API, you can access the documentation for your application at `{API endpoint}/docs` or `{API endpoint}/redoc`.

Note: FastAPI requires Python 3.6 and above

Let's install FastAPI and uvicorn, an **Asynchronous Server Gateway Interface (ASGI)** server, for production.

```
pip install fastapi uvicorn
```

Let's consider a loan prediction scenario, where the goal is to predict whether a customer's loan will be approved. The focus will not be on hyperparameter tuning and model optimization. However, you can optimize a model to improve its overall performance.

1. `# Importing the required packages`
2. `import pandas as pd`
3. `import numpy as np`
4. `import warnings`
5. `warnings.filterwarnings('ignore')`
6. `from sklearn.ensemble import RandomForestClassifier`
7.
8. `from sklearn import preprocessing`
9. `from sklearn.model_selection import train_test_split`
10. `from sklearn import metrics`
11. `import pickle`

Load the datasets, and capture numerical and categorical column names in separate variables for the next step:

1. `# Loading the data`
2. `data = pd.read_csv("loan_dataset.csv")`
3. `# Capturing numerical and categorical column names`

```

4. num_col = data.select_dtypes(include=['int64','float64']).columns.tolist()
5. cat_col = data.select_dtypes(include=['object']).columns.tolist()
6. cat_col.remove('Loan_Status')

```

To handle missing values, you can replace categorical missing values with mode and numerical missing values with the median.

```

1. # Missing value treatment (if found)
2. for col in cat_col:
3.     data[col].fillna(data[col].mode()[0], inplace=True)
4.
5. for col in num_col:
6.     data[col].fillna(data[col].median(), inplace=True)

```

Instead of removing extreme values (outliers) from numerical data, you can cap them at 5% on the lower side and 95% on the upper side.

In other words, data points lower than the 5% quantile will be replaced by a value at the 5% quintile, and on the other hand data points higher than the 95% quantile will be replaced by a value at the 95% quintile.

```

1. # Outlier treatment (if found)
2. data[num_col] = data[num_col].apply(
3.     lambda x: x.clip(*x.quantile([0.05, 0.95])))

```

Create a new feature, **TotalIncome**, which is the sum of applicant income and co-applicant income.

```

1. # Creating a new variable
2. data['TotalIncome'] = data['ApplicantIncome'] +
    data['CoapplicantIncome']
3. data = data.drop(['ApplicantIncome', 'CoapplicantIncome'], axis=1)

```

Convert categorical to numeric columns using the label encoding technique. The target column **Loan Status** will be encoded into numeric.

```

1. cat_col.remove('Loan_ID')
2.
3. # Encoding categorical features

```

```
4. for col in cat_col:  
5.     le = preprocessing.LabelEncoder()  
6.     data[col] = le.fit_transform(data[col])  
7.  
8. data['Loan_Status'] = le.fit_transform(data['Loan_Status'])
```

Now, you will build an ML model using a random forest classifier and will store the trained models in a pickle object. Here, you can use other ML algorithms and different ML techniques, like GridSearchCV, to improve accuracy. Here, you will build a baseline ML model.

```
1. # Model building  
2. X = data[['Gender', 'Married', 'TotalIncome', 'LoanAmount', 'Credit_History']]  
3. y = data['Loan_Status']  
4. features = X.columns.tolist()  
5.  
6. model = RandomForestClassifier(max_depth=4, random_state = 10)  
7. model.fit(X, y)  
8.  
9. # saving the model  
10.pickle_model = open("trained_model/model_rf.pkl", mode = "wb")  
11.pickle.dump(model, pickle_model)  
12.pickle_model.close()  
13.  
14.# Loading the trained model  
15.pickle_model = open('trained_model/model_rf.pkl', 'rb')  
16.model_rf = pickle.load(pickle_model)  
17.  
18.prediction = model_rf.predict([[1, 1, 6000, 150, 0]])  
19.print(prediction)
```

Next, create a *requirements.txt* file for the dependencies.

1. numpy==1.19.5
2. pandas==1.1.5
3. scikit-learn==0.24.2

Now, create a *loan_pred_app.py* file. First, load the dependencies and pickle the object of the trained model. Furthermore, create a FastAPI instance and assign it to the app. This will make the app a point of interaction while creating the API.

1. *# Importing Dependencies*
2. from fastapi import FastAPI
3. from pydantic import BaseModel
4. import uvicorn
5. import pickle
6. import numpy as np
7. import pandas as pd
- 8.
9. app = FastAPI()
10. *# Loading the trained model*
11. trained_model = 'trained_model/model_rf.pkl'
12. model = pickle.load(open(trained_model, 'rb'))

Create a class **LoanPred** that defines the input data type expected from the client.

You will use the **LoanPred** class for the data model that will define the data type expected from the users. It is inherited from **BaseModel**. Then add root view with the function that returns '**message**': '**Loan Prediction App**' on the home page.

1. class LoanPred(BaseModel):
2. Gender: float
3. Married: float
4. ApplicantIncome: float
5. LoanAmount: float
6. Credit_History: float
- 7.

```
8. @app.get('/')
9. def index():
10.    return {'message': 'Loan Prediction App'}
```

The following function will create the UI for user input. Here, the **/predict** class is created as an endpoint, also known as a route. Then, pass the data model, that is, the **LoanPred** class, to the **predict_loan_status()** function as a parameter.

```
1. # Define the function, which will make the prediction using the
   input data provided by the user
2. @app.post('/predict')
3. def predict_loan_status(loan_details: LoanPred):
4.    data = loan_details.dict()
5.    gender = data['Gender']
6.    married = data['Married']
7.    income = data['ApplicantIncome']
8.    loan_amt = data['LoanAmount']
9.    credit_hist = data['Credit_History']
10.
11.   # Make predictions
12.   prediction = model.predict([[gender,married,income,loan_
      amt,credit_hist]])
13.
14.   if prediction == 0:
15.     pred = 'Rejected'
16.   else:
17.     pred = 'Approved'
18.
19.   return {'status':pred}
20.
21. if __name__ == '__main__':
22.   uvicorn.run(app, host='127.0.0.1', port=8000)
```

Now, it's time to run the app and see standard UI auto-generated by FastAPI, which uses swagger, now known as openAPI.

```
uvicorn loan_pred_app:app --reload
```

The preceding command can be interpreted as follows:

- **loan_pred_app** refers to the name of the file where the API is created.
- The **app** is the instance defined in it.
- **--reload** will simply restart the FastAPI server every time a change is made in the app file.

The following figure shows the terminal output after running the preceding command:

```
~/webapi/fastapi$ uvicorn loan_pred_app:app --reload
INFO: Will watch for changes in these directories: ['/home/suhas/webapi/fastapi']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [20063] using statreload
INFO: Started server process [20065]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Figure 7.2: Running the FastAPI app in terminal

Once you see the **Application startup complete** on the terminal, open the browser and go to the path mentioned in the terminal. In this case, it is **127.0.0.1:8000**.

The following figure shows the UI of the FastAPI app in the browser. Here, the text message **Loan Prediction App** is displayed.



Figure 7.3: FastAPI app in terminal

The following figure shows the Swagger (openAPI) UI of the FastAPI app on the browser. Simply add **/docs** to the end of the URL, and you should see auto-generated docs for the app.

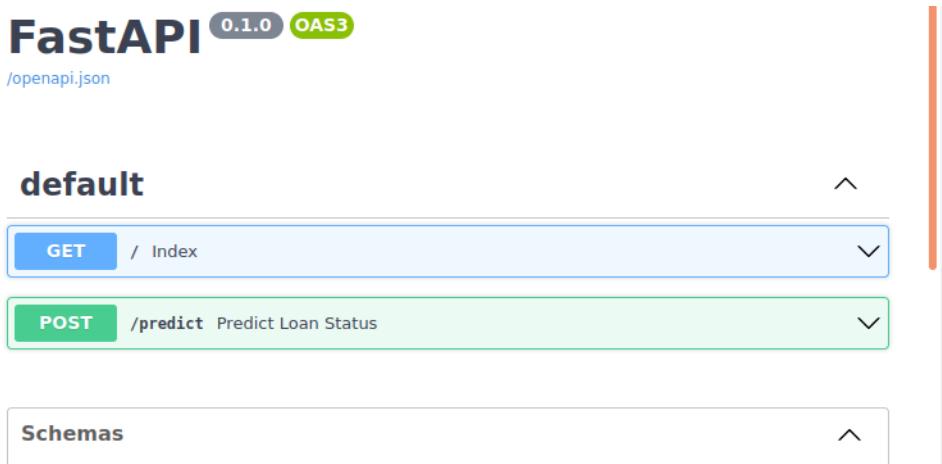


Figure 7.4: Docs of the FastAPI app

FastAPI provides the functionality to validate the data type. It detects the invalid data type at runtime and returns the bad input to the client, which eventually reduces the burden of managing exceptions at the developer's end.

The following figure shows the schema and the data type of the data expected from the client. As there are five variables in the mentioned model, and they can be seen in the figure, along with their data type.

This screenshot shows the detailed schema for the POST endpoint "/predict". The browser address bar indicates the URL is 127.0.0.1:8000/docs#/.

The schema for the "LoanPred" model is defined as:

```

LoanPred <-
  Gender*      number
  Title*       number
  Married*     number
  title: Married
  ApplicantIncome* number
  title: Applicantincome
  LoanAmount*   number
  title: Loanamount
  Credit_History* number
  title: Credit History
}

```

Below this, there is a section for "ValidationError" and another for "ModelError".

Figure 7.5: Schema of FastAPI app

The following figure explores the **predict()** of the app. Here, a sample JSON input is entered, which is expected from the client. By default, the values of all the variables are zero in the schema. To pre-test the app, you have to click on the **Try it out** button.

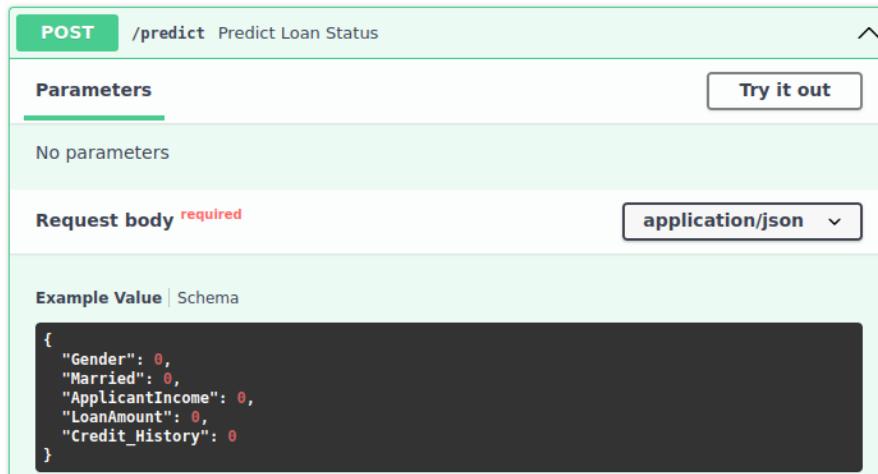


Figure 7.6: *Predict()* of FastAPI app

After clicking on the **Try it out** button, you can pass the variable values or data in JSON format. Figure 7.7 shows the data for given variables being provided by the user. Finally, click on the **Execute** button at the bottom to see the prediction returned by the API.

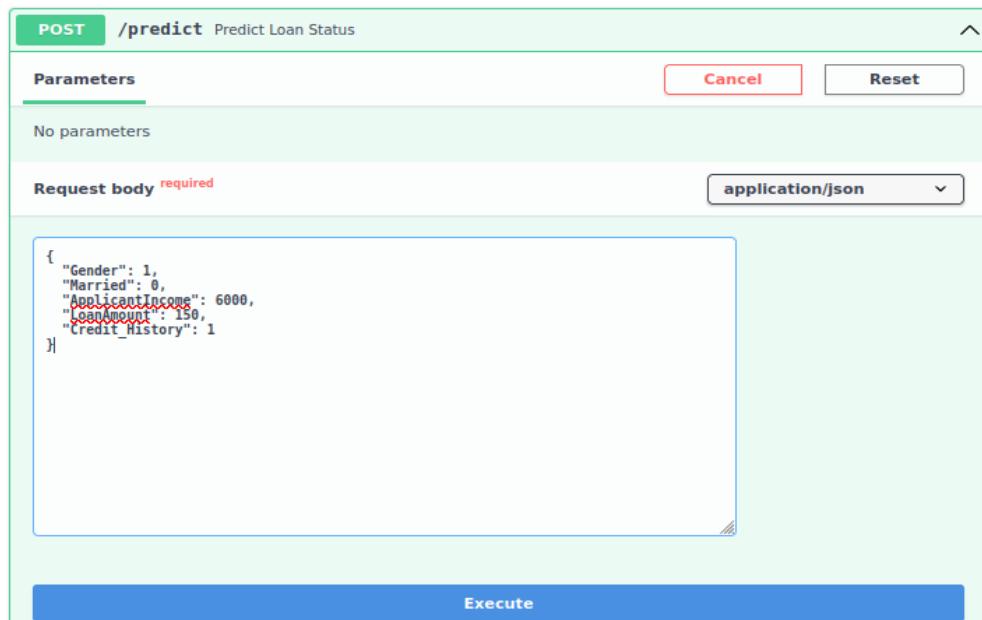
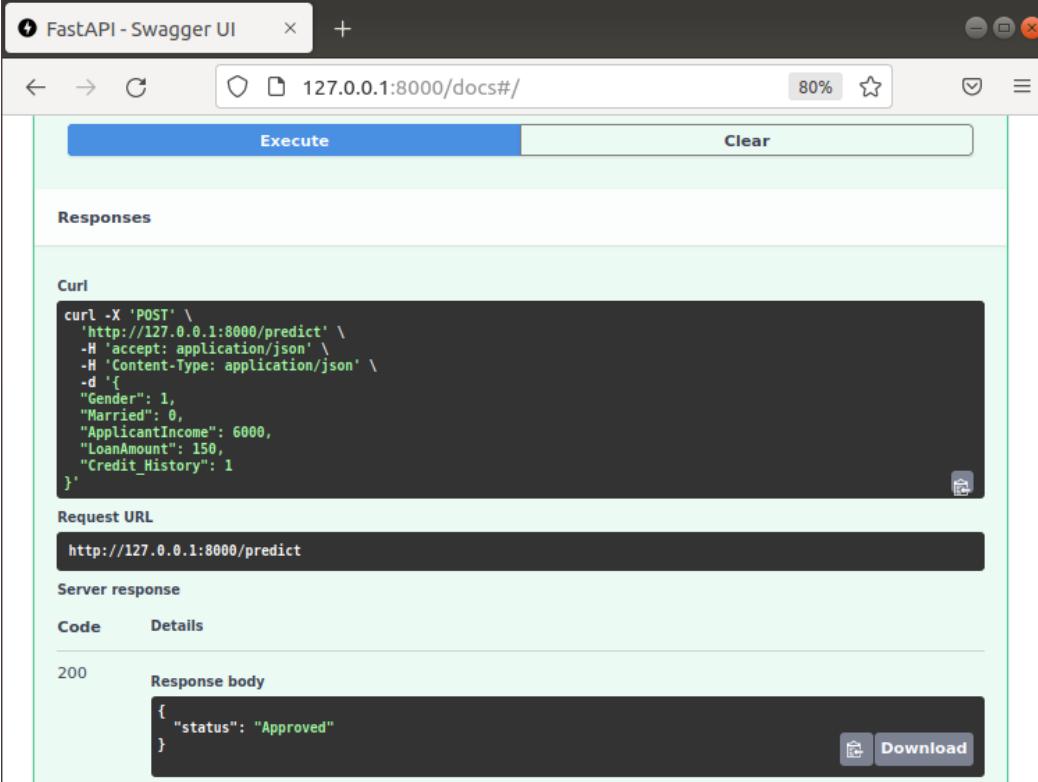


Figure 7.7: Parameters passing in the *predict* function

The following figure shows the predicted outcome under the **Response body** in JSON format. Based on the data provided, the predicted status of the loan is returned as ‘Approved’. Optionally, you can run the `curl` command to get the output in the terminal. Response code **200** shows that the request has succeeded without any error.



The screenshot shows the FastAPI - Swagger UI interface. At the top, there's a header bar with a logo, the title 'FastAPI - Swagger UI', and various icons. Below the header is a browser-style address bar with the URL '127.0.0.1:8000/docs#/' and a zoom level of '80%'. The main area is divided into sections: 'Responses' (which is currently selected), 'Curl' (containing a command-line example), 'Request URL' (containing the URL 'http://127.0.0.1:8000/predict'), and 'Server response' (containing the response details). Under 'Responses', the 'Code' tab is selected, showing a response code of '200'. The 'Details' tab is also visible. The 'Server response' section shows a JSON response body with the key 'status' and the value 'Approved'.

```

Curl
curl -X 'POST' \
  'http://127.0.0.1:8000/predict' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "Gender": 1,
    "Married": 0,
    "ApplicantIncome": 6000,
    "LoanAmount": 150,
    "Credit_History": 1
  }'

Request URL
http://127.0.0.1:8000/predict

Server response

Code Details

200 Response body
{
  "status": "Approved"
}

```

Figure 7.8: Prediction outcome of FastAPI

Streamlit

Streamlit is an open-source library in Python that enables users to build and share attractive UI for machine learning models. It comes with extensive documentation to learn and explore. With streamlit, you can add beautiful and interactive widgets to get the user inputs with a few lines of code, such as a dropdown selection box and a slider to change the values.

According to the makers, streamlit is the fastest way to build ML apps and deploy them on the cloud. It is a great framework to deploy ML apps using Python. In streamlit, everything can be coded in Python without the need for any front-end skills such as JavaScript to develop stunning UI for the app. Streamlit is a framework

that converts the Python code into interactive apps and enables data scientists to build data and model-based apps quickly.

Let's install streamlit using the following command:

```
pip install streamlit
```

Verify the streamlit installation using the following:

```
streamlit hello
```

Now, create a *streamlit_app.py* file for the ML-based streamlit application:

```
1. import pickle  
2. import streamlit as st  
3.  
4. # Loading the trained model  
5. trained_model = 'trained_model/model_rf.pkl'  
6. model = pickle.load(open(trained_model, 'rb'))  
7.  
8. @st.cache()
```

The following function will make the prediction based on input received from the front end:

```
1. # The following function will make the prediction based on data  
provided by the user  
2. def prediction(Gender, Married, ApplicantIncome, LoanAmount,  
Credit_History):  
3.     # Pre-processing user input  
4.     if Gender == "Male":  
5.         Gender = 1  
6.     else:  
7.         Gender = 0  
8.  
9.     if Married == "Unmarried":  
10.        Married = 1  
11.    else:
```

```
12.     Married = 0
13.
14.     if Credit_History == "Unclear Debts":
15.         Credit_History = 1
16.     else:
17.         Credit_History = 0
18.
19.     # Making predictions
20.     prediction = model.predict(
21.         [[Gender, Married, ApplicantIncome, LoanAmount, Credit_
History]])
22.
23.     if prediction == 0:
24.         pred = 'Rejected'
25.     else:
26.         pred = 'Approved'
27.     return pred
```

The following is a function that contains streamlit code, which will be responsible for taking input data from the user and displaying the prediction of the model:

```
1. # The Following function is to define the home page of the streamlit
application
2. def main():
3.
4.     # Front-end view for the app
5.     html_temp = """
6.         <div style ="background-color:green;padding:1px">
7.             <h1 style ="color:black;text-align:center;">
8.                 Loan Prediction App
9.             </h1>
10.            </div>
11.        """
12.
```

```
13.     # display the front-end aspect
14.     st.markdown(html_temp, unsafe_allow_html = True)
15.
16.     # Following code is to create a box field to get user data
17.     Gender = st.selectbox('Gender',("Male","Female"))
18.     Married = st.selectbox('Marital Status',("Unmarried","Married"))
19.     ApplicantIncome = st.number_input("Applicants monthly income")
20.     LoanAmount = st.number_input("Total loan amount")
21.     Credit_History = st.selectbox('Credit_History',("Unclear Debts",
22.                                     "No Unclear Debts"))
23.     result =""
24.
25.     # When Predict button is clicked it will make the prediction and
26.     # display it
27.     if st.button("Predict"):
28.         result = prediction(Gender, Married, ApplicantIncome,
29.                             LoanAmount, Credit_History)
30.         st.success('Your loan is {}'.format(result))
31. if __name__=='__main__':
32.     main()
```

Open the terminal and run the following command where the *streamlit_app.py* file is located:

streamlit run streamlit_app.py

Or

streamlit run streamlit_app.py &>/dev/null&

Figure 7.9 shows the front end of the streamlit app. Here, the prediction is based on the data provided by the user.

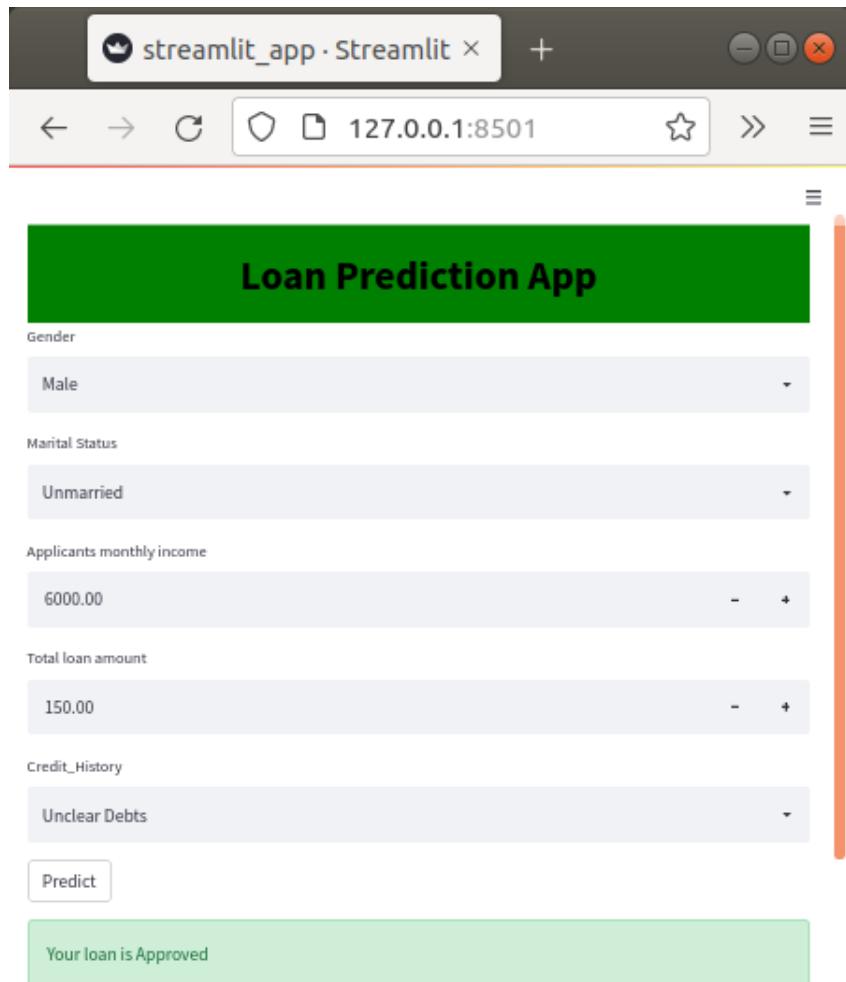


Figure 7.9: Streamlit app

To deploy your Streamlit app on the streamlit cloud, you can use Streamlit sharing. Upload your files with the *requirements.txt* file on GitHub. Create an account on their website at <https://streamlit.io/cloud> and then provide a GitHub link and streamlit app file. This will deploy your app on the streamlit cloud.

Flask

Flask is a web framework that allows you to build web applications using Python. It is a lightweight framework compared to Django. It follows the REST architecture. You can develop simple web applications with Flask, as it requires less base code. Flask is based on the **Web Server Gateway Interface (WSGI)** and Jinja2 engine.

To start a Flask application, you need to use the `run()` function. If you set `debug=True` inside the `run()` function, then it becomes easy to track the error. When you enable debug mode, the server will restart every time you make changes in the app file and save it. If an error occurs, then it shows the reason in the browser itself. However, you should not use this feature while deploying models in the production environment.

In a Flask, you can call static files like CSS or JavaScript files to render the web page of the app. The `route()` function guides the Flask to the URL called by the function.

```
pip install Flask
```

To run the Flask application in the terminal, go to the directory where `app.py` and other required files are located:

```
python app.py
```

Create an `index.html` file with the following HTML code to build the front end for the users. This will enable the users to provide the input data through UI.

```
1. <html>
2.   <head>
3.     <title>LOAN PREDICTION</title>
4.   </head>
5.   <body>
6.     <h2 align="center">LOAN PREDICTION</h2>
7.     <br/>
8.     <form action="/predict" method="POST">
9.       <table align="center" cellpadding = "10">
10.         <!-- First Name -----
11.           -----
12.           <tr>
13.             <td>FIRST NAME</td>
14.             <td><input type="text" name="First_Name" maxlength="30"
placeholder="characters a-z A-Z"/>
15.           </td>
16.           </tr>
17.         <!-- Last Name -----
18.           ----->
```

```
18.    <tr>
19.        <td>LAST NAME</td>
20.        <td>
21.            <input type="text" name="Last_Name" maxlength="30"
placeholder="characters a-z A-Z" />
22.        </td>
23.    </tr>
24.    <!--
Gender----->
25.    <tr>
26.        <td>GENDER</td>
27.        <td>
28.            <input type="radio" name="gender" id="gender"
value="1">Male</input>
29.            <input type="radio" name="gender" id="gender"
value="0">Female</input>
30.        </td>
31.    </tr>
32.    <!--
Marital
Status----->
33.    <tr>
34.        <td>MARRIED</td>
35.        <td>
36.            <input type="radio" name="married" id="married" value="1">
Yes </input>
37.            <input type="radio" name="married" id="married" value="0">
No </input>
38.        </td>
39.    </tr>
40.    <!--
Income ----->
41.    <tr>
42.        <td>TOTAL INCOME</td>
```

```
43.      <td>
44.          <input type="text" name="total_income" maxlength="30"
placeholder="$(thousands)" />
45.      </td>
46.      </tr>
47.      <!--- Loan Amount ----->
48.      <tr>
49.          <td>LOAN AMOUNT</td>
50.          <td>
51.              <input type="text" name="loan_amt" maxlength="30"
placeholder="$(thousands)" />
52.          </td>
53.          </tr>
54.          <!--- Credit History ----->
55.          <tr>
56.              <td>CREDIT HISTORY</td>
57.              <td>
58.                  <input type="radio" name="credit_history" id="credit_
history" value="1"> Yes </input>
59.                  <input type="radio" name="credit_history" id="credit_
history" value="0"> No </input>
60.              </td>
61.          </tr>
62.          <!--- Submit and Reset ----->
63.          <tr>
64.              <td colspan="2" align="center">
65.                  <input type="submit" value="Submit">
66.                  &nbsp;&nbsp; <!--- to add extra space ----->
67.                  <input type="reset" value="Reset" onclick="location.
href='/';">
```

```
68.      </td>
69.      </tr>
70.      </table>
71.      </form>
72.      <h3 align="center"> {{ prediction }} </h3>
73.    </body>
74. </html>
```

Here, the **result.html** file will display the prediction output to the users:

```
1. <!doctype html>
2. <html>
3.   <body>
4.     <h1>{{ prediction }}</h1>
5.   </body>
6. </html>
```

Now, create an *app.py* file for the Flask application:

```
1. # Importing Dependencies
2. from flask import Flask,render_template,url_for,request
3. import pandas as pd
4. from sklearn.ensemble import RandomForestClassifier
5. import numpy as np
6. import pickle
7. import os
```

The next step is to load the serialized model object to make the prediction:

```
1. app = Flask(__name__)
2. port = int(os.environ.get("PORT", 80))
3. # Loading the trained model
4. pickle_in = open('trained_model/model_rf.pkl', 'rb')
5. model = pickle.load(pickle_in)
```

Now, create the views for the Flask application. The following code will render the HTML template, which will be the landing page of the Flask application. `app.route('/')` will route to the home page URL; the trailing slash '`/`' is generally used as a convention for the home page.

```
1. # Views  
2. @app.route('/')  
3. def home():  
4.     return render_template('index.html')
```

Now, create a view for the `predict()` function. `app.route('/predict')` will route to the home page URL. The **POST** method type allows you to send data to a server to update or create a target resource.

```
5. @app.route('/predict',methods=['POST'])  
6. def predict():  
7.     if request.method == 'POST':  
8.         # Fetch Value for Gender  
9.         gender = request.form['gender']  
10.        if gender == "Female":  
11.            gender = int(0.0)  
12.        if gender == "Male":  
13.            gender = int(1.0)  
14.  
15.        # Fetch Value for Married  
16.        married = request.form['married']  
17.        if married == "No":  
18.            married = int(0.0)  
19.        if married == "Yes":  
20.            married = int(1.0)  
21.  
22.        # Fetch value for LoanAmount  
23.        loan_amt = float(request.form['loan_amt'])
```

```
24.  
25.      # Fetch value for Total_Income  
26.      total_income = float(request.form['total_income'])  
27.  
28.      # Fetch value for Prior_Credit_Score  
29.      credit_history = request.form['credit_history']  
30.      if credit_history == "No":  
31.          credit_history = int(0.0)  
32.      if credit_history == "Yes":  
33.          credit_history = int(1.0)  
34.  
35.      to_predict_list = [gender, married, total_income, loan_amt,  
36.                           credit_hi           story]  
37.      prediction_array = np.array(to_predict_list, dtype=np.  
float32).reshape(1, 5)  
38.      # Making Predictions using the trained model  
39.      prediction = model.predict(prediction_array)  
40.      prediction_value = prediction[0]  
41.      # print(prediction_value )  
42.  
43.  
44.      if int(prediction_value) == 1:  
45.          status="Congratulations! your loan approval request is  
        processed"  
46.      if int(prediction_value) == 0:  
47.          status="Sorry! your loan approval request is rejected"  
48.  
49.      return render_template('index.html',prediction = status)
```

Add error handling functionality, as follows:

```
1. @app.errorhandler(500)
2. def internal_error(error):
3.     return "500: Something went wrong"
4.
5. @app.errorhandler(404)
6. def not_found(error):
7.     return "404: Page not found",404
```

Finally, you will define the main function as shown below:

```
1. if __name__ == '__main__':
2.     app.run(host='0.0.0.0', port=port)
```

Gunicorn

The Gunicorn is an application server for running a Python app. Gunicorn is WSGI compatible, so it can communicate with multiple WSGI applications. In the current case, Gunicorn translates the request received from Ngnix for the Flask app and vice versa.

To install Gunicorn, execute the following command:

```
sudo apt-get install gunicorn3
```

Go to the directory where the *app.py* file is located and run:

```
gunicorn3 app:app
```

This command helps to know which IP and port are being used by Gunicorn. In this case, it is **8000** and IP can be **0.0.0.0** or **127.0.0.1**.

NGINX

NGINX is a high-performance, highly scalable open-source, and reverse proxy web server. It can perform load balancing and caching application instances. It accepts incoming connections and decides where they should go next. In the current case, it sits on top of a Gunicorn.

To install NGINX, execute the following command:

```
sudo apt-get install nginx
```

You can check the status of NGINX using the following command:

```
sudo service nginx status
```

Now, go to the following path:

```
cd /etc/nginx/sites-enabled/
```

Note: NGINX configuration files do not have any extension, and every line should be closed using ; (semicolons).

Create a new configuration file for the Flask app:

```
sudo nano flask_app
```

Then, add the following snippet and save the file:

```
1. server{  
2.     listen 80;  
3.     server_name 0.0.0.0;  
4.  
5.     location / {  
6.         proxy_pass http://unix:/home/suhas/webapi/flask/  
flaskapp.sock;  
7.     }  
8. }
```

Here, you are dictating NGINX to listen to port **80**. Inside the location block, pass the request to the socket using **proxy_pass**. After changing the NGINX file, restart the NGINX service using the following command:

```
sudo service nginx restart or sudo systemctl restart nginx
```

You can check the status of NGINX using the following command:

```
sudo service nginx status
```

Go to the directory where the *app.py* file is located and run the following command to start the Gunicorn server:

```
gunicorn3 app:app
```

Open a new tab in the browser and enter the IP in the address bar. You should see the Flask app up and running.

Create and run the service in the background:

Create a **systemd** unit file that allows the Ubuntu boot system to start Gunicorn automatically and serve up the Flask application every time the server starts.

First, go to the *system* directory using the following command:

```
cd /etc/systemd/system
```

Next, you need to create a service for the Flask app

```
sudo nano flaskapp.service
```

Then, add the following commands to it and save it:

```
1. [Unit]
2. Description=Flaskapp Gunicorn Service
3. After=network.target
4.
5. [Service]
6. User=suhas
7. Group=www-data
8. WorkingDirectory=/home/suhas/webapi/flask
9.
10. ExecStart=/usr/bin/gunicorn3 --workers 3 --bind unix:flaskapp.sock -m 007 app:app
11. Restart=on-failure
12. RestartSec=10
13.
14. [Install]
15. WantedBy=multi-user.target
```

In the preceding service, you are instructing Gunicorn to start three worker processes, which can be updated afterward. Then, create and link the UNIX socket file. In the current scenario, **007** is used for access so that the socket file allows access to the owner and group and restricts others. Then, pass the filename of the app.

```
sudo systemctl daemon-reload
```

```
sudo service flaskapp restart
```

Restart the **flaskapp** service by running the preceding command, and you should see *flaskapp.sock* file created in the app directory.

The following figure shows that the request received from the client goes to NGINX first. Next, it passes to the Gunicorn server; the Gunicorn translates and passes it to the Flask app and then back to the client.

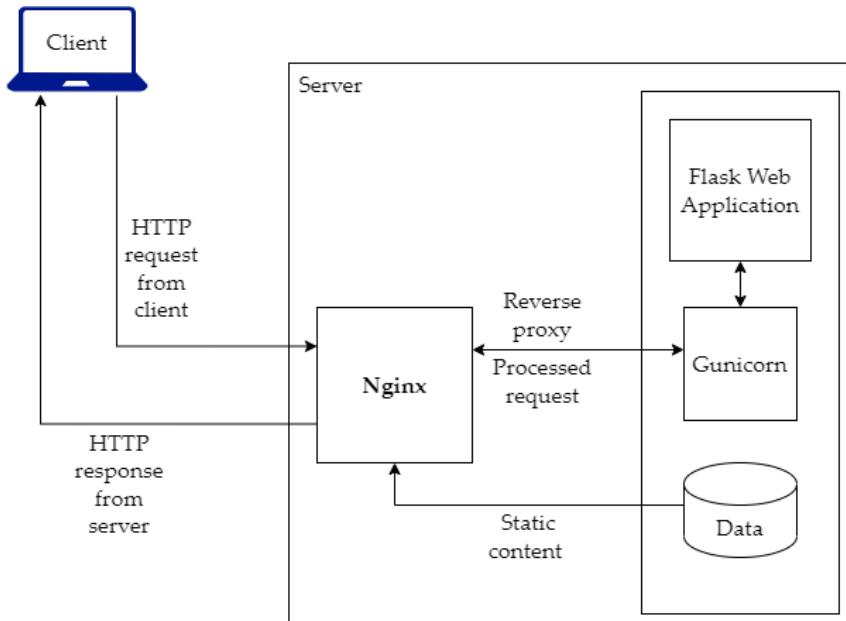


Figure 7.10: Client-server interaction

If everything goes well, open the browser and enter the IP in the address bar. You should see the app up and running.

The following figure shows the **Loan Prediction** app running in the browser. Provide the user data.

LOAN PREDICTION

FIRST NAME	<input type="text" value="Sam"/>
LAST NAME	<input type="text" value="Potter"/>
GENDER	<input checked="" type="radio"/> Male <input type="radio"/> Female
MARRIED	<input type="radio"/> Yes <input checked="" type="radio"/> No
TOTAL INCOME	<input type="text" value="6000"/>
LOAN AMOUNT	<input type="text" value="150"/>
CREDIT HISTORY	<input checked="" type="radio"/> Yes <input type="radio"/> No
<input type="button" value="Submit"/> <input type="button" value="Reset"/>	

Figure 7.11: Passing parameters in the Flask app

The following figure shows the prediction after providing user data and clicking on the **Submit** button.

A screenshot of a web browser window titled "LOAN PREDICTION". The URL bar shows "0.0.0.0/predict". The form fields are as follows:

FIRST NAME	characters a-z A-Z
LAST NAME	characters a-z A-Z
GENDER	<input type="radio"/> Male <input type="radio"/> Female
MARRIED	<input type="radio"/> Yes <input type="radio"/> No
TOTAL INCOME	\$ (thousands)
LOAN AMOUNT	\$ (thousands)
CREDIT HISTORY	<input type="radio"/> Yes <input type="radio"/> No

Below the form are two buttons: "Submit" and "Reset". Underneath the form, a message reads: "Congratulations! your loan approval request is processed".

Figure 7.12: Prediction outcome

Conclusion

This chapter demonstrated how to develop an ML-based web application using the most commonly used frameworks, viz FastAPI, Streamlit, and Flask with NGINX and Gunicorn. You created a user-friendly and simple UI to receive input data from users, and you studied the REST API; it's working between the client and server. This chapter also touched upon the salient features of each framework.

The next chapter will discuss building native applications for PC and Android devices.

Points to remember

- RESTful API provides a common platform for communicating with an application built in different programming languages.
- FastAPI requires Python 3.6 and above.
- Streamlit is an open-source library in Python that enables users to build and share attractive UI for machine learning models.
- FastAPI comes up with a built-in standard and interactive documentation.
- NGINX configuration files do not have any extension and every line should be closed using a ; (semicolon).

Multiple choice questions

1. How can you verify the installation of streamlit?
 - a) streamlit hello
 - b) streamlit hello world
 - c) streamlit run test
 - d) streamlit --
2. Docs of the FastAPI application can be viewed on a browser using which of the following?
 - a) /docs
 - b) /redoc
 - c) Both a and b
 - d) It is not accessible

Answers

1. a
2. c

Questions

1. What is NGINX?
2. What is the role of a Gunicorn?
3. What is the command to run a streamlit app?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 8

Build Native ML Apps

Introduction

In order to consume ML models as a native application, a GUI should be built for the users. In this chapter, you will learn to build native applications using Tkinter and Kivy packages, which can be converted into desktop applications for Windows and Android devices, respectively.

Python allows multiple ways to build a **Graphical User Interface (GUI)**, which is an application that contains the main window, buttons, widgets, input fields, and the like. End users can interact with it and see the response based on their actions and inputs. Python has numerous GUI frameworks or toolkits available.

Structure

This chapter discusses the following topics:

- Introduction to Tkinter
- Build an ML-based app using Tkinter
- Convert Python app into Windows EXE file using Pyinstaller
- Introduction to kivy

- Build an ML-based app using kivy and kivyMD
- Convert Python app into Android app using Buildozer

Objectives

After studying this chapter, you should be able to build and deploy ML-based native apps, such as Windows and Android apps. This chapter is divided into two sections. By the end of the first section, you should be comfortable using Tkinter for developing ML-based Windows apps. You can also convert a Python app to an independent Windows executable file. By the end of the second section, you should be able to develop ML-based kivy applications and convert them into Android apps using the Buildozer package.

Introduction to Tkinter

Tkinter is a free software released under a Python license.

Tkinter is the Python interface to the Tk GUI library. Tk is derived from the **Tool Command Language (TCL)**, which is a scripting language. The best part is that you do not need to install Tkinter explicitly as it comes with Python since 1994.

Tkinter enables developers to create widgets (GUI elements) easily, which are packaged in the Tk toolkit. With widgets, developers can create buttons, menus, and input fields to get user data. These widgets can be linked to Python features, methods, data, or other widgets.

For instance, a button widget can be used to perform a certain action upon click event, or it can call the function and pass the user's data as parameters.

Note: Get the latest version of Tkinter by installing Python 3.7 or a later version.

You can verify the Tkinter installation using the following command:

```
python -m tkinter
```

A pop-up window will appear, as shown below, which contains TCL/Tk version and text as **This should be a cedilla: ç**.

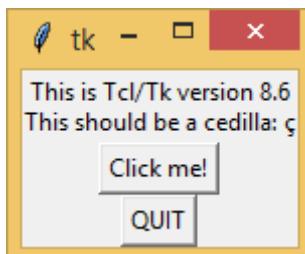


Figure 8.1: Tkinter

Note: A cedilla is a symbol that is written under the letter 'ç' in French, Portuguese, and some other languages to show that you pronounce it like a letter 's'.

Hello World app using Tkinter

Let's create a basic Tkinter app to give you a glimpse of Tkinter functionality. The following code creates a simple app displaying the **Hello World** text with the **Exit** button to close the app.

```
1. from tkinter import *
2. win = Tk()
3.
4. a = Label(win, text ="Hello World")
5. a.pack()
6.
7. b=Button(win, text='Exit', command=tk.destroy)
8. b.pack()
9.
10. win.mainloop()
```

In the preceding code, first, the **tkinter** package is imported to create the main window for the app using the **Tk()**, where the **win** is the name of the main window object that has been created using **Tk()**. A **Label()** widget is used to display text or images, and the **win** parameter is used to denote the parent window of the app.

The **pack()** organizes the widgets in blocks (such as buttons and labels). If you do not organize the widget, your app will still run, but that widget will be invisible. A **Button()** widget is used to add buttons, and here it is used to close the Tkinter app. When the user clicks on the **Exit** button, the Tkinter app window will close. Finally, the **mainloop()** is to keep the application running and execute user commands.

The following figure shows the output of the preceding code. It is a simple Tkinter app with a label and a button.

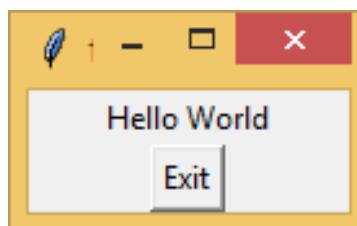


Figure 8.2: Hello World app using Tkinter

Here are some salient features of Tkinter:

- Tkinter is easy to use and quick to develop desktop applications.
- The syntax of Tkinter and its widget is simple to understand.
- Tkinter is shipped with Python, so you do not need to install it explicitly.
- Tkinter is open-source and free to use.

Build an ML-based app using Tkinter

Here, you need to develop the Python code for data loading, data cleaning, feature engineering, model building and finally, saving trained models to serialized objects like pickles. Then you can use trained models to get predictions on the Tkinter app.

Let's consider a loan prediction scenario, where the goal is to predict whether a customer's loan will be approved. The focus will not be on hyperparameter tuning and model optimization. However, you can optimize a model to improve its overall performance. First, create **train.py** file as follows:

```
1. # Import the required packages  
2. import pandas as pd  
3. import numpy as np  
4. import warnings  
5. warnings.filterwarnings('ignore')  
6. from sklearn.ensemble import RandomForestClassifier  
7.  
8. from sklearn import preprocessing  
9. from sklearn.model_selection import train_test_split  
10. from sklearn import metrics  
11. import pickle
```

For the next step, load the datasets and capture numerical and categorical column names in separate variables.

```
1. # Load the data  
2. data = pd.read_csv("loan_dataset.csv")  
3. # Missing value treatment (if found)
```

```

4. num_col = data.select_dtypes(include=['int64','float64']).columns.tolist()
5. cat_col = data.select_dtypes(include=['object']).columns.tolist()
6. cat_col.remove('Loan_Status')

```

To handle missing values, you can replace categorical missing values with mode and numerical missing values with the median.

```

1. # Missing value treatment (if found)
2. for col in cat_col:
3.     data[col].fillna(data[col].mode()[0], inplace=True)
4.
5. for col in num_col:
6.     data[col].fillna(data[col].median(), inplace=True)

```

Instead of removing extreme values (outliers) from numerical data, you can cap them at 5% on the lower side and 95% on the upper side.

In other words, data points lower than the 5% quantile will be replaced by a value at the 5% quintile, and on the other hand data points higher than the 95% quantile will be replaced by a value at the 95% quintile.

```

1. # Outlier treatment (if found)
2. data[num_col] = data[num_col].apply(
3.     lambda x: x.clip(*x.quantile([0.05, 0.95])))

```

There are many instances where a co-applicant income is not available. Create a new feature, **TotalIncome** which is the sum of applicant income and co-applicant income.

```

1. # Create a new variable
2. data['TotalIncome'] = data['ApplicantIncome'] + data['CoapplicantIncome']
3. data = data.drop(['ApplicantIncome', 'CoapplicantIncome'], axis=1)

```

You should avoid passing text data, such as Gender and Married (Marital status), to the model. Convert categorical features to numeric features using the label encoding technique. The target column **Loan_Status** will be encoded into numeric.

```

1. cat_col.remove('Loan_ID')
2.

```

```
3. # Encode categorical features  
4. for col in cat_col:  
5.     le = preprocessing.LabelEncoder()  
6.     data[col] = le.fit_transform(data[col])  
7.  
8. data['Loan_Status'] = le.fit_transform(data['Loan_Status'])
```

Now, you will build an ML model using a random forest classifier and will store the trained models in a pickle object. Here, you can use other ML algorithms and different ML techniques, like GridSearchCV, to improve accuracy. Here, you will build a baseline ML model.

```
1. # Model building  
2. X = data[['Gender', 'Married', 'TotalIncome', 'LoanAmount', 'Credit_History']]  
3. y = data['Loan_Status']  
4. features = X.columns.tolist()  
5.  
6. model = RandomForestClassifier(max_depth=4, random_state = 10)  
7.  
8. model.fit(X, y)  
9.  
10. # save the model  
11. pickle_model = open("trained_model/model_rf.pkl", mode = "wb")  
12. pickle.dump(model, pickle_model)  
13. pickle_model.close()  
14.  
15. # Load the trained model  
16. pickle_model = open('trained_model/model_rf.pkl', 'rb')  
17. model_rf = pickle.load(pickle_model)  
18.  
19. prediction = model_rf.predict([[1, 1, 6000, 150, 0]])  
20. print(prediction)
```

Next, create a *requirements.txt* file of dependencies, as follows:

```
1. pandas==1.1.5  
2. pyinstaller==4.1  
3. scikit-learn==0.24.0
```

Tkinter app

When you complete the preceding part, you should have a pickled object of the trained model, which will be used by the Tkinter app. Here, you will need to import two packages, viz **tkinter** to build GUI and **joblib** to load pickled ML model objects.

Create a Python file and name it *ml_app.py*.

Import the **tkinter** module to create a Tkinter desktop application.

```
1. from tkinter import *  
2.  
3. import joblib  
4.  
5. trained_model = 'E:/tkinter_ml_app/trained_model/model.pkl'  
6. model = joblib.load(trained_model)
```

Define **MyWindow** class for an ML app. In this class, add Tkinter widgets for GUI and the **predict()** function for making predictions based on user input. In the following code, labels for input files have been created using the **Label()** widget.

```
1. class MyWindow:  
2.     def __init__(self, win):  
3.         # Create a text Label  
4.         self.lbl0=Label(win, text="Loan Prediction App", font=(25))  
5.         self.lbl0.pack(pady=10)  
6.         self.lbl1=Label(win, text='Gender')  
7.         self.lbl2=Label(win, text='Married')  
8.         self.lbl3=Label(win, text='Total Income')  
9.         self.lbl4=Label(win, text='Loan Amount')
```

```
10.         self.lbl5=Label(win, text='Credit History')
11.         self.lbl6=Label(win, text='Loan Status')
```

The next step is to declare input boxes using the **Entry()** widget for the preceding fields. In this, declare the optional parameter **bd**, that is, the border inside the **Entry()** widget for the input box.

The **insert()** widget inserts the text at the given position. **0** (zero) is the first character, so it inserts the default text at the beginning.

The syntax of the **bind()** is as follows:

widget.bind(event, handler)

In the current case, an entry widget is linked to a **FocusIn** event using **bind()**. It means a specific entry widget is focused, that is, when the cursor is active in a given entry widget, then it should execute the lambda function defined inside the **bind()**. The lambda function is used to delete the content of the entry widget as soon as the user clicks on it.

Create **Predict** button using **Button()**.

```
1.         # Create an entry widget to accept the user input
2.         self.t1=Entry(bd=2)
3.         self.t1.insert(0, "0:F, 1:M")
4.             self.t1.bind("<FocusIn>", lambda args: self.
t1.delete('0', 'end'))
5.
6.         self.t2=Entry(bd=2)
7.         self.t2.insert(0, "0:No, 1:Yes")
8.             self.t2.bind("<FocusIn>", lambda args: self.
t2.delete('0', 'end'))
9.
10.        self.t3=Entry(bd=2)
11.        self.t3.insert(0, "E.g. 6000")
12.            self.t3.bind("<FocusIn>", lambda args: self.
t3.delete('0', 'end'))
13.
14.        self.t4=Entry(bd=2)
```

```
15.         self.t4.insert(0, "E.g. 150")
16.             self.t4.bind("<FocusIn>", lambda args: self.
t4.delete('0', 'end'))
17.
18.         self.t5=Entry(bd=2)
19.         self.t5.insert(0, "0:Clear Debts, 1:Unclear Debts")
20.             self.t5.bind("<FocusIn>", lambda args: self.
t5.delete('0', 'end'))
21.
22.         self.t6=Entry(bd=2)
23.
24. # Create a Predict button
25.         self.btn1 = Button(win, text='Predict')
```

Tkinter comes with three geometry managers: grid, place, and pack. A geometry manager's job is to arrange widgets in specific positions. The place geometry manager gives you more flexibility compared to the other two geometry managers. You can declare the vertical and horizontal position of the widget. The following code mentions the vertical and horizontal positions of each widget, including the **Predict** button.

```
1.     # Organize widgets appropriately
2.     self.lbl1.place(x=100, y=50)
3.     self.t1.place(x=200, y=50)
4.
5.     self.lbl2.place(x=100, y=100)
6.     self.t2.place(x=200, y=100)
7.
8.     self.lbl3.place(x=100, y=150)
9.     self.t3.place(x=200, y=150)
10.
11.    self.lbl4.place(x=100, y=200)
12.    self.t4.place(x=200, y=200)
```

```
13.  
14.         self.lbl5.place(x=100, y=250)  
15.         self.t5.place(x=200, y=250, width=165)  
16.  
17.             self.b1=Button(win, text='Predict', command=self.  
predict, fg='blue')  
18.             self.b1.place(x=170, y=300)  
19.  
20.             self.lbl6.place(x=100, y=350)  
21.             self.t6.place(x=200, y=350)
```

Finally, define the `predict()` to make the prediction using the ML model based on user inputs.

```
1.  # For making predictions  
2.  def predict(self):  
3.      self.t6.delete(0, 'end')  
4.      gender = float(self.t1.get())  
5.      married = float(self.t2.get())  
6.      income = float(self.t3.get())  
7.      loan_amt = float(self.t4.get())  
8.      credit_hist = float(self.t5.get())  
9.      prediction = model.  
predict([[gender, married, income, loan_amt, credit_hist]])[0]  
10.     if prediction == 0:  
11.         pred = 'Rejected'  
12.     else:  
13.         pred = 'Approved'  
14.  
15.     self.t6.insert(END, str(pred))
```

The class declaration is done here.

Use the **Tk** class to create the main window and call the **mainloop()** to keep the window displayed.

Also, the application window does not appear on the screen till **mainloop()** is not called, as it takes all the objects and widgets and renders them on screen.

```

1. window=Tk()
2.
3. mywin=MyWindow(window)
4.
5. # Create a title
6. window.title('ML App')
7.
8. # Define the size of the window
9. window.geometry("400x400+10+10")
10.
11.window.mainloop() #Keep the window displaying

```

Create a *ml_app.py* file for the Tkinter app and run it. It should make predictions based on the given input.

Convert Python app into Windows EXE file

Now, you can convert this Tkinter - ML app to Windows executable application using Pyinstaller, with the following steps. Pyinstaller packages a Python application and its dependencies into a single package that can be run independently without installing a Python interpreter.

Let's create a virtual environment and install all the required dependencies.

Here, a virtual environment is created with Python version 3.7 using conda, as follows:

```
conda create -n venv_tkinter python=3.7 -y
```

Once it is created, activate it using:

```
source activate venv_tkinter
```

Note: To remove the conda environment, you can execute the following command
conda env remove -n venv_tkinter:

```
pyinstaller --noconfirm --onefile --windowed --icon "E:/tkinter_ml_app/
python_104451.ico" --add-data "E:/tkinter_ml_app/trained_model
/model.pkl;." --hidden-import "sklearn" --hidden-import "sklearn.
ensemble._forest" --hidden-import "sklearn.neighbors._typedefs"
--hidden-import "sklearn.utils._weight_vector" --hidden-import "sklearn.
neighbors._quad_tree" "E:/tkinter_ml_app/ml_app.py"
```

The preceding command will convert the Python-based Tkinter app to a Windows executable file that can be used for prediction.

Where:

- **-y, --noconfirm**: Will replace output directory without asking for confirmation
- **-F, --onefile**: Will create one file bundle executable
- **-D, --onedir**: Will create one folder bundle containing the Windows executable file
- **--add-data**: Additional non-binary files or folders to be added to the executable
- **-c, --console, --nowindowed**: Opens a console window for standard i/o (default)
- **-w, --windowed, --noconsole**: Used to not provide a console window for standard i/o
- **--hidden-import MODULENAME**: Used to name an import not visible in the code of the script(s); this option can be used multiple times

For more details, refer to:

<https://pyinstaller.org/en/stable/usage.html>

The following figure shows the status message after executing the preceding command. It states the EXE file creation success message.

```
178062 INFO: Appending archive to EXE E:\tkinter_ml_app\dist\ml_app.exe
178236 INFO: Building EXE from EXE-00.toc completed successfully.
(venv_tkinter)
$ | Suhas/e/tkinter_ml_app
```

Figure 8.3: Converting Python app to Windows app

Now, open the *ml_app.exe* file located in the *dist* folder.

The following figure shows the files and folders of the ML-based Tkinter app:

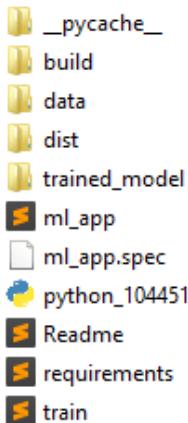


Figure 8.4: Tkinter ML app

Open the app; you should see the window of the Tkinter app. It should display the prediction based on user input.

The following figure displays the prediction based on user data provided:

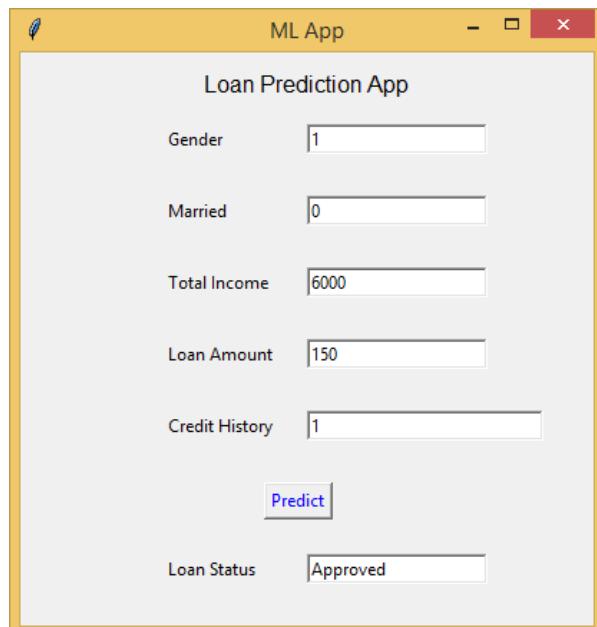


Figure 8.5: Tkinter ML app

Build an ML-based app using kivy and kivyMD

In this section, using the pickle object of the trained model generated earlier in this chapter, let's develop and deploy the FastAPI endpoint on the remote server, i.e., Heroku. You will learn how to deploy ML models on the Heroku platform with CI/CD pipeline in the upcoming chapter.

Moreover, you can deploy this FastAPI app using any other hosting service, such as PythonAnywhere or Amazon EC2. In the current scenario, a remote endpoint is used to make predictions in the Kivy app.

So, whenever users pass the data in the kivy app and press the **Predict** button, it will send the parameters to the API endpoint, where it will pass the received parameters to the model object and in return, send the prediction to API. The Kivy app will fetch the output (prediction) provided by the API.

Let's install FastAPI and uvicorn (an **ASGI (Asynchronous Server Gateway Interface)** server for production) as shown below:

```
pip install fastapi uvicorn
```

Note: FastAPI requires Python 3.6+

Now, create an *app.py* file. First, load the dependencies and pickle the object of the trained model. Also, create a FastAPI instance and assign it to the app. So, the app will be the point of interaction while creating API.

```
1. # Import dependencies
2. from fastapi import FastAPI
3. from pydantic import BaseModel
4. import uvicorn
5. import pickle
6. from fastapi.middleware.cors import CORSMiddleware
7.
8. app = FastAPI()
9.
10. origins = ["*"]
11.
```

```
12. app.add_middleware(  
13.     CORSMiddleware,  
14.     allow_origins=origins,  
15.     allow_credentials=True,  
16.     allow_methods=["*"],  
17.     allow_headers=["*"],)  
18.  
19. # Load the trained model  
20. trained_model = 'trained_model/model_rf.pkl'  
21. # pickle_in = open(trained_model, 'rb')  
22. model = pickle.load(open(trained_model, 'rb'))
```

Define the class **LoanPred**, which defines the datatype expected from the client.

The **LoanPred** class is used for the data model that is inherited from **BaseModel**. Add a root view of the function that returns '**message**': '**Loan Prediction App**' for the home page.

```
1. class LoanPred(BaseModel):  
2.     Gender: float  
3.     Married: float  
4.     ApplicantIncome: float  
5.     LoanAmount: float  
6.     Credit_History: float  
7.  
8.     @app.get('/')  
9.     def index():  
10.        return {'message': 'Loan Prediction App'}
```

The following function will create a UI for user input. Here, create a **/predict** as an endpoint, also known as the route. Then, add the parameter of the type data model created, which is **LoanPred**.

```
1. # Defining the function which will make the prediction using the data which the user inputs
2. @app.post('/predict_status')
3. def predict_loan_status(loan_details: LoanPred):
4.     data = loan_details.dict()
5.     gender = data['Gender']
6.     married = data['Married']
7.     income = data['ApplicantIncome']
8.     loan_amt = data['LoanAmount']
9.     credit_hist = data['Credit_History']
10.
11.    # Making predictions
12.    prediction=model.predict([[gender,    married,    income,    loan_
amt,    credit_hist]])
13.
14.    if prediction == 0:
15.        pred = 'Rejected'
16.    else:
17.        pred = 'Approved'
18.
19.    return {'status':pred}
20.
21. @app.get('/predict')
22. def get_loan_details(gender: float, married: float, income: float,
23.                      loan_amt: float, credit_hist: float):
24.     prediction = model.predict([[gender,    married,    income,    loan_
amt,    credit_hist]]).tolist()[0]
25.     print(prediction)
26.     if prediction == 0:
27.         pred = 'Rejected'
28.     else:
```

```
29. pred = 'Approved'  
30.  
31. return {'status':pred}  
32.  
33.if __name__ == '__main__':  
34. uvicorn.run(app, host='0.0.0.0', port=4000)
```

Now, it's time to run the app and see standard UI auto-generated by FastAPI, which uses swagger, now known as openAPI.

```
uvicorn app:app --reload
```

The preceding command can be interpreted as follows:

- The **app** refers to the name of the file in which the API is created.
- The **app** is the instance defined in it or a file.
- **--reload** will simply restart the FastAPI server every time changes are made in the app file.

KivyMD app

Kivy is an open-source software library for the quick development of apps with a simple GUI. Kivy is written in Python, so in order to use it, you should have Python installed on your system. Unlike Tkinter, it is not pre-installed with Python; you need to install it separately.

KivyMD is an extension of the kivy framework. It is a collection of Material Design (MD) widgets and is mainly used for GUI building along with kivy. It is a good idea to create a virtual environment first, as it helps maintain different packages easily. You can install kivy and KivyMD using the following commands:

```
pip install kivy  
pip install kivymd
```

Use the deployed endpoint on the remote server. Keep it simple for now. Import the required dependencies first. Here, import **MDApp**, which is a base app. Build the app on top of this base app as it takes care of initialization and booting up the app.

Then, import the **Builder** function for the front end of the app.

Now create a file named *main.py* and add the following code to it. First, import the required dependencies into it.

```
1. from kivymd.app import MDApp  
2. from kivy.lang.builder import Builder  
3. from kivy.uix.screenmanager import Screen, ScreenManager  
4. from kivy.core.window import Window  
5. from kivy.network.urlrequest import UrlRequest  
6. import certifi as cf
```

The screen is the base, and other components will be placed on top of it. It is similar to the web page. Here, you can assign a name to the screen. As you can see in the following code, a hierarchy is being maintained.

The screen managers mainly manage the different screens. Let's understand each component. **MDLabel** is used to display the text with properties like horizontal alignment, font style, and so on.

MDTextField is used to get user inputs. Here, **hint_text** is added to guide users while entering the data into the app. It has an **id** parameter, which will fetch the data entered by the user.

MDButton is used to perform actions. Here, it is being used to call the **predict()** with the parameter **on_press** to complete the action.

```
1. Builder_string = ''  
2. ScreenManager:  
3.     Main:  
4.         <Main>:  
5.             name : <main>  
6.             MDLabel:  
7.                 text: <Loan Prediction App>  
8.                 halign: <center>  
9.                 pos_hint: {<center_y>:0.9}  
10.                font_style: <H4>  
11.  
12.                MDLabel:  
13.                    text: <Gender>  
14.                    pos_hint: {<center_y>:0.75}
```

```
15.  
16.    MDTextField:  
17.        id: input_1  
18.        hint_text: <0:Female, 1:Male'  
19.        width: 100  
20.        size_hint_x: None  
21.        pos_hint: {<center_y>:0.75, <center_x>:0.50}  
22.  
23.    MDLabel:  
24.        text: <Marital Status>  
25.        pos_hint: {<center_y>:0.68}  
26.  
27.    MDTextField:  
28.        id: input_2  
29.        hint_text: <0>No, 1:Yes'  
30.        width: 100  
31.        size_hint_x: None  
32.        pos_hint: {<center_y>:0.68, <center_x>:0.50}  
33.  
34.    MDLabel:  
35.        text: <Applicant Income>  
36.        pos_hint: {<center_y>:0.61}  
37.  
38.    MDTextField:  
39.        id: input_3  
40.        hint_text: <6000>  
41.        width: 100  
42.        size_hint_x: None  
43.        pos_hint: {<center_y>:0.61, <center_x>:0.50}  
44.
```

```
45.     MDLabel:  
46.         text: <Loan Amount>  
47.         pos_hint: {<center_y>:0.54}  
48.  
49.     MDTextField:  
50.         id: input_4  
51.         hint_text: <150>  
52.         width: 100  
53.         size_hint_x: None  
54.         pos_hint: {<center_y>:0.54, <center_x>:0.50}  
55.  
56.     MDLabel:  
57.         text: <Credit History>  
58.         pos_hint: {<center_y>:0.47}  
59.  
60.     MDTextField:  
61.         id: input_5  
62.         hint_text: <0:Clear Debts, 1:Unclear Debts>  
63.         width: 100  
64.         size_hint_x: None  
65.         pos_hint: {<center_y>:0.47, <center_x>:0.50}  
66.  
67.     MDLabel:  
68.         pos_hint: {<center_y>:0.2}  
69.         halign: <center>  
70.         text: <>  
71.         id: output_text  
72.         theme_text_color: «Custom»  
73.         text_color: 0, 1, 1, 1  
74.
```

```
75.     MDRaisedButton:  
76.         pos_hint: {<center_y>:0.1, <center_x>:0.5}  
77.         text: <Predict>  
78.         on_press: app.predict()  
79.     '''
```

First store **ScreenManager()** into the **sm** object for ease of access, then add the main screen widget. After that, define the **MainApp** class (import **MDApp** in it as a base), which holds the core logic of the app. Define the **build()** for adding text and input widgets. The **predict()** will get the user data from the **id** directly as shown in the code.

Provide these input parameters to the remote API endpoint, which will return the prediction for it.

In kivy, you can use **UrlRequest()** to parse the URL and the **on_success** parameter to be triggered on request completion. In the current case, use **UrlRequest()** to call **res()** on completion of the API request, which will replace the blank text of **MDLabel** defined earlier: **MDRaisedButton**.

```
1. class Main(Screen):  
2.     pass  
3.  
4.     sm = ScreenManager()  
5.     sm.add_widget(Main(name='main'))  
6.  
7. class MainApp(MDApp):  
8.     def build(self):  
9.         self.help_string = Builder.load_string(Builder_string)  
10.        return self.help_string  
11.  
12.    def predict(self):  
13.        Gender = self.help_string.get_screen('main').ids.input_1.  
text  
14.        Married = self.help_string.get_screen('main').ids.  
input_2.text  
15.        ApplicantIncome = self.help_string.get_screen('main').ids.
```

```

    input_3.text
16.           LoanAmount = self.help_string.get_screen('main').ids.
input_4.text
17.           Credit_History = self.help_string.get_screen('main').ids.
input_5.text
18.   url = f'https://fastapi-               appl.
herokuapp.com/predict?gender={Gender}&married={Married}&income={Ap-
plicantIncome}&loan_amt={LoanAmount}&credit_hist={Credit_History}'
19.   self.request = UrlRequest(url=url, on_success=self.res, ca_
file=cfl.where(), verify=True)
20.
21.   def res(self, *args):
22.       self.data = self.request.result
23.       ans = self.data
24.           self.help_string.get_screen('main').ids.output_text.
text = ans['status']
25.
26.MainApp().run()

```

Run the app using **python main.py** and if everything goes well, you should see the app up and running.

The following figure shows the kivy app running successfully.

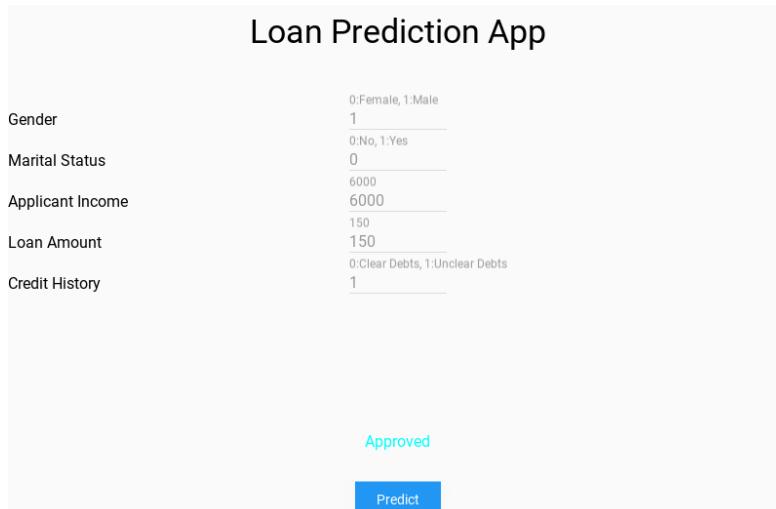


Figure 8.6: Kivy ML app

Convert the Python app into an Android app

Now, you will learn to convert this into an Android app using the Buildozer package. Make sure your kivy filename is the *main.py* for this step. You need to install the Buildozer package first, if it is not installed already.

```
pip install buildozer
```

Install the required dependencies, as follows:

```
pip install cython==0.29.19
```

```
sudo apt-get install -y \
    python3-pip \
    build-essential \
    git \
    python3 \
    python3-dev \
    ffmpeg \
    libsdl2-dev \
    libsdl2-image-dev \
    libsdl2-mixer-dev \
    libsdl2-ttf-dev \
    libportmidi-dev \
    libswscale-dev \
    libavformat-dev \
    libavcodec-dev \
    zlib1g-dev
```

```
sudo apt-get install -y \
    libgstreamer1.0 \
    gstreamer1.0-plugins-base \
    gstreamer1.0-plugins-good
```

```
sudo apt-get install build-essential libsqlite3-dev sqlite3 bzip2 libbz2-
```

```
dev zlib1g-dev libssl-dev openssl libgdbm-dev libgdbm-compat-dev liblzma-dev libreadline-dev libncursesw5-dev libffi-dev uuid-dev libffi6
```

```
sudo apt-get install libffi-dev
```

You may need to install additional dependencies as per system requirements.

Now, create the *buildozer.spec* file by running the following command:

```
buildozer init
```

The preceding command will create a standard *buildozer.spec* file. However, you need to modify it as per requirement. For instance, providing dependencies next to requirements under the **Application requirements** section, providing package name, title, and domain, providing internet permission under the **Permissions** section, and so on.

Finally, build and package the Android app using the following command:

```
buildozer -v android debug
```

Optionally, you can use Google Colab notebook.

The following figure shows the files and directories for the KivyML app:

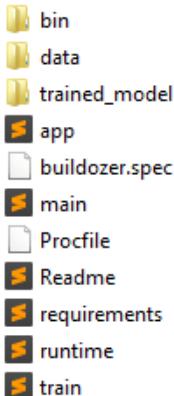


Figure 8.7: Kivy ML app

You have successfully built and run ML-based Tkinter and Kivy apps. Also, you have learned to package them in native applications.

Conclusion

In this chapter, you learned to develop an ML application using the most open-source frameworks: Tkinter, kivy, and KivyMD. You created a user-friendly yet

simple UI to receive input data from users and display responses. You explored the functionalities of Tkinter and kivy and converted the Tkinter app to a desktop app with Windows executable file using Pyinstaller and the kivyMD app to an Android app using Buildozer.

In the next chapter, you will learn how to build CI/CD pipelines for ML.

Points to remember

- Tkinter comes pre-installed with Python. The best way to get the latest version of Tkinter is to install Python version 3.7 or later.
- KivyMD is an extension of the kivy framework.
- Tkinter and kivy are both open-source and free to use.
- **MDLabel** is used to display text in kivyMD.

Multiple choice questions

1. In Tkinter, what is/are the geometry manager(s) is/are?
 - a) Grid
 - b) Place
 - c) Pack
 - d) All the above
2. The use of the `mainloop()` in Tkinter is
 - a) Keep app running
 - b) Execute any loop defined
 - c) Both
 - d) None of them

Answers

1. a
2. d

Questions

1. What is Tkinter?
2. What is the role of the `pack()` in Tkinter?
3. How to define buttons in the kivy app?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 9

CI/CD for

ML

Introduction

Nowadays, automation is all around us. By automating the manual process, you save time, effort, and cost. The best part about automation is that it reduces human errors. You can automate most of the parts while incorporating new changes or updates in the application.

CI/CD pipeline enables you to deploy the updates to the application in an automated way. This chapter will touch upon the elements of the CI/CD pipeline and ways to leverage it for ML applications. In this chapter, you will learn the different stages of the CI/CD pipeline, their importance in MLOps, and how to build one.

Structure

This chapter discusses the following topics:

- CI/CD pipeline for ML
 - Continuous Integration (CI)
 - Continuous Delivery/Deployment (CD)
 - Continuous Training (CT)
- Tools and platforms for building CI/CD pipeline

- Introduction to Jenkins
- Building CI/CD pipeline using GitHub, Docker, and Jenkins

Objectives

After studying this chapter, you should be able to build a CI/CD pipeline to deploy ML models in production, integrate GitHub with Jenkins, run the tests using Jenkins's job and generate a test summary in Jenkins, build a Docker image, and run a Docker container using Jenkins. Finally, you should be able to integrate Jenkins with an email account to get the status of the build and deployment.

CI/CD pipeline for ML

CI/CD is an acronym for **Continuous Integration/Continuous Delivery/Deployment (CI/CD)**. The purpose of the CI/CD pipeline is to automate the chain of interconnected steps to deploy an application or release a new version of the software.

When a new feature gets added to the application, any improvement needs to be integrated with the application. However, it involves different teams that execute multiple tasks and validate them before moving on to the production stage. Mostly, this is a manual and time-consuming process, and it can cause a delay in the release of the new version.

CI/CD pipeline helps in automating the testing, running error-free code for the application, faster deployments, saving time and cost for developers, high reliability, and so on.

CI/CD pipeline enables you to push the changes from development to deployment quickly, which usually consist of four stages:

- **Commit code changes:** After making changes to the file or code, the developer pushes the updates to the source repository. This activity is often performed in a team. CI/CD pipeline enables any team member to check the integrity of the code. Hence, you can automatically push the changes to the repository after it passes the tests.
- **Build:** In this phase, it fetches the changes from the repository for the build. It keeps a watch on the source repository for any changes. As soon as it detects the changes, it initiates the build process and validates the build results after build completion.
- **Test:** The test phase runs the automated tests, such as unit tests, pytest, and API tests, on top of the build. It is a vital stage of the CI/CD pipeline. This

ensures the overall integrity of the code and prevents any broken code from passing on to the next phase.

- **Deploy:** This phase deploys the changes to the production environment.

The following figure shows the different stages of the CI/CD pipeline. However, organizations can modify it as per their goals.

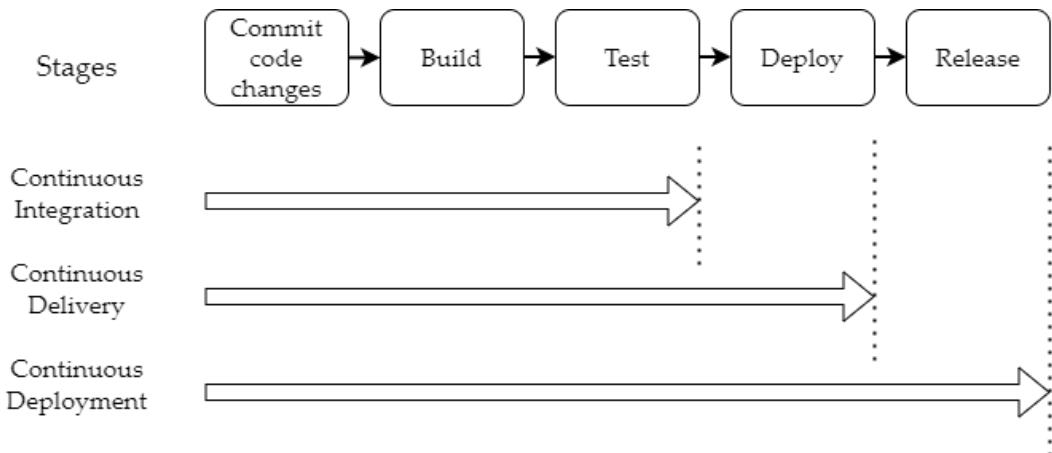


Figure 9.1: CI/CD Pipeline

Listing down some popular tools for CI/CD pipeline:

- Jenkins
- GitHub Actions
- Bamboo
- CircleCI
- GitLab CI/CD
- Travis CI

Continuous Integration (CI)

In **Continuous Integration (CI)**, the team of developers builds, run, and test code in local environments first. If everything goes well then, they push the updates to the repository. After this, the chain of steps starts to run, which involves the build, run, and test stages. The project members get notified at each step and get timely updates, such as the build outcome and test outcomes. Finally, the artifacts get stored and the report of the current status is sent via email or notified via Slack.

When a team of developers is working on the same application, they push the code changes to the repository. However, due to changes in the environment, such as the

production environment, the code can break or throw an error. On the other hand, there could be a conflict between updates, when multiple developers try to push the updates of the application to the central code repository. This issue is taken care of by the CI stage, where developers can push the changes that will pass through the defined stages, such as build, run, and test, to ensure that the code is working properly without any issues. However, if any of the CI stages fail, then you will be notified, and the further process stops. This way, you can avoid integrating any broken code into production. Developers can frequently push and check the functioning of the code, flow, and integrity of the code or applications before pushing it to the next stage for deployment.

Continuous Delivery/Deployment (CD)

CD refers to Continuous Delivery or Continuous Deployment (the terms are used interchangeably) based on the level of automation you are planning to implement. The CD stage depends on the CI stage. Once the CI stage is completed, it triggers the CD stage in the pipeline. The purpose of the **Continuous Delivery (CD)** stage is to deliver an error-free codebase or artifact to the pre-production environment. In this stage, you can add a series of test cases (wherever required) to ensure a stable build and functional application. It sends the test status report to the team or developer, and then the application is manually pushed to the production environment. If any of the steps fail, you may need to carry out the entire process again with the required updates.

On the other hand, continuous deployment goes one step further and deploys the application from the pre-production environment to the production environment quickly. This removes the step of manual deployment of an application to the production environment. However, it is optional, as it depends on the developer and operation team to choose the level of automation they want to implement as per the business and application's nature.

Continuous Training (CT)

A new stage introduced in the traditional CI/CD pipeline is Continuous Training (CT). In this stage, you expect the models to be trained continuously as new data comes in or any event occurs, such as the accuracy of the model dropping below the acceptable threshold. This may add slight complexity to CI/CD pipeline, but it is essential for Machine Learning deployment.

With CI/CD, **Continuous Training (CT)** is equally important in MLOps. Model retraining depends on scenarios and various other factors, such as how frequently data is changing and the schema of input data. It also depends on events such as accuracy dropping below an acceptable threshold, specific periods such as the end of every week, or manual triggers.

Introduction to Jenkins

Jenkins is an open-source, modular CI / CD automation tool written in Java that comes with several plugins. Jenkins enables the smooth and continuous flow of building, testing, and deploying apps with a recently updated or developed codebase. It has a large community support and is popular among developers.

If the build is successful, then Jenkins automatically executes a series of steps from the code repository, and if everything goes well, it deploys the application to the server.

Here are some salient features of Jenkins:

- Jenkins is an open-source, free, and modular tool.
- It is created by developers and for developers.
- Jenkins is easy to install, configure and can be installed on Linux, MacOS, and Windows.
- A large number of Jenkins plugins are available for popular cloud platforms.
- Jenkins's master can distribute the load to multiple slaves and enable faster processing.

Installation

Jenkins can be installed on any server that supports JAVA, as it is written in JAVA. Jenkins installation is described here with various options.

<https://www.jenkins.io/doc/book/>

- Docker
- Kubernetes
- Linux
- MacOS
- WAR files
- Windows
- Other systems
- Offline installations
- Initial settings

As you are installing Jenkins on Ubuntu, you can refer to the following link for a step-by-step installation guide.

<https://www.jenkins.io/doc/book/installing/linux/#debianubuntu>

Step 1: Install Jenkins

Add the repository key:

```
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -
```

Append the Debian package repository address to the server's **sources.list**:

```
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
```

Execute the following command so that the **apt** will use the latest repository:

```
sudo apt update
```

Finally, install Jenkins with its dependencies:

```
sudo apt install Jenkins
```

Step 2: Start Jenkins service

Now, start the Jenkins service using the following command:

```
sudo systemctl start Jenkins
```

To check the status of Jenkins, use the following command:

```
sudo systemctl status Jenkins
```

Step 3: Allow Jenkins's default port in the Firewall

By default, Jenkins will run on its default port, that is, **8080**. Hence, port **8080** needs to be allowed in the firewall. To do so, run the following command in the terminal:

```
sudo ufw allow 8080
```

Then, to confirm it, use the following command:

```
sudo ufw status
```

Step 4: Set up Jenkins

You can use Jenkins's web-based UI server IP or domain name to access Jenkins:

http://<server_ip_or_domain>:8080

Initially, you will get a screen asking for an administrator password; you can get this by using the following command:

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

Copy the alphanumeric password from the terminal and paste it into the **Administrator password** field.

In the next window, you can continue as admin or create the first admin user by filling up the form.

Now, add the Jenkins user to the Docker group using the following command:

```
sudo usermod -aG docker jenkins
```

Build CI/CD pipeline using GitHub, Docker, and Jenkins

Here, Jenkins is used for automated ML workflow. First off, you need to create a codebase in the local machine and run the ML app locally to make sure it is working properly on the local machine. Next, you will push the changes to the GitHub repository. Then, integrate the GitHub repo and Jenkins by webhook. Jenkins is to be installed on pre-production or production servers. Jenkins will pull the latest codebase from the linked GitHub repo and deploy the ML app on the server.

Develop codebase

Let's consider the scenario of loan prediction, where you need to predict whether a customer's loan will be approved. The focus will not be on hyperparameter tuning and model optimization. However, you can optimize a model to improve its overall performance.

In this chapter, you will use a machine learning package developed earlier.

You can access the code repository at the following link:

https://github.com/suhas-ds/docker_pkg_jenkins

The following is the directory structure for codebase:

```
.  
|   Dockerfile  
|   main.py  
|   requirements.txt  
\---src  
    |   MANIFEST.in  
    |   README.md
```

```
|   requirements.txt
|   setup.py
|   tox.ini
+---prediction_model
|   |   pipeline.py
|   |   predict.py
|   |   train_pipeline.py
|   |   VERSION
|   |   __init__.py
|   +---config
|   |       config.py
|   |       __init__.py
|   +---datasets
|   |       test.csv
|   |       train.csv
|   |       __init__.py
|   +---processing
|   |       data_management.py
|   |       preprocessors.py
|   |       __init__.py
|   \---trained_models
|       classification_v1.pkl
|       __init__.py
\---tests
    pytest.ini
    test_predict.py
```

main.py

FastAPI is a web framework for developing RESTful APIs in Python. Create a *main.py* file to run the FastAPI app.

Note: FastAPI requires Python 3.6+.

```
1. # Importing Dependencies  
2. from fastapi import FastAPI  
3. from pydantic import BaseModel  
4. import uvicorn  
5. import pickle  
6. import numpy as np  
7. import pandas as pd  
8. from fastapi.middleware.cors import CORSMiddleware  
9. from prediction_model.predict import make_prediction  
10. import pandas as pd
```

Create a FastAPI instance and assign it to the **app** so that the **app** will be a point of interaction while creating the API.

```
1. app = FastAPI(  
2.                 title="Loan Prediction Model API",  
3.                 description="A simple API that uses ML model to predict  
the Loan application status",  
4.                 version="0.1",  
5.             )
```

CORS (Cross-Origin Resource Sharing) refers to the situation when a front end running in a browser has JavaScript code that communicates with a back end, and the back end is of a different origin than the front end. However, it depends on your application and requirement whether to use it or not.

```
1. origins = ["*"]  
2.  
3. app.add_middleware(  
4.                     CORSMiddleware,  
5.                     allow_origins=origins,  
6.                     allow_credentials=True,  
7.                     allow_methods=["*"],  
8.                     allow_headers=["*"],  
9.                 )
```

Define the class **LoanPred**, which defines the data type expected from the client.

You can use the **LoanPred** class for the data model that is inherited from **BaseModel**. Then, add a root view of the function that returns '**message**': '**Loan Prediction App**' for the home page.

```
1. class LoanPred(BaseModel):
2.     Gender: str
3.     Married: str
4.     Dependents: str
5.     Education: str
6.     Self_Employed: str
7.     ApplicantIncome: float
8.     CoapplicantIncome: float
9.     LoanAmount: float
10.    Loan_Amount_Term: float
11.    Credit_History: float
12.    Property_Area: str
13.
14. @app.get('/')
15. def index():
16.     return {'message': 'Loan Prediction App'}
```

Here, create **/predict_status** as an endpoint, also known as the route. Then, add **predict_loan_status()** with a parameter of the type data model that you created as **LoanPred**.

```
1. #Defining the function which will make the prediction using the data
   which the user inputs
2. @app.post('/predict_status')
3. def predict_loan_status(loan_details: LoanPred):
4.     data = loan_details.dict()
5.     Gender = data['Gender']
6.     Married = data['Married']
7.     Dependents = data['Dependents']
```

```

8. Education = data['Education']
9. Self_Employed = data['Self_Employed']
10. ApplicantIncome = data['ApplicantIncome']
11. CoapplicantIncome = data['CoapplicantIncome']
12. LoanAmount = data['LoanAmount']
13. Loan_Amount_Term = data['Loan_Amount_Term']
14. Credit_History = data['Credit_History']
15. Property_Area = data['Property_Area']
16.
17. # Making predictions
18. input_data = [Gender, Married, Dependents, Education,
19.     Self_Employed, ApplicantIncome, CoapplicantIncome,
20.     LoanAmount, Loan_Amount_Term, Credit_History, Property_Area]
21. cols = ['Gender','Married','Dependents',
22.     'Education','Self_Employed','ApplicantIncome',
23.     'CoapplicantIncome','LoanAmount','Loan_Amount_Term',
24.     'Credit_History','Property_Area']
25. data_dict = dict(zip(cols,input_data))
26. prediction = make_prediction([data_dict])['prediction'][0]
27.
28. if prediction == 'Y':
29.     pred = 'Approved'
30. else:
31.     pred = 'Rejected'
32.
33. return {'status':pred}

```

The following function will create the UI for user input. Here, create **/predict** as an endpoint, also known as a route, and declare input data types expected from users.

```

1. @app.post('/predict')
2. def get_loan_details(Gender: str, Married: str, Dependents: str,

```

```
3. Education: str, Self_Employed: str, ApplicantIncome: float,
4. CoapplicantIncome: float, LoanAmount: float, Loan_Amount_Term: float,
5. Credit_History: float, Property_Area: str):
6.
7. input_data = [Gender, Married, Dependents, Education,
8.     Self_Employed, ApplicantIncome, CoapplicantIncome,
9.     LoanAmount, Loan_Amount_Term, Credit_History, Property_Area]
10. cols = ['Gender', 'Married', 'Dependents',
11.     'Education', 'Self_Employed', 'ApplicantIncome',
12.     'CoapplicantIncome', 'LoanAmount', 'Loan_Amount_Term',
13.     'Credit_History', 'Property_Area']
14.
15. data_dict = dict(zip(cols,input_data))
16. prediction = make_prediction([data_dict])['prediction'][0]
17. if prediction == 'Y':
18.     pred = 'Approved'
19. else:
20.     pred = 'Rejected'
21.
22. return {'status':pred}
23.
24.if __name__ == '__main__':
25. uvicorn.run(app)
```

The file for the FastAPI app is completed.

requirements.txt

Now, create the *requirements.txt* file, as follows. In this file, model requirements, test requirements, and FastAPI requirements are defined separately for better understanding and ease of management.

```
1. # Model building requirements
2. joblib==0.16.0
```

```
3. numpy==1.19.0
4. pandas==1.0.5
5. scikit-learn==0.23.1
6. scipy==1.5.1
7. sklearn==0.0
8.
9. # testing requirements
10. pytest<5.0.0,>=4.6.6
11.
12. # packaging
13. setuptools==40.6.3
14. wheel==0.32.3
15.
16. # FastAPI app requirements
17. fastapi>=0.68.0,<0.69.0
18. pydantic>=1.8.0,<2.0.0
19. uvicorn>=0.15.0,<0.16.0
20. gunicorn>=20.1.0
```

Dockerfile

Create a *Dockerfile* and name it Dockerfile without any file extension, and add the following commands to it.

```
1. FROM python:3.7-slim-buster
2.
3. RUN apt-get update && apt-get install -y \
4.     build-essential \
5.     libpq-dev \
6.     && rm -rf /var/lib/apt/lists/*
7.
8. RUN pip install --upgrade pip
9.
```

```
10. COPY . /code
11.
12. RUN chmod +x /code/src
13.
14. RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt
15.
16. EXPOSE 8005
17.
18. WORKDIR /code/src
19.
20. ENV PYTHONPATH "${PYTHONPATH}:/code/src"
21.
22. CMD pip install -e .
```

In the preceding *Dockerfile*, it will first pull the base image, that is, Python 3.7 slim buster, and install the necessary packages. After that, packages from the requirements file will be installed, and the **8005** port will be exposed so that it can be accessed outside the Docker container. Finally, it will assign **PYTHONPATH** in the **ENV** instruction and will execute the command mentioned in the CMD instruction to install the Python package in editable mode.

Create a Personal Access Token (PAT) on GitHub

If you perform any action like pull, push, or clone using the **git cli** command, then it won't work anymore with the GitHub password. As a matter of fact, GitHub has removed password authentication. Instead, you have to use a **Personal Access Token (PAT)**.

PATs are an alternative to using passwords for authentication to GitHub Enterprise Server when using the GitHub API or the command line.

First, you need to create a PAT. This is described in the following link:

<https://docs.github.com/en/enterprise-server@3.4/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>

To generate or regenerate your PAT, you can go to **Settings | Tokens** (login required):

<https://github.com/settings/tokens>

If you are not prompted for your username and password, your credentials may be cached in your machine.

Create a webhook on the GitHub repository

A webhook can be considered a lightweight API that enables one-way data sharing after being triggered by certain events. In GitHub, a webhook can be triggered when actions, such as push and pull, are performed on a repository. With the help of the GitHub webhook, you can trigger the CI stage on a remote server.

Go to your [GitHub repo](#) | [Settings](#) | [Webhooks](#).

Alternatively, you can directly jump to that page using the following URL:

<https://github.com/<username>/<your-repo-name>/settings/hooks>

In the current case, it is as follows:

https://github.com/suhas-ds/docker_pkg_jenkins/settings/hooks

You cannot use `https://localhost:8080/github-webhook/` in GitHub webhook's payload URL as the localhost URL needs to be exposed to the internet. You can use ngrok for exposing localhost URLs on the internet. Refer to the official site of ngrok <https://ngrok.com/> for more details.

Here, ngrok is being used for demonstration purposes; however, the remote server's IP or domain can be used in the deployment stage.

Use the command `ngrok http 8080` to expose the `8080` port publicly.

```
ngrok
      (Ctrl+C to quit)

Session Status           online
Account                  suhasp.ds@gmail.com (Plan: Free)
Version                 3.0.3
Region                  India (in)
Latency                10.529741ms
Web Interface          http://127.0.0.1:4040
Forwarding              https://32f7-103-111-133-78.in.ngrok.io -> http://localhost:8080

Connections             ttl     opn     rt1     rt5     p50     p90
                        50      0      0.00    0.00    0.12    5.43
```

Figure 9.2: ngrok

Provide a payload URL, as shown in the following figure. The payload URL format was discussed earlier in this chapter. Select the **Content type** as an **application/json** format. The **Secret** field is optional. Let's leave it blank for now.

Webhooks / Manage webhook

The screenshot shows a web-based configuration interface for a GitHub webhook. At the top, there are two tabs: "Settings" (which is active) and "Recent Deliveries". Below the tabs, a note states: "We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#)". The "Payload URL" field contains the value "https://32f7-103-111-133-78.in.ngrok.io/github-webhook/". The "Content type" dropdown menu is set to "application/json".

Figure 9.3: GitHub Webhook

Choose the **Enabled SSL verification** and **Just push the event** options, as shown in the following figure, so that it will send data only when someone pushes updates to the repository. However, you can also select the individual events, that is, **Let me select individual events**.

Now, click on the **Add Webhook** button to save Jenkins GitHub Webhook.

The screenshot shows the Jenkins GitHub Webhook configuration page. It includes the following sections:

- SSL verification:** A note says "By default, we verify SSL certificates when delivering payloads." with two radio button options: **Enable SSL verification** (selected) and **Disable (not recommended)**.
- Which events would you like to trigger this webhook?** Three radio button options are available: **Just the push event.** (selected), **Send me everything.**, and **Let me select individual events.**
- Active:** A checked checkbox with the note "We will deliver event details when this hook is triggered." Below this are two buttons: "Update webhook" (green background) and "Delete webhook" (gray background).

Figure 9.4: GitHub Webhook

Configure Jenkins

Now, mainly two sections are to be configured for the current scenario: GitHub webhook integration and extended email integration.

1. Go to Jenkins dashboard.
2. Go to **Manage-Jenkins | Manage Plugin**.
3. Install Git, GitHub, and email plugins without restart.

If the preceding plugins are already selected, you can continue to the next step.

Now, let's configure the GitHub webhook in Jenkins.

Go to **Manage Jenkins | Configure System**.

Scroll down and update the **API URL** in the **GitHub** section, as follows:

https://api.github.com

In the **Credentials** section, add GitHub's PAT generated in the previous step. You can leave the **Name** field blank.

Finally, click on the **Save** button.

The following figure shows GitHub webhook integration in Jenkins:

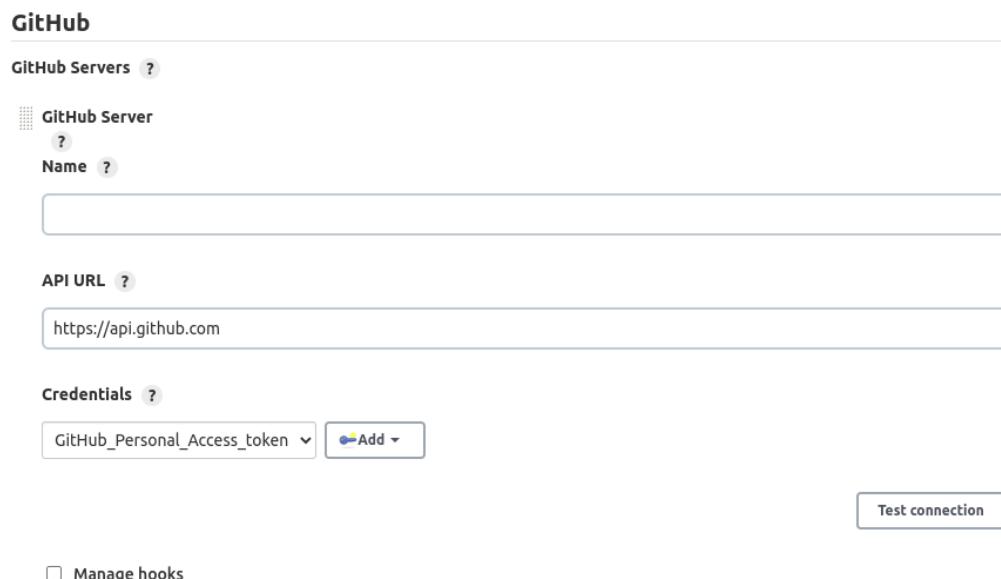


Figure 9.5: GitHub configuration in Jenkins

Now, whenever the developer pushes the updates to the repository, Jenkins will detect the changes and start executing a series of commands one by one.

Next, configure the email for a feedback loop or status updates, for example, succeeded or failed.

Go to **Manage Jenkins | Configure System**.

Scroll down and update the fields in the **GitHub** section.

In the following figure, the **Default Subject** field is updated by adding **Status** at the beginning. Also, an email body is added in the **Default Content** field. It is highly customizable, so you can update it as per requirement.

Default Subject ?

Status: \$PROJECT_NAME - Build # \$BUILD_NUMBER - \$BUILD_STATUS!

Maximum Attachment Size ?

-1

Default Content ?

```
Hello,  
Please find below the build details  
$PROJECT_NAME - Build # $BUILD_NUMBER - $BUILD_STATUS:  
Check console output at $BUILD_URL to view the results.  
Note: This is an auto generated email, please do not reply.  
Best,  
Automation Team
```

Figure 9.6: Email configuration

In the **Default Triggers** section, choose the **Always** checkbox so that Jenkins sends the status email for both success and failure scenarios, as shown in the following figure:

Default Triggers ?

Aborted
 Always

Figure 9.7: Email configuration - trigger

Now, select **HTML (text/html)** from the dropdown in the **Default Content Type** field and add your email ID in **Default Recipients**, as shown in the following figure:

Default Content Type ?

List ID ?

Add 'Precedence: bulk' E-mail Header ?

Default Recipients ?

Figure 9.8: Email configuration – content type and recipients

In the **Extended E-mail Notification** section, provide details for the email notifications. In the current scenario, Gmail details are provided, as follows:

- **smtp.gmail.com** to SMTP server
- **465** to SMTP Port
- Add email login credentials
- Check the **Use SSL** box
- Click on the **Save** button

Provide your email details as shown in the following figure:

Extended E-mail Notification

SMTP server

SMTP Port

Credentials

Use SSL

Use TLS

Figure 9.9: Extended email notification

Scroll down and click on the **Apply** and **Save** buttons to save the changes, as shown in the following figure:



Figure 9.10: Extended email notification

If you are integrating your Gmail account for email notifications, you will have to allow less secure apps in the settings of your Gmail account. You can go to the following URL to turn it on if it is off.

<https://myaccount.google.com/lesssecureapps>

For more details, visit <https://support.google.com/accounts/answer/6010255?hl=en>.

The required configuration for Jenkins is complete.

Create CI/CD pipeline using Jenkins

Now, you will build a simple CI/CD pipeline using GitHub and Jenkins. When developers push the updates to the GitHub repository, the GitHub webhook detects the changes and sends the notification to Jenkins. Jenkins pulls the latest code from the GitHub repository, builds the Docker image, and runs the container using that image. Next, it trains the model and exports the pickle object from the trained model. In the next step, pytest results are exported and displayed in Jenkins. After passing all the tests, it will run the ML web app. Finally, it will send the feedback via email to the developer or concerned team.

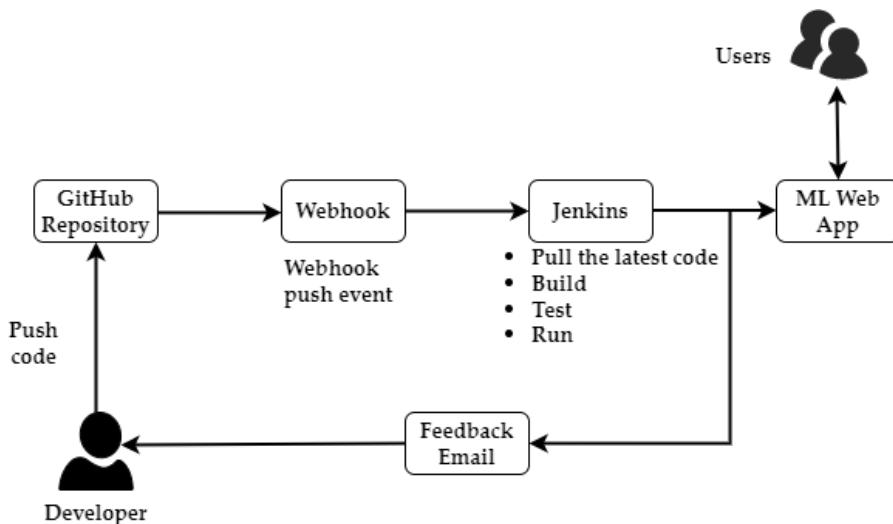


Figure 9.11: CI/CD pipeline using Jenkins

Stage 1: 1-GitHub-to-container

At this stage, Jenkins will pull the latest code and files from GitHub as soon as the developer pushes the updates to the linked GitHub repository. Let's understand this process step by step.

When the developer pushes the updates to the GitHub repository, the webhook detects the changes, and it gets triggered. GitHub webhook sends a message to Jenkins that new updates have been detected, and then Jenkins pulls the latest code files to start building the Docker image.

As shown in the following figure, select the **New Item** from the left panel of the Jenkins dashboard.

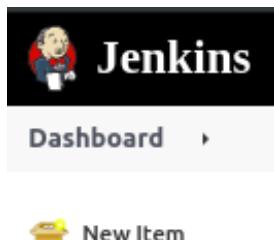


Figure 9.12: Jenkins's dashboard – New Item

As shown in the following figure, provide a job name in the **Enter an item name** field and choose a **Freestyle project**. In the current case, it is named **1-GitHub-to-docker-container**.

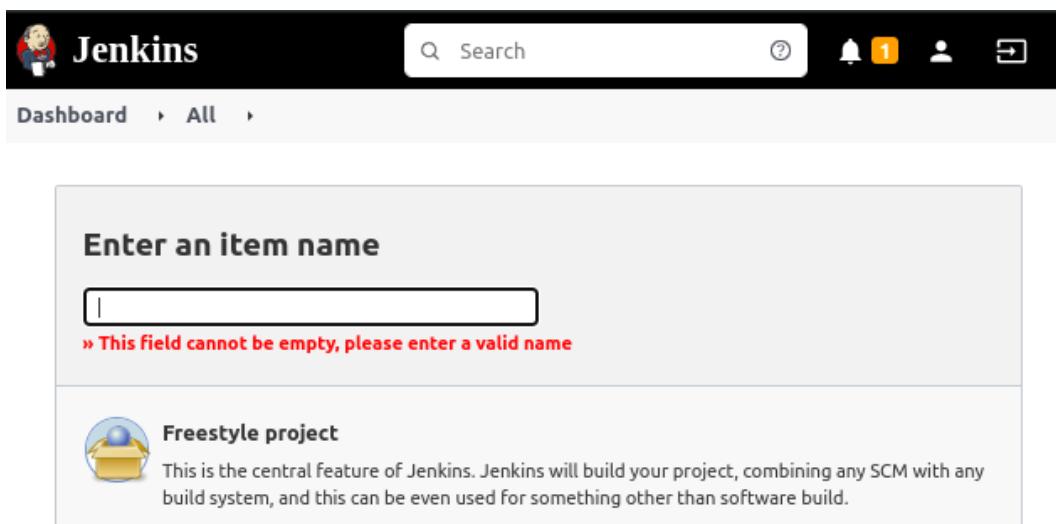


Figure 9.13: Jenkins's dashboard – Freestyle project

As shown in the following figure, select the **Git** radio button under the **Source Code Management** tab and provide the GitHub repository URL. By default, it will be the **master** branch, but you can update it. If your repository is private, then you may need to pass the credentials for the same.

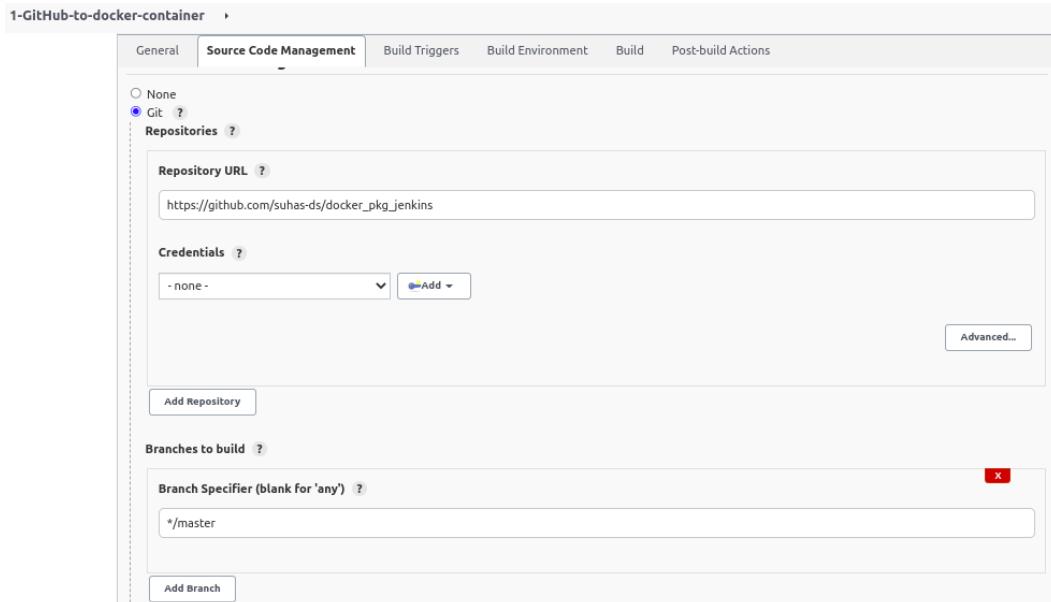


Figure 9.14: 1-GitHub-to-container

As shown in the following figure, choose the **GitHub hook trigger for GITScm polling** under the **Build Triggers** tab, to pull the latest updates from the GitHub repository.

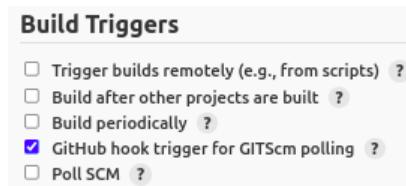


Figure 9.15: 1-GitHub-to-container - build triggers

In the following figure, Docker commands are being executed for building and running Docker images with the latest changes from the GitHub repository.

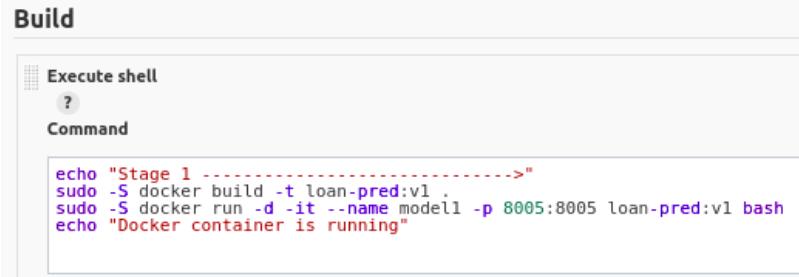


Figure 9.16: 1-GitHub-to-container - Build

As you can see in the following figure, Jenkins started building the Docker image as soon as the updates were pushed to the GitHub repository. The first line of output read **Started by GitHub push by suhas-ds**; it is showing who pushed the updates to the GitHub repository.

Console Output

```
Started by GitHub push by suhas-ds
Running as SYSTEM
Building in workspace /var/lib/jenkins/workspace/1-GitHub-to-docker-container
The recommended git tool is: NONE
No credentials specified
> git rev-parse --resolve-git-dir /var/lib/jenkins/workspace/1-GitHub-to-docker-container/.git # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url https://github.com/suhas-ds/docker_pkg_jenkins # timeout=10
Fetching upstream changes from https://github.com/suhas-ds/docker_pkg_jenkins
> git --version # timeout=10
> git --version # 'git version 2.17.1'
> git fetch --tags --progress -- https://github.com/suhas-ds/docker_pkg_jenkins +refs/heads/*:refs/remotes/origin/* # timeout=10
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
Checking out Revision 70ad4599683df499bd4e38e8d100f33a33650fc8 (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f 70ad4599683df499bd4e38e8d100f33a33650fc8 # timeout=10
Commit message: "Update"
> git rev-list --no-walk b5635257b0e9c8e643e0ee4b0be2ad3681107b45 # timeout=10
[1-GitHub-to-docker-container] $ /bin/sh -xe /tmp/jenkins998670132563554543.sh
+ echo Stage 1 ----->
Stage 1 ----->
+ sudo -S docker build -t loan-pred:v1 .
Sending build context to Docker daemon 241.7kB
```

Figure 9.17: 1-GitHub-to-container – Console Output

Stage 2: 2-training

At this stage, the model will be trained on training data by running the *train_pipeline.py* file inside the Docker container. As shown in the following figure, select the **None**

radio button under the **Source Code Management** tab, as it is the next stage; no need to pull the source code again.

Source Code Management

- None
- Git ?

Figure 9.18: 2-training – Source code management

As shown in the following figure, select the **Build after other projects are built** option under the **Build Triggers** tab. Next, provide the name of the previous stage in the **Projects to watch** field. Then, select the first radio button, that is, **Trigger only if the build is stable**, which means stage 2 will start executing only after the successful completion of the previous stage.

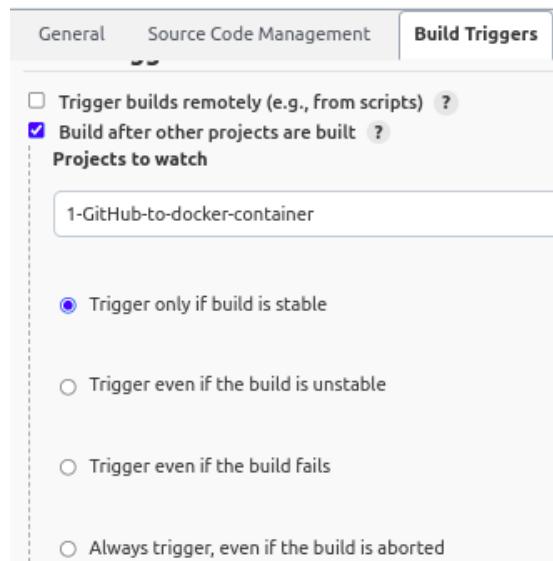


Figure 9.19: 2-training – Build triggers

As shown in the following figure, write Docker commands for training the model pipeline and exporting the latest pickle object of the trained model, under the **Build** tab.

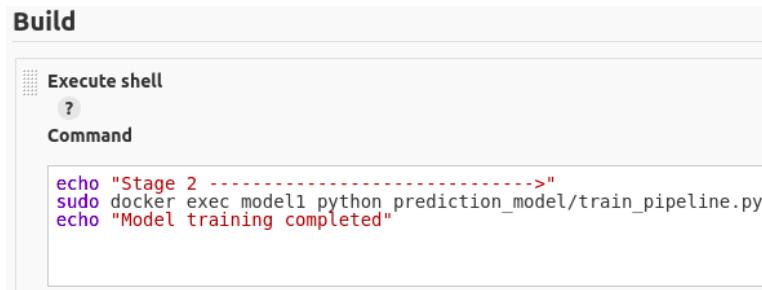


Figure 9.20: 2-training – Build

The console output of stage 2 is shown in the following figure. It tells you what caused stage 2 to trigger and the status of stage 2. After that, Jenkins is triggering stage 3 on the successful completion of stage 2.

Console Output

```
Started by upstream project "1-GitHub-to-docker-container" build number 20
originally caused by:
  Started by GitHub push by suhas-ds
  Running as SYSTEM
Building in workspace /var/lib/jenkins/workspace/2-training
[2-training] $ /bin/sh -xe /tmp/jenkins5775494677148366096.sh
+ echo Stage 2 -----
Stage 2 ----->
+ sudo docker exec modell python prediction_model/train_pipeline.py
Saved Pipeline : classification_v1.pkl
+ echo Model training completed
Model training completed
Triggering a new build of 3-testing
Finished: SUCCESS
```

Figure 9.21: 2-training – Console output

Stage 3: 3-testing

In this stage, Jenkins is running pytest and publishing test results using **JUnit**. Jenkins understands the JUnit test report in XML format. JUnit enables Jenkins to provide test results, including historical test result trends, failure tracking (if any), and a neat and clean web UI for viewing test reports. Choose the **None** radio button under the **Source Code Management** tab.

As shown in the following figure, select **Build after other projects are built**, and **Trigger only if build is stable** under **Build Triggers**, which denotes that this stage is the next one and will start execution after completion of the previous stage.

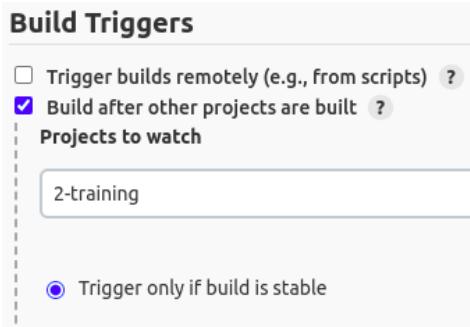


Figure 9.22: 3-testing – Build triggers

As shown in the following figure, under the **Build** tab, Jenkins is running tests inside the running Docker container using the Docker command. Next, the Docker command will export the pytest results in a *.xml* file. Then, the **mkdir** command will create a new directory as *reports* and copy test results to it.

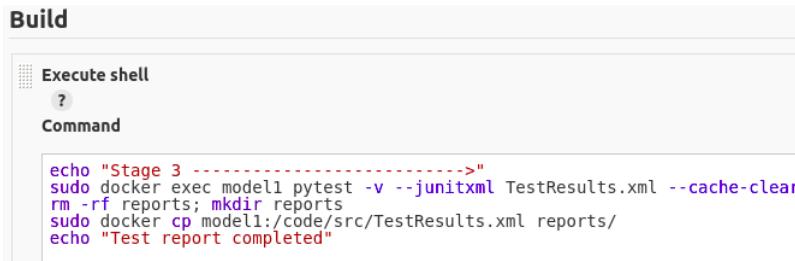


Figure 9.23: 3-testing – Build

Finally, it is publishing the test results using the JUnit plugin.

As shown in the following figure, choose **Publish JUnit test results report**, under **Post-build Actions**.

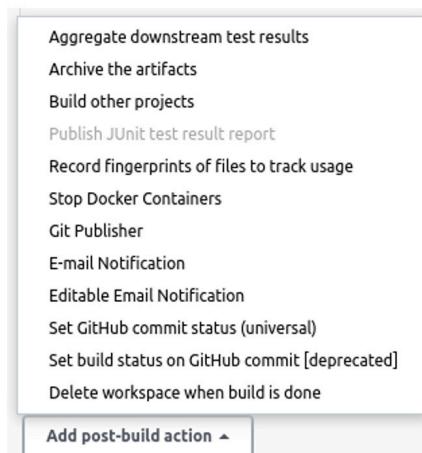


Figure 9.24: 3-testing – Post-build actions

As shown in the following figure, provide a path where you want to store the test results. Jenkins will fetch the test results from the mentioned path and publish the test results.

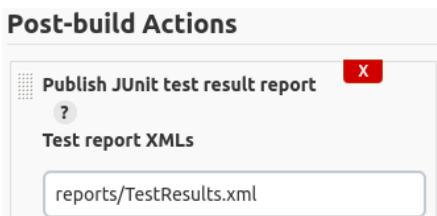


Figure 9.25: 3-testing – JUnit test result report

In the following figure, you can see the build numbers of Stage 1 and Stage 2. Then, it runs the pytest and exports test results in a *.xml* file using JUnit.

Console Output

```

Started by upstream project "2-training" build number 9
originally caused by:
  Started by upstream project "1-GitHub-to-docker-container" build number 19
originally caused by:
  Started by GitHub push by suhas-ds
Running as SYSTEM
Building in workspace /var/lib/jenkins/workspace/3-testing
[3-testing] $ /bin/sh -xe /tmp/jenkins5945505938338825077.sh
+ echo Stage 3 -----
Stage 3 -----
+ sudo docker exec modell pytest -v --junitxml TestResults.xml --cache-clear

```

Figure 9.26: 3-testing – Console output

In the following figure, you can see the test results of the three tests and the time taken for each test.

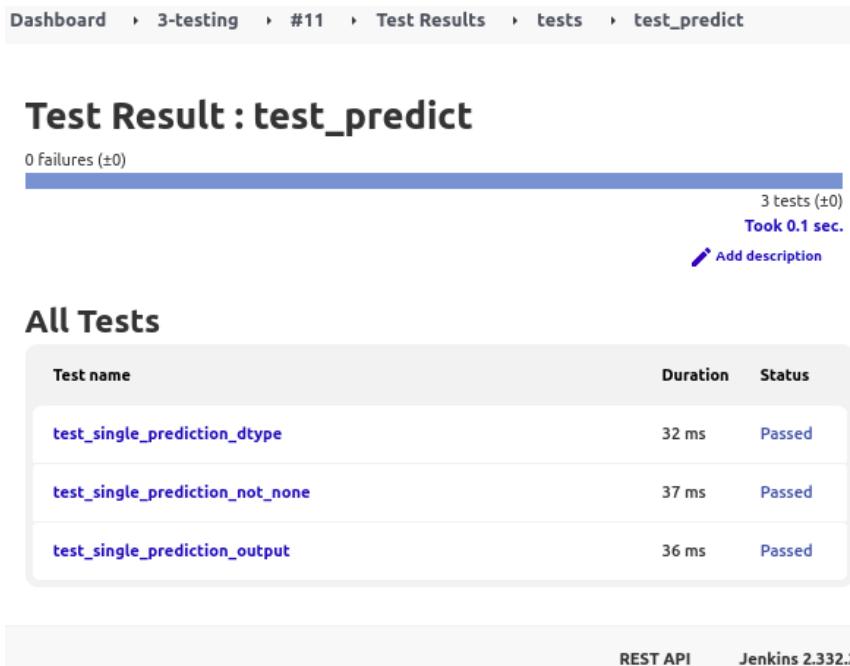


Figure 9.27: 3-testing – Test results

In the following figure, you can see the status of test results in a graphical format.

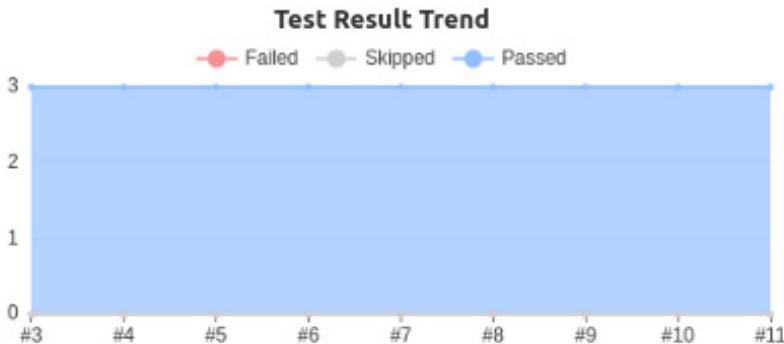


Figure 9.28: 3-testing – Test result trend

Stage 4: 4-deployment-status-email

At this stage, Jenkins is deploying an ML web app using FastAPI and a trained model.

As shown in the following figure, select **Build after other projects are built**, and **Trigger only if build is stable** under **Build Triggers**, which means this stage is the next one and will start execution after completion of the previous stage.

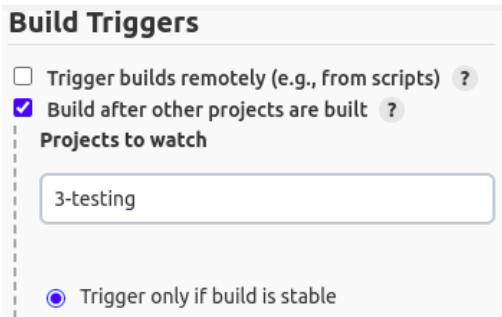


Figure 9.29: 4-deployment-status-email – Build triggers

As shown in the following figure, select **Execute shell** from the dropdown **Add build step**, under the **Build** tab, as it will execute the commands from the shell.

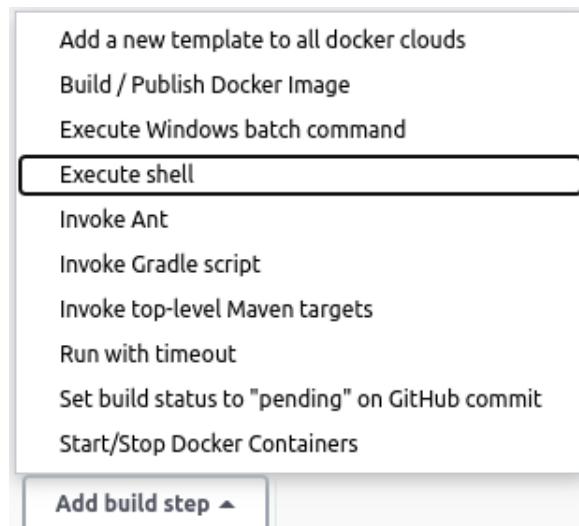


Figure 9.30: 4-deployment-status-email – Execute shell

In the following figure, Jenkins is executing a FastAPI command inside a running Docker container named **model1**, and **-d** in the following command is to instruct the Docker container to run the command in detached mode. And **-w /code** means

it will change the working directory to `/code`. Finally, `--host` and `--port` are the address and port on which the FastAPI web app will be running, respectively.

Build



```
echo "Stage 4 ----->"  
sudo docker exec -d -w /code modell uvicorn main:app --proxy-headers --host 0.0.0.0 --port 8005  
echo "Successfully Deployed"
```

Figure 9.31: 4-deployment-status-email – Build

As shown in the following figure, choose the **Extended Email Notification** from the **Add post-build action** dropdown.

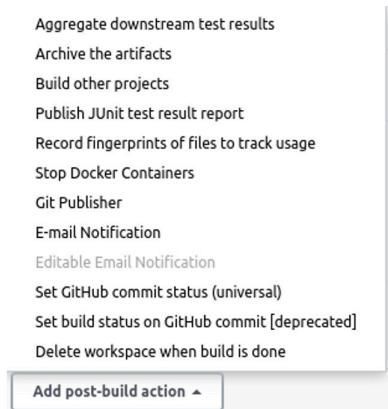


Figure 9.32: 4-deployment-status-email – Post build action

As shown in the following figure, select the **Content Type** as **HTML (text/html)** so that you can use HTML tags in the body; however, it is optional.

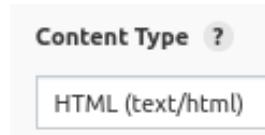


Figure 9.33: 4-deployment-status-email – Content type

As shown in the following figure, select the **Always** option under the **Triggers** section. This enables Jenkins to send an email despite the build status (Success or Failure). It will send the email to the mentioned people.

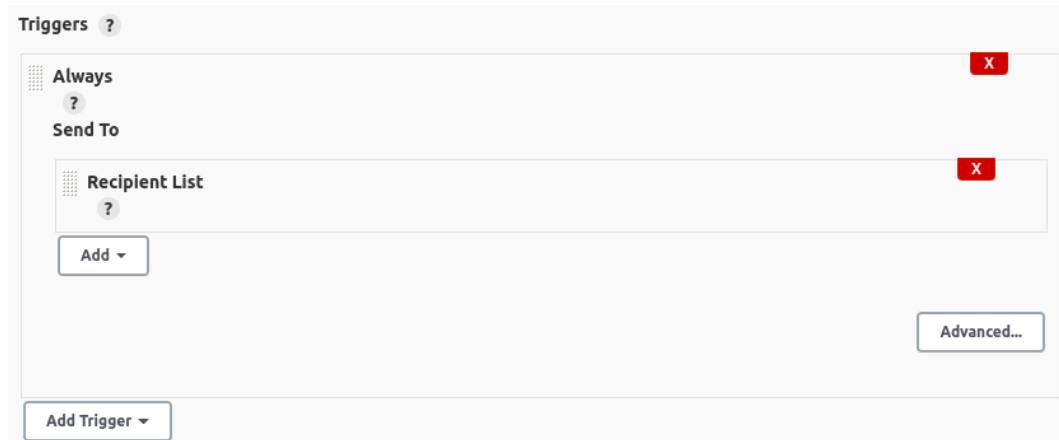


Figure 9.34: 4-deployment-status-email – Triggers

As shown in the following figure, choose the **Compress and Attach Build Log** option from the **Attach Build Log** dropdown. By enabling this option, people will get the console output of the current stage so that people come to know what happened in the job without visiting Jenkin's job; however, it is optional.



Figure 9.35: 4-deployment-status-email – Attach build logs

As you can see in the following figure, an email shows the result of stage 4. If the job fails due to any reason, then this email will be triggered because you selected the **Always** option in the extended email plugin. This email contains the job name, build number, and the status of the job. Here, the **Attach Build Log** option is not selected.

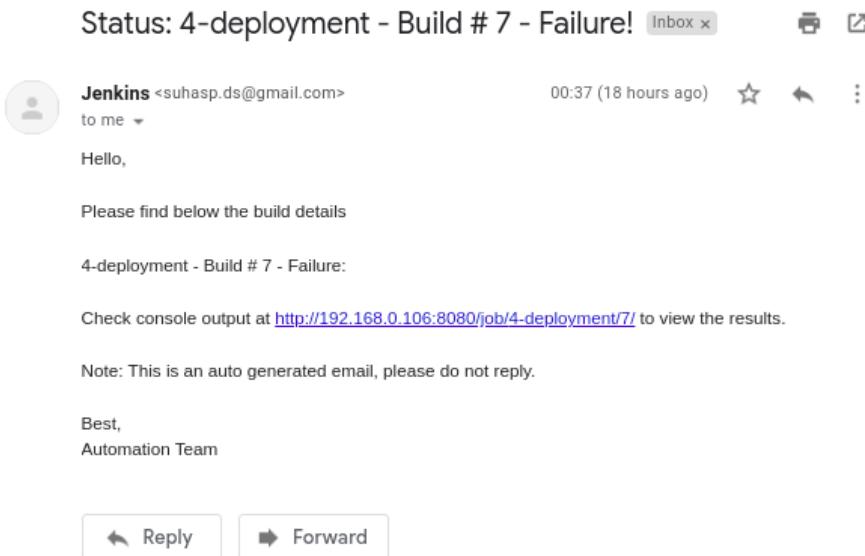


Figure 9.36: 4-deployment-status-email – Failure email

The following figure contains the output of stage 4. As you can see, it shows the status of the preceding stages with the build number. In the end, it also shows the process of sending the email.

Console Output

```

Started by upstream project "3-testing" build number 11
originally caused by:
Started by upstream project "2-training" build number 10
originally caused by:
Started by upstream project "1-GitHub-to-docker-container" build number 20
originally caused by:
Started by GitHub push by suhas-ds
Running as SYSTEM
Building in workspace /var/lib/jenkins/workspace/4-deployment-status-email
No emails were triggered.
[4-deployment-status-email] $ /bin/sh -xe /tmp/jenkins2922115474740220823.sh
+ echo Stage 4 -----
Stage 4 -----
+ sudo docker exec -d -w /code model1 uvicorn main:app --proxy-headers --host 0.0.0.0 --port 8005
+ echo Successfully Deployed
Successfully Deployed
Email was triggered for: Always
Sending email for trigger: Always
Request made to compress build log
Sending email to: suhasp.ds@gmail.com
Finished: SUCCESS

```

Figure 9.37: 4-deployment-status-email – Console output

As you can see in the following figure, the received email is the result of the successful completion of stage 4. After completion of the job without any errors, this email will be triggered as the **Always** option was chosen in the extended email plugin. This email contains the job name, build number, and the status of the job. You can see the build output of the job.

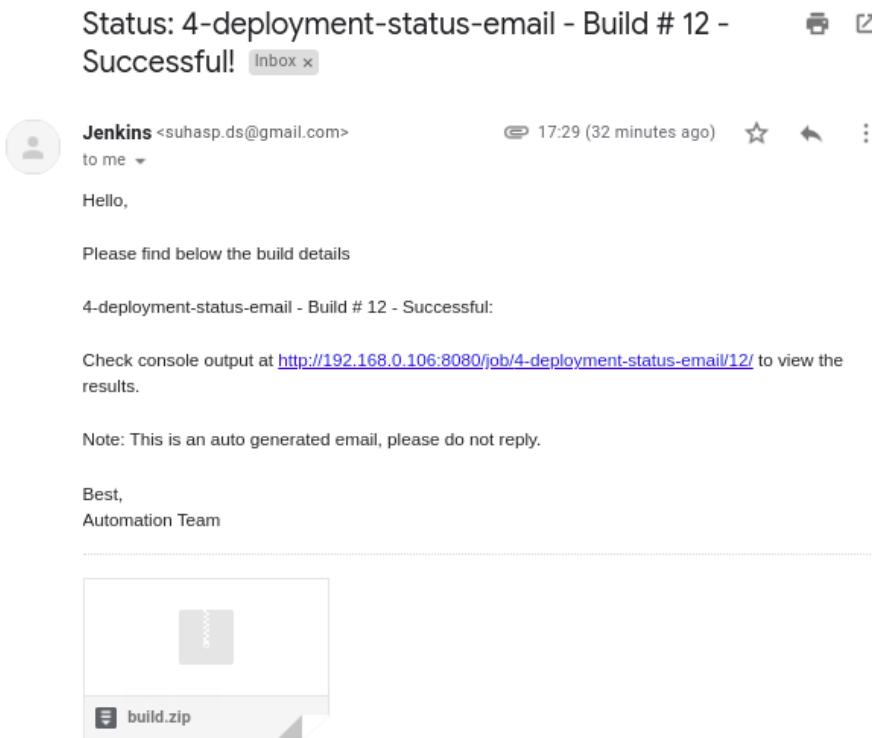


Figure 9.38: 4-deployment-status-email – Success email

The following figure shows the status of all jobs with the last success, last failure status, and duration to complete the job. You can see this status on the Jenkins dashboard.

S	W	Name	Last Success	Last Failure	Last Duration
✓	⌚	1-GitHub-to-docker-container	1 hr 59 min #20	N/A	1 min 44 sec ▶
✓	⌚	2-training	1 hr 57 min #10	2 days 17 hr #2	2.6 sec ▶
✓	⌚	3-testing	1 hr 57 min #11	20 hr #6	2.4 sec ▶
✓	☁️	4-deployment-status-email	1 hr 57 min #12	18 hr #8	0.26 sec ▶

Figure 9.39: Jenkins's dashboard – jobs overview

It's time to check the deployed loan prediction web app. This web app is running on the **8005** port, so you can access it by the server's IP address (or domain name), followed by port **8005**. When any changes are committed to a linked GitHub repository, Jenkins will trigger the first and subsequent stages one by one, and the latest web app will be deployed again. In short, after pushing the updates to the GitHub repository, you can sit and watch the status of each stage or just wait for the email.

Provide the user data and hit the **Execute** button, as shown in the following figure.

The screenshot shows a POST request to the endpoint `/predict` with the sub-action `Get Loan Details`. The form contains the following input fields:

Name	Description
Gender * required string (query)	Male
Married * required string (query)	Yes
Dependents * required string (query)	0
Education * required string (query)	Graduate
Self_Employed * required string (query)	No
ApplicantIncome * required number (query)	5720
CoapplicantIncome * required number (query)	0
LoanAmount * required number (query)	110
Loan_Amount_Term * required number (query)	360
Credit_History * required number (query)	1
Property_Area * required string (query)	Urban

At the bottom of the form is a large blue **Execute** button.

Figure 9.40: FastAPI – ML app

The following figure is the result of the preceding action. You can see the status as **Approved** in the response body. You can also integrate this API endpoint into other applications.

The screenshot shows a user interface for a FastAPI application's response. At the top, there is a 'Responses' section. Below it, under 'Curl', is a code block containing a cURL command to make a POST request to 'http://0.0.0.0:8005/predict' with specific parameters. Under 'Request URL', the URL is shown with query parameters. Under 'Server response', there is a table with one row for status code 200. The 'Details' column shows a JSON response body with a single key-value pair: 'status': 'Approved'. There are download and copy icons next to the response body.

```

curl -X 'POST' \
  'http://0.0.0.0:8005/predict?Gender=Male&Married=Yes&Dependents=0&Education=Graduate' \
  -H 'accept: application/json' \
  -d ''

```

Request URL

```

http://0.0.0.0:8005/predict?
Gender=Male&Married=Yes&Dependents=0&Education=Graduate&Self_Employed=No&ApplicantIncome=5720&CoapplicantIncome=0&LoanAmount=110&Loan_Amount_Term=360&Credit_History=1&Property_Area=Urban

```

Server response

Code	Details
200	Response body <pre>{ "status": "Approved" }</pre> <div style="display: flex; justify-content: space-around;"> </div>

Figure 9.41: FastAPI - Response

Thus, you learned to create a simple CI/CD pipeline using the open-source tool Jenkins. You can modify it as per business and application requirements.

Conclusion

In this chapter, you explored the process of Continuous Integration (CI), Continuous Delivery (CD), Continuous Deployment (CD), and Continuous Training (CT) in the CI/CD pipeline. Also, you learned to create a simple CI/CD pipeline using the popular open-source tool, Jenkins. Moving on, you integrated GitHub and Jenkins using GitHub webhook, and you built a Docker image and ran the container as Jenkins's job. You also executed and exported pytest results using the JUnit plugin. In the last stage, an ML web app was deployed on port 8005. Finally, it triggered the email with the status of the build, the job name, and the build logs.

In the next chapter, you will learn to build CI/CD pipelines that deploy ML apps on the Heroku platform using GitHub Actions.

Points to remember

- CD refers to Continuous Delivery or Continuous Deployment interchangeably, based on the level of automation you are planning to implement.

- Jenkins is an open-source, modular CI/CD automation tool written in Java, which comes with a large number of plugins.
- FastAPI requires Python 3.6 and above.
- Jenkins understands the JUnit test report XML format.

Multiple choice questions

1. Which plugin is used to display the test output in Jenkins?
 - a) pytest
 - b) Selenium
 - c) JUnit
 - d) GitHub
2. CD refers to which of the following?
 - a) Continuous Delivery
 - b) Continuous Deployment
 - c) Create Dictionary
 - d) Both a and b

Answers

1. c
2. d

Questions

1. What is CT in a CI/CD pipeline?
2. What are popular CI/CD tools / platforms?
3. Which plugin is required to show test results in Jenkins?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 10

Deploying ML Models on Heroku

Introduction

Heroku is a **Platform as a Service (PaaS)** platform that enables developers to build, run, and operate applications entirely in the cloud. You can push the Docker container to Heroku or provide GitHub repository details, such as the branch name, to auto-deploy the web app as soon as you push new changes to the model. This way, you can make the ML model accessible on the web, ready to make predictions.

If you are a beginner, then GitHub CI/CD is the simplest platform to build an end-to-end CI/CD pipeline. It is easy to manage as everything is in GitHub. You only need a GitHub repository to create and run a GitHub Actions workflow.

Refer to the previous chapters for the concepts discussed, such as packaging ML models, FastAPI, docker, and CI/CD pipeline.

Structure

This chapter discusses the following topics:

- Create a Heroku account and install Heroku CLI
- Create a Heroku app
- Deploy the web app to Heroku using Heroku CLI
- Build CI/CD pipeline using GitHub Actions

Objectives

After studying this chapter, you should be able to deploy the ML model on Heroku - Platform as a Service (PaaS) and integrate the GitHub repository into Heroku for automated deployment. Create the Heroku app from Heroku web UI and the terminal. You will learn to build and run CI/CD pipelines using GitHub Actions and Heroku. You will also learn to execute multiple tests using pytest and tox on GitHub Actions. Create YAML files for GitHub Actions to run the workflow.

Heroku

Heroku is a container-based cloud Platform as a Service (PaaS) platform. Developers use Heroku to deploy, manage, and scale modern apps. It has a pretty easy process to deploy your applications.

Heroku saves you time by removing the difficulty of maintaining servers, hardware, or infrastructure. It supports the most popular languages, such as Node, Ruby, Java, Clojure, Scala, Go, Python, and PHP.

Simply put, Heroku allows you to make your apps available on the internet for others to access, like a website. With a few steps, it allows you to make your app accessible to others. You can focus on app development without worrying about infrastructure, servers, and other such things.

It enables easy integration with GitHub to make it easy to deploy code available on the GitHub repository to apps that are running on Heroku. When GitHub integration is configured for a Heroku app, Heroku can automatically build and release (if the build is successful) the updates when pushed to the specified repository.

Heroku apps can be scaled to run on multiple dynos (a container that runs a Heroku app's code) simultaneously (except on Free or Hobby dynos) with simple steps to avoid any downtime. Heroku offers manual and auto scaling of the dynos. You can scale it horizontally by adding more dynos to handle the heavy traffic of requests. On the other hand, you can scale it vertically by increasing resources like memory and CPU as required.

Heroku offers Continuous Integration and Continuous Deployment. It has built-in test facilities to achieve Continuous Deployment. Heroku apps that share the same codebase can be organized into deployment pipelines, promoted easily from one stage to the next, and managed through a visual interface.

There are three methods to deploy the web app on the Heroku platform:

- Deployment with Heroku git
- Deployment with GitHub repository integration

- Deployment with Container Registry

Now that you must have got the gist of it, you are going to deploy a Machine Learning model to Heroku. After studying this chapter, you will be in a better position to work on Heroku for ML deployments.

Setting up Heroku

You need to create a Heroku account. This is a one-time activity. Go to the Heroku platform at <https://www.heroku.com/>

Step 1:

Create a Heroku account (if you don't have one) and log in.

Select the primary development language as **Python** and create a free account.

Step 2:

After logging in to the account, create a new app, as shown in the following figure:

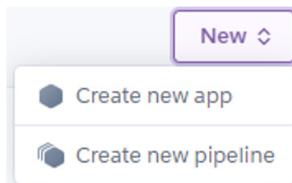


Figure 10.1: Create a new app

Click on the **Create new app** button and add the app name. In this case, it is **docker-ml-cicd**.

Hit the **Create app** button and complete the process.

Note: Two or more Heroku apps can't have the same name.

Step 3:

You can click the **Open app** button to see the app. For the current scenario, it is <https://docker-ml-cicd.herokuapp.com/>

It will display the following text:

Heroku | Welcome to your new app!

Now, go back to your Heroku app dashboard and install the Heroku **Command Line Interface (CLI)** for ease of access through the terminal. Heroku's CLI installation steps are mentioned at <https://devcenter.heroku.com/articles/heroku-cli>. You can

execute Heroku commands through the local terminal. At this stage, create one more app for production, and name it **docker-ml-cicd-prod**.

At this stage, your Heroku account should be ready with the apps you have created.

Deployment with Heroku Git

This method is pretty much straightforward. You need Heroku CLI for this method. Add Heroku's Git remote repository and push the changes to it directly.

From the Heroku application dashboard, go to the **Deploy** tab. You will see three methods of deployment there:

- Heroku Git
- GitHub
- Container Registry

The following figure shows the deployment methods available on the Heroku platform:

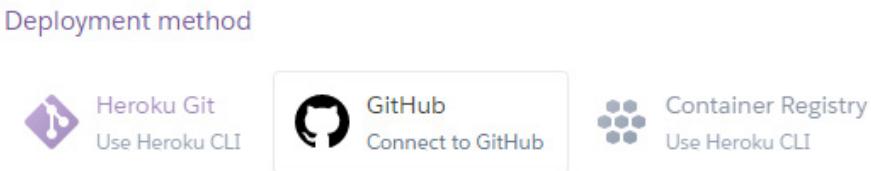


Figure 10.2: Deployment method

From the preceding methods, you need to select the first method, that is, the Heroku Git method, and follow the steps given below that option.

Deployment with GitHub repository integration

Heroku comes with the GitHub integration method, which enables you to automate the deployment process. Post GitHub and Heroku integration, any changes pushed to the repository will trigger the process that deploys the Heroku app at the end.

On Heroku, you can create a pipeline and add both apps to it. Heroku allows you to create review (temporary) apps before deploying the staging app. This additional option enables you to see the working of the app before pushing it to the staging or pre-deployment stage.

The pipeline has been divided into three stages:

- REVIEW APPS
- STAGING
- PRODUCTION

REVIEW APPS

This creates review apps for open pull requests. Each review app has a unique URL that can be shared for testing purposes. For standardizing it, you can set the pattern of the review app's URL.

When there is an open pull request, it gets created automatically for review purposes, and after the pull request is merged, the review app gets deleted. It means that the app is no longer accessible.

Note: You should enable both Heroku Pipelines and GitHub integration for the Heroku app to use review apps.

Under **REVIEW APPS**, hit the **Enable Review Apps** button.

After that, in the pop-up window, select the checkboxes for the following:

- **Create new review apps for new pull requests automatically:** When enabled, every new pull request opened will create a review app automatically, and the pull request that is closed will delete that Review app.
- **Destroy stale review apps automatically:** You get the dropdown option for the lifespan of the app, like After 1 day, 2 days, 5 days, or 30 days. The default value is 5 days.

Click on **Enable Review Apps**.

STAGING

Staging apps can be used to preview code changes and features before being deployed to production, which is known as the pre-production stage. Directly deploying it to the production environment by skipping staging deployment is not recommended. It may happen that an app that is working on your local machine may not work on Heroku (cloud). You might spend a lot of time finding and fixing an issue and lose face in front of customers. Once you verify an app that is working properly in the staging environment, you should push it to the production environment by using the **Promote to production** button.

Under **STAGING**, add the staging app.

PRODUCTION

Production apps run your customer-facing code. It is recommended to promote your code from a staging app only after it has been tested.

Under **PRODUCTION**, you should create a new app for production.

Heroku Pipeline flow

When someone pushes the changes to a branch (other than the master branch), then on the GitHub repository, hit the **Compare & pull request** button and click on the **Create pull request** button. You should see a new app getting built (under **REVIEW APPS**) upon pull request on GitHub. Once the app is ready, you can see that the review app is up and running.

If everything goes well, the lead developer can go back to GitHub and merge the pull request to the master branch of the GitHub repository. Within a few seconds, the staging app should start building. Once it is deployed, you can open it and verify the staging app. However, if you push the changes directly to the master branch, the staging app will start building, and it will bypass the **REVIEW APPS** stage.

You can pass the staging app to the tester or QA team and if everything goes well, one can finally push it to **PRODUCTION**.

When you hit the **Promote to production** button on Heroku, your production app will be live.

The following figure shows the workflow of the automated Heroku pipeline when integrated with the GitHub repository on Heroku.

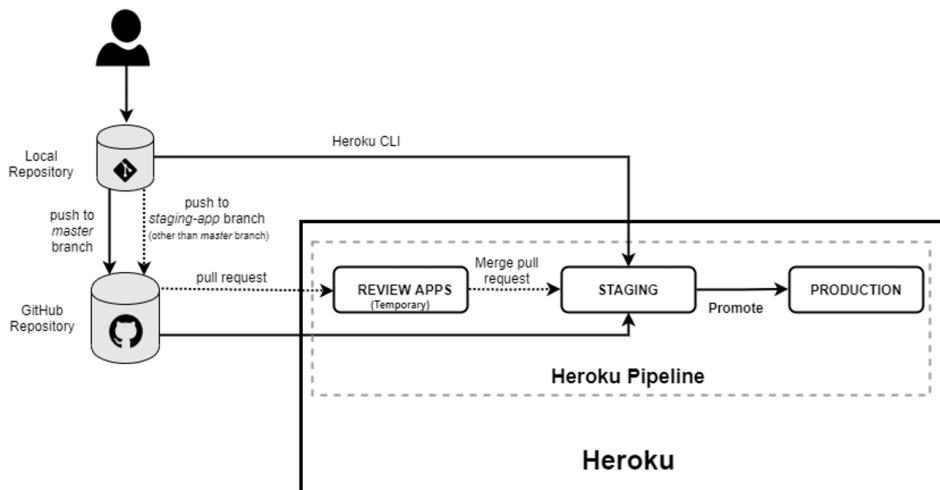


Figure 10.3: Heroku-automated pipeline

Deployment with Container Registry

Heroku Container Registry allows you to deploy your Docker images to Heroku.

Make sure you have a working Docker installation and are logged in to Heroku using the following command:

```
heroku login
```

You'll be prompted to enter any key to go to your web browser to complete the login process.

A new browser window will open up and will ask you to log in. You may add the **-i** or **--interactive** option to stay in the terminal and pass the login details when asked.

Make sure you are in the project directory where *Dockerfile* is located.

Now, log in to Heroku Container Registry:

```
heroku container:login
```

Build the image and push it to the container registry on Heroku using the following syntax:

```
heroku container:push <process-type> --app [app name]
```

In this case, it is a web process, and the **--app** parameter holds Heroku's app name.

Finally, release the image to your app:

```
heroku container:release <process-type> --app [app name]
```

Run the following command to see the app running in the browser:

```
heroku open --app [app name]
```

Note: Pipeline promotions are not supported in the container registry method.

GitHub Actions

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that helps you automate your development workflows at the same place as your code (that is, GitHub repository) and collaborate on pull requests and issues. You can write individual configurable tasks known as actions and combine them to create a custom workflow. Workflows are custom automated processes within GitHub.

The following figure shows the workflow created for the current use case:

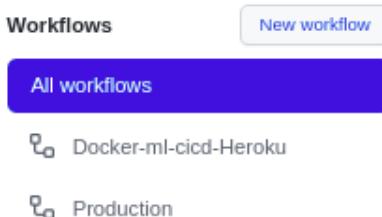


Figure 10.4: GitHub workflow

Unlike Jenkins, GitHub Actions comes with the runners (GitHub-owned servers available for common OSs like Windows, Linux, and macOS) to build, test and deploy the work. GitHub supports a plethora of languages, such as Node.js, Python, Java, Ruby, PHP, Go, Rust, and .NET.

GitHub Actions automation is managed using workflows. Workflows are nothing but the *yml* or *YAML* files you placed in the *.github/workflows* directory in the same project repository.

GitHub Actions are free to use, but some limits are set for them.

You can read more about GitHub Actions at <https://docs.github.com/en/actions>.

Configuration

GitHub Actions need a minimal GitHub configuration. Firstly, go to the GitHub repository where the codebase and required files are available. Next, by default, you will be redirected to the **Code** tab; switch to the **Settings** tab, and from the left panel, choose the **Secrets** option under the **Security** section. After that, select **Actions** under the **Secrets** menu. Finally, use the **New repository secret** button to add the following secrets:

HEROKU_API_KEY: You can get the Heroku API key from the following path on the Heroku platform.

[Heroku Login](#) | [Account settings](#) | [Account](#) | [API Key](#) | [Reveal/Regenerate API Key](#)

HEROKU_APP_NAME: This is the Heroku app name. Here, provide the app name that you have created for staging, that is, **docker-ml-cicd**.

HEROKU_PROD_APP_NAME: This is another app that is to be created for production. In this case, it is **docker-ml-cicd-prod**.

The following figure shows the repository secrets. You can either update or remove the secrets, but cannot see them.

Repository secrets			
HEROKU_API_KEY	Updated 2 days ago	<button>Update</button>	<button>Remove</button>
HEROKU_APP_NAME	Updated 8 hours ago	<button>Update</button>	<button>Remove</button>
HEROKU_PROD_APP_NAME	Updated 4 hours ago	<button>Update</button>	<button>Remove</button>

Figure 10.5: GitHub – Repository secrets

CI/CD pipeline using GitHub Actions and Heroku

Let's consider the scenario of loan prediction, where you need to predict whether a customer's loan will be approved. The focus will not be on hyperparameter tuning and model optimization. However, you can optimize a model to improve its overall performance.

In this chapter, the Machine learning package that was developed earlier (refer to *Chapter 4: Packaging ML Models*) will be used.

Firstly, create a package of ML code and build a web app using FastAPI. After that, create the test cases and dependencies file, along with the *Dockerfile* and *docker-compose.yml* file. Finally, create the GitHub workflow files (*.yml*) for GitHub Actions.

You can access the code repository at the following link:

<https://github.com/suhas-ds/heroku-docker-cicd>

The following directory structure shows the CI/CD pipeline files:

```
.
├── .github/
│   └── workflows/
│       ├── production.yml
│       └── workflow.yml
└── src/
    ├── prediction_model/
    │   └── config/
    │       ├── __init__.py
    │       └── config.py
```

```
|   |   └── datasets/
|   |       ├── __init__.py
|   |       ├── test.csv
|   |       └── train.csv
|   ├── processing/
|   |       ├── __init__.py
|   |       ├── data_management.py
|   |       └── preprocessors.py
|   ├── trained_models/
|   |       ├── __init__.py
|   |       └── classification_v1.pkl
|   ├── VERSION
|   ├── __init__.py
|   ├── pipeline.py
|   ├── predict.py
|   └── train_pipeline.py
|   └── tests/
|       ├── pytest.ini
|       └── test_predict.py
|   └── MANIFEST.in
|   └── README.md
|   └── requirements.txt
|   └── setup.py
|   └── tox.ini
└── .gitignore
└── Dockerfile
└── README.md
└── docker-compose.yml
└── main.py
└── pytest.ini
└── requirements.txt
└── runtime.txt
└── start.sh
└── test.py
└── tox.ini
```

In the `src` directory, files from *Chapter 4: Packaging ML Models* are being used.

`.gitignore`

This file contains files that Git should ignore.

```
1. venv/  
2. __pycache__/  
3. .pytest_cache  
4. .tox
```

`main.py`

First, load the dependencies and modules from `prediction_model`:

```
1. # Importing Dependencies  
2. from fastapi import FastAPI  
3. from pydantic import BaseModel  
4. import uvicorn  
5. import pickle  
6. import os  
7. import numpy as np  
8. import pandas as pd  
9. from fastapi.middleware.cors import CORSMiddleware  
10. from prediction_model.predict import make_prediction  
11. import pandas as pd
```

Create a FastAPI instance and assign it to the `app` so that the `app` will be a point of interaction while creating the API.

```
1. app = FastAPI(  
2.     title="Loan Prediction Model API",  
3.     description="A simple API that uses ML model to predict the Loan  
4.     application status",  
5.     version="0.1",  
6. )
```

CORS (Cross-Origin Resource Sharing) refers to situations when the front end running on a browser has JavaScript code that communicates with the back end,

and the back end is of a different origin from the front end. However, it depends on your application and requirement whether to use it.

```
1. origins = [
2.     "*"
3. ]
4.
5. app.add_middleware(
6.     CORSMiddleware,
7.     allow_origins=origins,
8.     allow_credentials=True,
9.     allow_methods=["*"],
10.    allow_headers=["*"],
11. )
```

Define the class **LoanPred**, which defines the data type expected from the client.

The **LoanPred** class for the data model is inherited from **BaseModel**. Then, add a root view with a function, which returns '**message**': '**Loan Prediction App**' for the home page.

```
1. class LoanPred(BaseModel):
2.     Gender: str
3.     Married: str
4.     Dependents: str
5.     Education: str
6.     Self_Employed: str
7.     ApplicantIncome: float
8.     CoapplicantIncome: float
9.     LoanAmount: float
10.    Loan_Amount_Term: float
11.    Credit_History: float
12.    Property_Area: str
13.
14. @app.get('/')
```

```
15. def index():
16.     return {'message': 'Loan Prediction App'}
17.
18. @app.get('/health')
19. def healthcheck():
20.     return {'status': 'ok'}
```

Here, create `/predict_status` as an endpoint, also known as a route. Then, add `predict_loan_status()` with a parameter of the type data model, that is, `LoanPred`.

```
1. #Defining the function which will make the prediction using the data
which the user inputs
2. @app.post('/predict_status')
3. def predict_loan_status(loan_details: LoanPred):
4.     data = loan_details.dict()
5.     Gender = data['Gender']
6.     Married = data['Married']
7.     Dependents = data['Dependents']
8.     Education = data['Education']
9.     Self_Employed = data['Self_Employed']
10.    ApplicantIncome = data['ApplicantIncome']
11.    CoapplicantIncome = data['CoapplicantIncome']
12.    LoanAmount = data['LoanAmount']
13.    Loan_Amount_Term = data['Loan_Amount_Term']
14.    Credit_History = data['Credit_History']
15.    Property_Area = data['Property_Area']
16.
17.    # Making predictions
18.    input_data = [Gender, Married, Dependents, Education,
19.                  Self_Employed, ApplicantIncome, CoapplicantIncome,
20.                  LoanAmount, Loan_Amount_Term, Credit_History, Property_Area]
21.    cols = ['Gender', 'Married', 'Dependents',
22.             'Education', 'Self_Employed', 'ApplicantIncome',
```

```
23.     'CoapplicantIncome', 'LoanAmount', 'Loan_Amount_Term',
24.     'Credit_History', 'Property_Area']
25. data_dict = dict(zip(cols,input_data))
26. prediction = make_prediction([data_dict])['prediction'][0]
27.
28. if prediction == 'Y':
29.     pred = 'Approved'
30. else:
31.     pred = 'Rejected'
32.
33. return {'status':pred}
```

The following function will create the UI for user input. Here, create **/predict** as an endpoint, also known as a route, and declare input data types expected from users.

```
1. @app.post('/predict')
2. def get_loan_details(Gender: str, Married: str, Dependents: str,
3. Education: str, Self_Employed: str, ApplicantIncome: float,
4. CoapplicantIncome: float, LoanAmount: float, Loan_Amount_Term: float,
5. Credit_History: float, Property_Area: str):
6.
7.     input_data = [Gender, Married, Dependents, Education,
8.                 Self_Employed, ApplicantIncome, CoapplicantIncome,
9.                 LoanAmount, Loan_Amount_Term, Credit_History, Property_Area]
10.    cols = ['Gender', 'Married', 'Dependents',
11.             'Education', 'Self_Employed', 'ApplicantIncome',
12.             'CoapplicantIncome', 'LoanAmount', 'Loan_Amount_Term',
13.             'Credit_History', 'Property_Area']
14.
15.    data_dict = dict(zip(cols,input_data))
16.    prediction = make_prediction([data_dict])['prediction'][0]
17.    if prediction == 'Y':
18.        pred = 'Approved'
```

```
19. else:  
20.     pred = 'Rejected'  
21.  
22. return {'status':pred}  
23.  
24.  
25.if __name__ == '__main__':  
26. pass
```

The file for FastAPI is completed.

requirements.txt

Now, create the *requirements.txt* file, as follows. In this file, you can define the model requirements, test requirements, and FastAPI requirements separately for better understanding and ease of management.

```
1. # Model building requirements  
2. joblib==0.16.0  
3. numpy==1.19.0  
4. pandas==1.0.5  
5. scikit-learn==0.23.1  
6. scipy==1.5.1  
7. sklearn==0.0  
8.  
9. # testing requirements  
10. pytest<5.0.0,>=4.6.6  
11. requests  
12.  
13. # packaging  
14. setuptools==40.6.3  
15. wheel==0.32.3  
16.  
17. # FastAPI app requirements  
18. fastapi>=0.68.0,<0.69.0
```

```
19. pydantic>=1.8.0,<2.0.0  
20. uvicorn>=0.15.0,<0.16.0  
21. gunicorn>=20.1.0
```

Dockerfile

Dockerfile contains a list of commands or instructions to be executed while building the Docker image. Docker uses this file to build the Docker image.

```
1. FROM python:3.7-slim-buster  
2.  
3. COPY ./start.sh /start.sh  
4.  
5. RUN chmod +x /start.sh  
6.  
7. ENV PYTHONPATH "${PYTHONPATH}:app/src/"  
8.  
9. COPY . /app  
10.  
11. RUN chmod +x /app  
12.  
13. RUN pip install --no-cache-dir --upgrade -r app/requirements.txt  
14.  
15. CMD ["./start.sh"]
```

docker-compose.yml

The web service builds from the *Dockerfile* in the current directory and mounts the *app* directory on the host to */app* inside the container.

```
1. version: "3.9" # optional since v1.27.0  
2. services:  
3.   web:  
4.     build: .  
5.     volumes:  
6.       - ./app:/app
```

pytest.ini

Pytest allows you to use a global configuration file, that is, *pytest.ini*, where you can keep the settings and additional arguments that are to be passed whenever you run the command. Add **-p no:warnings** to the **addopts** option, which will suppress the warnings. This section will execute the parameters when pytest runs.

```
1. [pytest]
2. addopts = -p no:warnings
```

runtime.txt

Declare the Python version to be used in this file.

```
1. python-3.7
```

start.sh

This is a shell script that contains a series of commands to be executed by the bash shell. The first line of this file tells which interpreter should be used to execute this script.

It will install the **prediction_model** package from **src/** placed in the **app/** folder and run the FastAPI app in the next command. Here, it will get the PORT from the Heroku environment, so you do not need to declare it explicitly. Heroku will run the app on any available port. The **main** is the file name located in the *app* directory, followed by **:app**, which is a FastAPI object to be called.

```
1. #!/bin/sh
2. pip install app/src/
3. uvicorn app.main:app --host 0.0.0.0 --port $PORT
```

Make sure you remove the **--reload** option if you are using it. The **--reload** option consumes much more resources; moreover, it is unstable. You can use it during development but should not use it in production.

test.py

This file will enable you to test the FastAPI app using **TestClient**. FastAPI provides the same **starlette.testclient** as **fastapi.testclient**; however, it comes directly from Starlette. With this, you can check the app's routes without running the app explicitly.

This file will test the root path of the app and sample prediction by passing the data.

```
1. # Importing dependencies
2. from main import app
3. from fastapi.testclient import TestClient
```

```
4. import pytest
5. import requests
6. import json
```

Create a **TestClient()** by passing a FastAPI application to it as an argument:

```
1. client = TestClient(app)
```

Define the functions with a name starting with **test_** as per standard pytest conventions:

```
1. def test_read_main():
2.     response = client.get("/")
3.     assert response.status_code == 200
4.     assert response.json() == {'message': 'Loan Prediction App'}
```

Write simple **assert** statements to check the output:

```
1. def test_pred():
2.     data = {
3.         "Gender": "Male",
4.         "Married": "Yes",
5.         "Dependents": "0",
6.         "Education": "Graduate",
7.         "Self_Employed": "No",
8.         "ApplicantIncome": 5720,
9.         "CoapplicantIncome": 0,
10.        "LoanAmount": 110,
11.        "Loan_Amount_Term": 360,
12.        "Credit_History": 1,
13.        "Property_Area": "Urban"
14.    }
15.
16.    response = client.post("/predict_status", json=data)
17.    assert response.json()["status"] != ''
18.    assert response.json() == {"status": "Approved"}
```

tox.ini

This is the tox configuration file. Tox automates and standardizes the testing in Python. It is a virtualenv management and test command-line tool to check if your package is compatible with different Python versions. Tox will first create a virtual environment based on the configuration provided, install dependencies and finally execute the commands provided in the configuration.

tox-gh-actions is a plugin that enables tox to run on GitHub Actions. Hence, you need to install **tox-gh-actions** in the GitHub Actions workflow before running the tox command.

This file aims to check the functioning of the package against Python-3.7. It will install the required dependencies and run the pytest command.

```
1. [tox]
2. envlist = py37
3. skipsdist=True
4.
5. [gh-actions]
6. python =
7.   3.7: py37
8.
9. [testenv]
10. install_command = pip install {opts} {packages}
11. deps =
12.   -r requirements.txt
13.
14. setenv =
15.   PYTHONPATH=src/
16.
17. commands=
18.   pip install requests
19.   pytest -v test.py
20.   pytest -v src/tests/
```

workflow.yml

GitHub Actions will look for this file for executing the workflow. This workflow is used for staging or pre-production app deployment on Heroku. It gets triggered on a push event, which means any updates pushed to the project's **master** branch of the GitHub repository will trigger this workflow to run. Under jobs, you should declare the OS and Python version on which it should run. Then, it will run the pytest using tox. After passing all tests, it will start the deployment on Heroku. For this, it will use the secrets defined in the settings of the GitHub repository.

```
1. name: Docker-ml-cicd-Heroku
2.
3. on:
4.   push:
5.     branches:
6.       - master
7.
8. jobs:
9.   build:
10.    runs-on: ubuntu-18.04
11.   strategy:
12.     matrix:
13.       python-version: ['3.7']
14.
15.   steps:
16.     - uses: actions/checkout@v2
17.     - name: Permissions
18.       run: chmod +x start.sh
19.     - name: Set up Python ${{ matrix.python-version }}
20.       uses: actions/setup-python@v2
21.       with:
22.         python-version: ${{ matrix.python-version }}
23.     - name: Install dependencies
24.       run: |
25.         python -m pip install --upgrade pip
```

```
26.      python -m pip install tox tox-gh-actions
27.      - name: Test with tox
28.      run: tox
29.      - name: Log in to Heroku Container registry
30.      env:
31.          HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
32.      run: heroku container:login
33.      - name: Build and push
34.      env:
35.          HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
36.      run: heroku container:push web --app ${secrets.HEROKU_APP_NAME }
37.      - name: Release
38.      env:
39.          HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
40.      run: heroku container:release web --app ${secrets.HEROKU_APP_NAME }
```

production.yml

On successful completion of *workflow.yml* for staging, *production.yml* workflow will run. This workflow will simply deploy a production app on Heroku using the latest updates, assuming everything is working fine.

```
1. name: Production
2.
3. on:
4.   workflow_run:
5.     workflows: [Docker-ml-cicd-Heroku]
6.     types:
7.       - completed
8. jobs:
9. build:
10.    runs-on: ubuntu-18.04
```

```
11.   steps:
12.     - uses: actions/checkout@v2
13.     - name: Log in to Heroku Container registry
14.   env:
15.     HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
16.   run: heroku container:login
17.     - name: Build and push
18.   env:
19.     HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
20.   run: heroku container:push web --app ${secrets.HEROKU_PROD_APP_NAME }
21.     - name: Release
22.   env:
23.     HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
24.   run: heroku container:release web --app ${secrets.HEROKU_PROD_APP_NAME }
```

Once all files and codebase are ready, run the app on the local machine. If the app is running on a local machine, then deploy the app to the Heroku container registry. For this, you can refer to the *Deployment with Container Registry* section discussed earlier in this chapter. If everything goes well, the app should run on the Heroku platform post deployment.

Next, create a repository on GitHub for the current scenario (if not created already) and push code files to it. Go to the **Actions** tab on GitHub; you will see GitHub has already started executing the *workflow.yml* file for staging or pre-deployment app.

As you can see in the following figure, updates have been pushed with the comment as **Staging**.

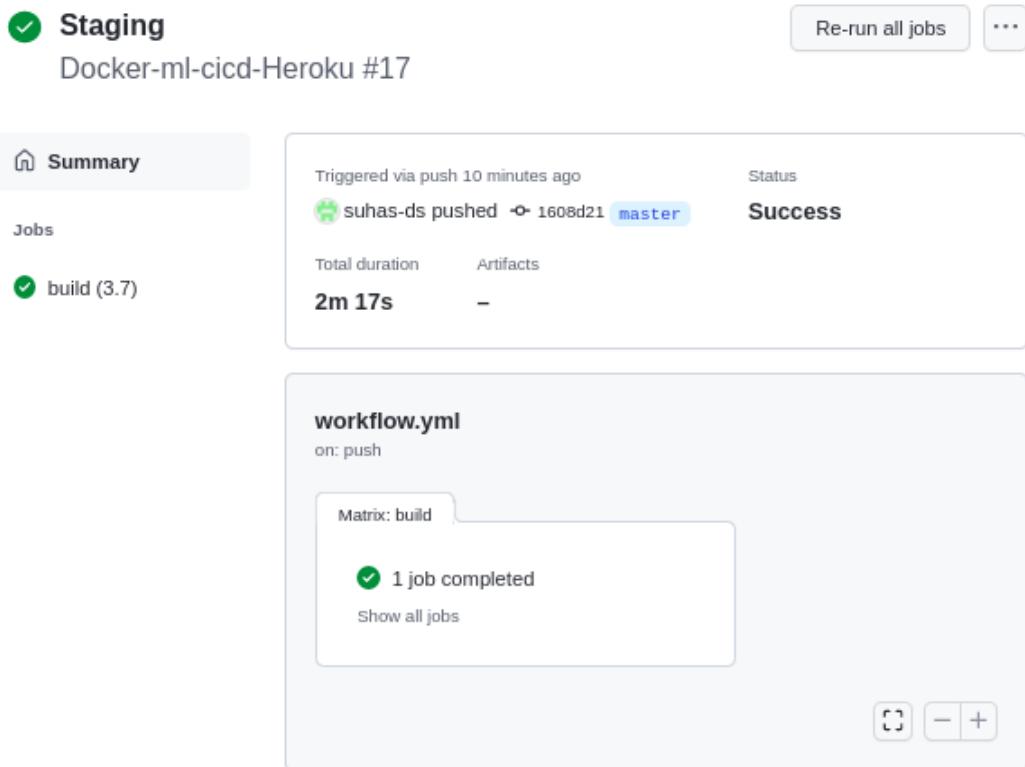
The screenshot shows a GitHub repository page for 'suhas-ds/heroku-docker-cicd'. The 'Actions' tab is active. There are two workflow runs listed:

- Production**: Completed by suhas-ds. Last run 6 minutes ago, took 1m 39s.
- Staging**: Docker-ml-cicd-Heroku #17: Commit 1608d21 pushed by suhas-ds. Last run 8 minutes ago, took 2m 17s. The status is labeled 'master'.

Figure 10.6: GitHub workflow runs

Click on **Staging**; it will take you to the summary of this workflow. You will see the status of the jobs you have defined in the workflow. Other execution details are mentioned, such as the time taken to complete this workflow, the username means who pushed the updates to the master branch, and the name of the workflow file.

In the following figure, you can see the summary of the staging stage:



The image shows a GitHub workflow interface. At the top, there's a green checkmark icon followed by the word "Staging". Below it, the text "Docker-ml-cicd-Heroku #17" is displayed. To the right are two buttons: "Re-run all jobs" and "...".

On the left, there's a sidebar with a house icon and the word "Summary". Below it, under "Jobs", there's a green checkmark icon next to "build (3.7)".

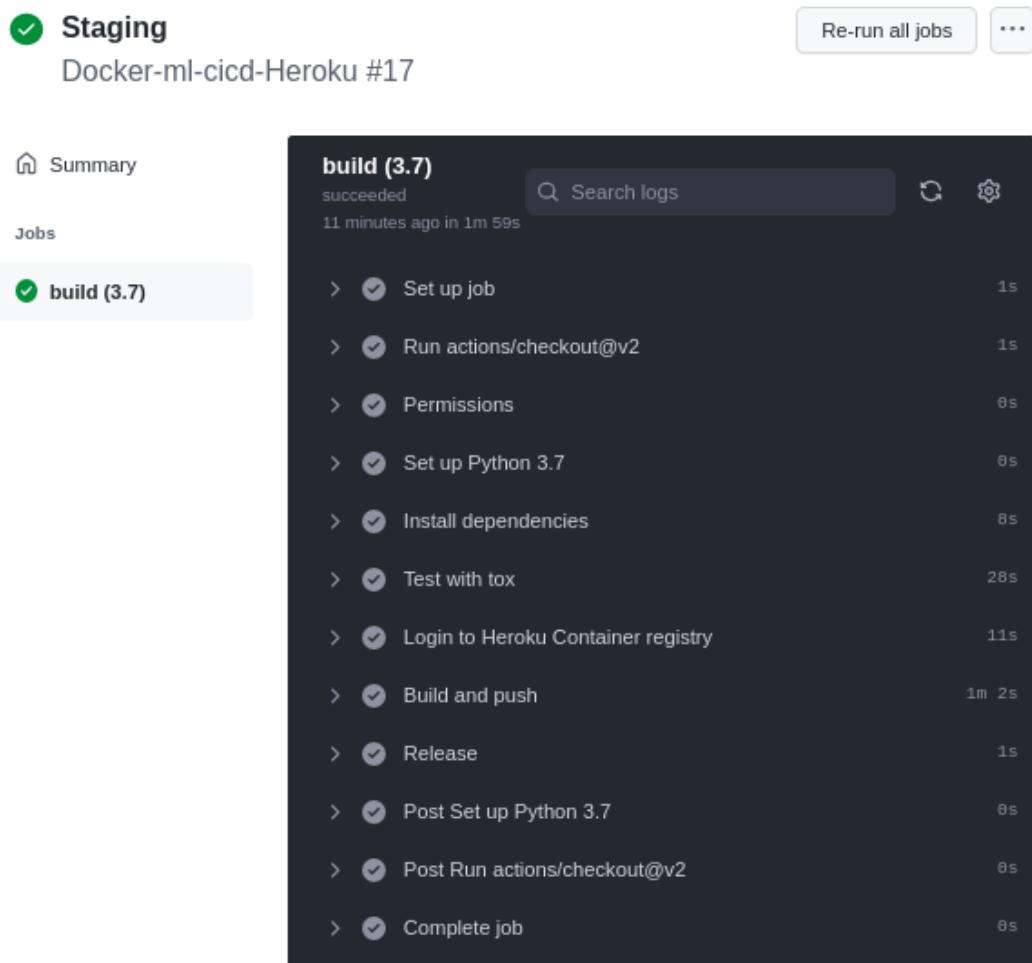
The main area has a light gray header with the text "Triggered via push 10 minutes ago" and "Status Success". It also shows "suhas-ds pushed → 1608d21 master" and "Total duration 2m 17s".

Below this, there's a section titled "workflow.yml" with the subtitle "on: push". It shows a "Matrix: build" section with a green checkmark icon and the text "1 job completed". There's also a "Show all jobs" link.

At the bottom right of the main area, there are three small icons: a square with a circle, a minus sign, and a plus sign.

Figure 10.7: GitHub workflow - staging

The following figure shows the steps executed in the workflow and their output. To see the output of the step, click on the > icon on the left side of the step.

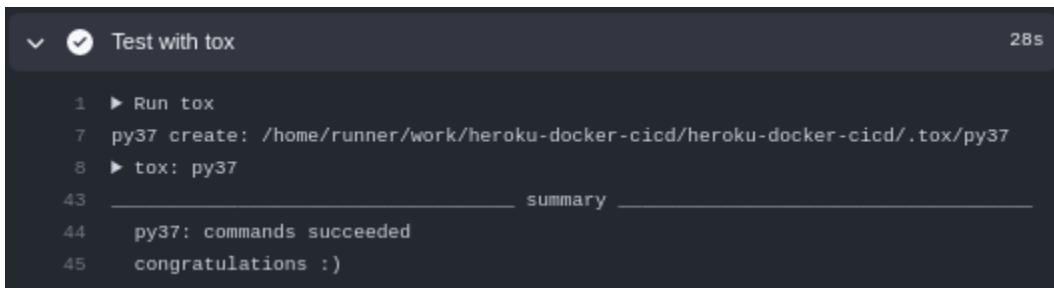


The screenshot shows a GitHub Actions workflow named "build (3.7)" for a repository "Docker-ml-cicd-Heroku #17". The workflow has completed successfully 11 minutes ago in 1m 59s. It consists of the following steps:

- > Set up job (1s)
- > Run actions/checkout@v2 (1s)
- > Permissions (0s)
- > Set up Python 3.7 (0s)
- > Install dependencies (8s)
- > Test with tox (28s)
- > Login to Heroku Container registry (11s)
- > Build and push (1m 2s)
- > Release (1s)
- > Post Set up Python 3.7 (0s)
- > Post Run actions/checkout@v2 (0s)
- > Complete job (0s)

Figure 10.8: GitHub workflow – staging steps

The following figure shows the output of the tox command, which executed pytest commands using the py37 environment.



```

1 ► Run tox
7 py37 create: /home/runner/work/heroku-docker-cicd/heroku-docker-cicd/.tox/py37
8 ► tox: py37
43 _____ summary _____
44 py37: commands succeeded
45 congratulations :)

```

Figure 10.9: GitHub workflow – tox

The following figure shows the app running on the Heroku platform on successful completion of *workflow.yml* workflow.

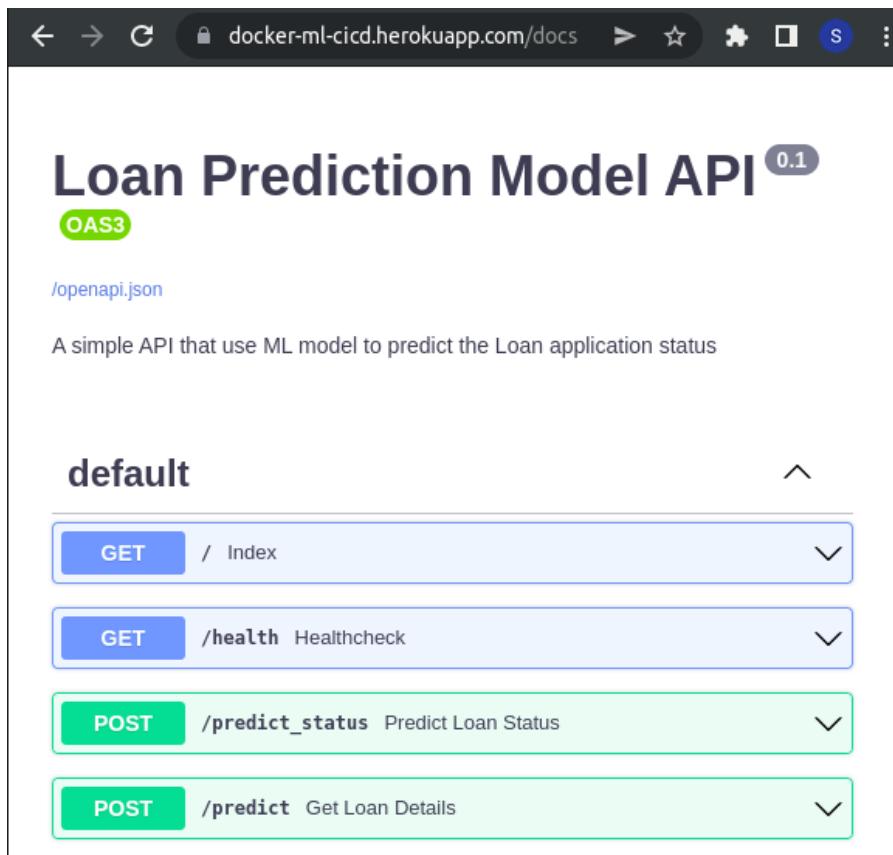
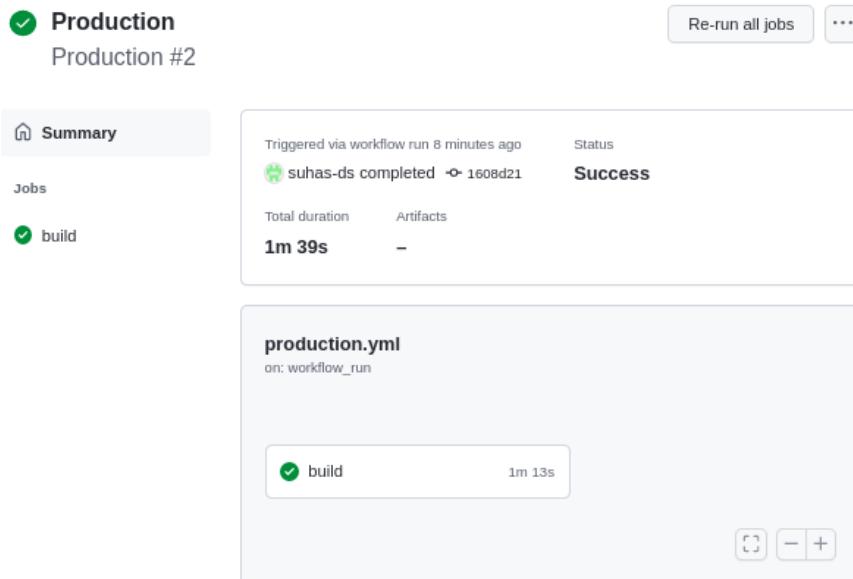


Figure 10.10: Heroku app – docker-ml-cicd

Click on **Production**; it will take you to the summary of this workflow. You will see the status of the jobs you have defined in the workflow. Other execution details are mentioned, such as the time taken to complete the workflow, the username that pushed the updates to the master branch, the name of the workflow file, and build no., in this case, it is #2.

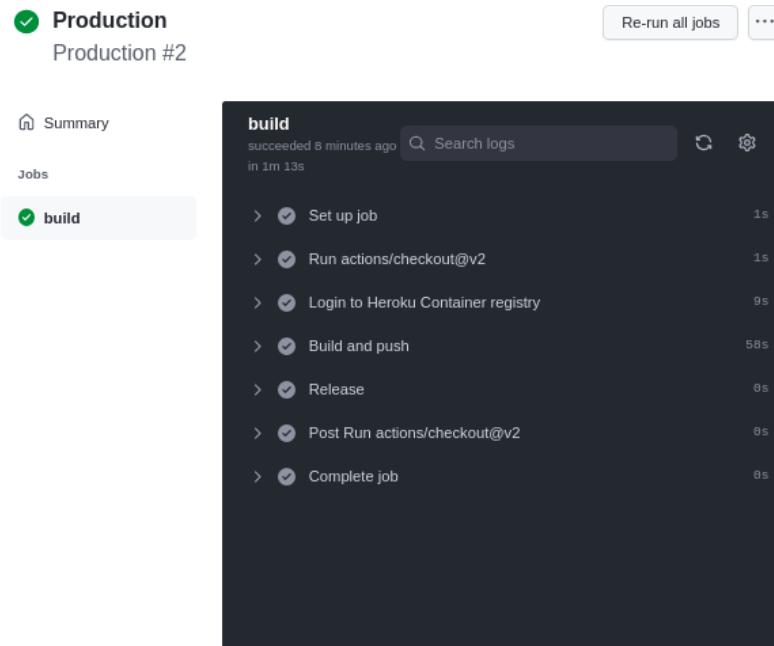
In the following figure, you can see the summary and status of the production stage:



The image shows two panels from a GitHub workflow interface. The top panel is titled 'Production' and 'Production #2'. It has a 'Summary' tab selected, showing a green checkmark icon next to 'build'. The status is 'Success'. The job was triggered via workflow run 8 minutes ago, completed at 16:08:21, and had a total duration of 1m 39s. There are no artifacts. The bottom panel shows the 'production.yml' configuration file, specifically the 'on: workflow_run' section, which triggers a 'build' step. This step succeeded 8 minutes ago, took 1m 13s, and has a green checkmark icon.

Figure 10.11: GitHub workflow – production steps

The following figure shows the steps executed in the workflow and their output. To see the output of the step, click on the > icon on the left side of the step.



This panel provides a detailed view of the 'build' step from the previous screenshot. It lists the individual actions and their execution times. The actions are: Set up job (1s), Run actions/checkout@v2 (1s), Login to Heroku Container registry (9s), Build and push (58s), Release (0s), Post Run actions/checkout@v2 (0s), and Complete job (0s). Each action has a green checkmark icon and a right-pointing arrow indicating its status.

Figure 10.12: GitHub workflow – production steps

The following figure shows the app running on the Heroku platform on successful completion of the *production.yml* workflow.

The screenshot shows a browser window with the URL `docker-ml-cicd-prod.herokuapp.com/docs`. The page title is "Loan Prediction Model API". It includes a link to `/openapi.json` and a green "OAS3" badge. A brief description states: "A simple API that uses ML model to predict the Loan application status". Below this, there is a section titled "default" containing four API endpoints:

- GET / Index**
- GET /health Healthcheck**
- POST /predict_status Predict Loan Status**
- POST /predict Get Loan Details**

Figure 10.13: Heroku app – docker-ml-cicd-prod

Thus, you have now learned to create a simple CI/CD pipeline using GitHub to deploy ML apps on the Heroku platform. Furthermore, you can modify it as per business and application requirements.

Conclusion

In this chapter, you have learned about the ML app deployment on the Heroku platform (PaaS), built an automated CI/CD pipeline using GitHub Actions, and deployed staging or pre-deployment and production app on the Heroku platform using GitHub CI/CD pipeline. Finally, you integrated tox with GitHub Actions to run the test as a part of the CI/CD pipeline before deploying the staging app to Heroku.

In the next chapter, you will explore the Azure platform for deploying ML apps.

Points to remember

- GitHub Actions come with runners (GitHub-owned servers for major OSs such as Windows, Linux, and macOS) to build, test and deploy the workflow.

- The Heroku app must enable both Heroku Pipelines and GitHub integration to use review apps.
- You can either update or remove the GitHub secrets but cannot see them.
- GitHub Actions is free to use; however, some limits are set for them.

Multiple choice questions

1. _____ is a plugin that enables tox running on GitHub Actions.
 - a) JUnit
 - b) tox-gh-actions
 - c) GitHub'
 - d) tox-github
2. GitHub Actions automation is managed using _____.
 - a) .git
 - b) tox
 - c) Workflow
 - d) All the above

Answers

1. b
2. c

Questions

1. What is the file format of the GitHub Actions workflow?
2. What are the three methods to deploy the app on the Heroku platform?
3. Explain the working stages of the Heroku pipeline.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 11

Deploying ML Models on Microsoft Azure

Introduction

Microsoft Azure is a popular cloud platform among developers, as it comes with a wide variety of services that help developers and organizations to deliver quality solutions to customers with less effort and less time in a secure environment.

In this chapter, you will be acquainted with MLaaS, that is, Machine Learning as a Service, offered by Microsoft Azure. This chapter is mainly divided into two parts. In the first part, GitHub Actions and Azure web app containers will be used in the CI/CD pipeline to deploy the ML app on the Azure cloud. In the second part, Azure DevOps and **Azure Machine Learning (AML)** service will be used to deploy ML apps on the Azure cloud.

Structure

This chapter discusses the following topics:

- Create an Azure account and install Azure CLI
- Run the ML app using the Docker container locally
- Deploy the app to Azure web service using Azure container

- Build a CI/CD pipeline using GitHub Actions
- Azure Machine Learning (AML) service
- Build a CI/CD pipeline using Azure Machine Learning (AML) service and Azure DevOps

Objectives

After studying this chapter, you should be able to deploy ML models on **Platform as a Service (PaaS)** and **ML as a Service (MLaaS)**. You should also be able to integrate the GitHub repository to Azure for automated deployment of the app. You should know how to create a web app and container for it. Additionally, you will be familiar with how to build and run CI/CD pipelines using GitHub Actions and Azure and execute multiple test cases using pytest and tox on GitHub Actions. Further on in the chapter, you will learn how to create YAML files for GitHub Actions to run the workflow. By the end of the chapter, you should also be able to build and run CI/CD pipelines using Azure DevOps.

Azure

Microsoft Azure (also known as Azure) is a cloud computing service offered by Microsoft. Azure provides different forms of cloud computing options, such as Software as a Service (SaaS), Platform as a Service (PaaS), ML as a Service (MLaaS), and Infrastructure as a Service (IaaS), for deploying applications and services on Azure.

Microsoft first introduced its cloud computing services as Windows Azure in 2008, but it was commercially launched in 2010. Later, in 2014, they expanded their services and re-launched it as Microsoft Azure.

Azure is a ready-to-go, resourceful, flexible, and fast yet economical cloud platform. It comes with 200+ products and cloud services; however, running **Virtual Machines (VM)** or containers is popular among developers. It is compatible with open-source technologies like Docker, Jenkins, and Kubernetes. Azure has a built-in Continuous Integration and Continuous Deployment pipeline.

Azure enables integration with GitHub to make it easy to deploy code available on the GitHub repository to apps running on Azure. When GitHub integration is configured for an Azure app, Azure can automatically build and release (if the stage is completed successfully) and push it to Azure.

Azure App Service can be scaled using **Azure Kubernetes Service (AKS)**. You can choose different tiers that come with a set of computational power and set the auto scale limit. When required, it will make replicas of the service to serve the requests seamlessly.

There are numerous ways to deploy the app on the Azure platform, but you will explore two methods in this chapter:

- Deployment using GitHub Actions
- Deployment using Azure DevOps and Azure ML

Azure primarily uses a pay-as-you-go pricing plan, which allows you to pay only for the services you have used. If any application uses multiple services, then each service will be billed based on the plan/tier obtained for it. Microsoft offers a discounted rate if the user or organization is looking for a long-term commitment.

Azure is a paid service, but it allows new users to explore its functionality and services for free for a limited duration using free credits. After that, you would have to activate a pay-as-you-go pricing plan.

Set up an Azure account

You need to set up an Azure account. This is a one-time activity, and once it is done, it is ready for use. Go to the Azure platform <https://azure.microsoft.com/>.

Create an Azure account (if you don't have one), and log in.

Note: You can explore Azure for free; however, you need to provide credit/debit card details. Azure will send a notification if any payment is to be made. Also, you will get access to popular services for free for 12 months, with an additional \$200 credit for 30 days.

Azure DevOps will be used in this chapter. For this, you need to log in to Azure DevOps.

<https://azure.microsoft.com/en-us/services/devops/>

To connect and communicate with Azure, the Azure command-line tool is required, that is, **azure-cli** and **azureml-sdk**.

Install Azure CLI with your local terminal:

```
pip install azure-cli==2.37.0  
pip install --upgrade azureml-sdk[cli]
```

You can create a resource group. It is a container that holds related resources for Azure projects. You can allocate the resources to this resource group as per requirement.

Deployment using GitHub Actions

In this part, an ML app will be deployed in Azure web service using Docker. GitHub Actions will be used for building, testing, and deploying ML apps to Azure. First

off, GitHub Actions will build and push a Docker image to the **Azure Container Registry (ACR)** and pull it into the Azure App Service.

Firstly, prepare the required codebase for this implementation. Next, create an ACR, and then build and push the Docker image to the ACR. In the next step, create a web app in the Azure App Service, which will pull container images from the ACR to deploy a web-based ML app.

The following figure shows the workflow of deployment on Azure App Service (web app) using GitHub Actions:

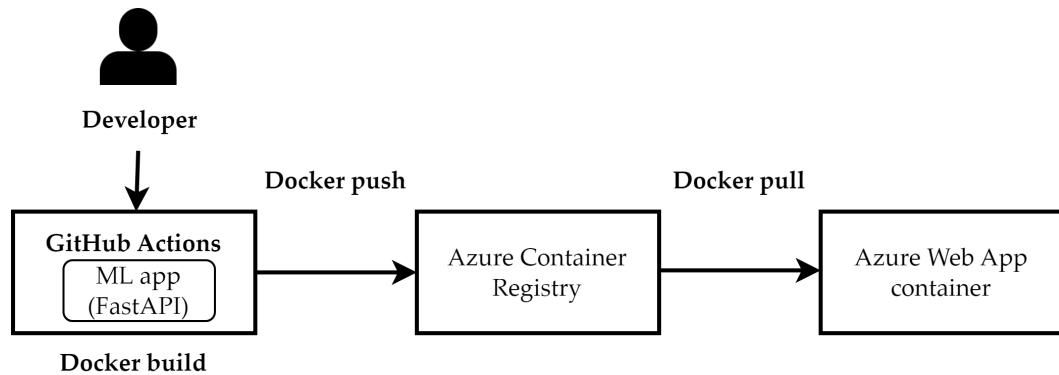


Figure 11.1: GitHub Actions and web app workflow

You can access the code at the following GitHub repository:

<https://github.com/suhas-ds/mlapp-cd>

The following directory structure shows the files that will be used for the CI/CD pipeline:

```

.
├── .github
│   └── workflows
│       └── prod.workflow.yml
└── src
    ├── prediction_model
    │   ├── config
    │   │   ├── __init__.py
    │   │   └── config.py
    │   └── datasets
    │       └── __init__.py
  
```

```
|   |   |   └ test.csv
|   |   └ train.csv
|   └ processing
|       |   └ __init__.py
|       |   └ data_management.py
|       └ preprocessors.py
|   └ trained_models
|       |   └ __init__.py
|       └ classification_v1.pkl
|   └ VERSION
|   └ __init__.py
|   └ pipeline.py
|   └ predict.py
|   └ train_pipeline.py
|   └ tests
|       |   └ pytest.ini
|       └ test_predict.py
|   └ MANIFEST.in
|   └ README.md
|   └ requirements.txt
|   └ setup.py
|   └ tox.ini
└ Dockerfile
└ docker-compose.yml
└ main.py
└ pytest.ini
└ requirements.txt
└ runtime.txt
└ start.sh
└ test.py
└ tox.ini
```

Let's discuss the code and the concepts from the preceding files. In the `src` directory, files from *Chapter 4: Packaging ML Models* will be used.

`.gitignore`

This file contains the names of files and directories that Git should ignore.

```
1. venv/  
2. __pycache__/  
3. .pytest_cache  
4. .tox
```

main.py

This file will load the pickle object of the model and run a FastAPI app. First, load the dependencies and modules from **prediction_model**:

```
1. # Importing Dependencies  
2. from fastapi import FastAPI  
3. from pydantic import BaseModel  
4. import uvicorn  
5. import pickle  
6. import os  
7. import numpy as np  
8. import pandas as pd  
9. from fastapi.middleware.cors import CORSMiddleware  
10. from prediction_model.predict import make_prediction  
11. import pandas as pd
```

Create a FastAPI instance and assign it to the **app** so that the **app** becomes a point of interaction while creating the API.

```
1. app = FastAPI(  
2.     title="Loan Prediction Model API",  
3.     description="A simple API that uses ML model to predict the Loan  
application status",  
4.     version="0.1",  
5. )
```

CORS (Cross-Origin Resource Sharing) refers to the situations when a front end running in a browser has JavaScript code that communicates with a back end, and the back end is of a different origin than the front end. However, it depends on your application and requirement whether to use it.

```
1. origins = [
```

```
2.      "*"
3.  ]
4.
5. app.add_middleware(
6.     CORSMiddleware,
7.     allow_origins=origins,
8.     allow_credentials=True,
9.     allow_methods=["*"],
10.    allow_headers=["*"],
11. )
```

Define the class **LoanPred**, which defines the data type expected from the client.

The **LoanPred** class for the data model is inherited from **BaseModel**. Then, add a root view with a function, which returns '**message**': '**Loan Prediction App**' for the home page.

```
1. class LoanPred(BaseModel):
2.     Gender: str
3.     Married: str
4.     Dependents: str
5.     Education: str
6.     Self_Employed: str
7.     ApplicantIncome: float
8.     CoapplicantIncome: float
9.     LoanAmount: float
10.    Loan_Amount_Term: float
11.    Credit_History: float
12.    Property_Area: str
13.
14. @app.get('/')
15. def index():
16.     return {'message': 'Loan Prediction App'}
17.
```

```
18. @app.get('/health')
19. def healthcheck():
20.     return {'status': 'ok'}
```

Here, create `/predict_status` as an endpoint, also known as the route. Then, add `predict_loan_status()` with a parameter of the type data model, that is, `LoanPred`.

```
1. #Defining the function which will make the prediction using the data
which the user inputs
2. @app.post('/predict_status')
3. def predict_loan_status(loan_details: LoanPred):
4.     data = loan_details.dict()
5.     Gender = data['Gender']
6.     Married = data['Married']
7.     Dependents = data['Dependents']
8.     Education = data['Education']
9.     Self_Employed = data['Self_Employed']
10.    ApplicantIncome = data['ApplicantIncome']
11.    CoapplicantIncome = data['CoapplicantIncome']
12.    LoanAmount = data['LoanAmount']
13.    Loan_Amount_Term = data['Loan_Amount_Term']
14.    Credit_History = data['Credit_History']
15.    Property_Area = data['Property_Area']
16.
17.    # Making predictions
18.    input_data = [Gender, Married, Dependents, Education,
19.                  Self_Employed, ApplicantIncome, CoapplicantIncome,
20.                  LoanAmount, Loan_Amount_Term, Credit_History, Property_Area]
21.    cols = ['Gender', 'Married', 'Dependents',
22.             'Education', 'Self_Employed', 'ApplicantIncome',
23.             'CoapplicantIncome', 'LoanAmount', 'Loan_Amount_Term',
24.             'Credit_History', 'Property_Area']
```

```
25. data_dict = dict(zip(cols,input_data))
26. prediction = make_prediction([data_dict])['prediction'][0]
27.
28. if prediction == 'Y':
29.     pred = 'Approved'
30. else:
31.     pred = 'Rejected'
32.
33. return {'status':pred}
```

The following function will create the UI for user input. Here, create **/predict** as an endpoint, also known as a route, and declare the input data types expected from users.

```
1. @app.post('/predict')
2. def get_loan_details(Gender: str, Married: str, Dependents: str,
3. Education: str, Self_Employed: str, ApplicantIncome: float,
4. CoapplicantIncome: float, LoanAmount: float, Loan_Amount_Term: float,
5. Credit_History: float, Property_Area: str):
6.
7.     input_data = [Gender, Married, Dependents, Education,
8.                 Self_Employed, ApplicantIncome, CoapplicantIncome,
9.                 LoanAmount, Loan_Amount_Term, Credit_History, Property_Area]
10.    cols = ['Gender','Married','Dependents',
11.             'Education','Self_Employed','ApplicantIncome',
12.             'CoapplicantIncome','LoanAmount','Loan_Amount_Term',
13.             'Credit_History','Property_Area']
14.
15.    data_dict = dict(zip(cols,input_data))
16.    prediction = make_prediction([data_dict])['prediction'][0]
17.    if prediction == 'Y':
18.        pred = 'Approved'
```

```
19. else:  
20.     pred = 'Rejected'  
21.  
22. return {'status':pred}  
23.  
24.if __name__ == '__main__':  
25.     uvicorn.run("main:app", host="0.0.0.0", port=port, reload=False)
```

The FastAPI file is completed.

requirements.txt

Now, let's create a *requirements.txt* file, as follows. In this file, model requirements, test requirements, and FastAPI requirements are defined separately for better understanding and ease of management.

```
1. # Model building requirements  
2. joblib==0.16.0  
3. numpy==1.19.0  
4. pandas==1.0.5  
5. scikit-learn==0.23.1  
6. scipy==1.5.1  
7. sklearn==0.0  
8.  
9. # testing requirements  
10. pytest<5.0.0,>=4.6.6  
11. requests  
12.  
13. # packaging  
14. setuptools==40.6.3  
15. wheel==0.32.3  
16.  
17. # FastAPI app requirements  
18. fastapi>=0.68.0,<0.69.0
```

```
19. pydantic>=1.8.0,<2.0.0  
20. uvicorn>=0.15.0,<0.16.0  
21. gunicorn>=20.1.0
```

Dockerfile

Dockerfile contains a list of commands or instructions to be executed while building the Docker image.

```
1. FROM python:3.7-slim-buster  
2.  
3. COPY ./start.sh /start.sh  
4.  
5. RUN chmod +x /start.sh  
6.  
7. ENV PYTHONPATH "${PYTHONPATH}:app/src/"  
8.  
9. COPY . /app  
10.  
11. RUN chmod +x /app  
12.  
13. # expose the port that uvicorn will run the app on  
14. ENV PORT=8000  
15. EXPOSE 8000  
16.  
17. RUN pip install --no-cache-dir --upgrade -r app/requirements.txt  
18.  
19. CMD ["./start.sh"]
```

docker-compose.yml

This builds the web service from the *Dockerfile* in the current directory and mounts the *app* directory on the host to **/app** inside the container.

```
1. version: "3.9" # optional since v1.27.0  
2. services:
```

```
3.   web:  
4.     build: .  
5.   volumes:  
6.     - ./app:/app
```

pytest.ini

Pytest allows you to use a global configuration file, that is, *pytest.ini*, where you can keep settings and additional arguments that you pass whenever you run the command. You can add **-p no:warnings** to **addopts** field, which will suppress the warnings. This section will execute the parameters when pytest runs.

```
1. [pytest]  
2. addopts = -p no:warnings
```

runtime.txt

This file is optional. In this file, you can declare the Python version to be used.

```
1. python-3.7
```

start.sh

This is a shell script that contains a series of commands to be executed by the bash shell. The first line of this file dictates which interpreter should be used to execute this script.

It will install the **prediction_model** package from **src/** placed in **app/** and run the FastAPI app in the next command. The *main.py* is the file name located in the *app* directory.

```
1. #!/bin/bash  
2. pip install app/src/  
3. python app/main.py
```

Make sure you remove the **--reload** option if you are using it. The **--reload** option consumes much more resources, and it is also unstable. You can use it during development but should not use it in production.

test.py

This test file will enable testing the FastAPI app using **testclient**. FastAPI provides the same **starlette.testclient** as **fastapi.testclient**; however, it comes directly from Starlette. With this, you can check the app's routes without running them from the app explicitly.

This file will test the root path of the app and model predictions by passing the data.

```
1. # Importing dependencies
2. from main import app
3. from fastapi.testclient import TestClient
4. import pytest
5. import requests
6. import json
```

Create a **TestClient()** and pass a FastAPI application to it.

```
1. client = TestClient(app)
```

Define the test functions with a name starting with **test_** as per standard naming conventions of pytest:

```
1. def test_read_main():
2.     response = client.get("/")
3.     assert response.status_code == 200
4.     assert response.json() == {'message': 'Loan Prediction App'}
```

Write simple assert statements in the test function to check the output.

```
1. def test_pred():
2.     data = {
3.         "Gender": "Male",
4.         "Married": "Yes",
5.         "Dependents": "0",
6.         "Education": "Graduate",
7.         "Self_Employed": "No",
8.         "ApplicantIncome": 5720,
9.         "CoapplicantIncome": 0,
10.        "LoanAmount": 110,
11.        "Loan_Amount_Term": 360,
12.        "Credit_History": 1,
13.        "Property_Area": "Urban"
14.    }
```

```
15.  
16.     response = client.post("/predict_status", json=data)  
17.     assert response.json()["status"] != ''  
18.     assert response.json() == {"status": "Approved"}
```

tox.ini

This is the tox configuration file. The tox automates and standardizes the testing in Python. It is a virtualenv management and test command-line tool to check whether your package is compatible with different Python versions. The tox will first create a virtual environment based on the configuration provided and then install dependencies. Finally, it will execute the commands provided in the configuration.

tox-gh-actions is a plugin that enables tox to run on GitHub Actions. Hence, you need to install **tox-gh-actions** in the GitHub Actions workflow before running the tox command.

This file will check the package compatibility with Python-3.7. First, it will install the required dependencies and then run the pytest commands.

```
1. [tox]  
2. envlist = py37  
3. skipsdist=True  
4.  
5. [gh-actions]  
6. python =  
7.     3.7: py37  
8.  
9. [testenv]  
10. install_command = pip install {opts} {packages}  
11. deps =
```

```
12.      -r requirements.txt  
13.  
14. setenv =  
15.  PYTHONPATH=src/  
16.  
17. commands=  
18.      pip install requests  
19.      pytest -v test.py  
20.      pytest -v src/tests/
```

Once all files and codebase are ready, run the app on the local machine. If the app is running on a local machine, then deploy the app to the Azure Container Registry.

Build a Docker image on a local machine using the following command:

```
sudo docker build . -t mlappcd.azurecr.io/mlapp-cd:v1
```

Then, run the Docker image using the following command:

```
sudo docker push mlappcd.azurecr.io/mlapp-cd:v1
```

At this point, you should see an ML app up and running on the local machine.

Infrastructure setup

First off, set up infrastructure on Azure cloud. In this section, you will learn to set up an **Azure Container Registry (ACR)** and create a web app container using the Azure App Service.

Azure Container Registry

To begin with, push the Docker image to the Azure Container Registry from the local machine, and from the next time onward, GitHub Actions will push the image. This Docker image will be pulled by the Azure App Service to run a web app container.

Create an Azure Container Registry, as shown in the following figure:

The screenshot shows the 'Create container registry' wizard in the Azure portal. The title bar says 'Create container registry'. The top navigation bar includes 'Home > Container registries > Create container registry'. Below the title, there are tabs: 'Basics' (selected), 'Networking', 'Encryption', 'Tags', and 'Review + create'. A descriptive text block explains the purpose of Azure Container Registry. The 'Project details' section contains fields for 'Subscription' (Free Trial) and 'Resource group' ((New) mlapp-cd). The 'Instance details' section includes 'Registry name' (mlappcd.azurecr.io), 'Location' (East US), and 'Availability zones' (checkbox 'Enabled'). An info message about availability zones is present. The 'SKU' dropdown is set to 'Basic'. At the bottom are buttons for 'Review + create', '< Previous', and 'Next: Networking >'.

Figure 11.2: Azure Container Registry

After providing the values to fields, complete the process by clicking on the **Review + create** button.

After completing the preceding process, you can check the container registry to see the status. The following figure shows the status of the container registry as **OK**:

The screenshot shows the Microsoft Container Registry Overview page for a deployment named "mlappcd". The deployment status is marked as "complete" with a green checkmark. Deployment details include a start time of 5/29/2022, 10:13:41 PM, and a correlation ID of e9a576b4-1d34-4c02-9b80-745a5a61e9f. A "Go to resource" button is visible at the bottom.

Figure 11.3: Deployment status of Azure Container Registry

Next, enable the **Admin user** option in the **Access keys**. The following figure shows that admin user access is enabled under **Settings**:

The screenshot shows the "Access keys" settings page for the "mlappcd" container registry. The "Admin user" toggle switch is set to "Enabled". Two password fields are shown: "password" and "password2", both containing partially obscured text.

Name	Password
password	LaF1LMYt889iyL...
password2	F2IcsGYeYliZCld...

Figure 11.4: Access keys of container registry

After completing the preceding process, go back to your local terminal. Log in to Azure using Azure CLI, as follows:

```
az login
```

After executing the preceding command in the local terminal, a browser window will open where you need to log in. Then, you can close the browser window and continue in the terminal. The following figure shows the output of the preceding command:

```
az login
A web browser has been opened at https://login.microsoftonline.com/organizations/
oauth2/v2.0/authorize. Please continue the login in the web browser. If no web br
owser is available or if the web browser fails to open, use device code flow with
`az login --use-device-code`.
```

Figure 11.5: Azure login

After that, log in to the Azure Container Registry using the following command in the terminal:

```
sudo az acr login --name mlappcd
```

The following figure shows the execution of the preceding command in the terminal:

```
sudo az acr login --name mlappcd
[sudo] password for suhas:
Login Succeeded
```

Figure 11.6: Azure Container Registry login

First, build the Docker image in the local machine:

```
sudo docker build . -t mlappcd.azurecr.io/mlapp-cd:v1
```

In the following figure, you can see the image has been built and tagged successfully:

```
Successfully built aba19c23c8e5
Successfully tagged mlappcd.azurecr.io/mlapp-cd:v1
```

Figure 11.7: Docker image built and tagged in local machine

Now, push that image to Azure container registry:

```
sudo docker push mlappcd.azurecr.io/mlapp-cd:v1
```

As shown in the following figure, the image has been pushed to the Azure Container Registry:

```

sudo docker push mlappcd.azurecr.io/mlapp-cd:v1
The push refers to repository [mlappcd.azurecr.io/mlapp-cd]
3b6d870fb83d: Pushed
8f1f0492f672: Pushed
887b37ce3cfa: Pushed
b53d06fba293: Pushed
5fb14adc05b1: Pushed
fa8a2adf7020: Pushed
1286c8c60b62: Pushed
3c97f5d9ffd6: Pushed
832439eadb07: Pushed
0ad3ddf4a4ce: Pushed
v1: digest: sha256:41546d3a93f60b03d1b3a8e202eb65e412c19426

```

Figure 11.8: Docker image pushed to ACR

As shown in the following figure, verify that the image (which is pushed to the repo) is available in the Azure Container Registry. In this case, you can see that the image **mlapp-cd** is available in the repo.

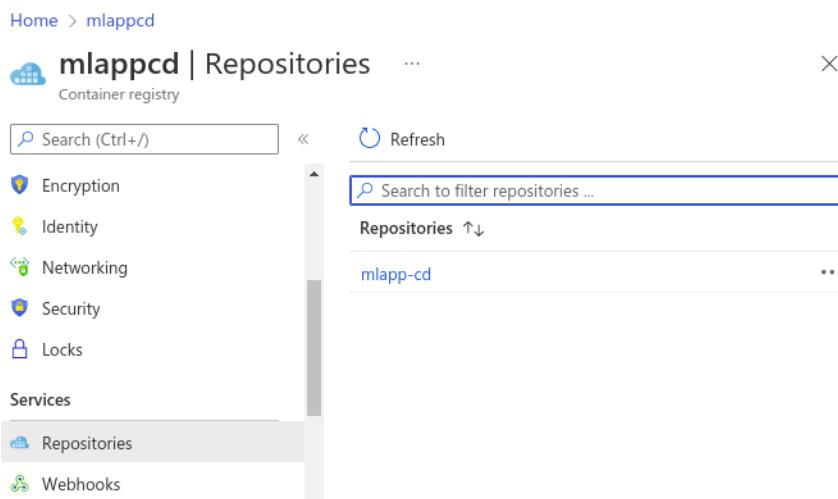


Figure 11.9: Azure Container Registry

Azure App Service

Now, create an Azure App Service resource. Using the Azure App Service, you can create a container-based web app. This will pull the image from the ACR. Azure App Service enables you to build web, mobile, and API apps quickly, which can be scaled as per requirement.

As shown in the following figure, select the subscription and service group under the **Basic** tab. Then, provide the name of the instance as **mlapp-cd**. Choose the **Docker Container** radio button, as the ML app is based on the Docker container.

Home > App Services >

Create Web App

Basics Docker Networking (preview) Monitoring Tags Review + create

App Service Web Apps lets you quickly build, deploy, and scale enterprise-grade web, mobile, and API apps running on any platform. Meet rigorous performance, scalability, security and compliance requirements while using a fully managed platform to perform infrastructure maintenance. [Learn more](#)

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ Free Trial

Resource Group * ⓘ mlapp-cd

Create new

Instance Details

Need a database? [Try the new Web + Database experience.](#)

Name * mlapp-cd .azurewebsites.net

Publish * Code Docker Container Static Web App

Operating System * Linux Windows

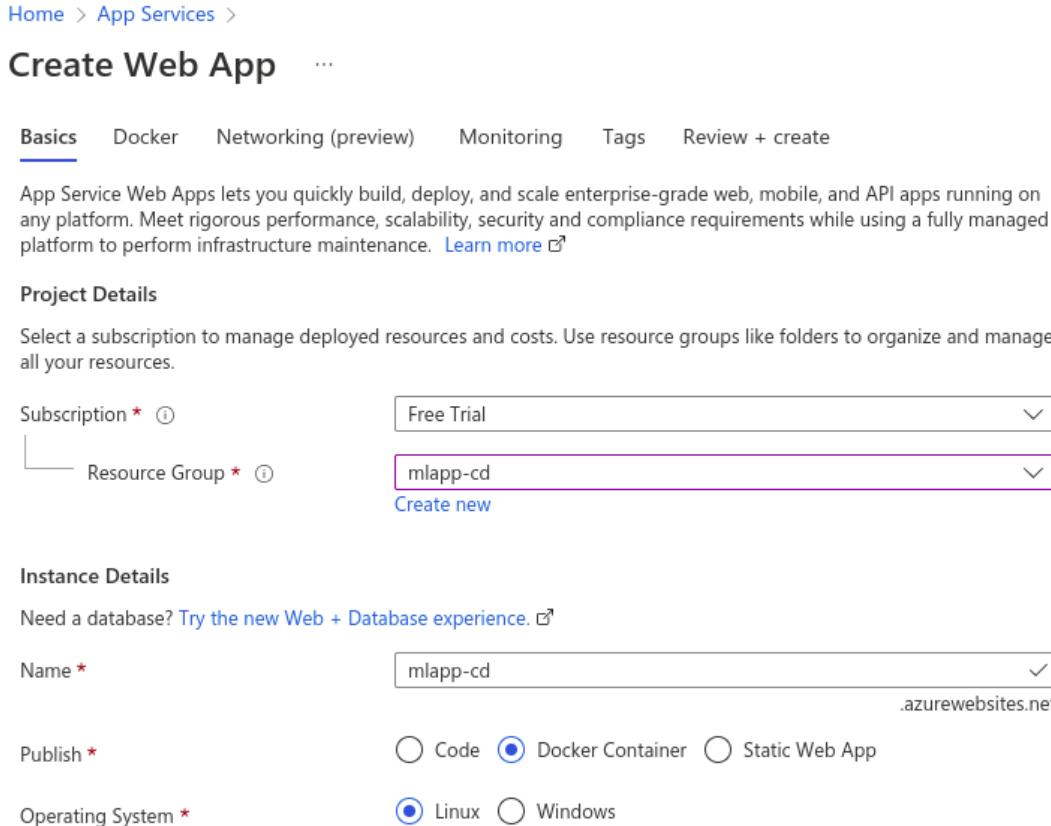


Figure 11.10: Azure Service

Next, choose the operating system **Linux** and select the region where the web app needs to be deployed. Finally, select the **Sku and size** from the plan. You can change the **App Service Plan** as per the application and business requirements.

Home > App Services >

Create Web App

Operating System * Linux Windows

Region * East US

Not finding your App Service Plan? Try a different region or select your App Service Environment.

App Service Plan

App Service plan pricing tier determines the location, features, cost and compute resources associated with your app. [Learn more](#)

Linux Plan (East US) * (New) ASP-mlappcd-856c

Sku and size * Basic B1
100 total ACU, 1.75 GB memory

Zone redundancy

An App Service plan can be deployed as a zone redundant service in the regions that support it. This is a deployment time only decision. You can't make an App Service plan zone redundant after it has been deployed [Learn more](#)

Zone redundancy Enabled: Your App Service plan and the apps in it will be zone

Figure 11.11: Azure Service

In the next tab, that is, **Docker** configuration, make sure the image from the Azure Container Registry is selected. As shown in the following figure, select the **Image Source** as **Azure Container Registry**, and then select the registry, image, and tag from the **Azure container registry options** section.

Home > App Services >

Create Web App

Basics Docker Networking (preview) Monitoring Tags Review + create

Pull container images from Azure Container Registry, Docker Hub or a private Docker repository. App Service will deploy the containerized app with your preferred dependencies to production in seconds.

Options Single Container

Image Source Azure Container Registry

Azure container registry options

Registry * mlappcd

Image * mlapp-cd

Tag * v1

Startup Command

Figure 11.12: Azure Service-Docker configuration

After providing the values to fields, complete the process by clicking on the **Review + create** button. You should see the status of the web app in the newly created resource. The following figure shows the status of the Azure App Service resource:

The screenshot shows the Azure portal interface for a Web App service. The top navigation bar includes 'Home >', the service name 'Microsoft.Web-WebApp-Portal-fd29ca31-a959 | ...', and standard actions like 'Delete', 'Cancel', 'Redeploy', and 'Refresh'. On the left, a sidebar menu lists 'Overview', 'Inputs', 'Outputs', and 'Template'. The main content area displays a success message: 'Your deployment is complete' with a green checkmark icon. Below this, deployment details are listed: Deployment name: Microsoft.Web-WebApp-Portal-fd29ca31-a959, Subscription: Free Trial, Resource group: mlapp-cd, Start time: 5/29/2022, 10:54:02 PM, and Correlation ID: 449bbc7f-efae-4679-a7d1-d2c3ce524e5d. A link to 'Deployment details (Download)' is provided. At the bottom, a table summarizes the resources:

Resource	Type	Status
mlapp-cd	Microsoft.Web/sites	OK
ASP-mlappcd-b304	Microsoft.Web/serverf...	OK

Figure 11.13: Azure Service resource-status

After completing the preceding process, your app should be deployed. You can access the ML web app from anywhere via the internet.

GitHub Actions

GitHub Actions will automate the deployment on Azure. To automate the deployment of the ML app, the workflow (.yml) file is used, as follows:

prod.workflow.yml

GitHub Actions will look for this file for executing the workflow. This workflow is used for app deployment on Azure. It gets triggered by push events, which means any changes pushed to the repo's master branch of the GitHub repository will trigger this workflow to run. Under jobs, declared the OS and Python versions to be used. Then, it will run the pytest using tox. After passing all tests, it will start the deployment on Azure. For this, it will use the secrets defined in the settings of the

GitHub repository. First off, log in to the Azure Container Registry, and then build and push the Docker image to the registry. Finally, this workflow will deploy the app to the Azure web app service and log out from Azure.

```
1. name: Build and deploy to production
2.
3. on:
4.   push:
5.     branches:
6.       - master
7.
8. jobs:
9.   build-and-deploy:
10.    runs-on: ubuntu-18.04
11.    strategy:
12.      matrix:
13.        python-version: ['3.7']
14.
15.    steps:
16.      - name: Checkout GitHub Actions
17.        uses: actions/checkout@master
18.
19.      - name: Set up Python ${{ matrix.python-version }}
20.        uses: actions/setup-python@v2
21.        with:
22.          python-version: ${{ matrix.python-version }}
23.
24.      - name: Install dependencies
25.        run: |
26.          python -m pip install --upgrade pip
27.          python -m pip install tox tox-gh-actions
28.
29.      - name: Test with tox
```

```
30.      run: |
31.      tox
32.
33.      - name: Login via Azure CLI
34.        uses: azure/login@v1
35.        with:
36.          creds: ${{ secrets.AZURE_CREDENTIALS }}
37.
38.      - name: Login to Azure Container Registry
39.        uses: azure/docker-login@v1
40.        with:
41.          login-server: mlappcd.azurecr.io
42.          username: ${{ secrets.REGISTRY_USERNAME }}
43.          password: ${{ secrets.REGISTRY_PASSWORD }}
44.
45.      - name: Build and push container image to the registry
46.        run: |
47.          docker build . -t mlappcd.azurecr.io/mlapp-cd:${{ github.sha }}
48.          docker push mlappcd.azurecr.io/mlapp-cd:${{ github.sha }}
49.
50.      - name: Deploy to App Service
51.        uses: azure/webapps-deploy@v2
52.        with:
53.          app-name: 'mlapp-cd'
54.          images: 'mlappcd.azurecr.io/mlapp-cd:${{ github.sha }}'
55.
56.      - name: Azure logout
57.        run: |
58.          az logout
```

Next, create the repository on GitHub for this (if not created already) and push the codebase into that repo.

Service principal

You need to provide a service principal in the GitHub Actions workflow for Azure authentication and app deployment. To get the credentials, execute the following command with your subscription id in the terminal.

```
az ad sp create-for-rbac --name "github-actions" --role contributor  
--scopes /subscriptions/<Subscription ID>/resourceGroups/mlapp-cd --sdk-auth
```

With the preceding command, you will create an Azure **Role Based Access Control (RBAC)** named **github-actions** with a contributor role and scope.

Note: You can get the subscription ID from the subscription you have used while creating a resource group.

Azure RBAC is an authorization system built on Azure Resource Manager that provides access management of Azure resources.

You should get the following output:

```
{  
  "clientId": "████████████████████████████████████████",  
  "clientSecret": "████████████████████████████████████████████████████████",  
  "subscriptionId": "████████████████████████████████████████████████████████",  
  "tenantId": "████████████████████████████████████████████████████████",  
  "activeDirectoryEndpointUrl": "https://login.microsoftonline.com",  
  "resourceManagerEndpointUrl": "https://management.azure.com/",  
  "activeDirectoryGraphResourceId": "https://graph.windows.net/",  
  "sqlManagementEndpointUrl": "https://management.core.windows.  
net:8443/",  
  "galleryEndpointUrl": "https://gallery.azure.com/",  
  "managementEndpointUrl": "https://management.core.windows.net/"  
}
```

In the preceding response, **clientId**, **clientSecret**, **subscription**, and **tenantId** are unique for the individual account. Copy and save the response of the preceding command for future use.

Now, go to GitHub repo settings and create three GitHub secrets:

- **AZURE CREDENTIALS:** Entire JSON response from preceding
- **REGISTRY_USERNAME:** `clientId` value from JSON response
- **REGISTRY_PASSWORD:** `clientSecret` value from JSON response

Configure Azure App Service to use GitHub Actions for CD

You need to configure the Azure App Service so that it can be automated using GitHub Actions. Head over to the **Azure App Service | Deployment Center** and link the GitHub repo **master** branch, as shown in the following figure.

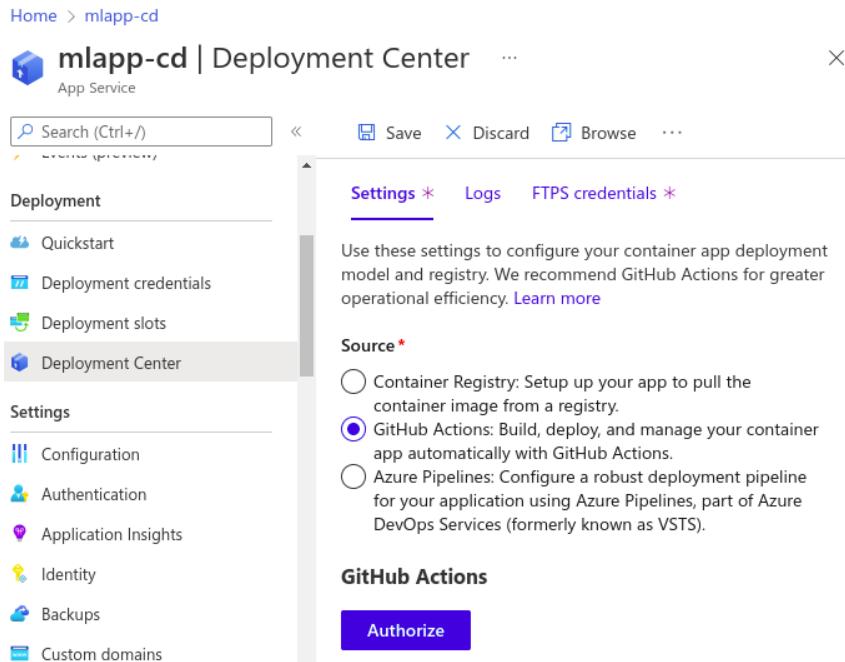


Figure 11.14: Azure Service-GitHub configuration

You need to authorize GitHub Actions to automate CD to provide the required details. You can add deployment slots, such as staging or pre-production. It is useful when you don't want to expose the updated Azure App Service directly to production. You need to do the same configuration for the deployment center as per the previous step.

After completing the configuration in the app service, place the `prod.workflow.yml` workflow file in the linked GitHub repository. The path would be `.github/workflows/prod.workflow.yml`. When you push any changes to the GitHub repo, it will trigger the GitHub Actions workflow. After completing all the stages, the ML app will be deployed on Azure.

Go to the **Actions** tab in the GitHub repository, and you will see that GitHub has already started running the `prod.workflow.yml` workflow for app deployment on Azure.

Push the changes with a **Deploy** text as a comment, as you can see in the following figure.

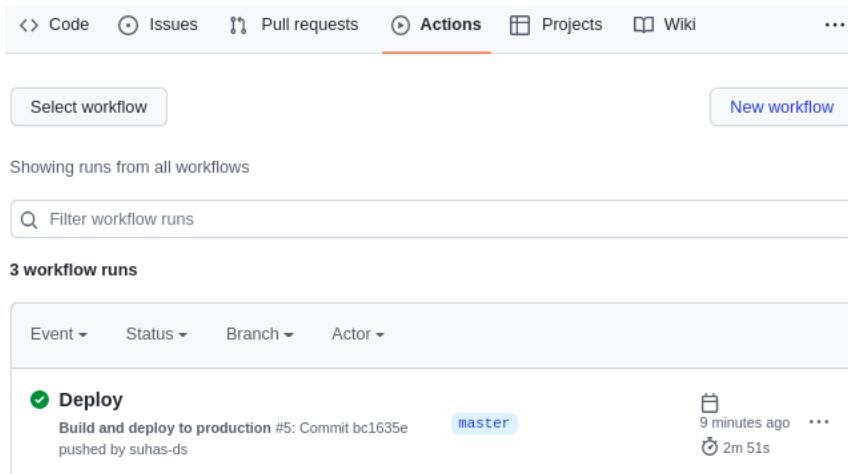


Figure 11.15: GitHub Actions-workflow runs

Click on **Deploy**, and it will take you to the summary of this workflow. You will see the status of the jobs you have defined in the workflow. Other execution details are also mentioned, such as the time taken to complete this workflow, the username that pushed the updates to the **master** branch, and the name of the workflow file.

The screenshot shows the GitHub Actions workflow interface. At the top, there's a green checkmark icon followed by the word "Deploy". Below it, a button says "Build and deploy to production #5". To the right are two buttons: "Re-run all jobs" and an ellipsis (...). On the left, there's a sidebar with "Summary" and "Jobs" sections. Under "Jobs", "build-and-deploy (3.7)" is selected, indicated by a green checkmark. The main area displays the workflow steps:

Step	Description	Duration
> Set up job		6s
> Checkout GitHub Actions		1s
> Set up Python 3.7		0s
> Install dependencies		6s
> Test with tox		32s
> Login via Azure CLI		14s
> Login to Azure Container Registry		0s
> Build and push container image to registry		1m 15s
> Deploy to App Service		22s
> Azure logout		0s
> Post Set up Python 3.7		0s
> Post Checkout GitHub Actions		0s
> Complete job		0s

Figure 11.16: GitHub Actions-workflow steps

The following figure shows the output of app deployment to the Azure step from GitHub Actions. You can see that the app is successfully deployed and running on Azure. An app service application URL is printed at the end.

The screenshot shows the deployment status for the "Deploy to App Service" step. It includes the command run, deployment logs, and deployment URL.

```

v Deploy to App Service 22s
1 ► Run azure/webapps-deploy@v2
13 Updating App Service Configuration settings. Data:
    ****"linuxFxVersion":"DOCKER|mlappcd.azurecr.io/mlapp-cd:bc1635e1f1eb4414b3c18957fa424068ddacb624"****
14 Updated App Service Configuration settings.
15 Restarting app service: mlapp-cd
16 Deployment passed
17 Restarted app service: mlapp-cd
18 Successfully updated deployment History at https://mlapp-
    cd.scm.azurewebsites.net/api/deployments/bc1635e1f1eb4414b3c18957fa424068ddacb6241653943290526
19 App Service Application URL: http://mlapp-cd.azurewebsites.net

```

Figure 11.17: GitHub Actions-deployment status

Head over to the Azure App Service; you should see container logs in the Azure App Service, as shown in the following figure. This helps you debug the issue while deploying the app.

The screenshot shows the Azure App Service Deployment Center for the 'mlapp-cd' application. The left sidebar includes links for Home, Security, Events (preview), Deployment (Quickstart, Deployment slots, Deployment Center), Settings (Configuration, Authentication, Application Insights, Identity, Backups, Custom domains), and a search bar. The main area has tabs for Settings, Container Logs (which is selected), Build Logs, and FTPS credentials. A message states: "These are the logs from Docker Engine emitted during the provisioning phase of your container image." Below this, a log viewer displays several lines of text, including environment variables and deployment logs:

```

WEBSITES_ENABLE_APP_SERVICE_STORAGE=false -e
WEBSITE_SITE_NAME=mlapp-cd -e WEBSITE_AUTH_ENABLED=False -e
PORT=8000 -e WEBSITE_ROLE_INSTANCE_ID=0 -e
WEBSITE_HOSTNAME=mlapp-cd.azurewebsites.net -e
WEBSITE_INSTANCE_ID=4956fd0f71e6a2c01e79df39cf93832dc1b8832
2e9676900fc844a8843ac6026 mlappcd.azurecr.io/mlapp-
cd:e23dc8ace9807c51a881ba5a45108753b04b7195

2022-05-29T19:21:22.117Z INFO - Logging is not enabled for
this container.
Please use https://aka.ms/linux-diagnostics to enable
logging to see container logs here.
2022-05-29T19:21:24.477Z INFO - Initiating warmup request
to container mlapp-cd_1_e2ccd285 for site mlapp-cd
2022-05-29T19:21:33.672Z INFO - Container mlapp-
cd_1_e2ccd285 for site mlapp-cd initialized successfully
and is ready to serve requests.

```

Figure 11.18: App service-container logs

If everything goes well, the app should work on Azure post deployment.

The following figure shows the app running on the Azure platform on successful completion of the *prod.workflow.yml* workflow.

The screenshot shows the Azure API Management interface for the 'Loan Prediction Model API'. At the top, it says '0.1 OAS3'. Below that is a link to '/openapi.json'. The main content area is titled 'default' and contains four API endpoints:

- GET / Index**
- GET /health Healthcheck**
- POST /predict_status Predict Loan Status**
- POST /predict Get Loan Details**

Figure 11.19: App service-ML web app running

Azure provides a monitoring service to track the usage of the Azure app. You can select metrics from the dropdown.

The following figure shows the monitoring service of the Azure app.

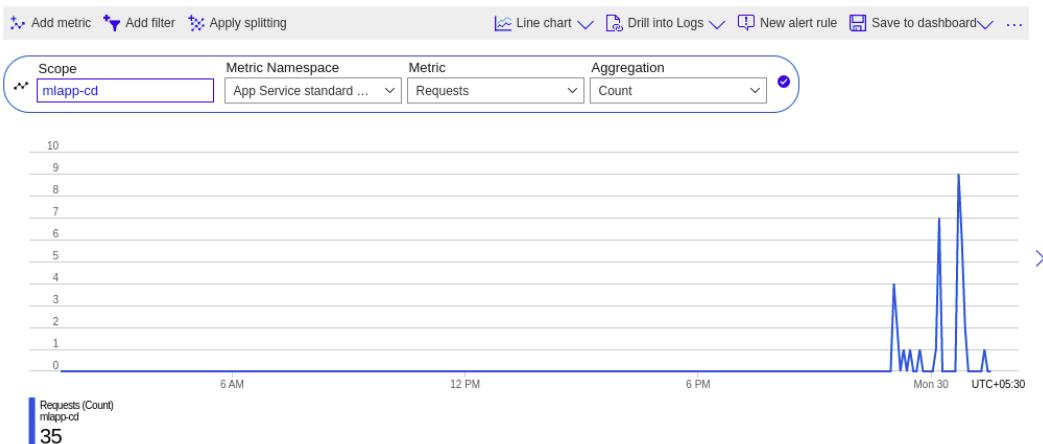


Figure 11.20: App service-monitoring service

Thus, you have learned to create a simple CI/CD pipeline using GitHub to deploy the ML app on the Azure platform. However, you can modify it as per business and application requirements.

Deployment using Azure DevOps and Azure ML

In this part, you will use Azure DevOps and AML to deploy an ML app on the Azure cloud. Unlike the previous part, the CI pipeline and CD pipeline are in the Azure cloud. This approach requires parallelism to be enabled. If you are using free credits, then by default, parallelism is not enabled; however, you can activate it by sending the request via email. You can use Azure Git repo or other sources, such as GitHub. This approach gives more flexibility, such as integration with other platforms, manual approval before deploying to the production, auto redeploy triggers, and such.

The following figure shows the automated ML workflow:

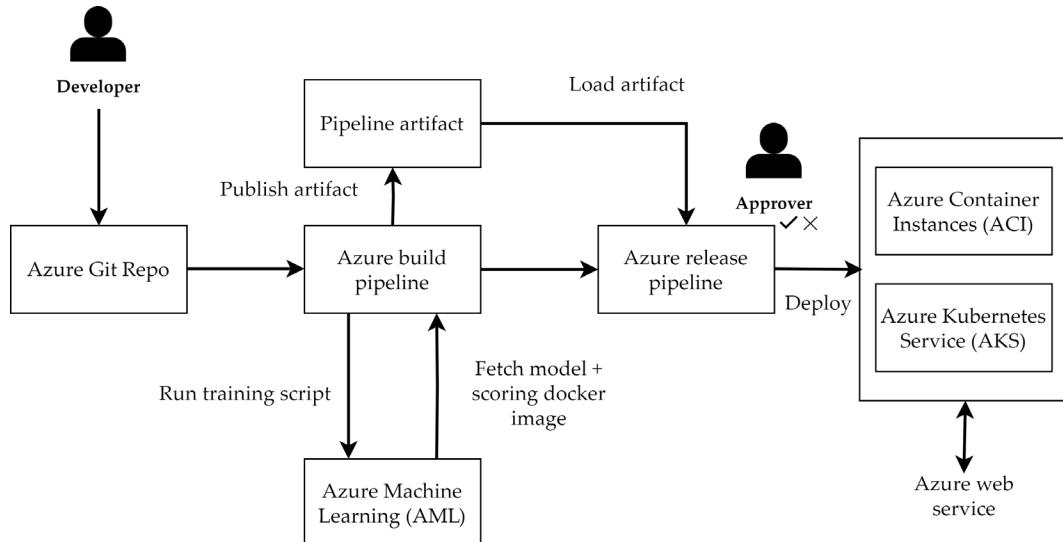


Figure 11.21: Azure DevOps pipeline for ML

Azure Machine Learning (AML) service

Azure Machine Learning (AML) service enable the creation of a reproducible CI/CD pipeline for ML. It is an Azure cloud-hosted environment that allows developers and organizations to deploy ML models quickly in the production environment with minimal code. In Azure Machine Learning Studio, you can run the notebooks on the cloud, use Automated ML for automated training and tuning the ML model using a metric, or use a designer to deploy a model from data preparation using a drag-and-drop interface.

The following are the salient features of Azure Machine Learning Studio:

- Storage provision to store your data and artifacts
- MLFlow to track your experiments and log the run details like timestamp, model metrics, and so on
- Model registry to store trained ML models for reusability
- Key vault to store credentials and variables
- Supports open-source libraries and frameworks
- Compute instance, which enables building and training in a secure VM
- Pipelines and CI/CD for faster training and deployment

- Endpoints for ML model
- Data drift functionality to deliver consistent accuracy and predictions
- Monitoring service to track the model, logs, and resources

Workspace

A machine learning workspace is the main resource of AML. It contains experiments, models, endpoints, datasets, and such. It also includes other Azure resources, such as Azure Container Registry (ACR), which registers Docker containers at the time of Docker image deployment, storage, application insights, and key vault.

Experiments

An experiment consists of several runs initiated from the script. Run details and metadata are stored under the experiment. When you run the script, the experiment name needs to be provided to store the run information. However, it will create a new experiment if the name provided in the script does not exist in the given workspace.

Runs

A run can be defined as a single iteration of the training script. Multiple runs are recorded under an experiment. A run logs model metrics and metadata, such as timestamps.

The best thing about AML is that you do not need to create a separate web service API using frameworks like Flask, Django, and FastAPI. AML creates endpoints for trained ML models and tracks the web service.

In this section, you will be using the Azure Machine Learning template and modifying it as per your requirements. You can build your codebase from the scratch; however, this template saves time as it has reusable and required steps already defined. The Azure DevOps demo generator can be accessed at the following link:

<https://azureddevopsdemogenerator.azurewebsites.net/environment/createproject>

Go to DevOps Labs, select the **Azure Machine Learning template** and add it to your project by providing the project name and organization in the next step.

Alternatively, you can directly go to the following URL:

<https://azureddevopsdemogenerator.azurewebsites.net/?name=machinelearning>

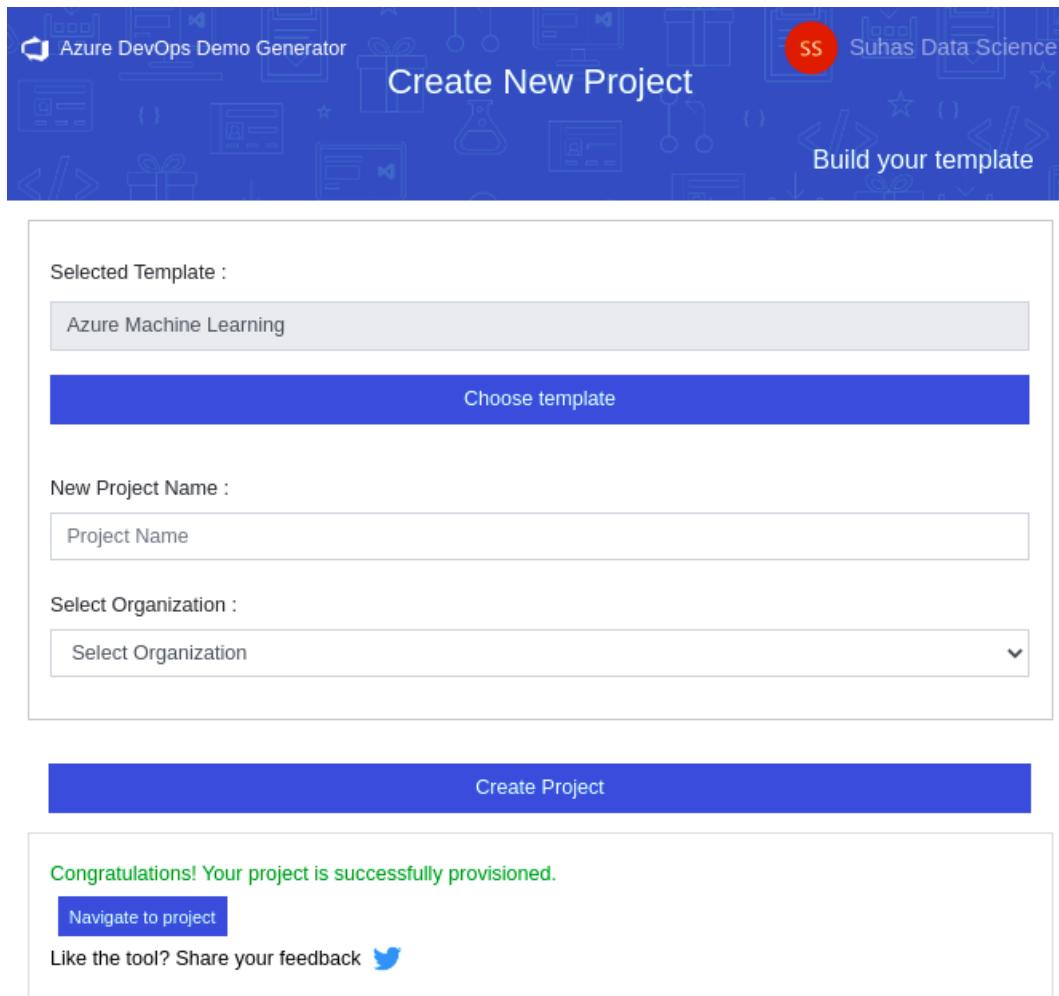


Figure 11.22: Azure DevOps Demo Generator

As you can see, a codebase is imported along with the template. This template contains code and pipeline definitions for a machine learning project to demonstrate how to automate the end-to-end ML / AI project. In this, most steps can be reused with minor modifications wherever required. A codebase is residing in the Azure Git repository; however, you can import a codebase from the GitHub repository. Note that this template is built for a regression algorithm and uses a diabetes dataset.

However, you need to update the files for the classification algorithm, and the loan dataset will be used for it.

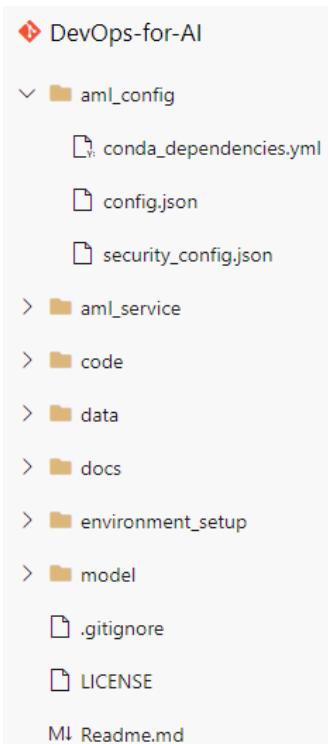


Figure 11.23: Azure code repository

Next, head over to the pipeline section from the left menu. You can see that the steps have already been defined in the template. Don't worry if you want to create all the steps from the beginning. First off, create a new project, and inside that project, go to the pipeline section from the left menu. Next, select a codebase from the available source options, such as GitHub, Azure Repos, and Git. Then, create a new pipeline from a YAML file, or use the classic editor to create a pipeline without YAML.

Next, go to the newly created project settings and create a new service connection, if it is not created. Choose **Service Principal (automatic)**, and then choose the subscription and resource group from the dropdown and provide the service connection name in the next window. Make sure the **Grant access permission to all pipeline** checkbox is selected.

The following figures show the steps discussed:

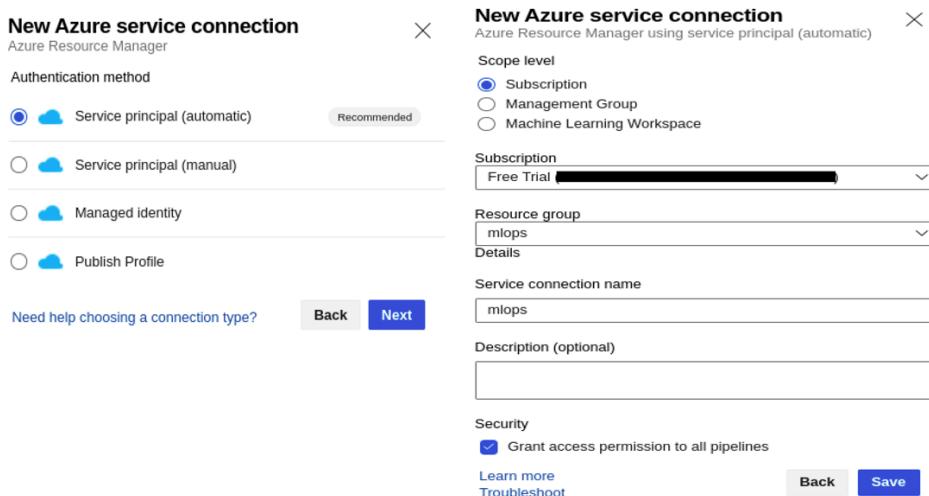


Figure 11.24: New Azure service connection

Now, you need to update the files that will be used in the Azure CI/CD pipeline. You can refer to the following code to modify respective scripts. Only code modifications are discussed here, as you can keep the rest of the code as it is in the original script.

code/training/train.py

This file will build a classification model using logistic regression. After model training, it will log Cross-Validation (CV) scores and accuracy for later use. Then, it will save the model as a pickle object. This script will be called by the *aml_service/10-TrainOnLocal.py* script.

The following code update is incorporated into the existing script:

```

1. import pickle
2. from azureml.core import Workspace
3. from azureml.core.run import Run
4. import os
5. from sklearn.linear_model import LogisticRegression
6. from sklearn.model_selection import train_test_split, cross_val_score
7. from sklearn.metrics import accuracy_score
8. from sklearn.preprocessing import LabelEncoder
9. from sklearn.externals import joblib

```

```
10. import pandas as pd
11. import numpy as np
12. import json
13. import subprocess
14. from typing import Tuple, List
15.
16. run = Run.get_submitted_run()
17.
18. url = "https://gist.githubusercontent.com/
suhas-ds/a318d2b1dda8d8cbf2d6990a8f0b7e8a/
raw/9b548fab0952dd12b8fdf057188038c8950428f1/loan_dataset.csv"
19. df = pd.read_csv(url)
20.
21. # fill the missing values for numerical cols with the median
22. num_col = ['LoanAmount', 'Loan_Amount_Term', 'Credit_History']
23. for col in num_col:
24.     df[col].fillna(df[col].median(), inplace=True)
25.
26. # fill the missing values for categorical cols with mode
27. cat_col = ['Gender', 'Married', 'Dependents', 'Self_Employed']
28. for col in cat_col:
29.     df[col].fillna(df[col].mode()[0], inplace=True)
30.
31. # Total Income = Applicant Income + Coapplicant Income
32. df['Total_Income'] = df['ApplicantIncome'] +
df['CoapplicantIncome']
33.
34. # drop unnecessary columns
35. cols = ['ApplicantIncome', 'CoapplicantIncome', "LoanAmount",
"Loan_Amount_Term", 'Loan_ID']
36. df = df.drop(columns=cols, axis=1)
37.
38. # Label encoding
39. cols = ['Gender', "Married", "Education", 'Self_Employed', "Property_"
Area", "Loan_Status", "Dependents"]
```

```
40. le = LabelEncoder()
41. for col in cols:
42.     df[col] = le.fit_transform(df[col])
43.
44. # Train test data preparation
45. target = 'Loan_Status'
46.
47. X = df.drop(columns=['Loan_Status'], axis=1)
48. y = df['Loan_Status']
49. X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size=0.30, random_state=42)
50.
51. print("Running train.py")
52.
53. model = LogisticRegression()
54. model.fit(X_train, y_train)
55. print("Accuracy is", model.score(X_test, y_test)*100)
56. # cross validation - it is used for better validation of the
model
57. # eg: cv-5, train-4, test-1
58. cv_score = cross_val_score(model, X, y, cv=5)
59. print("Cross-validation is", np.mean(cv_score)*100)
60. run.log("CV_score", np.mean(cv_score)*100)
61. y_pred = model.predict(X_test)
62. print("Accuracy = ", accuracy_score(y_test, y_pred))
63. run.log("accuracy", accuracy_score(y_test, y_pred))
64.
65. # Save the model as part of the run history
66. model_name = "sklearn_classification_model.pkl"
67. # model_name = "."
68.
69. with open(model_name, "wb") as file:
70.     joblib.dump(value=model, filename=model_name)
```

code/score/score.py

This script will be used for testing the web service. Here, update the saved model path, and then update the input test data that is to be predicted using web service.

The following code update is incorporated into the existing script:

```
1. model_path = Model.get_model_path(model_name="sklearn_
classification_model.pkl")
```

Update test data as per the ML model requirement:

```
1. test_row = '{"data": [[1,1,0,1,0,1,2,3849],[1,1,3,0,0,0,1,5540]]}'
```

aml_config/config.json

Here, you need to update the details from the Azure portal. Other scripts will use the variables declared in this script.

The following code update is incorporated into the existing script:

```
1. {
2.     "subscription_id": "████████████████",
3.     "resource_group": "mllops",
4.     "workspace_name": "mllops_ws",
5.     "location": "centralus"
6. }
```

environment_setup/install_requirements.sh

This bash script is responsible for installing Python packages using **pip**.

The following code update is incorporated into the existing script:

```
1. python --version
2. pip install azure-cli==2.37.0 ==2.0.69
3. pip install --upgrade azureml-sdk[cli]
4. pip install -r requirements.txt
```

aml_service/00-WorkSpace.py

This file will get the details of the existing workspace; however, if not found, it will create a new one.

aml_service/10-TrainOnLocal.py

This script triggers the *code/training/train.py* script to run on the local compute (host agent in case of build pipeline). If you are training on a remote VM, you do not need this script in the build pipeline. All the training scripts generate an output file *aml_*

config/run_id.json, which records the **run_id** and **run history name** of the training run. *run_id.json* is used by *20-RegisterModel.py* to get the trained model. Update the experiment name to **mlops-classification**.

The following code update is incorporated into the existing script:

```
1. experiment_name = "mlops-classification"
```

aml_service/15-EvaluateModel.py

At this step, you will get the run history for both the production model and the newly trained model to compare the accuracy. If the new model's accuracy is better than that of the existing model, the new model will be promoted to production; otherwise, the existing model will remain as it is. However, for the first time, there won't be any model to compare, so it will go to the **except** block and promote the new model. You can change the metrics that need to be compared and conditions that decide whether the new model is to be promoted to production. At the end of the script, it will capture the experiment name and run id of the new model in the *aml_config/run_id.json* file if the new model is promoted to production.

The following code update is incorporated into the existing script:

```
1. production_model_acc = production_model_run.get_metrics().  
get("accuracy")  
2.     new_model_acc = new_model_run.get_metrics().get("accuracy")  
3.     print(  
4.         "Current Production model accuracy: {}, New trained model  
accuracy: {}".format(  
5.             production_model_acc, new_model_acc  
6.         )  
7.     )  
8.  
9.     promote_new_model = False  
10.    if new_model_acc > production_model_acc:  
11.        promote_new_model = True  
12.        print("New trained model performs better, thus it will be  
registered")
```

aml_service/20-RegisterModel.py

This script is responsible for registering new models. This will look for *aml_config/run_id.json*; if **run_id** is not found, it will print the message **No new model to**

register as production model perform better written inside the **except** block and exit from the code.

If the run id is found in *aml_config/run_id.json*, it will register the model. Finally, it will write the registered model details to */aml_config/model.json*.

The following code update is incorporated into the existing script:

```
1. # Download Model to Project root directory
2. model_name = "sklearn_classification_model.pkl"
3. run.download_file(
4.     name=".outputs/" + model_name, output_file_path=".model/" +
model_name
5. )
6. print("Downloaded model {} to Project root directory".
format(model_name))
7. os.chdir("./model")
8. model = Model.register(
9.     model_path=model_name, # This points to a local file
10.    model_name=model_name, # This is the name the model is
registered as
11.    tags={"area": "loan", "type": "classification", "run_id": run_
id},
12.    description="Classification model for loan dataset",
13.    workspace=ws,
14. )
```

aml_service/30-CreateScoringImage.py

In the beginning, it will look for the *aml_config/model.json* file generated from the previous step. If it is not found, the script will be terminated with the message **No new model to register thus no need to create new scoring image**.

The following code update is incorporated into the existing script:

```
1. image_name = "loan-model-score"
2.
3. image_config = ContainerImage.image_configuration(
4.     execution_script="score.py",
5.     runtime="python-slim",
```

```
6.     conda_file="conda_dependencies.yml",
7.     description="Image with logistic regression model",
8.     tags={"area": "loan", "type": "classification"},
9. )
```

This will create the image for the model, and at the end, capture the image details, such as **name**, **version**, **creation_state**, **image_location**, and **image_build_log_uri** in the */aml_config/image.json*.

aml_service/50-deployOnAci.py

Azure Container Instances (ACI) is an Azure service that enables users to run a container on the Azure cloud without demanding the use of a **Virtual Machine (VM)**.

In the beginning, it will look for the *aml_config/image.json* file generated from the previous step. If it is not found, the script will be terminated with the message **No new model, thus no deployment on ACI**.

The following code update is incorporated into the existing script:

```
1. aciconfig = AciWebservice.deploy_configuration(
2.     cpu_cores=1,
3.     memory_gb=1,
4.     tags={"area": "loan", "type": "classification"},
5.     description="A loan prediction app",
6. )
```

This script will create a web service and capture the ACI details in */aml_config/aci_webservice.json* at the end of the script.

aml_service/60-AciWebserviceTest.py

In the beginning, it will look for the *aml_config/aci_webservice.json* file generated from the previous step. If it is not found, the script will be terminated with the message **No new model, thus no deployment on ACI**.

The following code update is incorporated into the existing script:

```
1. # Input for Model with all features
2. input_j = [[1,1,0,1,0,1,2,3849],[1,1,3,0,0,0,1,5540]]
```

Configure CI pipeline

At this stage, you will configure the CI pipeline. This pipeline will execute the following tasks:

- Set up Python version 3.6.
- Install required dependencies.
- Create or get Azure Machine Learning (AML) workspace.
- Build and train the ML model.
- Evaluate newly trained model performance against the existing model to decide whether the model is to be promoted to production.
- Register model in Azure Machine Learning (AML) service.
- Create a scoring Docker image with the required dependencies.
- Publish the artifacts (all files) to the repo.

The following figure shows the steps of the CI pipeline:

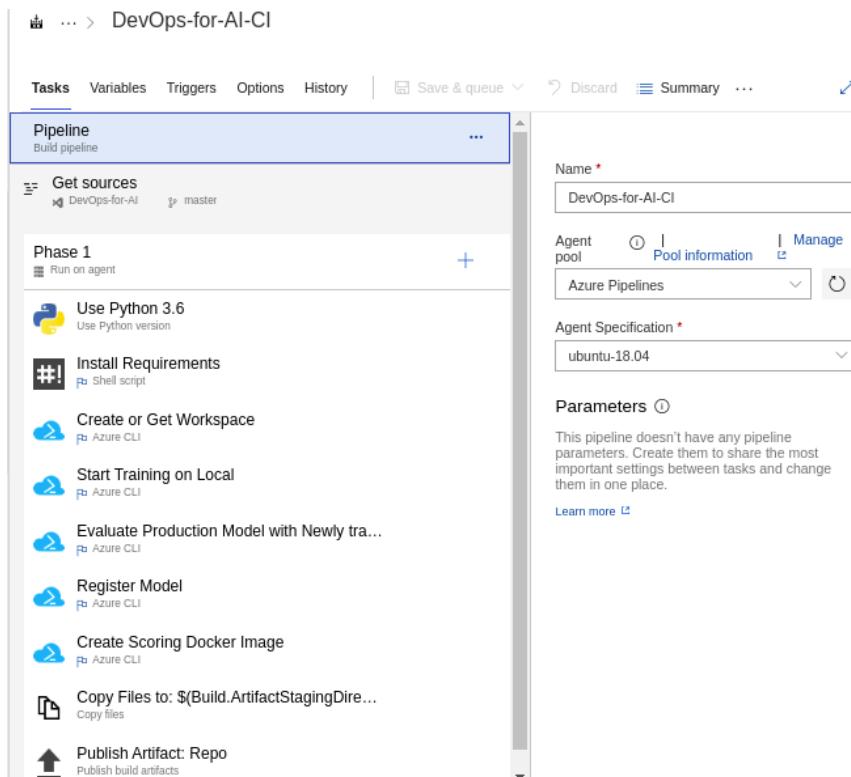


Figure 11.25: CI pipeline for ML

Note: You need to select Azure service connection from the Azure subscription dropdown wherever necessary.

Next, in the Triggers tab, select **Enable continuous integration** checkbox. This enables the CI pipeline to run as soon as any changes are pushed to the code repository.

You can trigger the CI pipeline by pushing any changes to the repository. The Azure CI pipeline detects the changes in the repo and starts building the pipeline. To see the progress of the CI pipeline, switch to the **Summary** tab.

The following figure displays the **Summary** tab. You can see the commit message on the top, followed by the summary and the jobs. The job status and progress of steps can be seen by clicking on **Phase 1**.

The screenshot shows the Azure DevOps CI pipeline summary for pull request #50, titled "Updated train.py". The pipeline was triggered by Suhas Data Science. The summary card includes details like the repository version (DevOps-for-AI master commit 6c318972), start time (Today at 18:59), duration (2m 45s), and work items (0 consumed). A "View change" button is available. Below the summary card is a "Jobs" section showing a single job named "Phase 1" which is currently "Running" for 2m 41s.

Figure 11.26: CI pipeline for ML-Jobs summary

After clicking on **Phase 1** under the **Jobs** section, you should see the progress of the CI pipeline. The following figure shows the terminal output of each CI pipeline step:

The screenshot displays a CI pipeline interface. On the left, a list of jobs is shown under 'Jobs in run #50'. One job, 'Phase 1', is expanded, revealing its sub-steps and execution times. On the right, a detailed log for the 'Finalize Job' step is displayed, showing the following sequence of events:

```

1 Starting: Finalize Job
2 Cleaning up task key
3 Start cleaning up orphan processes.
4 Finishing: Finalize Job

```

Figure 11.27: CI pipeline for ML-logs

Configure CD pipeline

Now that you have completed the CI pipeline configuration, it's time to configure the CD pipeline. The CD pipeline will deploy the image generated through the CI pipeline to the **Azure Container Instance (ACI)** and **Azure Kubernetes Service (AKS)**. In the current case, the scope of the project is limited to ACI. However, you can deploy it on AKS to handle large amounts of traffic and make it scalable.

This pipeline executes the following tasks:

- Set up Python version 3.6.
- Install the required dependencies.
- Deploy a web service on the Azure container instance (ACI).
- Test the ACI web service by passing data that needs to be tested.

- Deploy a web service on Azure container instance (AKS).
- Test the AKS service by passing the data to be tested.

The following figure shows the CD pipeline:

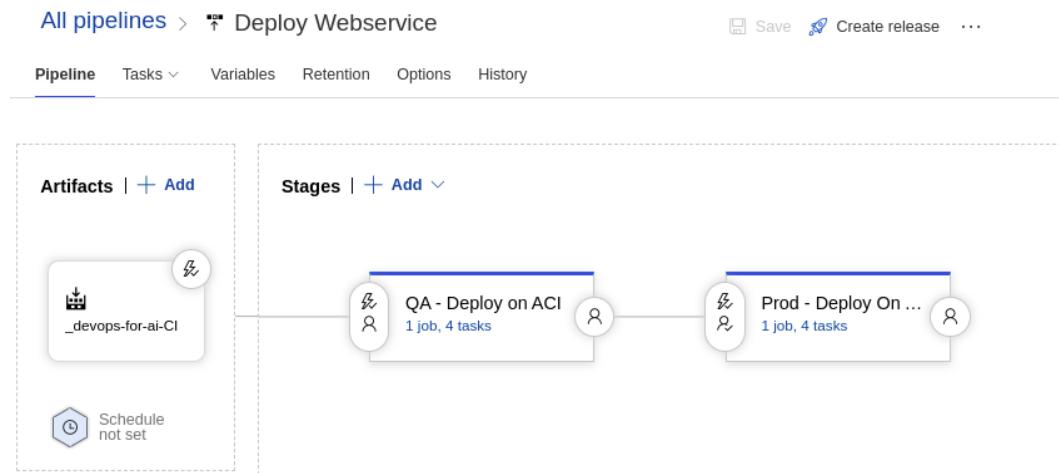


Figure 11.28: CD pipeline for ML

Now, navigate to **Pipeline | Releases**, select **Deploy Web service** and click on **Edit pipeline**.

The following figure shows the **QA – Deploy on ACI** stage of the CD pipeline when switched to the **Tasks** tab.

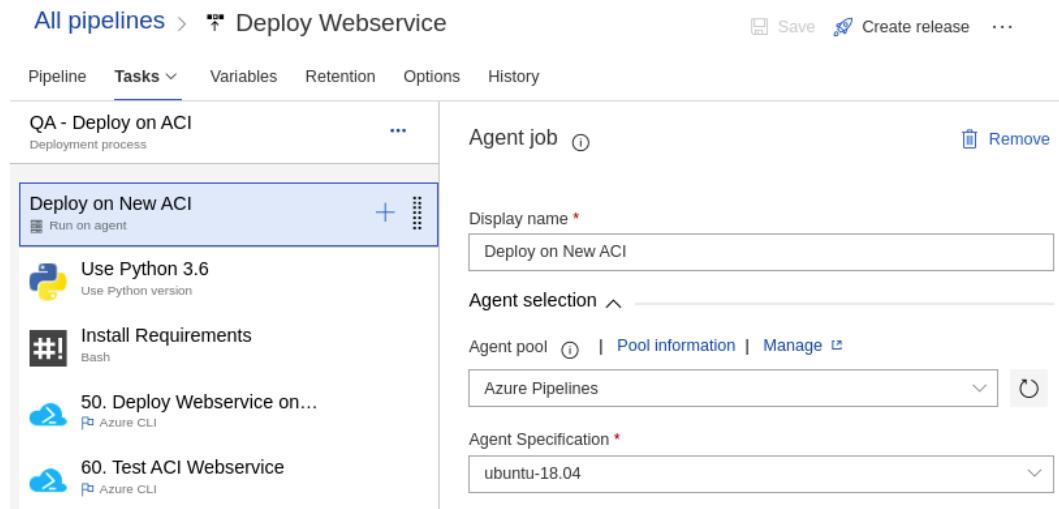


Figure 11.29: CD pipeline for ML-QA stage

Note: You need to select Azure service connection from the Azure subscription dropdown wherever necessary.

In the following figure, you can see the entire file path provided by the artifacts that are retrieved from the CI pipeline:

The screenshot shows the Azure DevOps Pipeline interface. At the top, there's a header with 'All pipelines > Deploy Webservice'. Below it is a navigation bar with 'Pipeline', 'Tasks', 'Variables', 'Retention', 'Options', and 'History'. The 'Tasks' tab is selected. The main area displays a pipeline named 'QA - Deploy on ACI' which is a 'Deployment process'. It contains several tasks: 'Deploy on New ACI' (Run on agent), 'Use Python 3.6' (Use Python version), 'Install Requirements' (Bash), '50. Deploy Webservice on ACI' (Azure CLI), and '60. Test ACI Webservice' (Azure CLI). The 'Install Requirements' task is currently selected, indicated by a blue border around its row. To the right of the pipeline list, there's a detailed view of this task. It shows the 'Task version' set to '3.*'. The 'Display name' is 'Install Requirements'. Under 'Type', 'File Path' is selected (indicated by a checked radio button). The 'Script Path' field contains the value: \$(System.DefaultWorkingDirectory)/_devops-for-ai-CI/devops-for-ai/environment_setup/install_requirements.sh. There's also a '...' button next to the script path field.

Figure 11.30: CD pipeline for ML-QA stage requirement step

When you update the CD pipeline, you can run the CD pipeline manually by hitting the **Create Release** button or setting the auto trigger, which means that the CD pipeline starts executing upon completion of the CI pipeline. You need to make sure that continuous deployment triggers are enabled in the **Artifacts** stage and the **QA -Deploy on ACI** stage of the CD pipeline.

In the following figure, you can see the CD pipeline triggers upon completion of the CI pipeline, which caused the **QA -Deploy on ACI** stage to start executing:

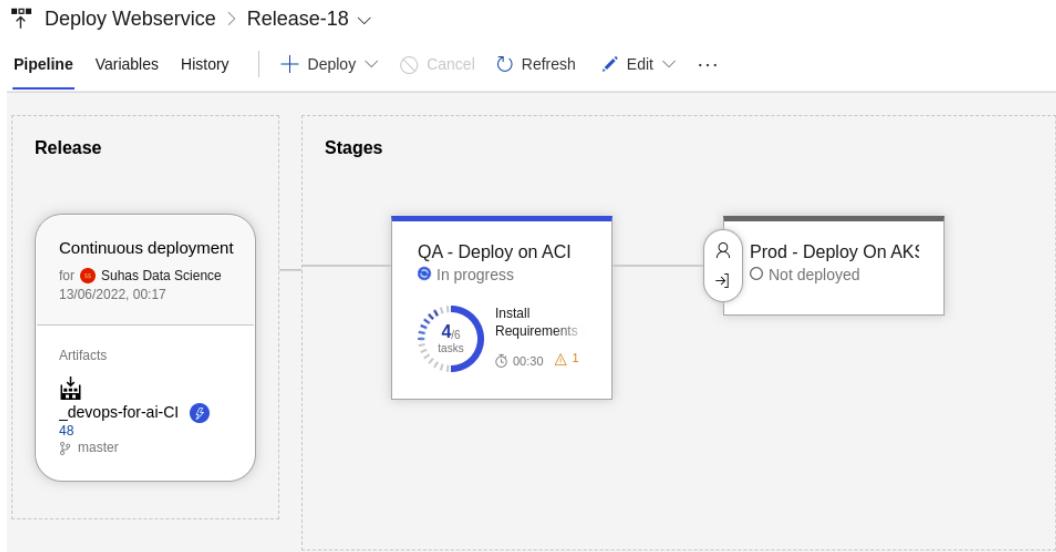


Figure 11.31: CD pipeline for ML-triggered after CI pipeline

It takes a few minutes to execute all the steps. You should see the number of tasks completed and the status of the stage. If you want to see the details of each step, click on the **Logs** option.

The following figure shows the status of each step with logs.

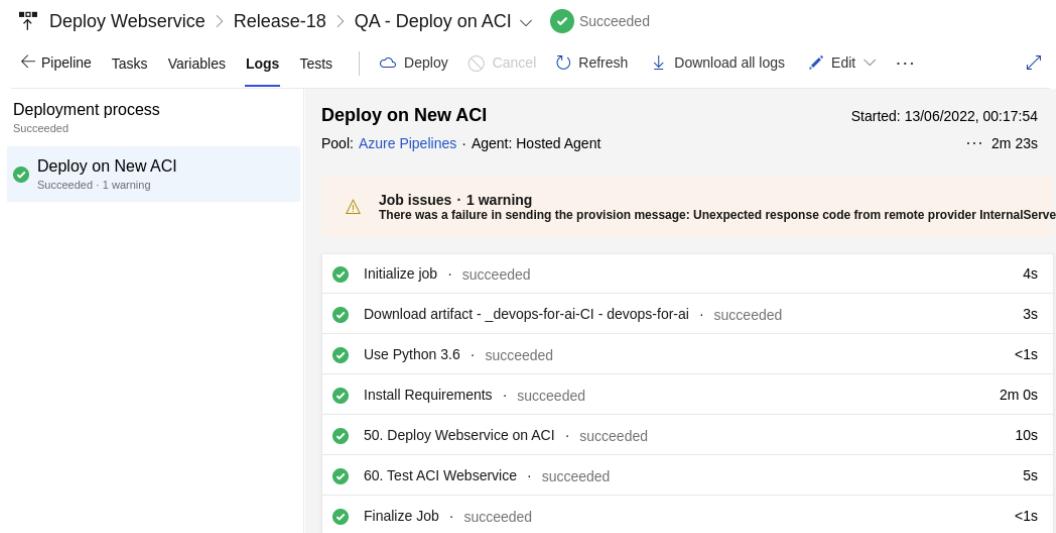


Figure 11.32: CD pipeline for ML-QA stage logs

When ACI deployment is completed, it waits for manual approval from the developer but for a limited time. You can update the time from the options. After approval, it will deploy to the AKS. If you want to automate this step, you can remove the manual approval or set auto approval. However, it is recommended to keep manual approval before deploying it on AKS.

In the following figure, you can see that the **QA – Deploy on ACI** stage is completed, and it is pending approval for deploying it on AKS. You can set the time for approval requests to be active. After approval, the pipeline will deploy the app using AKS.

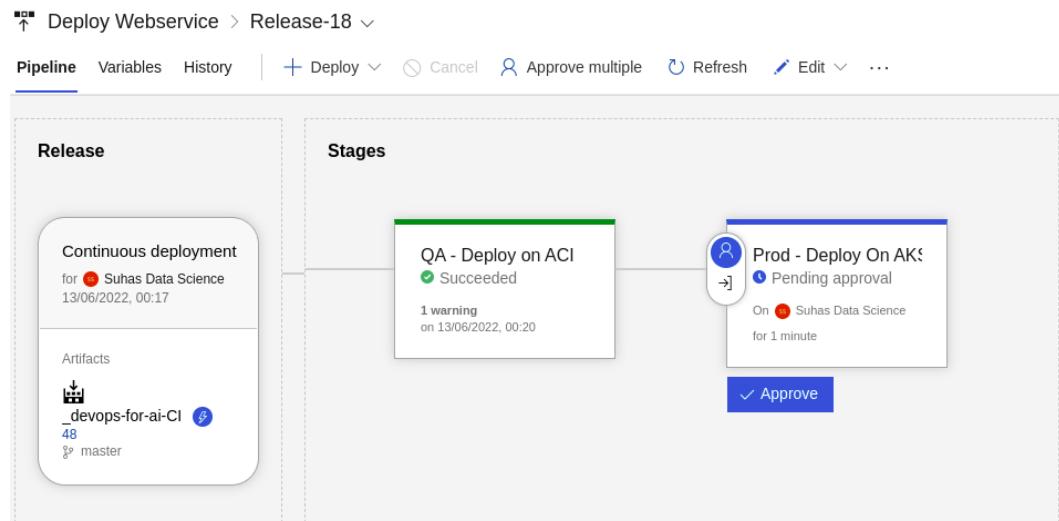


Figure 11.33: Azure pipeline for ML-completed

Now, head over to the Azure Machine Learning studio, and go to **Jobs** from the left panel. You should see the experiment with the name provided in the training script, that is, **mlops-classification**. On the right side, you can see the details like the experiment name, the date of creation, and the last submission. When you go to the experiment, you will see the run information with status and other metadata. On the top, you can see metrics details with graphs. In the current case, accuracy and CV score are being logged. You can assess each job's details by clicking on it.

The following figure shows the model metrics, that is, accuracy and CV score, along with run information and other details. In the graphs, you can see the value of each metric logged in the runs separately.



Figure 11.34: Azure Machine Learning-Model metrics

When a model is deployed in an **Azure Container Instance (ACI)**, you will get the scoring URI, which can be consumed by the subsequent applications.

The following figure shows the endpoint created by the script in the CI/CD pipeline:

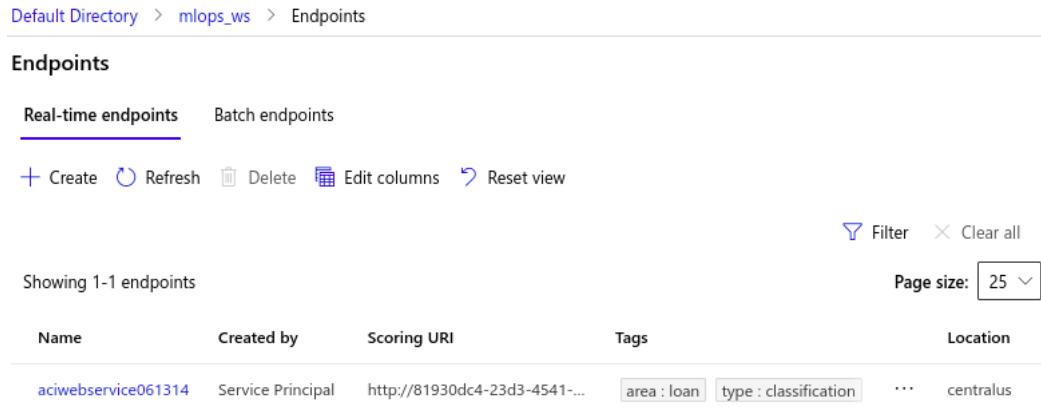


Figure 11.35: AML Real-time endpoints

You can assess the model endpoints by evaluating the outcome after passing the input data. Moreover, you can consume the endpoint using other languages, such as C#, Python, and R.

In the following figure, you can see the output produced by the model's endpoints under **Test results** after passing the test data:

The screenshot shows the AML Real-time endpoints testing interface. At the top, there is a breadcrumb navigation: Default Directory > mlops_ws > Endpoints > aciwebservice061314. Below this, the endpoint name "aciwebservice061314" is displayed. Underneath it, there are four tabs: Details, Test (which is selected), Consume, and Deployment logs. The "Input data to test real-time endpoint" section contains the JSON input:

```
{"data": [[1, 1, 0, 1, 0, 1, 2, 3849], [1, 1, 3, 0, 0, 0, 1, 5540]]}
```

. To the right, the "Test result" section displays the JSON response:

```
{"result": [1, 0]}
```

.

Figure 11.36: AML Real-time endpoints testing

You can assess the same endpoint in the Postman as well. The following figure shows the testing of scoring URI in the Postman:

The screenshot shows the Postman interface. At the top, it says "Azure CI/CD". Below that, there are buttons for "Examples 0" and "BUILD". The main area shows a "POST" request to the URL <http://81930dc4-23d3-4541-8d1b-d4eff6677c3c.azuredl.com/v1/predict>. The "Body" tab is selected, showing the JSON input:

```
1  {"data": [[1, 1, 0, 1, 0, 1, 2, 3849], [1, 1, 3, 0, 0, 0, 1, 5540]]}]
```

. The "Send" button is highlighted. Below the body, the response status is shown as "200 OK" with a total time of "261 ms" and a size of "273 B". The response body is also shown:

```
1  {"result": [1, 0]}
```

.

Figure 11.37: AML Real-time endpoints testing in Postman

Till now, you have learned to automate CI/CD workflow using Azure DevOps and Azure Machine Learning Service. You have explored GitHub Actions and

Azure DevOps approaches to build an automated workflow to deploy models in production. By now, you should be familiar with the Azure cloud and the services provided by Azure and its implementation.

Conclusion

In this chapter, you learned to deploy the app on the Azure platform (PaaS). You also built an automated CI/CD pipeline using GitHub Actions. Deployed ML web app in the Azure App Service using Azure Container Registry (ACR). Further on, you integrated tox with GitHub Actions to run the test cases as a part of the CI/CD pipeline before deploying the ML web app to Azure.

After that, you learned to deploy an ML app using Azure DevOps and Azure Machine Learning service (AML), which provides ML as a service (MLaaS). You analyzed the metrics logged by runs in the Azure Machine Learning studio, and you passed the sample data to the endpoint and assessed the outcome.

In the next chapter, you will explore the GCP platform to deploy ML apps.

Points to remember

- You need to authorize GitHub Actions to automate the CD part of the pipeline and provide the required details.
- Azure Role Based Access Control (RBAC) is an authorization system built on Azure Resource Manager that provides fine-grained access management of Azure resources.
- An experiment consists of several runs initiated from the script.
- In the Azure DevOps template, the *aml_service/15-EvaluateModel.py* script will compare the performance metric of the latest model against the deployed model before the deployment stage. It means the set of rules is written in this script, which will decide whether the model can be promoted to the next step.

Multiple choice questions

1. _____ can be defined as a single iteration of the training script.
 - a) A run
 - b) An experiment
 - c) A workspace
 - d) ACR

2. Azure primarily uses the _____ pricing plan.
 - a) Unlimited (for all Azure services)
 - b) Quarterly
 - c) Monthly
 - d) Pay-as-you-go

Answers

1. a
2. d

Questions

1. What is the use of Azure Machine Learning (AML) service?
2. How can you log in to Azure from the local terminal?
3. Explain the working of the Azure pipeline stages.

CHAPTER 12

Deploying ML Models on Google Cloud Platform

Introduction

Cloud computing is the on-demand delivery of computer system resources, such as servers, databases, analytics, storage, and networking. These resources and services are available off-premises; however, they can be accessed over the cloud (the internet) as per requirement. This means you do not need to set up big infrastructure on-premises. This is beneficial for businesses and individuals as they get the required system resources and services instantly under a pay-as-you-go plan. Resources and services can be added or removed quickly, which allows you to spend money efficiently.

Refer to the previous chapters to learn about concepts like Packaging ML Models, FastAPI, Docker, and CI/CD pipeline, as they are pre-requisites for this chapter.

Structure

In this chapter, the following topics will be covered:

- Create and set up an account on the Google Cloud Platform (GCP)
- Cloud Source Repositories
- Cloud Build

- Container Registry
- Deploy web-based ML app to Google Kubernetes Engine (GKE) with manual trigger
- Build an automated CI/CD pipeline to deploy web-based ML apps on Google Kubernetes Engine (GKE)

Objectives

After studying this chapter, you should be able to deploy an ML model on Google Kubernetes Engine (GKE). You can create a fully automated CI/CD pipeline with simple steps, without the need to integrate any external tool, service, or platform. You will create a remote Git repository on GCP using Cloud Source Repository and Kubernetes cluster to make a scalable ML app. You will also learn to create manifest files for Kubernetes in this chapter. By the end of it, you should be able to create triggers in Cloud Build for the automated deployment of ML apps.

Google Cloud Platform (GCP)

Google Cloud Platform (GCP) comprises cloud computing services offered by Google, which uses the same infrastructure as the one used by YouTube, Gmail, and other Google platforms or services. The platform offers a range of services for compute, Machine learning and AI, networking, IoT, and BigData. Here are a few services offered by the GCP:

- Google's compute engine provides VM instances to run the code and deploy the apps.
- AI and Machine learning services like Vertex AI, which is an end-to-end ML life cycle management and unified platform. Data Scientists can upload the data, build, train, and test ML models easily.
- AI building blocks, such as Vision AI, help derive insights from images using AutoML.
- Container services, such as Container Registry and **Google Kubernetes Engine (GKE)**, manage Docker images and allow developers to build scalable applications.
- BigQuery and data proc to process and analyze large amounts of data.
- Databases, such as Cloud SQL and Cloud Bigtable, store data on the cloud.
- Developer tools, such as Cloud Build, Cloud Source Repositories, and Google Cloud Deploy to automate CI/CD process.

- Management tools, such as Deployment Manager and Cost Management, help you to track the deployment and cost of tools or services used in projects.
- Networking services, such as **Virtual Private Cloud (VPC)**, let you create a virtual private cloud environment within a public cloud. Multiple projects created in different regions can communicate with each other without openly communicating through the public internet.
- Security services, such as cloud key management, firewalls, and security center.
- Storage services, such as Cloud Storage, allow you to store artifacts.
- Serverless computing, such as Cloud Function, is an event-driven serverless compute platform. This Function as a Service (FaaS) lets you run the code with no server or containers.
- Operations services, such as Cloud Logging and Cloud Monitoring, let you track the performance, delay, and such of the deployed models or applications.
- It also provides other services, such as migration, IoT, event management, identity and access, hybrid and multi-cloud, backup, and recovery.

You must have got the gist of GCP and its services; now, you are going to learn to deploy the Machine Learning model on GCP. After completing this chapter, you will be in a better position to work on GCP for ML model deployments.

GCP offers a free trial account for 90 days with \$300 credits. It will give hands-on experience to new customers so that they can explore the services offered by GCP.

Set up the GCP account

First of all, set up a GCP account; however, this is a one-time activity. It should be ready for use immediately on logging in:

Step 1:

Create a GCP account (if you don't have one) and log in. The free trial account can be created on the GCP platform at <https://cloud.google.com/free>.

First, log in using your Gmail ID; it will then redirect you to the GCP Free-trial page. Select your country from the dropdown, accept the Terms of Service, and then click on **Continue**. Next, choose **Individual** (for personal use) or **Business** (if it is a business account). After that, provide personal details like name, address, and city. Finally, provide payment mode details and complete the process.

Step 2:

After logging in to the account, create a new project, as shown in the following figure:



Figure 12.1: Create a new project

Click on the **NEW PROJECT** button and provide the project name. In this case, it is **MLOps**, and the project ID will be auto-generated; however, it can be edited. In this case, it is **mllops-54321**. Location can be pre-selected or can be changed.

Hit the **Create** button to complete the process.

Note: The project ID is unique to each project.

Cloud Source Repositories

In this case, the Git repository, that is, the Cloud Source Repository, is used; however, other remote Git repositories, such as GitHub, can also be used. Cloud Source Repositories are private Git repositories hosted on GCP. Multiple Git repositories can be created within a single project. It supports the standard set of Git commands, such as push, pull, clone, and log. Cloud Source Repositories can be added to a local Git repository as remote repositories. It allows collaboration and provides security. However, it is recommended not to store any personal or confidential data in it. The good part is that GCP's Cloud Source Repositories allow you to store up to 50 GB per month for free.

Now, search and enable **Cloud Source Repositories API**, which allows access to source code repositories hosted by Google.

Next, create a new repository in Cloud Source Repositories, and provide the repository name and project ID, as shown in the following figure:

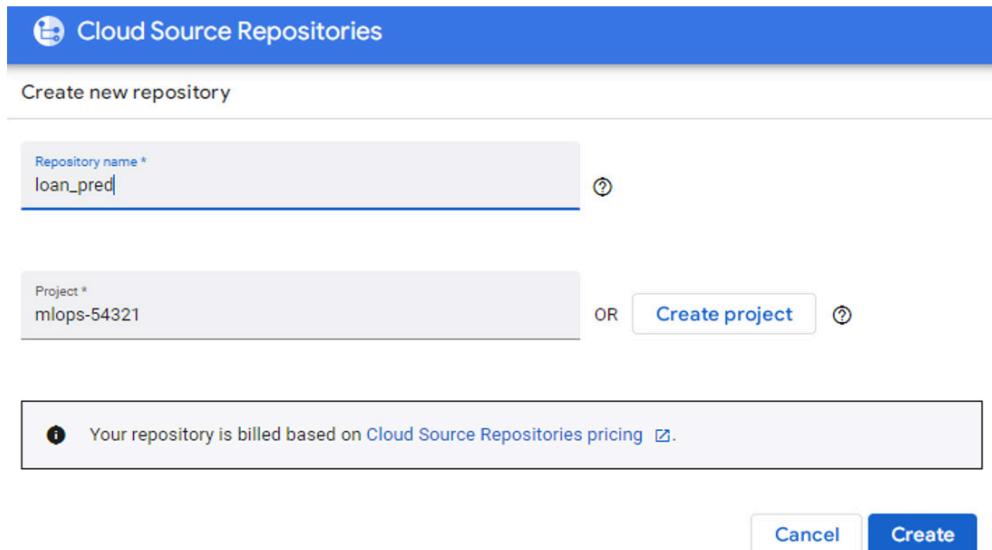


Figure 12.2: Create a new repository in Cloud Source Repositories

After creating a Cloud Source Repository, it will be empty. So, choose **Clone your repository to a local Git repository** option. Next, head to **Manually generated credentials** and follow the instructions to generate credentials by following the link **Generate and store Git credentials**. Simply copy the commands and run them in the local terminal. In the following figure, an empty Cloud Source Repository is created and options are selected, as discussed:

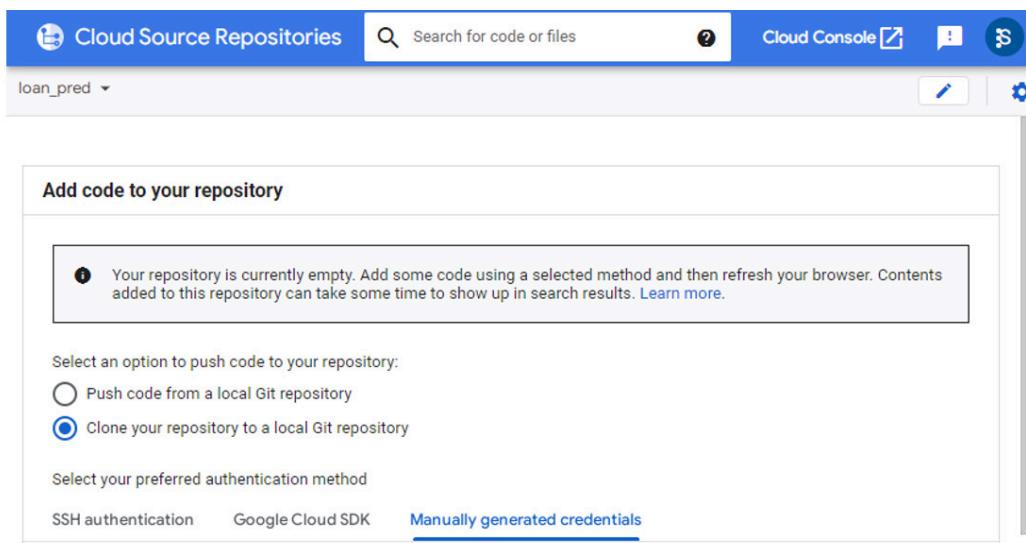


Figure 12.3: Cloning repository to local Git repository

Hereafter, the repository should be accessible from the local terminal. Now, create a new directory and clone the Cloud Source Repository to your local machine, include the code files in it, and push it to the Cloud Source Repository from the local machine.

Let's consider the scenario of loan prediction, where the problem statement is to predict whether a customer's loan will be approved. Feel free to implement hyper-parameter tuning and optimize the model.

First, create a package of ML code, and build a web app using FastAPI. Then, create the test cases and dependencies file, *Dockerfile*, and *docker-compose* file. Finally, create configuration files for GCP deployment, namely, *cloudbuild.yaml*, *deployment.yaml*, *service.yaml*.

Note: For code files in the *src* directory, refer to the ML model package developed in *Chapter 4: Packaging ML Models*, and for the rest of the code files, refer to *Chapter 11: Deploying ML Models on Microsoft Azure*. New code files added are *cloudbuild.yaml*, *deployment.yaml*, and *service.yaml*. Remove workflow directory *.github*. Also, the *tox.ini* file is updated for this scenario.

The code repository is also available on GitHub at <https://github.com/suhas-ds/GCP>.

The following directory structure shows the CI/CD pipeline files:

```
.
├── k8s
│   ├── deployment.yaml
│   └── service.yaml
└── src
    ├── build
    │   ├── bdist.linux-x86_64
    │   └── lib
    ├── prediction_model
    │   ├── config
    │   ├── datasets
    │   ├── processing
    │   ├── __pycache__
    │   ├── trained_models
    │   ├── __init__.py
    │   ├── pipeline.py
    │   ├── predict.py
```

```
|   |   ├── train_pipeline.py
|   |   └── VERSION
|   ├── prediction_model.egg-info
|   |   ├── dependency_links.txt
|   |   ├── PKG-INFO
|   |   ├── requires.txt
|   |   ├── SOURCES.txt
|   |   └── top_level.txt
|   ├── tests
|   |   ├── pytest.ini
|   |   └── test_predict.py
|   ├── MANIFEST.in
|   ├── README.md
|   ├── requirements.txt
|   ├── setup.py
|   └── tox.ini
├── cloudbuild.yaml
├── docker-compose.yml
└── Dockerfile
├── main.py
├── pytest.ini
└── requirements.txt
├── runtime.txt
└── start.sh
├── test.py
└── tox.ini
```

13 directories, 31 files

tox.ini

The *tox* is a free and open-source tool used for testing Python packages or applications. It creates the virtual environment and installs the required dependencies in it. Finally, it runs the test cases for that Python package ***prediction_model***. Update the *tox.ini* file as shown in the following code:

```
1. [tox]
2. envlist = my_env
3. skipsdist=True
4.
5. [testenv]
6. install_command = pip install {opts} {packages}
7. deps =
8.     -r requirements.txt
9.
10. setenv =
11. PYTHONPATH=src/
12.
13. commands=
14.     pip install requests
15.     pytest -v test.py --junitxml=test_log.xml
16.     pytest -v src/tests/ --junitxml=src_test_log.xml
```

The **skipsdist=True** flag indicates not to perform a packaging operation. It means the package will not be installed in the virtual environment before performing any test. Set **PYTHONPATH** to the *src* directory, where Python package files are placed. Under commands, pytest commands will be executed, and the results of the tests will be exported in *.xml* files.

After a successful push from the local terminal, code files will be displayed in the Cloud Source Repository, as shown in the following figure:

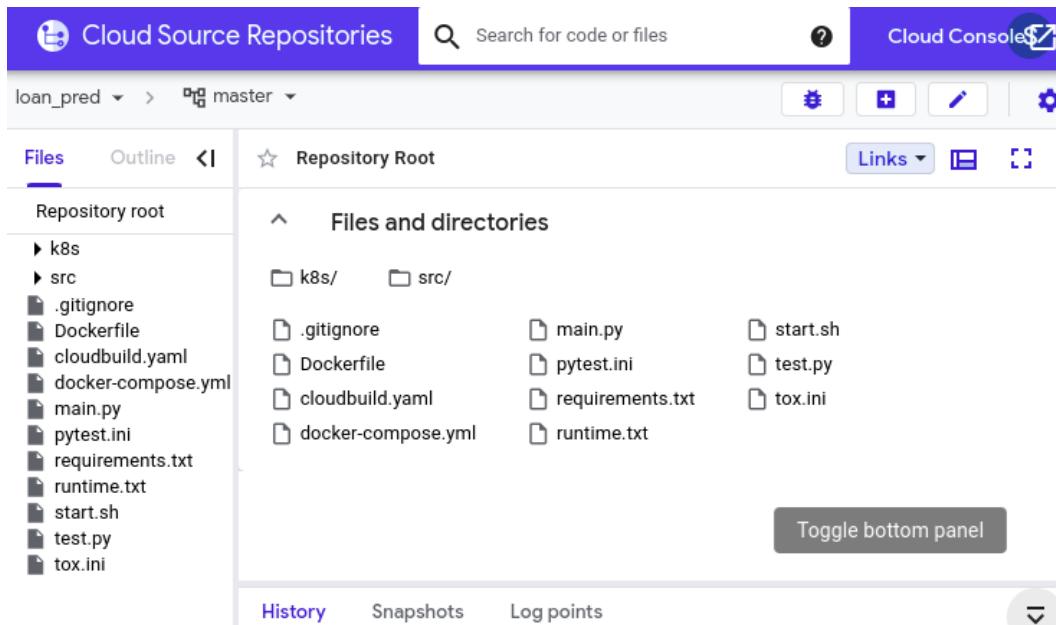


Figure 12.4: Code files pushed to Cloud Source Repository

Cloud Build

Cloud Build is a serverless platform that allows users to automate build, test, and deploy containers in the cloud quickly. Cloud Build works with deployment environments of the App Engine, Kubernetes Engine, Cloud Run, Cloud Functions, and Firebase. It will charge you for build minutes that are utilized if your usage exceeds the free quota allotted by GCP.

Cloud Build config file type can be YAML or JSON. This file contains a series of steps and commands to execute the build, specified by the developer. Each build step runs in its own Docker container. However, these containers are connected through a local Docker network with a Cloud Build. This permits **build** steps to communicate and share information.

Search and enable **Cloud Build API** in APIs and services. Cloud Build uses a special service account to execute builds on your behalf. The email format of the Cloud Build service account is **[PROJECT_NUMBER]@cloudbuild.gserviceaccount.com**. It has permission to execute tasks like fetching code from the repository, getting project details, and writing objects to storage associated with the project. However, this default service account can be changed to the developer's own service account to execute builds on the developer's behalf, and the developer can set specific permissions.

cloudbuild.yaml

Cloud builders are container images with common languages and tools installed in them. Cloud Build offers pre-built Docker images that can be used in the config file to execute commands. These pre-built images are available in the Container Registry at gcr.io/cloud-builders/.

In this case, the *cloudbuild.yaml* file will run the tests using tox and pytest. Next, it will build the Docker container image with the latest tag. Following that, it will push the image to the Container Registry and run the bash command to check the list of files in the current directory. Finally, it will pull the image from the Docker container to deploy it on **Google Kubernetes Engine (GKE)**.

```
1. steps:
2. # Run test
3. - name: 'python:3.7-slim'
4.   id: Test
5.   entrypoint: /bin/sh
6.   args:
7.     - -c
8.     - 'python -m pip install tox && tox'
9.
10. # Build the image
11. - name: 'gcr.io/cloud-builders/docker'
12.   id: Build
13.   args: ['build', '-t', 'gcr.io/$PROJECT_ID/mlapp:latest', '.']
14.   timeout: 200s
15.
16. # Push the image
17. - name: 'gcr.io/cloud-builders/docker'
18.   id: Push
19.   args: ['push', 'gcr.io/$PROJECT_ID/mlapp:latest']
20.
21. - name: 'gcr.io/cloud-builders/gcloud'
22.   id: Bash
23.   entrypoint: /bin/sh
```

```
24. args:  
25. - -c  
26. - |  
27. echo "List of files and directories within the current working  
directory"  
28. ls -l  
29.  
30. # Deploy container image to GKE  
31. - name: "gcr.io/cloud-builders/gke-deploy"  
32. id: Deploy on GKE  
33. args:  
34. - run  
35. - --filename=k8s/  
36. - --image=gcr.io/$PROJECT_ID/mlapp:latest  
37. - --location=us-west1-b  
38. - --cluster=mlkube
```

In the preceding file, mainly three (supported) builder images are used:

gcr.io/cloud-builders/docker: To build and push the Docker container image to the Container Registry

gcr.io/cloud-builders/gcloud: To run the inline bash script

gcr.io/cloud-builders/gke-deploy: To deploy containerized app in GKE

Also, **python:3.7-slim** is the publically available image used for testing the code using tox. If any image is to be used from the Docker hub, then simply provide the image name in single quotes. However, if an image is from other registries, then the full registry path needs to be specified in single quotes. The **args** field of a build step accepts a list of arguments and passes them to the image referred to by its **name** field.

Container Registry

Container Registry is a container image managing and storing private container images service offered by GCP. It allows users to push and pull container images securely. In this case, the Docker container image will be pushed to the Container Registry. The Kubernetes engine will pull the image to deploy the ML app on the cloud.

Container Registry uses the hostname, project ID, image, and tag or image digest to access images, where image digest is the sha256 hash value of the image contents. The format is as follows:

HOSTNAME/PROJECT-ID/IMAGE : TAG

Or

HOSTNAME/PROJECT-ID/IMAGE :@IMAGE-DIGEST

The following figure shows the GCP Container Registry after enabling the Container Registry API, where **mlapp** is an app name and **gcr.io** is a hostname. Currently, **gcr.io** hosts the images in the United States.

The screenshot shows the Google Cloud Container Registry interface. The top navigation bar includes 'Google Cloud' and 'MLOps'. The left sidebar has 'Container registry' selected under 'Images' and 'Settings'. The main area is titled 'Repositories' and shows a single entry: 'MLOps'. Below it is a 'Filter' section with fields for 'Name' (set to 'mlapp'), 'Hostname' (set to 'gcr.io'), and 'Visibility' (set to 'Private').

Name	Hostname	Visibility
mlapp	gcr.io	Private

Figure 12.5: Container Registry

Kubernetes

Container orchestration is an automation of the operational process needed to run containerized workloads and services. It automates tasks like deployment, scaling up-down, lifecycle management, load balancing, configuration, security, resource allocation, health monitoring, and networking of containers.

Kubernetes (also known as K8s or Kube) is an open-source platform for container orchestration. It allows the application to scale on the fly, without interrupting the application running in production. It creates multiple replicas of containerized applications quickly to handle increased traffic and automatically reduces the number of replicas when traffic decreases.

Deployment on Kubernetes creates Pods with containers inside them. Pods are the smallest unit in the Kubernetes environment. A Pod can have one or more containers. It always runs on a Node; however, a Node can have multiple Pods. A Node is just a worker (VM or physical machine) in a Kubernetes environment. All nodes are managed by a control plane.

Every Node runs the Kubelet and container runtime, such as Docker. Kubelet is the medium of communication between the control plane and the Node. It also manages Pods and running containers inside the Node. Container runtime such as Docker will pull the image from the registry and it will run the containerized application.

Google Kubernetes Engine (GKE)

Google Kubernetes Engine (GKE) offers the infrastructure to manage, deploy, and scale containerized applications. The underlying architecture of GKE consists of a set of compute engine instances assembled to form a cluster.

Manifest is a file (JSON or YAML) containing a description of all the components you want to deploy. These manifest files guide Kubernetes to network between containers. Kubernetes schedules the deployment of containers into clusters and identifies the best host for the container. After deciding on a host, it manages the lifecycle of the container based on pre-planned specifications.

Search and enable **Kubernetes Engine API** in the GCP console, as shown in the following figure:

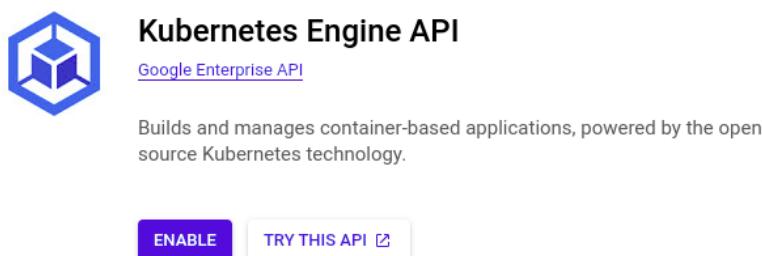


Figure 12.6: Kubernetes Engine API

Next, click on the **Activate Cloud Shell** icon in the top-right corner of the GCP account. It will open a cloud-based terminal:

```
gcloud container clusters create mlkube --zone "us-west1-b" --machine-type "n1-standard-1" --num-nodes "1" --project mlops-54321
```

```
Created [https://container.googleapis.com/v1/projects/mlops-54321/zones/us-west1-b/clusters/mlkube].
To inspect the contents of your cluster, go to: https://console.cloud.google.com/kubernetes/workload_/gcldns/us-west1-b/mlkube?project=mlops-54321
kubeconfig entry generated for mlkube.
NAME: mlkube
LOCATION: us-west1-b
MASTER_VERSION: 1.22.8-gke.202
MASTER_IP: 34.168.38.219
MACHINE_TYPE: n1-standard-1
NODE_VERSION: 1.22.8-gke.202
NUM_NODES: 1
STATUS: RUNNING
```

Figure 12.7: Creating Kubernetes cluster with cloud shell

After the successful creation of the Kubernetes cluster, its details can be checked using the GKE service, as shown in the following figure:

The screenshot shows the GKE Overview page. At the top, there are buttons for CREATE, DEPLOY, REFRESH, OPERATIONS, and HELP ASSISTANT. Below this is a navigation bar with tabs: OVERVIEW (selected), MONITORING, PREVIEW, and COST OPTIMISATION. A search bar labeled 'Filter Enter property name or value' is present. The main table displays one cluster entry:

Status	Name	Location	Number of nodes	Total vCPUs	Total memory	Notifications
<input type="checkbox"/>	mlkube	us-west1-b	1	1	3.75 GB	💡 Upcoming nodes upgrade

Figure 12.8: Kubernetes cluster

All manifest files contain the **apiVersion** field. Each Kubernetes configuration file has three sections: metadata, specification, and status. Metadata and specification are provided by the developers in the configuration files; however, the status part is managed by Kubernetes. For instance, if 2 replicas are declared but only one replica is up and running, then Kubernetes scrutinize the status and creates one more replica to match the desired state and the actual state. As shown as follows, Kubernetes configuration files are placed in the *k8s* directory. The **labels** and **selectors** are responsible for connecting *service* and *deployment* files. The same labels should be used in the *service.yaml* and *deployment.yaml* files for successful communication between service and deployment.

k8s

```
└── deployment.yaml
└── service.yaml
```

deployment.yaml

This file contains the configuration of Kubernetes deployment, such as the number of replicas to be created and the container image to be deployed. The role of *deployment.yaml* is to launch a Pod with a containerized app and ensure that the necessary number of replicas are always up and running in the Kubernetes cluster. In the following file, a **template** is a blueprint for a Pod. It has its own metadata and specification. In this case, the **replicas** field is set to **2**, which means it will create two replicas of the Pods.

1. **apiVersion: apps/v1**
2. **kind: Deployment**
3. **metadata:**

```
4.   name: loan-prediction
5. spec:
6.   replicas: 2
7.   selector:
8.     matchLabels:
9.       app: mlapp
10.  template:
11.    metadata:
12.      labels:
13.        app: mlapp
14.    spec:
15.      containers:
16.        - name: loan-prediction-app
17.          image: gcr.io/mlops-54321/mlapp:latest
18.        ports:
19.          - containerPort: 8000
```

service.yaml

This file connects the containerized application to the end user. The role of *service.yaml* is to expose an interface to a Pod created by *deployment.yaml*, which enables network access between cluster and service. It exposes the deployed application to the external world. Port **8000** of containerized application is mapped with port **80**. In ports, you can specify more than one port. In this case, the type of service is **LoadBalancer**, which will expose the service through GCP's (cloud provider's) load balancer. Label selector helps to locate Pods. In this case, **mlapp** is a label.

```
1. apiVersion: v1
2. kind: Service
3. metadata:
4.   name: mlapp
5. spec:
6.   type: LoadBalancer
7.   selector:
8.     app: mlapp
9.   ports:
```

```

10.  - port: 80
11.  targetPort: 8000

```

Note: In the `targetPort` field, the uppercase letter 'P' is used, as it is a key-value pair. In the key-value pair, the second word's first character should be uppercase.

Deployment using Cloud Shell – Manual Trigger

First, open a cloud shell and clone the remote Git repository using the `git clone` command. Now, code files are accessible in the GCP cloud shell terminal. Run the `cloudbuild.yaml` file manually with the following command:

```
gcloud builds submit --config cloudbuild.yaml --project=mlops-54321
```

This command will take a few minutes to execute. In this scenario, the pipeline is triggered manually, where a series of steps will be executed sequentially from the `cloudbuild.yaml` file.

The following figure shows the output of the preceding command, in which details of the deployed app using Kubernetes are given, such as the IP address of the deployed ML app, source, duration, status, and GKE URLs for workloads, services, configuration, and storage.

```

Step #3: #####
Step #3: > Deployed Objects
Step #3:
Step #3: NAMESPACE      KIND          NAME        READY
Step #3: default        Deployment    loan-prediction  Yes
Step #3: default        Service       mlapp        Yes      http://35.197.16.21
Step #3:
Step #3: #####
Step #3: > GKE
Step #3:
Step #3: Workloads:      https://console.cloud.google.com/kubernetes/workload?project=mlops-54321
Step #3: Services & Ingress: https://console.cloud.google.com/kubernetes/discovery?project=mlops-54321
Step #3: Applications:   https://console.cloud.google.com/kubernetes/application?project=mlops-54321
Step #3: Configuration:  https://console.cloud.google.com/kubernetes/config?project=mlops-54321
Step #3: Storage:        https://console.cloud.google.com/kubernetes/storage?project=mlops-54321
Step #3:
Finished Step #3
PUSH
DONE
-----
ID: 6798570d-1507-4565-9a9c-0dfe4578883a
CREATE_TIME: 2022-07-28T03:13:13+00:00
DURATION: 1M52S
SOURCE: gs://mlops-54321_cloudbuild/source/1658977991.765627-926cc43c19884630a9eb67f0da5ca957.tgz
IMAGES: -
STATUS: SUCCESS

```

Figure 12.9: Deploying ML app on GKE with manual trigger

CI/CD pipeline using Cloud Build

GCP comes with a set of services for the CI/CD pipeline. In the previous section, the build was triggered manually with the cloud shell terminal. To automate the manual process, CI/CD pipeline needs to be built with an automated trigger. The following figure shows the list of CI/CD services provided by GCP:

CI/CD		
Integrate and deliver continuously		
	Name	Description
Cloud Build	Cloud Build	Continuous integration delivery platform
Container registry	Container registry	Private container registry storage
Source Repositories	Source Repositories	Hosted private Git repos
Artifact Registry	Artifact Registry	Universal build artifact management
Cloud Deploy	Cloud Deploy	Managed continuous delivery to GKE

Figure 12.10: CI/CD services provided by GCP

When an update is pushed to **Cloud Source Repository (CSR)**, it will trigger the CI/CD pipeline. Cloud Build configuration file first runs the tests, and then builds and pushes the container image to the Container Registry. Simultaneously, it reads the Kubernetes manifest and deploys it on the Kubernetes engine. Kubernetes engine will pull the container image pushed by Cloud Build and eventually run the containerized ML app that is accessible to the end user.

The following figure shows the automated CI/CD pipeline to deploy the containerized app on GKE:

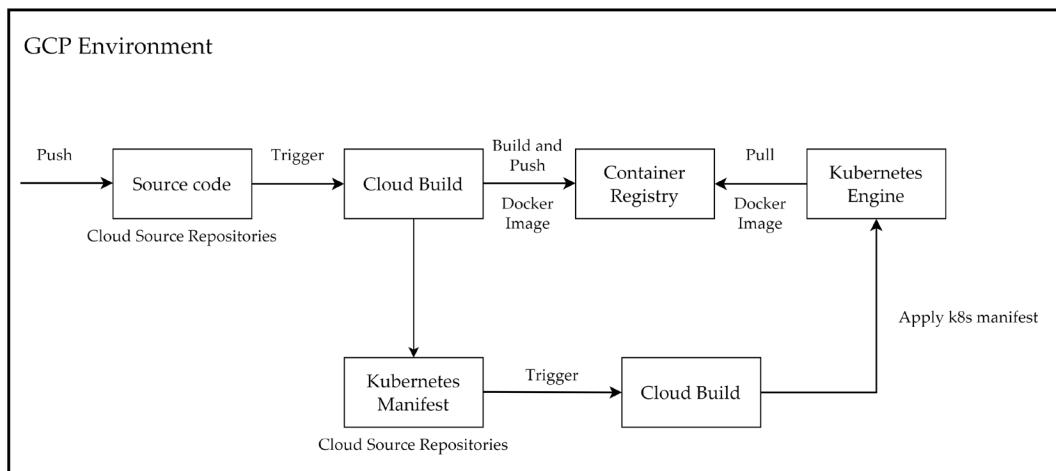


Figure 12.11: CI/CD pipeline to deploy ML app on GKE

Create a trigger in Cloud Build

In this section, create a trigger in Cloud Build that will look for a Cloud Build configuration file, that is, a YAML file, to build and push the Docker image to the Container Registry. It is also responsible for applying the Kubernetes manifest in the Kubernetes engine.

First, head to the **Triggers** section from the left side panel in Cloud Build. For the first time, it will show **No triggers found in global**, as shown in the following figure:

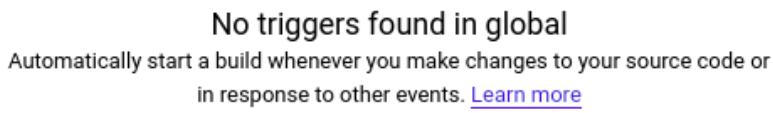


Figure 12.12.: Creating trigger in Cloud Build

Create a new trigger using the **CREATE TRIGGER** button. Next, provide a unique name within the project's region for triggers and select the region from the dropdown. **Description** and **Tags** are optional.

In this case, the **Name** is **AutoDeploy**, the **Region** is **global(non-regional)** and the **Description** is **M1app**, as shown in the following figure:

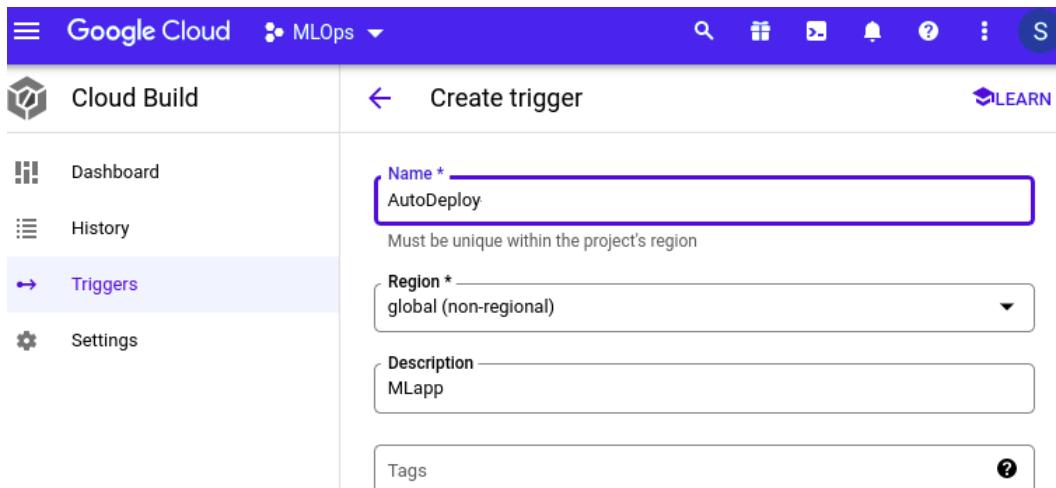


Figure 12.13: Creating trigger in Cloud Build – providing a name and selecting a region

Following that, select the repository event that will trigger the pipeline. It provides multiple options; however, **Push to a branch** is selected. So, any changes pushed to the **master** branch of the repository will trigger the pipeline. This will look for changes in the repository and clone the repository when the trigger is invoked.

In the following figure, the **Push to a branch** option is selected, along with the Cloud Source Repository and the master branch as a source:

[← Create trigger](#)

Event

Repository event that invokes trigger

Push to a branch

Push new tag

Pull request
Not available for Cloud Source Repositories

Or in response to

Manual invocation

Pub/Sub message

Webhook event

Source

Repository *

Select the repository to watch for events and clone when the trigger is invoked

Branch *

Trigger only for a branch that matches the given regular expression [Learn more](#)

Figure 12.14: Creating trigger in Cloud Build – selecting an event and source

Finally, complete the process by selecting the configuration file type and location. In this case, the **Cloud Build configuration file (YAML or JSON)** option is selected for **Configuration Type** and the *cloudbuild.yaml* file location from the repository is provided, as shown in the following figure:

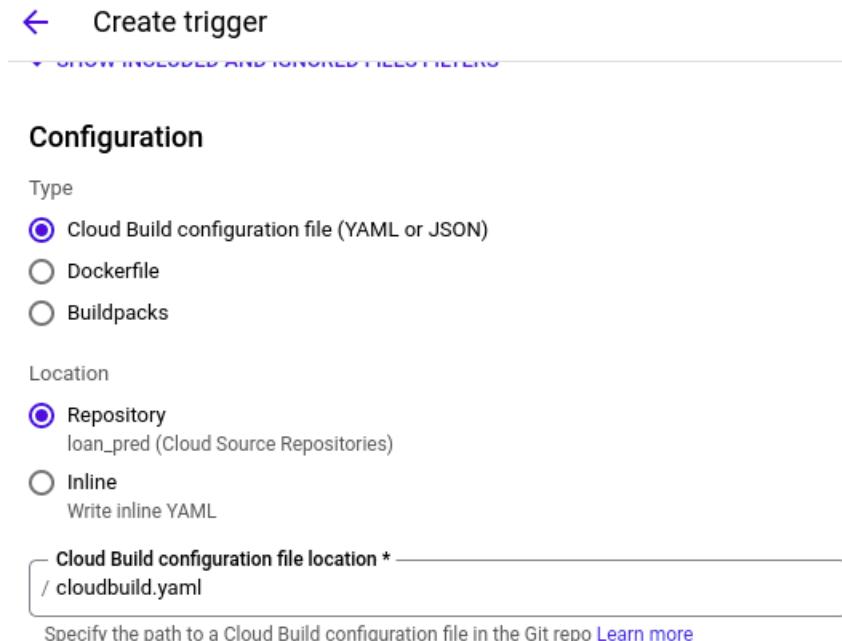


Figure 12.15: Creating trigger in Cloud Build – selecting configuration file type and its location

Now, any update pushed to the Cloud Source Repository will trigger the pipeline. To see the build history, go to the **History** section from the side panel or view it through the link from the dashboard.

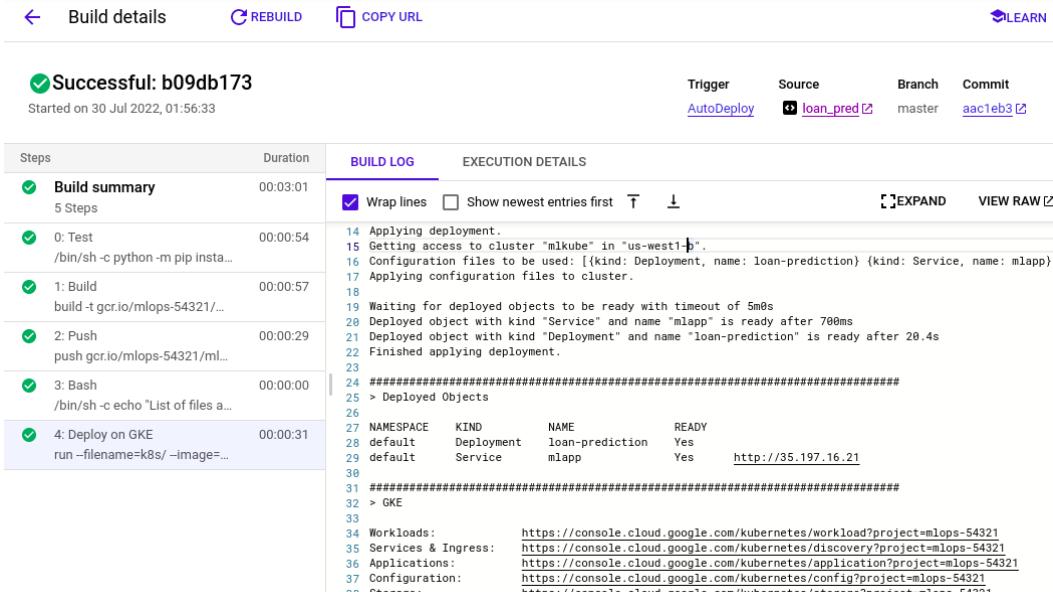
The following figure shows the build history with **Status**, **Build ID**, the **Source - loan_pred**, that is, the Cloud Source Repository in this case, and **Ref - the master branch** of that repository in this case. It also shows the **Trigger type**, **Trigger name**, and **Duration**, that is, the time taken to complete the build.

Build history		STOP STREAMING BUILDS		LEARN	
Region		global (non-regional)		?	
Filter Enter property name or value					? ☰
Status	Build	Source	Ref	Commit	Trigger Type
<input type="checkbox"/>	b09db173	loan_pr...	master	aac1eb3	Push to branch
<input checked="" type="checkbox"/>					AutoDeploy
					3 min 1 sec

Figure 12.16: Build history

You can see the live build logs for each step. It helps you to debug the errors that occurred during the build process. In the last step, you should see GKE deployment details, such as status, name, and URLs to access the app.

The following figure shows the build summary with the build ID and duration or time taken by each step:



The screenshot shows the Cloud Build details page for a successful build (b09db173). The build was triggered by AutoDeploy and originated from the branch master. The commit hash is aac1eb3. The build summary indicates 5 steps: Test, Build, Push, Bash, and Deploy on GKE, all completed successfully within 0:03:01. The execution details show the build log, which includes applying deployment, getting access to the cluster, and deploying objects. The log ends with deployment details for a service named mlapp with URL http://35.197.16.21.

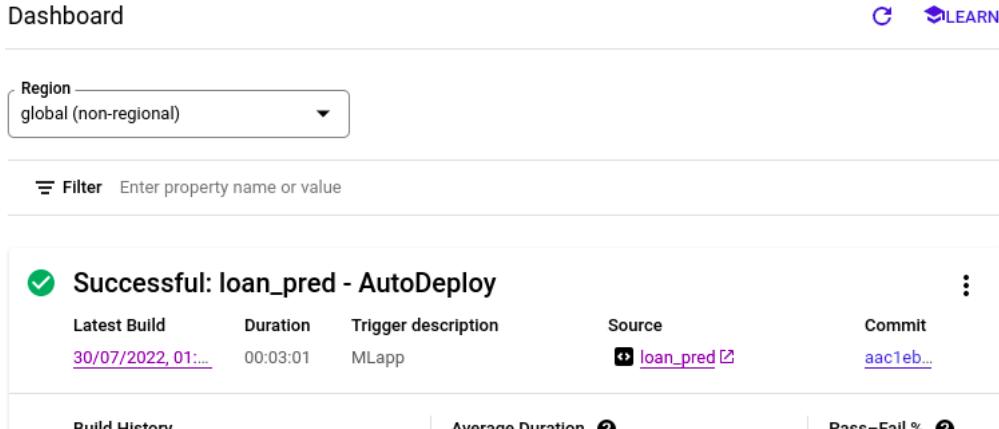
```

BUILD LOG
14 Applying deployment.
15 Getting access to cluster "mlkube" in "us-west1-b".
16 Configuration files to be used: [{kind: Deployment, name: loan-prediction} {kind: Service, name: mlapp}]
17 Applying configuration files to cluster.
18
19 Waiting for deployed objects to be ready with timeout of 5m0s
20 Deployed object with kind "Service" and name "mlapp" is ready after 700ms
21 Deployed object with kind "Deployment" and name "loan-prediction" is ready after 20.4s
22 Finished applying deployment.
23
24 #####
25 > Deployed Objects
26
27 NAMESPACE KIND NAME READY
28 default Deployment loan-prediction Yes
29 default Service mlapp Yes http://35.197.16.21
30
31 #####
32 > GKE
33
34 Workloads: https://console.cloud.google.com/kubernetes/workload?project=mlops-54321
35 Services & Ingress: https://console.cloud.google.com/kubernetes/discovery?project=mlops-54321
36 Applications: https://console.cloud.google.com/kubernetes/application?project=mlops-54321
37 Configuration: https://console.cloud.google.com/kubernetes/config?project=mlops-54321
38 Storage: https://console.cloud.google.com/kubernetes/storage?project=mlops-54321

```

Figure 12.17: Build details – Build log

The dashboard of the Cloud Build display summarizes details, such as the status of the **Latest Build**, **Duration**, **Build History of Pass-Fail** builds, the **Average Duration** of the build, and **Pass-Fail%**, as shown in the following figure:



The dashboard shows a successful build for the project loan_pred - AutoDeploy. The latest build was completed on 30/07/2022 at 01:... with a duration of 00:03:01 and triggered by MLApp. The build history section shows the same successful build entry.

Build History	Average Duration	Pass-Fail %
Successful: loan_pred - AutoDeploy 30/07/2022, 01:... 00:03:01 MLApp	?	?

Figure 12.18: Dashboard - Cloud Build

After successful deployment, you should see the ML app up and running in GKE, as shown in the following figure:

Loan Prediction Model API 0.1 OAS3

[/openapi.json](#)

A simple API that use ML model to predict the Loan application status

default

GET	/ Index	▼
GET	/health Healthcheck	▼
POST	/predict_status Predict Loan Status	▼
POST	/predict Get Loan Details	▼

Figure 12.19: ML app running in GKE

Google Kubernetes also offers an app monitoring service. Go to the service ingress URL in GKE. This includes resource usage, details, events, and logs. The following figure shows the overview of the monitoring of deployed apps:

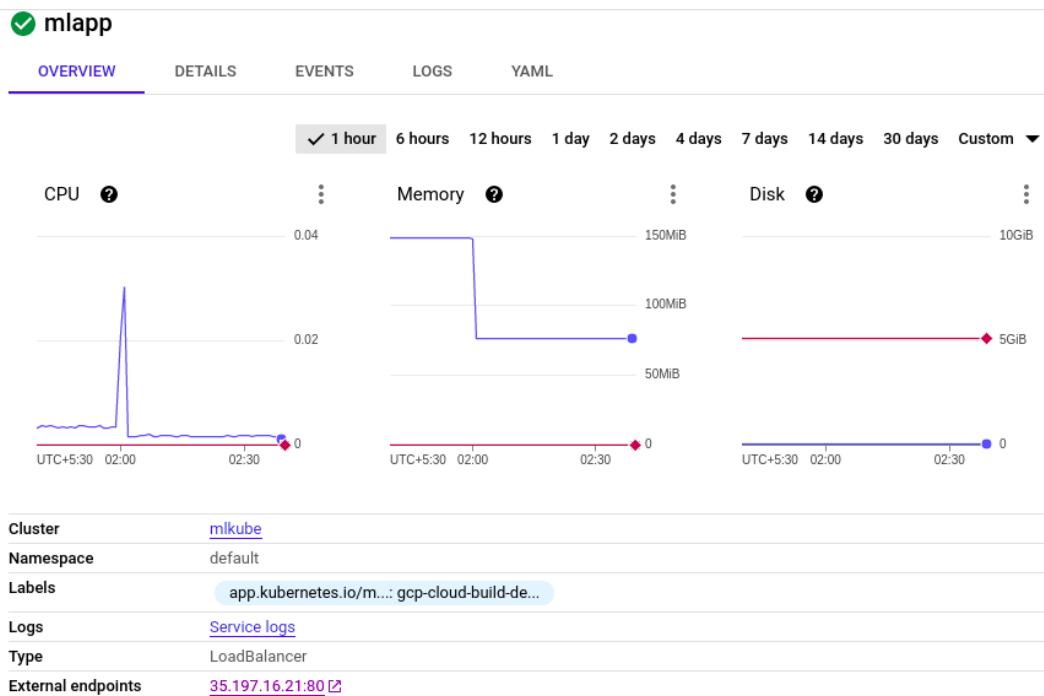


Figure 12.20: Monitoring mlapp

You have now learned how to create a simple CI/CD pipeline using GitHub to deploy ML apps on the GCP platform. However, you can modify it as per business and application requirements.

Conclusion

In this chapter, the ML app was deployed on Google Kubernetes Engine (GKE) with CI/CD pipeline. First off, you created a GCP account and created code files, including *cloudbuild.yaml* and Kubernetes manifest files for GCP deployment, and pushed them to Cloud Source Repositories (CSR). Then, you created a Container Registry to push-pull Docker container images and a Kubernetes cluster **ml-kube** to deploy an ML app. Finally, you enabled Cloud Build API and created an **AutoDeploy** trigger by integrating Cloud Source Repositories (CSR) to automate ML app deployment.

In the next chapter, you will learn to deploy an ML app on Amazon Web Services (AWS).

Points to remember

- Kubernetes cluster can be shared among multiple projects.
- Gmail email ID is required to create an account on GCP.
- Cloud Build is a serverless platform that allows you to automate build, test, and deploy containers quickly.
- Cloud builders are container images with common languages and tools installed in them.
- The same labels should be used in *service.yaml* and *deployment.yaml* files for successful communication between service and deployment.

Multiple choice questions

1. _____ is CI/CD service provided by Google.
 - a) Cloud Build
 - b) BigQuery
 - c) Virtual Private Cloud (VPC)
 - d) Pub/Sub

2. Manifest file's recommended file format is _____.
- a) .git
 - b) YAML
 - c) Python
 - d) Shell script

Answers

- 1. a
- 2. b

Questions

1. What is the location of the Cloud builder's pre-built container images?
2. What is the role of a Container Registry?
3. What is container orchestration?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 13

Deploying ML Models on Amazon Web Services

Introduction

Cloud computing refers to storing and accessing data and applications over the internet. Usually, data is stored on a remote server. Simply put, you can access data and applications from anywhere on the internet without worrying about the physical or on-premises infrastructure, which makes cloud computing more popular among organizations.

Amazon Web Services (AWS) helps solve on-premises infrastructure issues. AWS can spin up 100-1000 servers in a few minutes and, extra or unused servers will be removed. It is easy to scale applications with AWS. You can add more storage for applications or data. AWS helps focus on building and deploying applications on the cloud without worrying about setting up infrastructure from scratch.

Refer to the previous chapters if you need to study the concepts discussed, such as packaging ML models, FastAPI, Docker, and CI/CD pipeline.

Structure

In this chapter, the following topics will be covered:

- Introduction to Amazon Web Services (AWS)

- AWS Elastic Container Registry (ECR)
- AWS CodeCommit
- Amazon Elastic Container Service (ECS)
- AWS CodeBuild
- Application Load Balancer (ALB)
- Deploy web-based ML app to Elastic Container Service (ECS) with service and Application Load Balancer (ALB)
- AWS CodePipeline
- Build an automated CI/CD CodePipeline to deploy a web-based ML app on Amazon Web Services (AWS)

Objectives

After studying this chapter, you should be able to deploy an ML model on Amazon Elastic Container Service (ECS) without the need to integrate any external tool, service, or platform except AWS. You will create a remote Git repository on AWS using AWS CodeCommit and Amazon Elastic Container Service (ECS) cluster to run scalable ML apps. You will also learn to integrate Application Load Balancer (ALB) with Amazon Elastic Container Service (ECS) for routing requests coming from the external world. Moving on, you will integrate the service port with the Docker port via port mapping, create a security group for Application Load Balancer (ALB) and learn to push the Docker container image to AWS Elastic Container Registry (ECR). You should be able to create a fully automated CI/CD pipeline with AWS CodePipeline, AWS CodeBuild, AWS Elastic Container Registry (ECR), Application Load Balancer (ALB), and Amazon Elastic Container Service (ECS) after completing this chapter.

Introduction to Amazon Web Services (AWS)

Amazon Web Services (AWS) is a cloud infrastructure where you can host applications. In 2006, AWS started offering IT services to the market in the form of web services. AWS is one of the leading cloud service providers.

AWS compute services

Amazon Web Services (AWS) offers compute services for managing workloads that comprise many servers or instances.

Here are some of the widely used compute services offered by AWS that can be used as per the business or application requirements. These services will be discussed later in the chapter.

Amazon Elastic Compute Cloud (EC2)

Amazon EC2 is a virtual machine that represents a remote server. Amazon EC2 service is grouped under **Infrastructure-as-a-Service (IaaS)**. Amazon EC2 enables applications with resizable computing capacity, and these EC2 machines are known as instances. You can create multiple instances with different computing sizes. You can even upgrade the ram, vCPU, and so on after creation, so you do not need to recreate a new instance and configure it again. This is why the elastic term is used.

Amazon Elastic Container Service (ECS)

Amazon **Elastic Container Service (ECS)** enables you to deploy, scale, and manage containerized applications. It manages containers and enables developers to run containerized applications across the cluster of EC2 instances. However, it is not based on Kubernetes. Amazon ECS is free, meaning you don't need to pay for the ECS cluster. However, additional charges are to be paid for EC2 instances running in ECS tasks. There are mainly two ways to launch an ECS clusters:

- Fargate Launch
- EC2 Launch

Amazon ECS is a technology owned exclusively by AWS. ECS easily integrates with **AWS Application Load Balancer (ALB)** and **Network Load Balancer (NLB)**.

Amazon Elastic Kubernetes Service (EKS)

Amazon **Elastic Kubernetes Service (EKS)** is a Kubernetes service backed by AWS, which enables you to build Kubernetes clusters on AWS without manually installing Kubernetes on EC2. Amazon Elastic Kubernetes Service (EKS) allows you to manage or orchestrate containers in a Kubernetes environment. You need to pay for the EKS cluster, with additional charges for EC2 instances running inside the Kubernetes pod. Amazon EKS service set up and manages the Kubernetes control plane for you.

It is a good choice if you are looking for multi-cloud functionality and additional features compared to the Amazon **Elastic Container Service (ECS)**.

Amazon Elastic Container Registry (ECR)

Amazon **Elastic Container Registry (ECR)** allows developers to store, share and deploy Docker images on Amazon ECS. It provides security to images stored in

it. Amazon Elastic Container Registry (ECR) is integrated with Amazon ECS. It is similar to the Docker Hub container registry but is managed by AWS. It allows you to store private Docker container images in it.

AWS Fargate

Simply put, AWS Fargate is a serverless compute for containers. You pay for the usage per minute for the resources used by containers like virtual CPU (vCPU) and memory. AWS Fargate is compatible with both Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS).

AWS Lambda

AWS Lambda is simple and less expensive. It is a serverless and event-driven compute service; it is a **Function as a Service (FaaS)**. That means you don't need to manage servers or clusters. Events could be any simple activity, such as a user clicking on links to get the latest news. The term Lambda is supposed to be borrowed from functions of lambda calculus and programming. Mostly, it comprises three components:

- A function: It is the actual code to perform the task.
- A configuration: It dictates the execution of a function.
- An event source: This is the event that triggers the function. However, this component is optional.

AWS Lambda is a good choice for event-driven programming or when you need to access several services.

Amazon SageMaker

AWS also provides a machine learning platform as a service, that is, Amazon SageMaker. It removes the overhead of managing and maintaining servers manually. It also reduces the time and cost of machine learning model deployment on the AWS cloud.

Amazon SageMaker is a cloud-based machine learning platform that enables data scientists and developers to build, train, tweak, and deploy machine learning models in production environments. It uses Amazon **Simple Storage Service (S3)** to store the data and comes with over 15 most commonly used built-in machine learning algorithms for training the data. It deploys the ML models to SageMaker endpoints. Amazon SageMaker uses the Amazon Elastic Container Registry (ECR) to store container images.

You must have got the gist of AWS and its services. You are going to deploy a Machine Learning model to the Amazon Elastic Kubernetes Service (EKS). After this chapter, you will be comfortable working on AWS for ML model deployment.

Set up an AWS account

First of all, set up an AWS account; however, this is a one-time activity. It is ready to use immediately on log in.

The AWS Free Tier provides customers the ability to explore and try out AWS services free of charge up to specified limits for each service. The Free Tier consists of three different types of offerings, a 12-month Free Tier, an Always Free offer, and short-term trials.

Step 1: Create or log in to the existing AWS account

Create an AWS account (if you don't have one) and sign in. A free trial account can be created on the AWS platform at <https://aws.amazon.com/free>.

First, click on the **Create a Free Account** button. Next, provide the login details, such as email ID, password, and AWS account name. Then, select the account type (Professional or Personal) and provide contact information. After that, issue PAN and payment details. Finally, complete the verification process on the identity verification page.

After a few minutes, the account will be activated and ready to use. Select a plan as per your requirements.

Step 2: Create an IAM user and provide the required permissions

After signing in to the AWS Management Console, open the IAM console - <https://console.aws.amazon.com/iam/>. IAM stands for Identity and Access management. Next, click on **Users** in the navigation pane and choose **Add users**. Then, provide the user name for that user and choose the access type. After that, select the existing policies from the **attached existing policies** or create a new one for that user. In this scenario, Administrator access is given to the IAM user. **Tags** are optional. Finally, review and complete the process. Do not forget to save the user details like Access key ID, Secret access key, and user name. In this case, an IAM user will be used to execute all the tasks. Hit the **Create user** button and complete the process.

The following figure shows that the IAM user is created:

Add user

1 2 3 4 5

Success
You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.

Users with AWS Management Console access can sign-in at: [https://\[REDACTED\].signin.aws.amazon.com/console](https://[REDACTED].signin.aws.amazon.com/console)

Download .csv

	User	Access key ID	Secret access key	Email login instructions
▶	✓ suhas	[REDACTED] WWZ6W	***** Show	Send email

Figure 13.1: AWS IAM user

The **Secret access key** is available only when it is created, so you need to download and save it. If lost, then you have to create another one.

Creating access keys for the AWS account root user is not recommended unless required. Rather, create one or more IAM users with the required permissions to execute the tasks.

Step 3: Install and configure AWS CLI on the local machine

AWS **Command-Line Interface (CLI)** is a tool that allows the management of several AWS services. It is an open-source tool that enables you to interact with AWS services through commands. First, download and install it, then configure it with AWS credentials. To get started with AWS CLI, follow this link <https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html>.

For Linux, download the installation file using the **curl** command, where the **-o** option specifies the filename that the downloaded package is written to:

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
```

Unzip the installer as follows:

```
unzip awscliv2.zip
```

The installation command uses a file name *install* from unzipped *aws* directory:

```
sudo ./aws/install
```

Verify the installation by checking the version of AWS CLI using the following command:

```
aws --version
```

Restart the terminal if the **aws** command cannot be found.

Next, configure the AWS CLI with AWS IAM credentials, as shown in the following figure:

```
aws configure
AWS Access Key ID [None]: AKI[REDACTED]
AWS Secret Access Key [None]: CPdHawC5[REDACTED]
Default region name [None]: us-west-2
Default output format [None]:
```

Figure 13.2: AWS CLI configuration

AWS CodeCommit

In this case, the Git repository, that is, AWS CodeCommit, is used; however, other remote Git repositories like GitHub can be used. AWS CodeCommit is a private Git repository hosted on AWS. It allows you to create multiple Git repositories and supports a standard set of Git commands, such as push, pull, clone, and log. AWS CodeCommit can be added to a local Git repository as a remote. It allows collaboration and provides security; however, it is a standard practice not to store any personal or sensitive data in it. The good part is that AWS CodeCommit is available to both new and existing users for free up to a certain limit. It does not expire even after 12 months of free tier usage. It is free for the first 5 active users, which includes 1,000 repositories per account, 50 GB of storage per month, and 10,000 Git requests per month.

Go to **CodeCommit** from Services:

Services | **All Services** | **Developer Tools** | **CodeCommit**

Create a new repository in CodeCommit and then provide the repository name; the description is optional. Repository names are included in the URL of that repository.

Once the repository is created, specific permissions need to be provided; then, create Git credentials to access the **CodeCommit** repository from the local machine. Go to the IAM console, select **Users** from the navigation pane and choose the user for which CodeCommit is to be configured. Then, attach the **AWSCodeCommitPowerUser** policy from the policies list and complete the process. In this case, the IAM user has admin privileges, so there is no need to attach the above-mentioned policy to the IAM user.

Choose the same IAM user and locate HTTPS Git credentials for AWS CodeCommit. Next, select that user and click on the **Generate credentials** button. It will populate Git credentials, that is, username and password. Password can be seen and

downloaded, save it for future use. This password cannot be recovered later on; however, it can be regenerated.

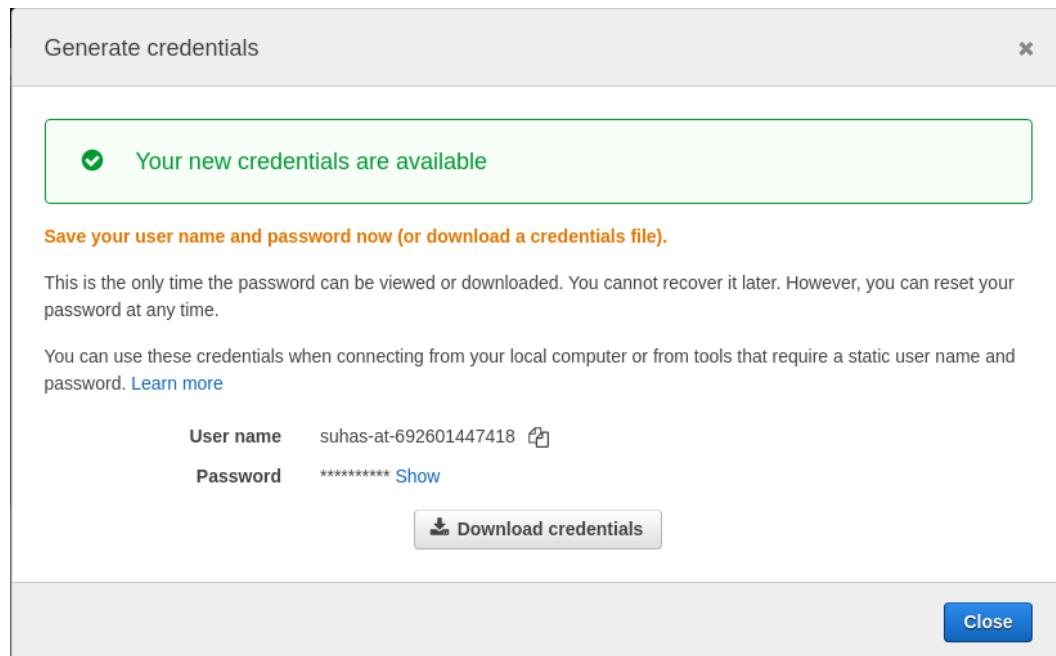


Figure 13.3: CodeCommit–Git credentials

After creating the code commit repository, clone it to the local machine. Copy project code files into the directory on the local machine. Re-initialize the Git repository using the **git init** command. Finally, commit and push changes to the CodeCommit repository from the local machine.

Let's consider the scenario of loan prediction where the problem statement is to predict whether a customer's loan will be approved. Feel free to implement hyperparameter tuning and tweak the model.

First, create an installable package of ML code, and then build a web app using FastAPI; then, create the tests and dependencies file. After that, create *Dockerfile* and *docker-compose.yml* files. Finally, create the configuration file *buildspec.yaml* for AWS deployment.

Note: For code files, refer to the previous chapters. You should create *buildspec.yaml* file and update the *tox.ini* file for the current scenario. You need to update *src\tests\test_predict.py*, *src\prediction_model\predict.py*, and *start.sh* for continuous training.

You access the code repository at <https://github.com/suhas-ds/AWS-CICD>

The following directory structure shows the code files for a CI/CD pipeline:

```
.  
├── src  
│   ├── prediction_model  
│   │   ├── config  
│   │   │   ├── __init__.py  
│   │   │   └── config.py  
│   │   ├── datasets  
│   │   │   ├── __init__.py  
│   │   │   ├── test.csv  
│   │   │   └── train.csv  
│   │   ├── processing  
│   │   │   ├── __init__.py  
│   │   │   ├── data_management.py  
│   │   │   └── preprocessors.py  
│   │   ├── trained_models  
│   │   │   ├── __init__.py  
│   │   │   └── classification_v1.pkl  
│   │   ├── VERSION  
│   │   ├── __init__.py  
│   │   ├── pipeline.py  
│   │   ├── predict.py  
│   │   └── train_pipeline.py  
│   ├── tests  
│   │   ├── pytest.ini  
│   │   └── test_predict.py  
│   ├── MANIFEST.in  
│   ├── README.md  
│   ├── requirements.txt  
│   ├── setup.py  
│   └── tox.ini  
└── .gitignore  
└── Dockerfile
```

```
|── README.md  
|── buildspec.yaml  
|── docker-compose.yml  
|── main.py  
|── pytest.ini  
|── requirements.txt  
|── runtime.txt  
|── start.sh  
|── test.py  
└── tox.ini
```

tox.ini

The tox is a free and open-source tool used for testing Python packages or applications. It creates the virtual environment and installs the required dependencies in it; finally, it runs the tests for that Python package. You need to update the *tox.ini* file as shown in the following code.

```
1. [tox]  
2. envlist = my_env  
3. skipsdist=True  
4.  
5. [testenv]  
6. install_command = pip install {opts} {packages}  
7. deps =  
8.     -r requirements.txt  
9.  
10. setenv =  
11. PYTHONPATH=src/  
12.  
13. commands=  
14.     pip install requests  
15.     pytest -v src/tests/ --junitxml=pytest_reports/Prediction_  
        test_report.xml  
16.     pytest -v test.py --junitxml=pytest_reports/API_test_report.  
        xml
```

The **skipstdist=True** flag indicates not to perform a packaging operation. It means the package will not be installed in a virtual environment before performing any test. Set **PYTHONPATH** to the **src/** directory where Python package files are placed. In the commands section, pytest commands will be executed, and the results of the tests will be exported in **.xml** files in the **pytest_reports** directory.

```
.\src\prediction_model\predict.py
```

Here, add the **train_accuracy** function to the *predict.py* file that will return the accuracy score.

```
1. def train_accuracy(input_data):  
2.     """ Checking accuracy score of training data """  
3.  
4.     # Read Data  
5.     data = pd.DataFrame(input_data)  
6.     y_train = np.where(data['Loan_Status']=='Y', 1, 0).tolist()  
7.  
8.     # Prediction  
9.     prediction = _loan_pipe.predict(data[config.FEATURES])  
10.    y_pred = prediction.tolist()  
11.    score = accuracy_score(y_train,y_pred)*100  
12.    return score
```

```
.\src\tests\test_predict.py
```

Now, add the **test_score** function to the *test_predict.py* file that will test whether the training accuracy score is between 70 to 95. You can change this range.

```
1. from prediction_model.predict import train_accuracy  
2.  
3. def test_score():  
4.     ''' This function will check the accuracy score of training data  
    '''  
5.     data = load_dataset(file_name=config.TRAIN_FILE)  
6.     score = train_accuracy(data)  
7.     assert 70 <= score <= 95
```

After a successful push from the local terminal, code files will be displayed in the cloud source repository.

Continuous Training

With a CI/CD pipeline, you can add **Continuous Training** (CT) stage. So when the CI/CD pipeline will run, it will also train the model on the latest available data as per the configuration settings.

start.sh

To train the model on the latest data, you need to add the following command to the *start.sh* file:

```
python app/src/prediction_model/train_pipeline.py
```

This will run the *train_pipeline.py* file and generate the latest pickle file of the trained model. This file will first install the package, then train the model, and finally, run the FastAPI app.

```
1. #!/bin/bash  
2.  
3. pip install app/src/  
4. python app/src/prediction_model/train_pipeline.py  
5. python app/main.py
```

Amazon Elastic Container Registry (ECR)

Amazon Elastic Container Registry (ECR) is a container registry service managed by AWS. It stores, manages, and provides security to private container images. If these container images are to be accessed through code or other services, such as **CodeBuild**, then container registry access needs to be provided to the specified service roles.

Go to Elastic Container Registry from Services:

[Services](#) | [All Services](#) | [Containers](#) | [Elastic Container Registry](#)

Create a repository with the name **mlapp-cicd**. By default, the visibility of the repository will be private; it means access will be managed by IAM and pre-defined permissions will be granted to the repository policy. The repository name should be concise yet meaningful, that is, it should be based on the content of the repository. The name must start with a letter and can only contain lowercase letters, numbers, hyphens, underscores, periods, and forward slashes.

Then, go to the repository and click on the **View push commands** button in the top-right corner. It will display the push commands and instructions you have to follow while pushing a Docker container image from a local machine using AWS CLI. First,

you need to log in to AWS ECR using the command given in the pop-up window. Next, build a Docker image on a local machine using the **docker build** command. If the Docker image is already built, the Docker build step can be skipped. Then, tag the Docker image and push the image to the AWS ECR repository using the commands given in the pop-up window.

The following figure shows the push commands for the **mlapp-cicd** repository:

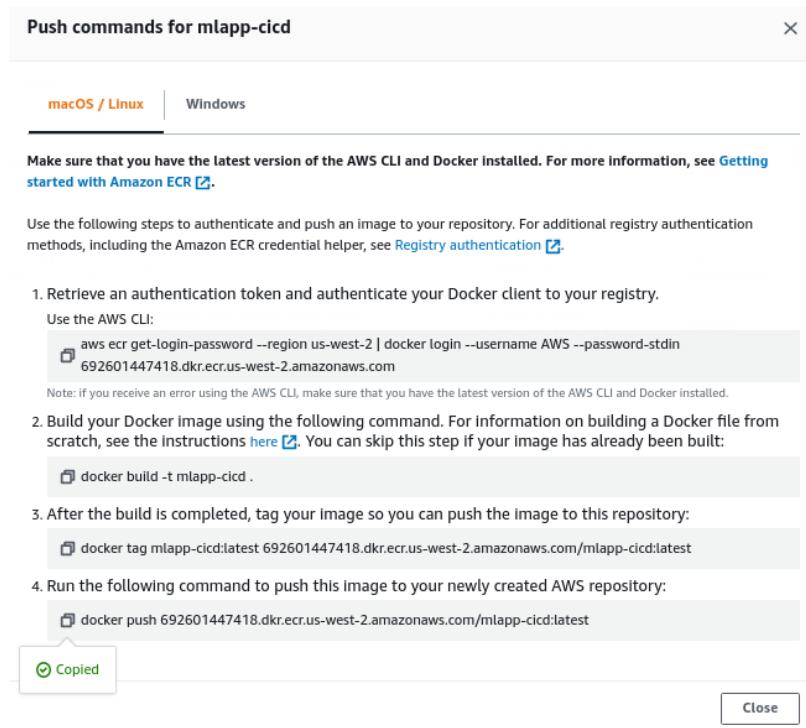


Figure 13.4: Amazon ECR – Push commands

Note: It is recommended to install and configure the latest version of AWS CLI and Docker on a local machine.

Docker Hub rate limit

From November 20, 2020, anonymous and free user are limited to 100 and 200 container image pull requests every 6 hours. After the specified limit, it will throw an error that contains **ERROR: toomanyrequests: Too Many Requests. or You have reached your pull rate limit. You may increase the limit by authenticating and upgrading: https://www.docker.com/increase-rate-limits**. To increase your pull rate limits, you can upgrade your account to a **Docker Pro or Team subscription**. Docker Pro and Docker Team accounts enable 5,000 pulls in 24 hours from Docker Hub.

To avoid this error, create a new repository on AWS ECR for the current scenario. Next, pull the base Python 3.7 image from the Docker Hub into the local machine or VM. Then, push that base Python 3.7 image to the newly created AWS ECR repository. In the current scenario, the base image is being maintained in a separate repository, as shown in the following figure:

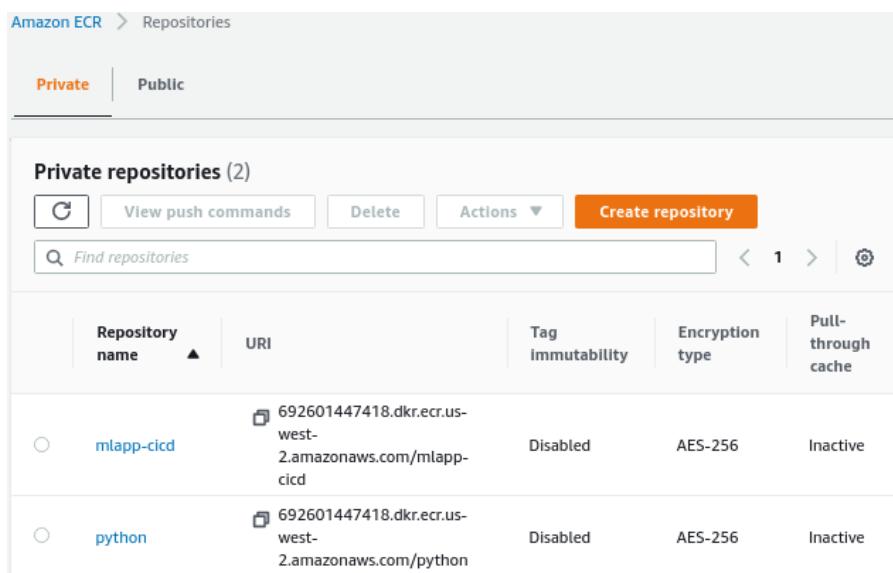


Figure 13.5: Amazon ECR–Repositories

Use this image path in the *Dockerfile* so that when the **docker build** command runs for *Dockerfile*; it will pull the base Python 3.7 images from AWS. For this, you need to update *Dockerfile* as follows:

1. `# Pull base Python:3.7 image from AWS ECR repository`
2. `ARG REPO=692601447418.dkr.ecr.us-west-2.amazonaws.com`
3.
4. `FROM ${REPO}/python:3.7`
5.
6. `COPY ./start.sh /start.sh`
7.
8. `RUN chmod +x /start.sh`
9.
10. `ENV PYTHONPATH "${PYTHONPATH}:app/src/"`
11.

```
12. COPY . /app
13.
14. RUN chmod +x /app
15.
16. # Exposing the port that uvicorn will run the app on
17. ENV PORT=8000
18. EXPOSE 8000
19.
20. RUN pip install --upgrade pip
21.
22. RUN pip install --no-cache-dir --upgrade -r app/requirements.txt
23.
24. CMD ["./start.sh"]
```

In the current scenario, the **python:3.7-slim-buster** image is built and tagged with 3.7, and then it is pushed to AWS ECR for reusability. This way, every time, the **python:3.7-slim-buster** image will be pulled from AWS ECR instead of the Docker Hub to avoid any rate limit error.

AWS CodeBuild

A **CodeBuild** is a fully managed **Continuous Integration (CI)** service backed by AWS infrastructure that allows developers to automate the build, test, and deploy containers or packages quickly. It works on an on-demand or pay-as-you-go model, that is, it charges users based on the minute for the compute resources they have used.

The **CodeBuild** config file type is **YAML**. This file contains a series of phases and commands to execute the build specified by the developer.

buildspec.yaml

In this case, the *buildspec.yaml* file first installs the tox package in the **install** phase. Next, it will run the tests using tox and log in to AWS ECR in the **pre_build** phase. Then, the *buildspec.yaml* file will build a Docker container image with the latest tag in the **build** phase and push the image to the AWS ECR repository in the **post_build** phase. Finally, specify the details for pytest reports, such as files, a base directory, and file format. Also, provide the filename to be stored as an artifact in the S3 bucket.

```
1. version: 0.2
2.
3. phases:
4.   install:
5.     runtime-versions:
6.       python: 3.7
7.     commands:
8.       - pip install tox
9.   pre_build:
10.    commands:
11.      - echo Running test...
12.      - tox
13.      - echo Logging into Amazon ECR...
14.      - aws --version
15.      - aws ecr get-login-password --region us-west-2 | docker
login --username AWS --password-stdin 692601447418.dkr.ecr.us-
west-2.amazonaws.com
16.      - REPOSITORY_URI=692601447418.dkr.ecr.us-west-2.amazonaws.
com/mlapp-cicd
17.      - COMMIT_HASH=$(echo $CODEBUILD_RESOLVED_SOURCE_VERSION |
cut -c 1-7)
18.      - IMAGE_TAG=${COMMIT_HASH:=latest}
19.   build:
20.     commands:
21.       - echo Build started on `date`
22.       - echo Building the Docker image...
23.       - docker build -t $REPOSITORY_URI:latest .
24.       - docker tag $REPOSITORY_URI:latest $REPOSITORY_URI:$IMAGE_
TAG
25.   post_build:
26.     commands:
27.       - echo Build completed on `date`
28.       - echo Pushing the Docker images...
```

```
29.      - docker push $REPOSITORY_URI:latest
30.      - docker push $REPOSITORY_URI:$IMAGE_TAG
31.      - echo Writing container image definitions file...
32.      - printf '[{"name":"mlapp-cicd","imageUri":"%s"}]' 
$REPOSITORY_URI:$IMAGE_TAG > imagedefinitions.json
33. reports:
34.   pytest_reports:
35.     files:
36.       - Prediction_test_report.xml
37.       - API_test_report.xml
38.     base-directory: pytest_reports/
39.   file-format: JUNITXML
40. artifacts:
41.   files: imagedefinitions.json
```

In the preceding file, mainly three supported builder images are used.

You can see, **python:3.7** is the publicly available image used for testing the code using tox. If any image is to be used from the Docker Hub, then simply provide the image name in single quotes. However, if an image is from other registries, then the full registry path needs to be specified in single quotes. The **args** field of a build phase accepts a list of arguments and passes them to the image referenced by the name field.

To create a build, **CodeBuild** is used. Search and select **CodeBuild** from developer tools and choose **Build projects** from the **Build** section. Click on the **Create build project** button to create a new one. First off, under the **Project configuration**, provide the project name.

In this case, the project name **mlapp-cicd** is provided, as shown in the following figure:

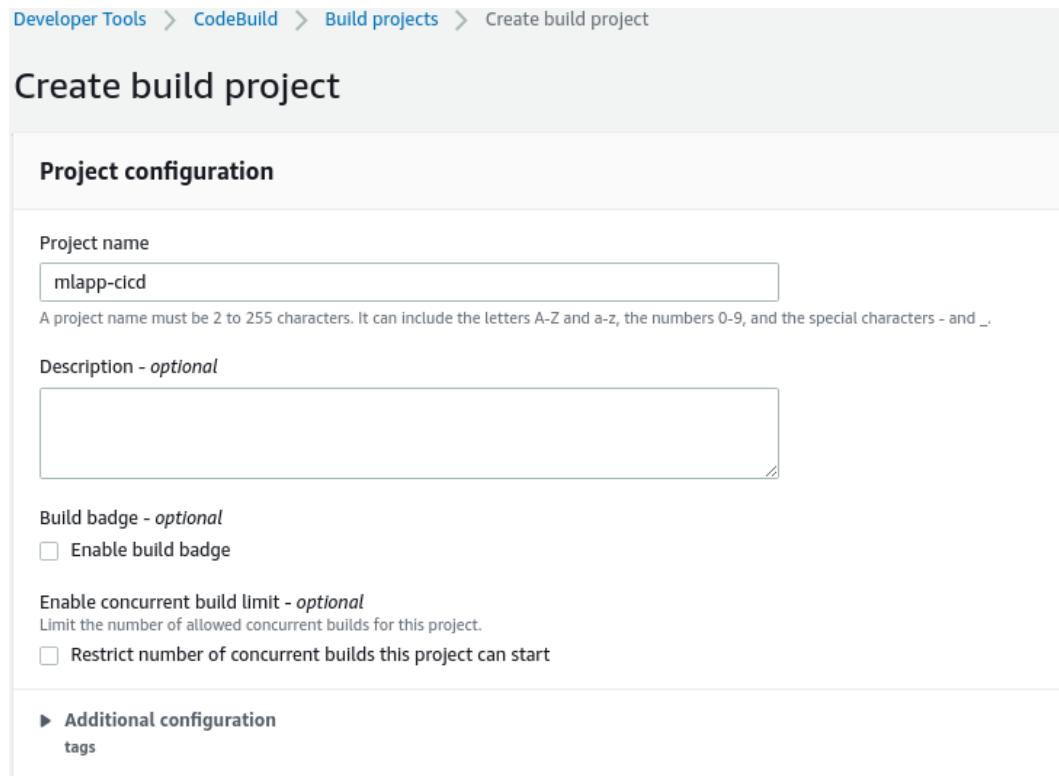


Figure 13.6: CodeBuild—Project configuration

Next, under the **Source** section, select the **Source provider** from the dropdown list and choose the subsequent details, such as **Repository** and **Branch**. Source providers contain details and code files to be used as input source code for the build project. In the current scenario, choose **AWS CodeCommit** as a **Source provider**, as shown in the following figure. However, other source providers can be selected instead of **AWS CodeCommit**, such as BitBucket, GitHub, and S3.

Source

Add source

Source 1 - Primary

Source provider

AWS CodeCommit

Repository

loan_pred

Reference type

Choose the source version reference type that contains your source code.

Branch

Git tag

Commit ID

Branch

Choose a branch that contains the code to build.

main

Commit ID - optional

Choose a commit ID. This can shorten the duration of your build.

Source version [Info](#)

refs/heads/main

[312d4a49](#) Update

► Additional configuration
Git clone depth, Git submodules

Figure 13.7: CodeBuild–Source

Then, under the **Environment** section, choose the **Managed Image** option and choose the operating system as **Ubuntu** from the dropdown. It provides other operating systems such as Windows Server and Amazon Linux. In the current scenario, **aws/CodeBuild/standard:4.0** image is chosen. The remaining selections can be kept as default. Make sure you select the **Privileged** checkbox, which allows you to build Docker images.

The following figure shows the selection made for the **Environment** section:

The screenshot shows the 'Environment' configuration section of the AWS CodeBuild console. It includes fields for 'Environment image' (set to 'Managed image'), 'Operating system' (set to 'Ubuntu'), 'Runtime(s)' (set to 'Standard'), 'Image' (set to 'aws/codebuild/standard:4.0'), 'Image version' (set to 'Always use the latest image for this runtime version'), 'Environment type' (set to 'Linux'), and a 'Privileged' checkbox which is checked. A note in the middle of the form states: 'The programming language runtimes are now included in the standard image of Ubuntu 18.04, which is recommended for new CodeBuild projects created in the console. See Docker Images Provided by CodeBuild for details.'.

Figure 13.8: CodeBuild–Environment

After that, under the **Buildspec** section, choose **Use a buildspec file**, as shown in the following figure:

The screenshot shows the 'Buildspec' configuration section of the AWS CodeBuild console. It includes fields for 'Build specifications' (set to 'Use a buildspec file') and a 'Buildspec name - optional' field which is empty. A note below the field states: 'By default, CodeBuild looks for a file named buildspec.yml in the source code root directory. If your buildspec file uses a different name or location, enter its path from the source root here (for example, buildspec-two.yml or configuration/buildspec.yml)'.

Figure 13.9: CodeBuild–Buildspec

Finally, under the **Logs** section, select **Cloudwatch logs**. This will upload build output logs to cloudwatch. This enables you to analyze the logs and output generated after building the project. However, this is optional. At the bottom of the page, click on the **Create build project** button and complete the configuration process.

Attach container registry access to CodeBuild's service role

Now, you need to provide access to the service role of CodeBuild so that it can build the Docker container images. For this, go to **IAM management console | Roles** and select the CodeBuild service role. In the current scenario, it is **CodeBuild-mlapp-cicd-service-role**. Click on it, and the summary page will open. Click on **Attach policies**, and then search for **EC2ContainerRegistry** and select **AmazonEC2ContainerRegistryFullAccess**. It provides administrative access to Amazon ECR resources. If the required access is not provided, CodeBuild will be unable to build Docker images.

The following figure shows container registry access allowed for CodeBuild's service role:



The screenshot shows the AWS IAM Permissions tab for the 'CodeBuild-Service role'. The 'Permissions' tab is selected. There are two managed policies attached: 'CodeBuildBasePolicy-mlapp-cicd-us-west-2' (Customer managed) and 'AmazonEC2ContainerRegistryFullAccess' (AWS managed). Both policies are listed with their respective descriptions.

	Policy name	Type	Description
<input type="checkbox"/>	CodeBuildBasePolicy-mlapp-cicd-us-west-2	Customer managed	Policy used in trust relationship with CodeBuild
<input type="checkbox"/>	AmazonEC2ContainerRegistryFullAccess	AWS managed	Provides administrative access to Amazon ECR resources

Figure 13.10: CodeBuild–Service role

Now, go to the main page of CodeBuild, and let's manually execute the build phase by hitting the **Start build** button in the top-right corner. Build status will be displayed as **In Progress**. Logs are available under the **Build logs** tab. By default, it will show the last 1000 lines of the build log. After completion of the build phase, the latest container image should be available in Amazon Elastic Container Registry

(ECR). Also, the test report should be available under the **Reports** tab, as shown in the following figure:

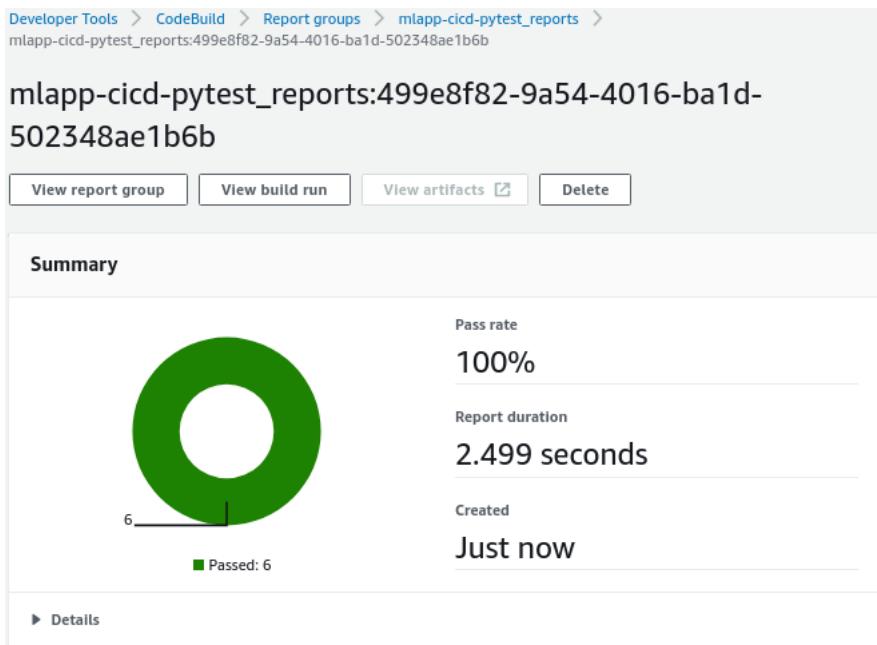


Figure 13.11: CodeBuild—Test report

Amazon Elastic Container Service (ECS)

Amazon Elastic Container Service (ECS) is a container orchestration tool used for managing and running Docker containers. Amazon Elastic Container Service (ECS) is a fully managed container management service offered by AWS. In the current scenario, ECS with the Fargate model will be used. It will take care of managing the cluster and load balancing. It will also make sure the application is up and running.

Let's understand the terms used in the Amazon Elastic Container Service (ECS):

- **Task Definition:** The task definition allows you to specify which Docker image to use, which ports to expose, how much CPU and memory to allot, how to collect logs, and how to define environment variables. This is a blueprint that dictates how a Docker container should launch.
- **Task:** A task resembles an **instance** of task definition. It can be created independently, that is, without an Amazon ECS cluster. This will spin up a container with an application running in it. It will not replace itself automatically if it stops or fails due to some error.

- **Service:** A Service is responsible for creating and maintaining the desired number of tasks up and running all the time. If any task fails or is stopped due to some error, then the ECS Service will replace that task with a new one. It refers to the **task definition** file to create tasks.
- **Cluster:** It is a logical group of container instances.
- **Container:** This is the Docker container created during task instantiation.

The following figure resembles the Amazon Elastic Container Service (ECS) architecture.

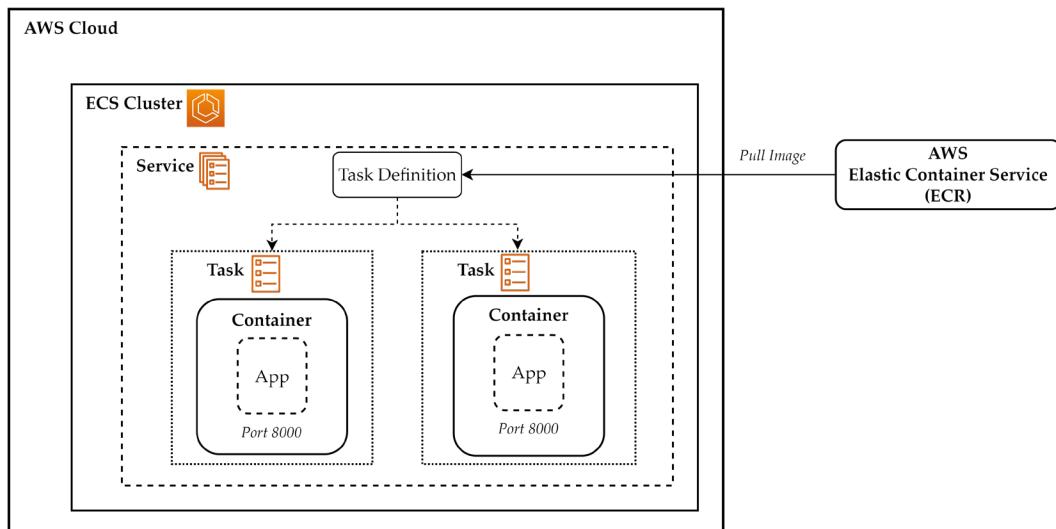


Figure 13.12: Amazon Elastic Container Service (ECS)–Architecture

AWS ECS deployment models

AWS ECS Deployment model can be chosen as per your requirement. Let's look at the models AWS Elastic Container Service (ECS) mainly offers for cluster deployment.

EC2 instance

EC2 (Elastic Compute Cloud) is a virtual machine in the cloud. First, configure and deploy EC2 instances in the cluster to run the containers. It provides more granular control over the instances. You can choose the instances as per requirement.

This model is a better choice if you want to do the following:

- Run containerized applications continuously

- Have better control over the auto-scaling configuration
- Use a Classic Load Balancer (CLB) to distribute workloads
- Use Graphical Processing Unit (GPU)
- Use Elastic Block Storage (EBS)
- Deploy large and complex applications

Fargate

This is a serverless pay-as-you-go model. You will be charged based on the computing capacity selected and the time of usage. ECS with Fargate is a container orchestration tool used for running Docker based containers without having to manage the underlying infrastructure.

This model is a better choice if you want to do the following:

- Run the task occasionally or for a short period
- Containerized applications should be able to handle sudden spikes in incoming traffic
- Use application and network load balancers to distribute workloads
- Save time from different configurations, regular maintenance, and security management

Note: In this chapter, classic UI is used. By default, you might see a new UI; however, you can switch back to the classic UI.

Go to Elastic Container Service from Services:

Services | All Services | Containers | Elastic Container Service

First off, go to the cluster page and create an ECS cluster by clicking on the **Create Cluster** button. AWS provides templates for creating clusters to simplify the process of cluster creation. In the current scenario, **Networking only (AWS Fargate)** is selected as the type of instance. On the next screen, under **Cluster configurations**, the **Cluster name** is to be provided. In this scenario, it is **mlapp-cluster**, as shown in the following figure:

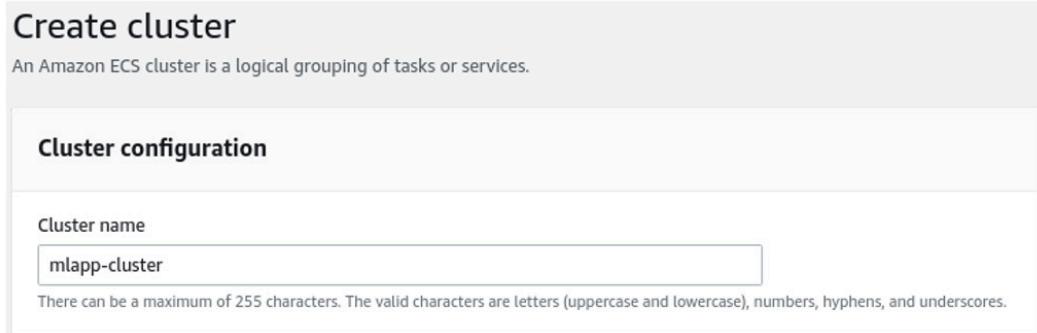


Figure 13.13: Amazon Elastic Container Service (ECS)–Cluster configuration

Then, click on the **Create** button, and it will launch the ECS cluster. You will see ECS cluster is created with the type Fargate, but no instances are running.

Task definition

After that, create a task definition. Go to the **Task Definition** page and click on **Create new Task Definition**. Choose the type as **Fargate** in step 1. In step 2, provide the task definition name. In this case, it is **mlapp-cicd**. Select the **Task role ecsTaskExecutionRole** from the dropdown and select the **Operating system family** as **Linux**, as shown in the following figure:

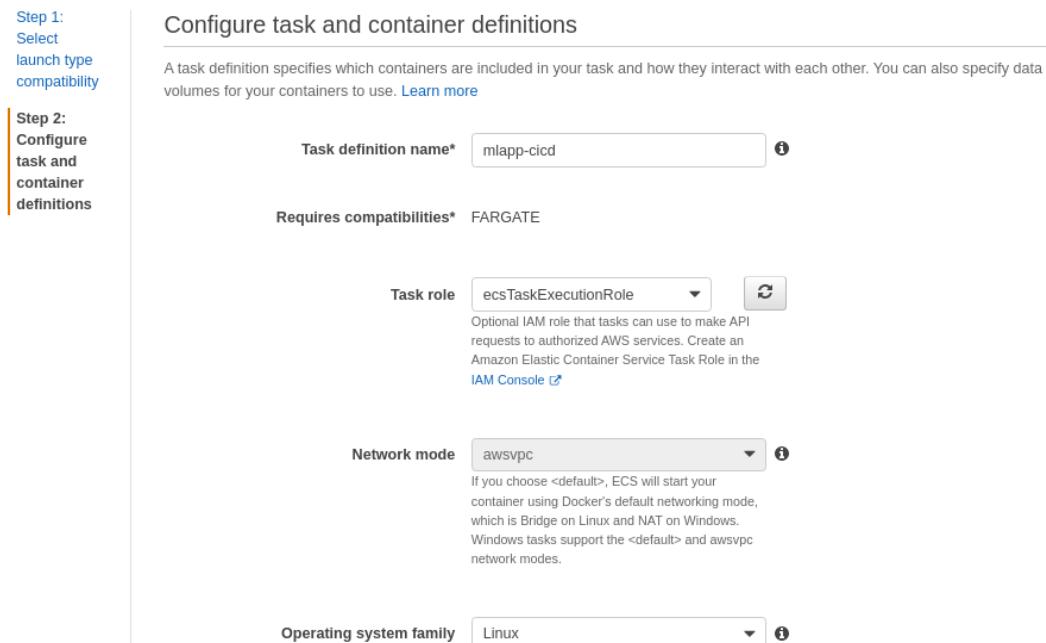


Figure 13.14: Amazon Elastic Container Service (ECS) –task definition

Keep the **Task execution role** as default, that is, **ecsTaskExecutionRole**. Under the **Task size** section, choose the required memory and CPU to be allotted for tasks based on application requirements or use cases. A **Task memory (GB)** of **1GB** and **Task CPU (vCPU)** of **0.25 vCPU** is chosen for the current scenario, as shown in the following figure:

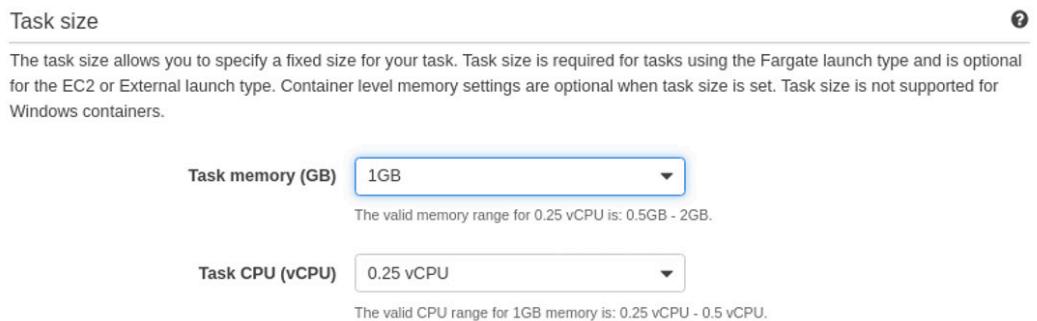


Figure 13.15: Task definition–Task Size

Finally, add the container details, such as container name, image URI to be taken from AWS ECR, and memory limits (in MiB) for containers. Hard and soft limits correspond to the **memory** and **memoryReservation** parameters in task definitions. The **Port mappings** parameter is important. Here, you need to provide a port used by the container to run the application and expose it in the *Dockerfile*. In the current scenario, the container port is **8000**. You can leave the rest of the configurations as they are and add a container.

In the end, click on **Create** and complete step 2 of the task definition.

Running task with the task definition

The task can be run independently using the task definition created earlier.

The following figure shows the various options available for task definition. Now, to run the task, choose the first option, that is, **Run Task**.

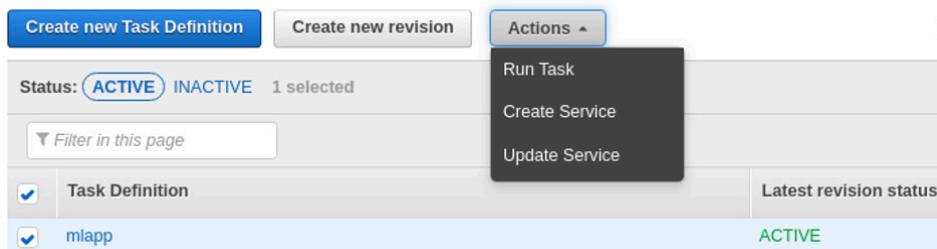


Figure 13.16: Running an independent task

Next, choose the launch type. In the current scenario, **FARGATE** is chosen. Choose **mlapp-cluster** from the **Cluster** dropdown.

Amazon VPC and subnets

Amazon Virtual Private Cloud (Amazon VPC) enables you to launch AWS resources into a virtual network that you've defined. These virtual networks are similar to the traditional networks used in data centers. The merit of a virtual network is that it comes with scalable infrastructure managed by AWS. Each VPC network consists of one or more IP address ranges called subnets.

Next, under **VPC and security groups**, choose cluster VPC and choose subnets. In the current scenario, choose **2a** and **2b**; however, the remaining **2c** and **2d** can also be chosen. Make sure **the Auto-assign public IP is ENABLED**. This will allow you to automatically assign available public IPs to access the ML app from anywhere.

Then, edit the default security group, and the security groups window will show up. Here, create a new security group that will be used in a later stage. Choose **Create new security group** from **Assigned security groups** and provide the name for the security group as **mlapp-sg**, where **sg** stands for the security group. After that, two ports are allowed, that is, port **80** and port **8000**, for the communication of containerized applications with the external world.

After that, add **HTTP port 80** and **Custom TCP port 8000**, on which the containerized application is running. Choose **Anywhere** in the source option for both ports.

Finally, in the bottom-right corner, hit the **Run Task** button. It will take some time to get the task up and running. The task's status will be shown as **RUNNING** in the **Tasks** tab, as shown in the following figure:

The screenshot shows the AWS Lambda console interface. The top navigation bar includes 'Lambda', 'Actions', 'Logs', 'Metrics', 'Functions', 'Events', 'Tasks', and 'APIs'. Below the navigation, there are sections for 'Network' and 'Containers'.

Network section details:

- Network mode: awsvpc
- ENI Id: eni-054be77aeef109267
- Subnet Id: subnet-0e03a48656136de99
- Private IP: 172.31.35.132
- Public IP: 54.191.99.95
- Mac address: 06:53:09:bd:eb:7f

Containers section details:

- Last updated on August 26, 2022 4:52:46 AM (2m ago)
- Task ARN: arn:aws:lambda:us-east-1:9ba7781.../mlapp-cicd
- Table headers: Name, Container Runtime ID ..., Status, Image, Image Digest, CPU Uni..., Hard/Sof..., Essential..., Resource...
- Table data row: mlapp-cicd, bd33c48a2336431d913..., RUNNING, 6926014474..., sha256:13d2e5e333..., --, --/--, true, 9ba7781...

Figure 13.17: Running an independent task—task created

The public IP of the task is **54.191.99.95**. It means an ML app can be accessed with this public IP from anywhere on the internet.

The issue with running a task independently is that if the task fails or stops due to any reason, then the application also stops; this makes the application inaccessible to the external world. To overcome this issue, ECS service can be used. Don't forget to stop the running task. With ECS service, multiple tasks can be created.

Load balancing

In parallel computing, load balancing is a process of distributing application traffic across different available computes or resources in order to make overall request handling more efficient. The main objective of the technique is to avoid any downtime for end users or customers.

The following figure resembles the high-level architecture of an **Application Load Balancer (ALB)**:

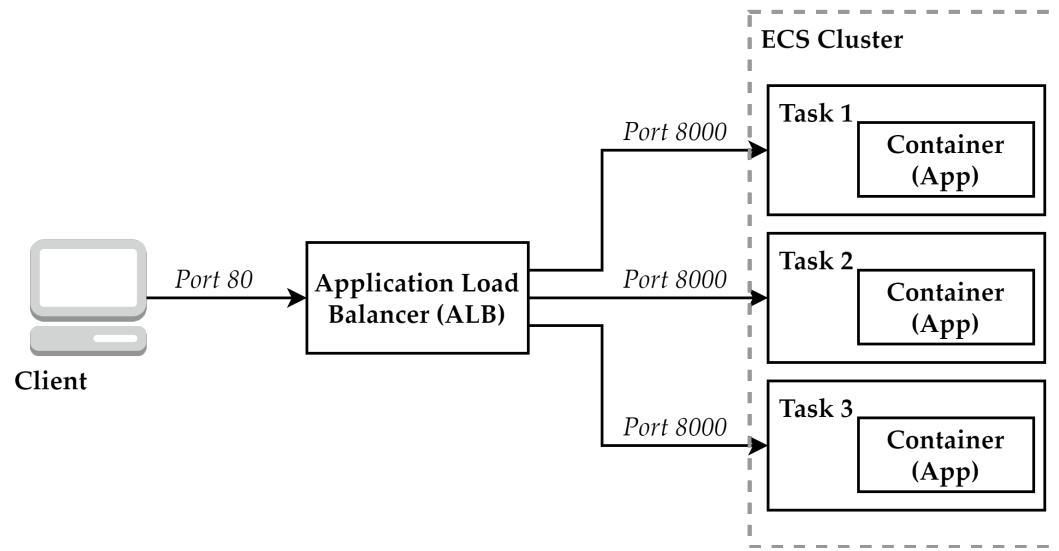


Figure 13.18: Application Load Balancer (ALB)

An application load balancer can only be added to a service while creating a service. Next, go to the load balancing part and create the target groups, which takes care of dynamically adding the IP addresses that get created while creating the tasks.

Go to load balancer from Services:

[Services](#) | [All Services](#) | [Developer Tools](#) | [Load balancing](#)

The following figure shows the components of Load Balancing:

▼ Load Balancing

Load Balancers

Target Groups New

Figure 13.19: AWS-Load Balancing

Let's understand the role and configuration of the two components of Load Balancing.

Target group

With a load balancer, a single DNS name is enough to access the containerized application. A load balancer will take the user requests on port **80** and route them to tasks with dynamic IP through the service. The target group will ensure connectivity between the load balancer and tasks with dynamic IP through service.

Go to the target group and create a new target group by hitting the **Create target group** button.

Creating a target is a three-step process:

- Specify group details
- Register targets
- Review IP targets to include in your group

Step 1: Specify group details

A configuration setting cannot be changed after the creation of a target group. In this step, the target type is **IP addresses**; its features are listed in the following figure:

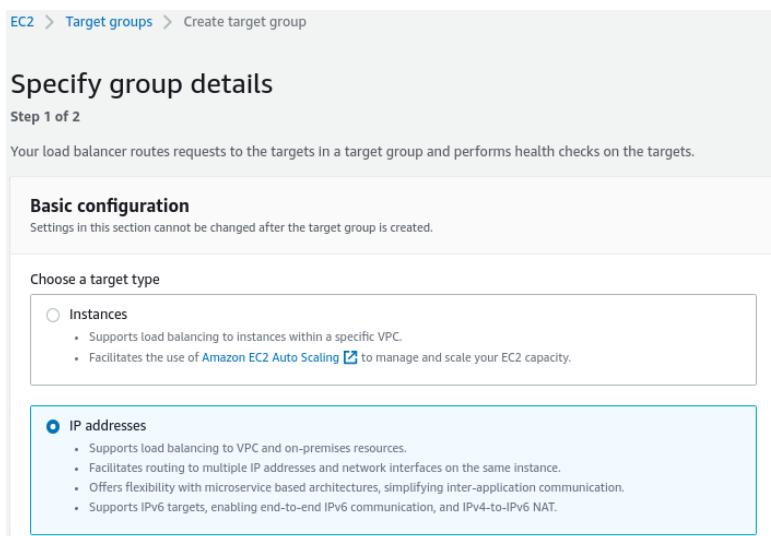


Figure 13.20: Target Groups—Specify group details

Next, provide the **Target group name** as **mlapp-tg**. The Target group name should not start or end with a hyphen (-). Then, provide port **8000** with **HTTP** type **Protocol** as the application is exposed to port **8000**. Leave the rest of the selections as they are. This configuration is shown in the following figure:

Target group name
mlapp-tg
A maximum of 32 alphanumeric characters including hyphens are allowed, but the name must not begin or end with a hyphen.

Protocol	Port
HTTP	8000

IP address type
Only targets with the indicated IP address type can be included in this target group.
 IPv4
 IPv6

VPC
Select the VPC that hosts the load balancer. Only VPCs that support the IP address type selected above are available in this list. On the Register targets page, you can register IP addresses from this VPC, or from private IP addresses located outside of this load balancer's VPC (such as a peered VPC, EC2-Classic, or on-premises targets that are reachable over Direct Connect or VPN).
 -
 vpc-0794b53fdc52faed4
 IPv4: 172.31.0.0/16

Protocol version
 HTTP1
Send requests to targets using HTTP/1.1. Supported when the request protocol is HTTP/1.1 or HTTP/2.

Figure 13.21: Target Groups–Basic configuration

Under the health check section, by default, the target group will check the root or index path ('/'), or you can specify a custom path if preferred. The target group will periodically send the requests to this endpoint of the application to check whether it is healthy.

Step 2: Register targets

In this step, IP addresses and ports can be specified manually from the selected network. By default, the VPC network will be selected. Remove the pre-selected IP, as this part is optional. This can be configured later on. The following figure shows this step:

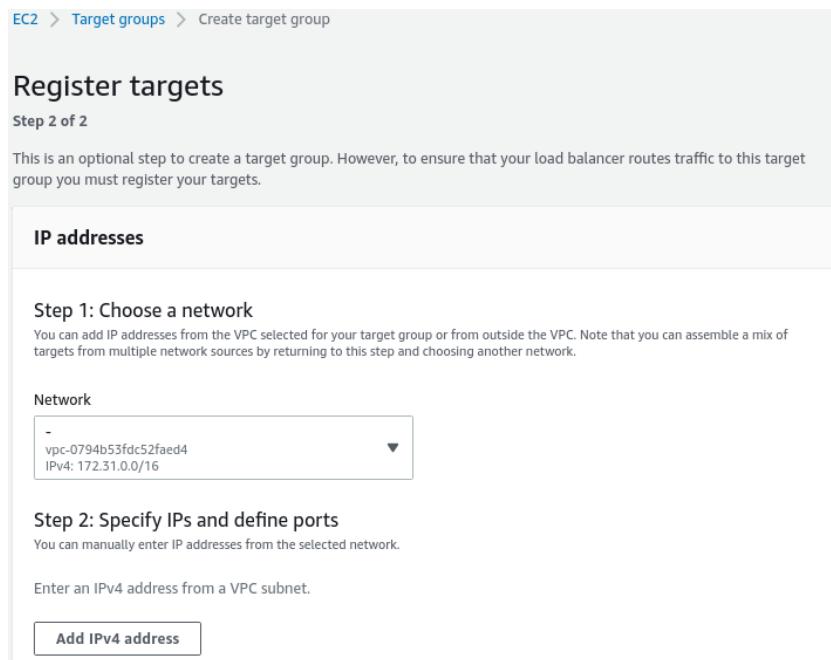


Figure 13.22: Target Groups–Register targets

After that, specify the ports for routing to this target. In the current case, the application's port **8000** is specified, as shown in the following figure:

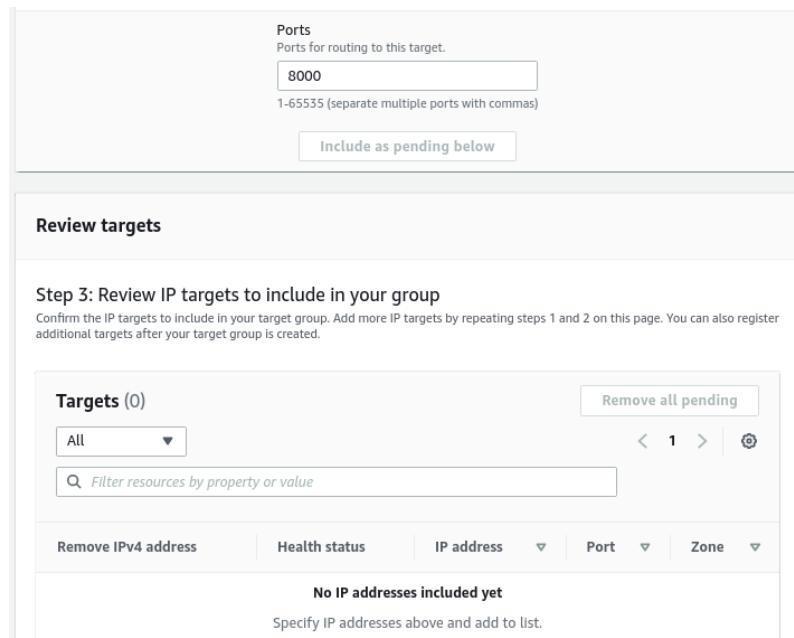


Figure 13.23: Target Groups – Review targets and ports for routing to target

Step 3: Review IP targets to include in your group

At this point, you can repeat steps 1 and 2 if you want to add additional IP targets. However, you have not specified any IP address is specified in step 2, so go ahead and hit the **Create target group** button at the bottom. It will take you back to the main page of the targets group and display a successful message, as shown in the following figure:

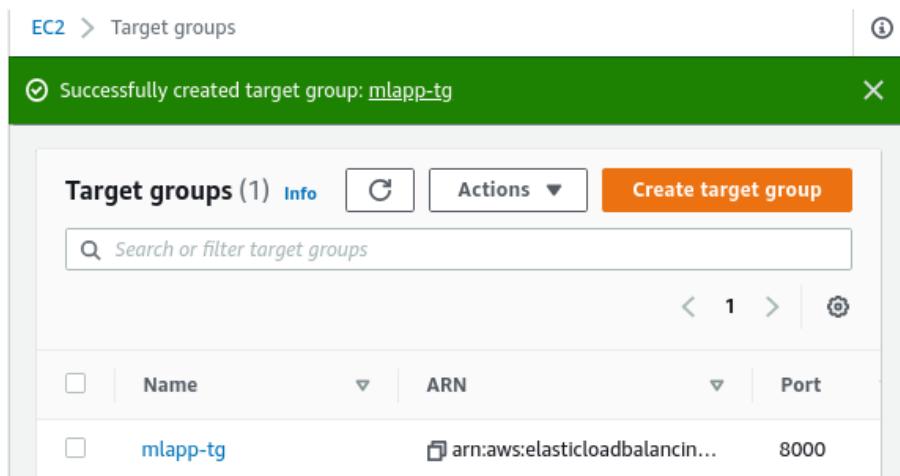


Figure 13.24: Target Groups—mlapp-tg

Amazon Resource Names (ARNs) are unique identifiers of AWS resources. AWS ARN format could be any one of the following formats:

- arn:partition:service:region:account-id:resource-id
- arn:partition:service:region:account-id:resource-type / resource-id
- arn:partition:service:region:account-id:resource-type:resource-id

A partition is a group of AWS regions, and each account has a limited scope: one partition.

Security Groups

A security group is a set of firewall rules that control the traffic toward the load balancer. It has not been created yet, so let's go ahead and create a new security group. Follow these steps:

[Services](#) | [All Services](#) | [EC2](#) | [Network & Security](#) | [Security Groups](#)

First off, specify basic details for the security group, such as name and description, as shown in the following figure. A **Security group name** is **mlapp-1b-sg**, where **1b** stands for the load balancer and **sg** stands for the security group.

The screenshot shows the 'Create security group' wizard. In the 'Basic details' section, the 'Security group name' is set to 'mlapp-lb-sg'. The 'Description' is 'Allows end users to access ml app through load balancer'. Under 'VPC', a VPC ID 'vpc-0794b53fdc52faed4' is selected. A note below the name field states: 'Name cannot be edited after creation.'

Figure 13.25: Security Group for load balancer—Create a security group

Next, configure inbound rules for the security group. Hit the **Add rule** button. A load balancer should be able to access port **80** from anywhere, so specify port **80** under the **Port range** and choose **Source type** as anywhere, as shown in the following figure. More rules can be added through the **Add rule** button.

The screenshot shows the 'Inbound rules' configuration page. It displays one rule named 'Inbound rule 1'. The rule details are: Type: Custom TCP, Protocol: TCP, Port range: 80, Source type: Anywhere-IPv4, and Description - optional: (empty). A 'Delete' button is visible next to the rule. At the bottom left is an 'Add rule' button.

Figure 13.26: Security Group for load balancer—Inbound rules

Then, hit the **Create security group** button in the bottom-right corner of the page and complete the process. The following figure shows the details of the security group after creation:

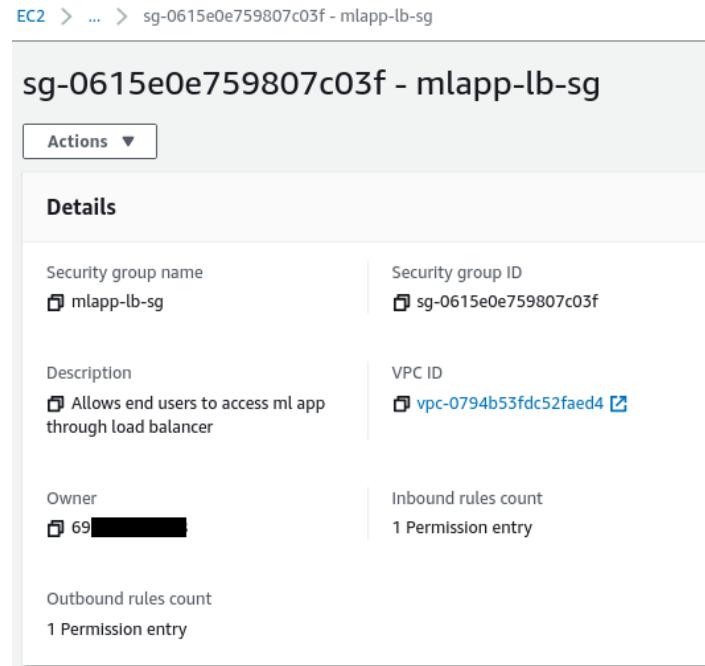


Figure 13.27: Security Group for load balancer–mlapp-lb-sg

This security group, **mlapp-lb-sg** will be used for the load balancer.

Application Load Balancers (ALB)

Application Load Balancer (ALB) accepts incoming requests and routes them to registered targets, such as EC2 instances. It also checks the health of its registered targets (by default, the root path is set to '/'; however, it can be changed to a different path of application). Application Load Balancer (ALB) is client-facing. It routes the traffic to healthy targets only. It can be easily integrated with Amazon Elastic Container Service (Amazon ECS), Amazon Elastic Container Service for Kubernetes (Amazon EKS), AWS Fargate, and AWS Lambda.

Once security and target group is created, create a load balancer. Go to the **Load balancing** section again and choose **Load Balancers**.

Services | All Services | EC2 | Load balancing | Load Balancers

Then, hit the **Create Load Balancer** button. The following figure shows the main page of the load balancer:

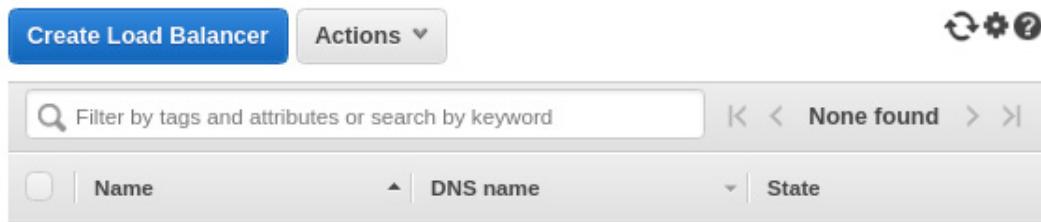


Figure 13.28: Application Load Balancer—Create Load Balancer

After clicking the **Create Load Balancer** button, it will show three options with brief descriptions and architecture to choose from, as shown in the following figure:

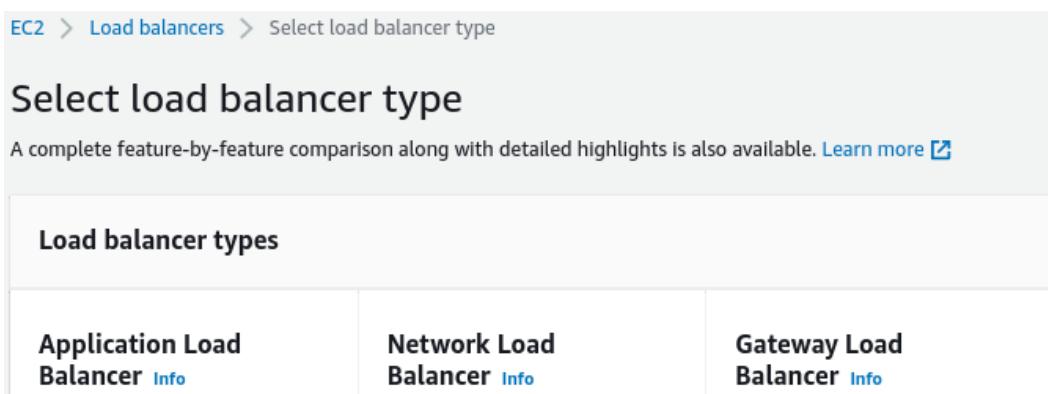


Figure 13.29: Application Load Balancer—Select load balancer type

The application load balancer distributes the incoming requests across targets, such as containers, microservices, and EC2 instances, depending on the requests. Before passing the incoming requests to the targets, it evaluates whether the listeners' rules are configured.

In the current scenario, choose the first type, which is **Application Load Balancer**, by hitting the **Create** button.

Now, configure the load balancer. First, specify the name for the load balancer **mlapp-1b**, where **1b** stands for the load balancer. Next, choose the scheme as **Internet-facing**. The main difference between the **Internet-facing** and the **Internal** scheme is the internet. An **Internet-facing** load balancer routes the requests coming from clients over the internet to specified targets. On the other hand, an **Internal** load balancer will only route the request from clients with private IPs to specified

targets. Select the IP address type as **IPv4**, which is used by specified subnets. The following figure shows the basic configuration of the **Application Load Balancer**:

EC2 > Load balancers > Create Application Load Balancer

Create Application Load Balancer Info

The Application Load Balancer distributes incoming HTTP and HTTPS traffic across multiple targets such as Amazon EC2 instances, microservices, and containers, based on request attributes. When the load balancer receives a connection request, it evaluates the listener rules in priority order to determine which rule to apply, and if applicable, it selects a target from the target group for the rule action.

► How Application Load Balancers work

Basic configuration

Load balancer name

Name must be unique within your AWS account and cannot be changed after the load balancer is created.

mlapp-lb

A maximum of 32 alphanumeric characters including hyphens are allowed, but the name must not begin or end with a hyphen.

Scheme Info

Scheme cannot be changed after the load balancer is created.

Internet-facing

An internet-facing load balancer routes requests from clients over the internet to targets.

Requires a public subnet. [Learn more](#) 

Internal

An internal load balancer routes requests from clients to targets using private IP addresses.

Figure 13.30: Application Load Balancer–Basic configuration

Then, configure the network mapping. Network mapping enables the load balancer to route incoming requests to specific subnets and IP addresses. For mapping, select at least two availability zones and one subnet per zone where the load balancer can route the traffic. By default, it will display the available zones for selection. A load balancer will route the requests to targets in these availability zones only. Choose the default VPC and two **Subnet (2a and 2b)** from the dropdown, as shown in the following figure:

Network mapping Info

The load balancer routes traffic to targets in the selected subnets, and in accordance with your IP address settings.

VPC Info

Select the virtual private cloud (VPC) for your targets. Only VPCs with an internet gateway are enabled for selection. The selected VPC cannot be changed after the load balancer is created. To confirm the VPC for your targets, view your [target groups](#)



- vpc-0794b53fdc52faed4
IPv4: 172.31.0.0/16



Mappings Info

Select at least two Availability Zones and one subnet per zone. The load balancer routes traffic to targets in these Availability Zones only. Availability Zones that are not supported by the load balancer or the VPC are not available for selection.

us-west-2a

Subnet

subnet-08c67a67021afa335



IPv4 settings

Assigned by AWS

us-west-2b

Figure 13.31: Application Load Balancer–Network mapping

After that, remove the default security group and choose the security group from the dropdown created in the previous section. A security group can also be created through the **Create new security group** link. It will take you to the main page of the security group only, as seen in the previous section. The following figure shows the security group **mlapp-1b-sg** chosen for the load balancer:

Security groups Info

A security group is a set of firewall rules that control the traffic to your load balancer.

Security groups

Select up to 5 security groups



[Create new security group](#)

mlapp-lb-sg sg-0615e0e759807c03f

VPC: vpc-0794b53fdc52faed4

Figure 13.32: Application Load Balancer–Security groups

Finally, choose the target group you created earlier, that is, **mlapp-tg** with port **80**, as shown in the following figure:

The screenshot shows the 'Listeners and routing' section of the AWS Application Load Balancer configuration. A single listener named 'HTTP:80' is listed. It has the following settings:

- Protocol:** HTTP
- Port:** 80
- Default action:** Forward to target group 'mlapp-tg' (Target type: IP, IPv4)
- Actions:** Remove, Create target group

At the bottom left, there is a 'Add listener' button.

Figure 13.33: Application Load Balancer–Listeners and routing

Scroll down and hit the **Create load balancer** button in the bottom-right corner. The following figure shows the main page of the load balancer, which shows the newly created load balancer is created:

The screenshot shows the main page of the AWS Application Load Balancer. The search bar at the top contains 'mlapp-lb'. The main table displays one item:

Name	DNS name	State
mlapp-lb	mlapp-lb-700497658.us-wes...	Active

Below the table, there are tabs for 'Description', 'Listeners', 'Monitoring', 'Integrated services', and 'Tags'. Under 'Basic Configuration', the following details are shown:

- Name:** mlapp-lb
- ARN:** arn:aws:elasticloadbalancing:us-west-2:692601447418:loadbalancer/app/mlapp-lb/88eb44bdd1c0f776
- DNS name:** mlapp-lb-700497658.us-west-2.elb.amazonaws.com (A Record)
- State:** Active

Figure 13.34: Application Load Balancer–mlapp-lb

Now, first, delete the service if you created it earlier, as the load balancer setting cannot be added after the creation of the service. Next, create a service with a load balancer.

Service

An AWS ECS service allows you to keep running a specified number of instances of a task definition simultaneously in an AWS ECS cluster. If the task(s) fails or stops due to any reason, the ECS service will launch another instance of task definition to ensure that the desired number of tasks are up and running. The benefit of this is that the application will be up and running despite the failure of any task.

Go to the **Services** tab on the **mlapp-cluster** page and hit the **Create** button to create a new service.

Service creation involves four steps, as shown in the following figure. The configuration and setting for each step are discussed in the following figure:

Create Service

| Step 1: Configure service

Step 2: Configure network

Step 3: Set Auto Scaling
(optional)

Step 4: Review

Figure 13.35: ECS Service creation steps

Step 1: Configure service

In this step, select the launch type. As the ECS cluster is configured with FARGATE, choose the launch type as **FARGATE**. The operating system family will be **Linux**.

Choose **Task Definition** as **mlapp** and **Cluster** as **mlapp-cluster** from the dropdown as shown in the following figure:

The screenshot shows the 'Configure service' section of the AWS ECS console. It includes the following fields:

- Launch type:** FARGATE (selected)
- Operating system family:** Linux
- Task Definition:**
 - Family: mlapp
 - Revision: 1 (latest)
- Platform version:** LATEST
- Cluster:** mlapp-cluster

Figure 13.36: ECS Service—Configure service

Next, in the same step, issue the name for the service; in this scenario, it is **mlapp-cicd**, and Set the **Number of tasks** to **2**, which refers to the desired number of tasks that will be influenced by the service. However, this count can be updated (1 and above) as per the requirement. The remaining options can be kept as it is. The following figure is the continuation of step 1:

The screenshot shows the 'Deployments' section of the AWS ECS console. It includes the following fields:

- Service name:** mlapp-cicd
- Service type***: REPLICA
- Number of tasks**: 2
- Minimum healthy percent**: 100
- Maximum percent**: 200
- Deployment circuit breaker**: Disabled

Figure 13.37: ECS Service—Configure service

Step 2: Configure network

In this step, the first choose **Cluster VPC** and the **Subnets** associated with it from the dropdown. Again, choose two subnets, that is, **2a** and **2b**, similar to standalone tasks but with additional configuration, as shown in the following figure. Make sure the **Auto-assign public IP** is **ENABLED**. This will allow you to assign available public IPs to access the ML app from anywhere.

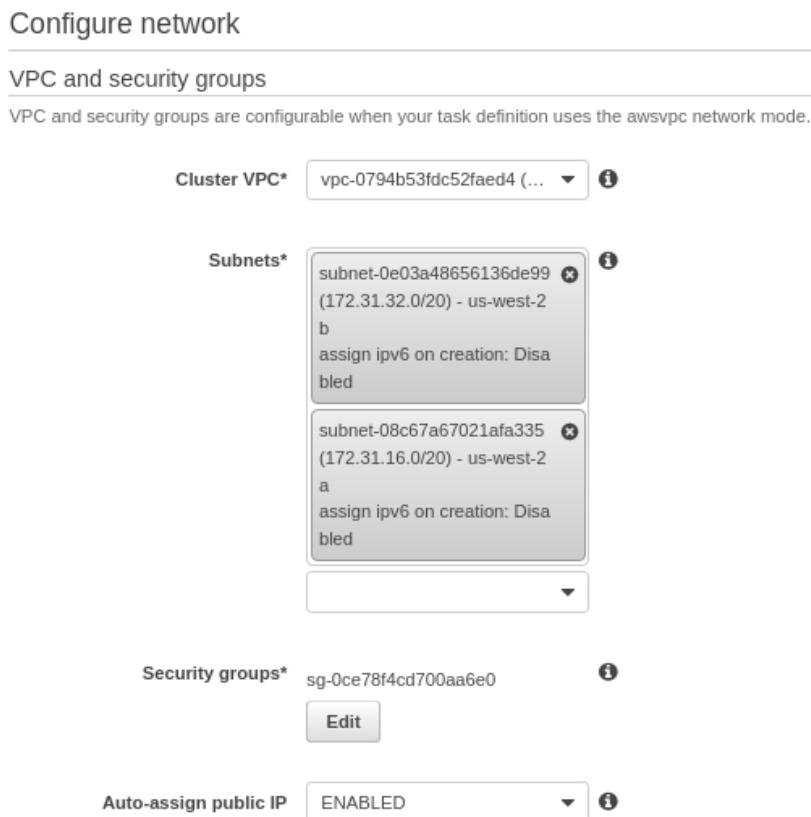


Figure 13.38: ECS Service—Configure network

Next, click on the **Edit** button in the security groups. It will open a window to configure security groups. Here, the existing security group is chosen, which you created for running independent tasks. It has inbound rules defined for ports **80**

and **8000**, as shown in the following figure. Save this configuration for the security group.

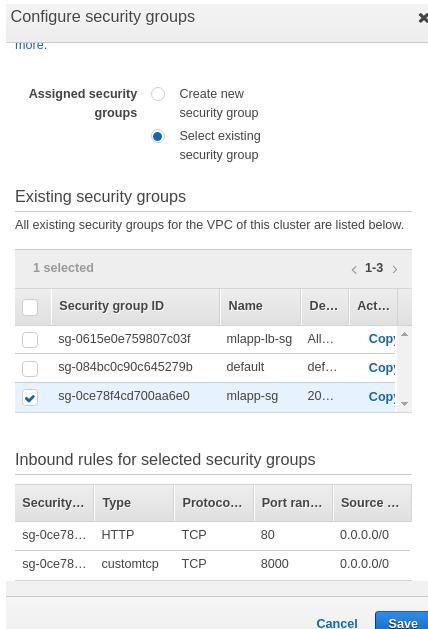


Figure 13.39: ECS Service—Configure security groups

Then, select **Application Load Balancer** as the load balancer type. This enables it to distribute across the tasks running in the service without letting the end user know. A load balancer of the type **Application Load Balancer** allows containers to use dynamic host port mapping, that is, multiple tasks are allowed per container instance. Multiple services can use the same listener port on a single load balancer with a rule-based routing path.

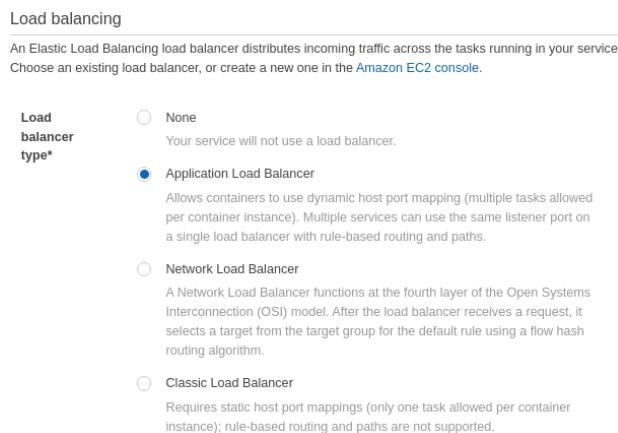


Figure 13.40: ECS Service – Load balancer type

After that, choose the **Load balancer name** from the dropdown as shown in the following figure. A load balancer will listen to HTTP port **80**. Choose the **Target group name** as **mlapp-tg** from the dropdown, which you created and configured in the load balancer section.

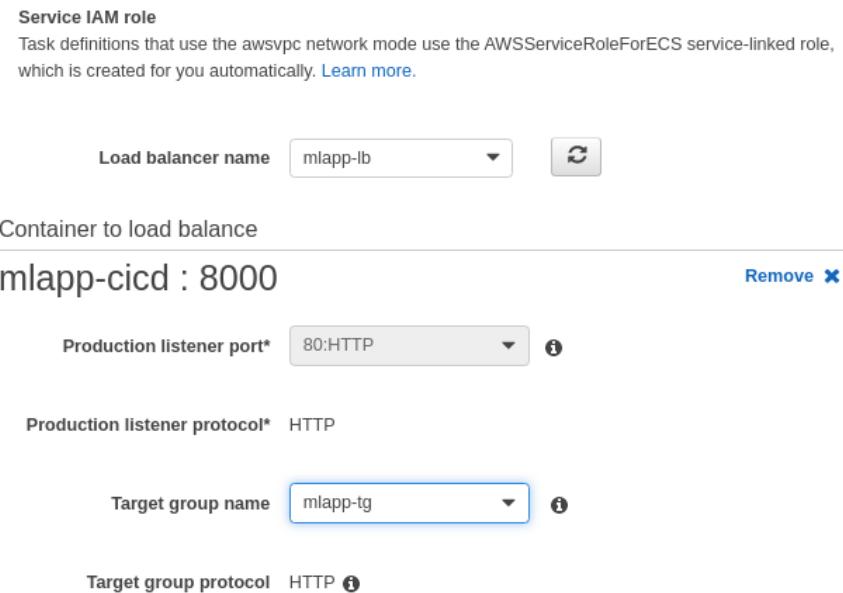


Figure 13.41: ECS Service–target and port

Step 3: Set Auto Scaling (optional)

Auto-scaling is optional. This will automatically update the service's desired count within a specified range based on CloudWatch alarms. In the current scenario, the default is used, that is, **Do not adjust the service's desired count**. Click on **Next step**.

Step 4: Review

In this step, review the configuration for the service. Here, you can go to the previous step to update the configuration. If it looks fine, then go ahead and create a service by hitting the **Create Service** button, as shown in the following figure:



Figure 13.42: ECS Service–Review

After step 4, the service will be created; however, it will take a few minutes to create the desired number of tasks for that service. The progress of task creation can be

checked under the **Events** and **Logs** tab. After a few minutes, the tasks' status will change to **RUNNING**, as shown in the following figure:

Task	Task Definition ...	Last status	Desired status ...	Group
84e6c806a3494...	mlapp:1	RUNNING	RUNNING	service:mlapp-cicd
9a7f618ec3574f...	mlapp:1	RUNNING	RUNNING	service:mlapp-cicd

Figure 13.43: ECS Service—mlapp-cicd

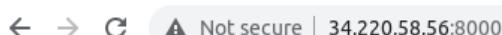
Now, click on the first task, copy its public IP, and enter the public IP, followed by port **8000**, in the browser. It should display a message, as shown in the following figure:



```
{"message": "Loan Prediction App"}
```

Figure 13.44: ECS Service—Accessing ml app with a public IP of the first task

Go back and click on the second task, copy its public IP, and enter the public IP followed by port 8000 in the browser. It should display a message, as shown in the following figure:



```
{"message": "Loan Prediction App"}
```

Figure 13.45: ECS Service—Accessing ml app with a public IP of the second task

The good thing about service over standalone tasks is that if any one of the tasks fails or shuts down due to any reason, the service will replace it with a new task to maintain the desired number of tasks up and running. The issue with a service without a load balancer is that when it replaces a task or creates a new one, the public IP will get changed, which makes it a bit difficult to keep a track of the latest or running tasks for that service.

These public IPs can be accessed by the external world through the internet. However, you can restrict access to the public IP of tasks by updating inbound rules in the security group of service **mlapp-sg**. Go to the **mlapp-sg** security group, remove inbound rules (if they exist), add a new rule and specify container port, that is, **8000** in **Port range**, and choose the security group of the load balancer, that is, **mlapp-lb-sg** instead of **0.0.0.0/0** or anywhere in the source. Finally, you can save the rules. This will restrict the direct access to public IPs of tasks for the external world through the internet.

Let's check the deployed containerized app with an application load balancer. Go to the load balancer page, copy its DNS name, and run it on the browser (paste and enter). It should display a text message. This is the root page of the application.

Then, pass the input data in **<ALB DNS name>/docs** and check the prediction output, as shown in the following figure. Here, you don't need to pass the port and IP address as the load balancer is configured with port and target group, which will handle dynamic IPs of tasks.



Figure 13.46: Application Load Balancer—Prediction response of ML app

Note: A load balancing setting can only be set on service creation.

CI/CD pipeline using CodePipeline

An AWS comes with a set of tools and services for the CI/CD pipeline. In the previous section, the build was triggered manually with CodeBuild. To automate the manual process, you must build the CI/CD pipeline with an automated trigger.

To achieve this, AWS CodePipeline will be used. Before creating a CI/CD pipeline, make sure previous services and topics are studied and implemented on the AWS cloud. An AWS CodePipeline will connect those services to create an automated pipeline. AWS services like CodeCommit, ALB, ECS FARGATE cluster, and such need to be created and configured. Make sure you provide the necessary permissions to services wherever applicable. For the current scenario, any external service, such as GitHub, is not being used. However, you can integrate external services or third-party services into this process. Here, AWS services are leveraged to deploy an application.

When there is any update pushed to CodeCommit, it will trigger the CI/CD pipeline. A CodeBuild configuration file *buildspec.yaml* will first run the tests. Next, log in to Amazon ECR, then build and push Docker container images to the **Elastic Container Registry (ECR)**. It will also write a container image definitions file *imagedefinitions.json*, in which a repository, followed by an image tag, will be captured. This container image definitions file will be stored in an Amazon S3 bucket as an artifact. A CodeBuild will export the test reports (*.xml*) in the **pytest_reports/** directory. These test results can be seen on the CodeBuild page. After that, in the deploy stage, specify the ECS cluster details. ECS cluster will pull the latest container image from Elastic Container Registry (ECR). Finally, the containerized app will be deployed into the ECS cluster. Logs will be captured in CloudWatch.

When the client or end user will access this ML app through **Domain Name System (DNS)** name, **DNS** will translate the human-readable name to numerical IP addresses that the machine can understand. This request will go to Application Load Balancer (ALB). Next, the Application Load Balancer (ALB) will check the specified listener's rules, such as port numbers and IP addresses. Then, it will send incoming requests to a specific target group which will route requests toward running tasks located in the Elastic Container Service (ECS) FARGATE cluster. The application will process the request and send the prediction to the client or end user.

The following figure shows the architecture of the automated CI/CD pipeline to deploy the containerized app on Elastic Container Service (ECS):

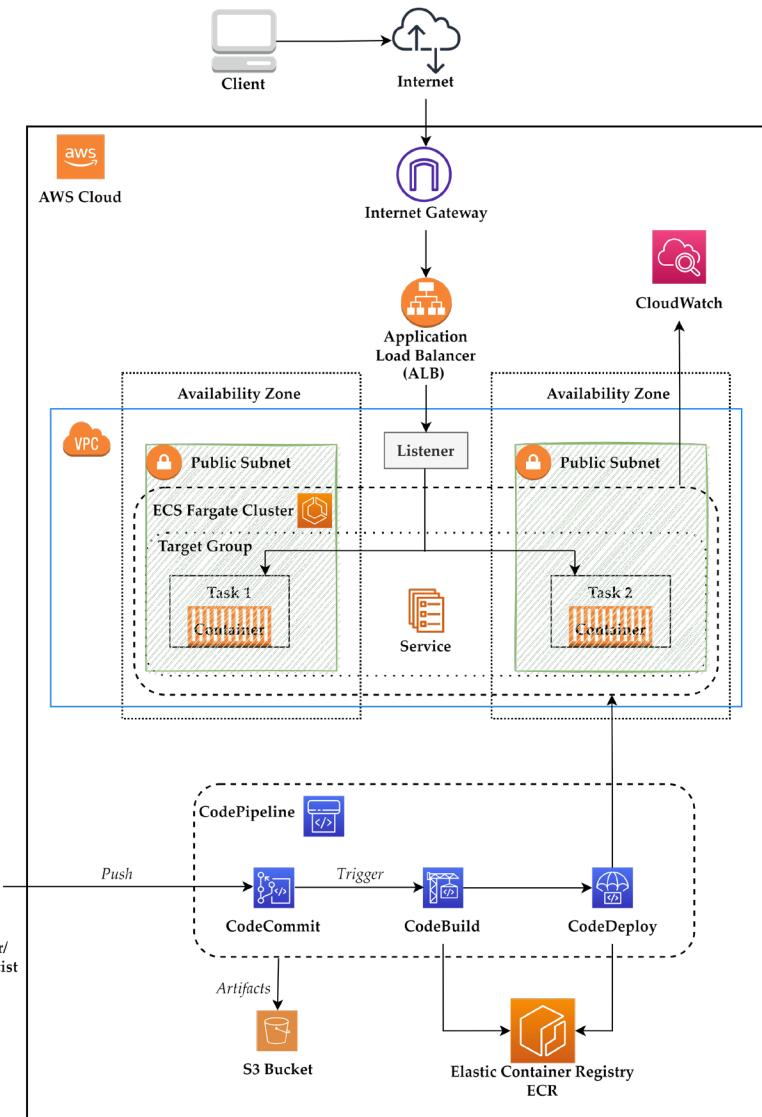


Figure 13.47: CI/CD pipeline with ECS reference architecture

AWS CodePipeline

A pipeline is a workflow that is responsible for automating the deployment or release of the application. A CodePipeline is a fully managed continuous delivery (CD) service in the AWS cloud. It enables faster and easy deployment of an application. It automates the different phases involved in the CI/CD process, such as build, test, and deploy. Each phase comprises a series of actions to be performed. AWS CodePipeline can easily integrate with third-party services like GitHub.

The following figure shows the flow of CodePipeline and AWS services used to create a CI/CD pipeline.

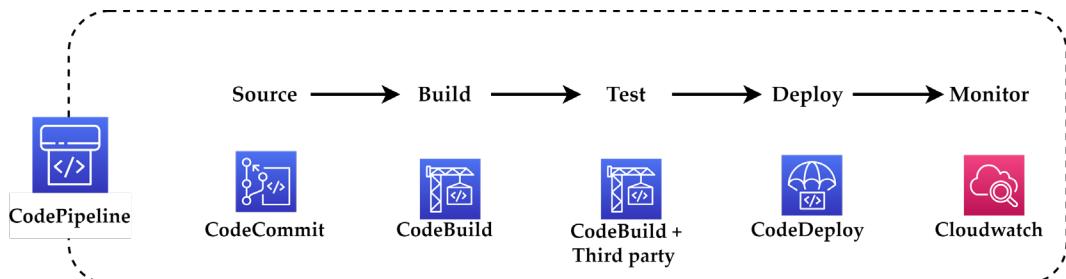


Figure 13.48: CI/CD pipeline using CodePipeline

Go to CodePipeline from Services.

Services | All Services | Developer Tools | CodePipeline

Let's start by creating a pipeline. Hit the **Create pipeline** button. It consists of 5 steps, namely:

- Choose pipeline settings
- Add source stage
- Add build stage
- Add deploy stage
- Review

Step 1: Choose pipeline settings

In this step, you need to specify the **Pipeline name** as `mlapp-cicd`. A pipeline name cannot be changed after creation. Choose a **New service role** for CodePipeline. Check to **Allow AWS CodePipeline to create a service role so it can be used with this new pipeline** checkbox. The **Role name** should auto populate.

The screenshot shows the 'Choose pipeline settings' step in the AWS CodePipeline console. The top navigation bar includes 'Developer Tools > CodePipeline > Pipelines > Create new pipeline'. The main title is 'Choose pipeline settings' with an 'Info' link. Below it, 'Step 1 of 5' is indicated. A large box titled 'Pipeline settings' contains the configuration fields:

- Pipeline name:** 'mlapp-cicd' (input field, max 100 characters)
- Service role:** Two options are shown:
 - New service role** (selected): 'Create a service role in your account'
 - Existing service role**: 'Choose an existing service role from your account'
- Role name:** 'AWSCodePipelineServiceRole-us-west-2-mlapp-cicd' (input field, type your service role name)
- Allow AWS CodePipeline to create a service role so it can be used with this new pipeline:** A checked checkbox.

Figure 13.49: CodePipeline – Choose pipeline settings

Step 2: Add source stage

In this step, you need to specify the details of the source from which the latest code and updates are to be pulled. Choose **AWS CodeCommit** as a **Source provider**. Next, choose the **Repository name** as **loan_pred**. Then, choose the **main** branch from which the latest code and updates are to be pulled. From the **Change detection options**, choose **Amazon Cloudwatch Events (recommended)** as shown in the

following figure. This will trigger the pipeline as soon as it detects the changes in the source repository's branch.

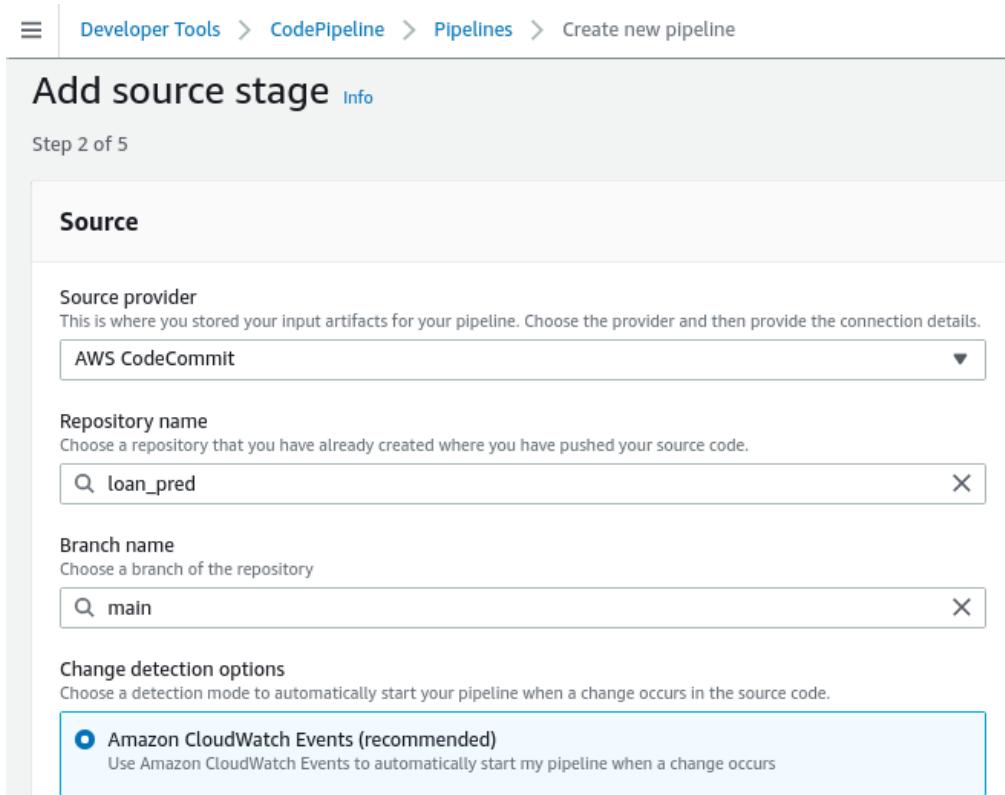


Figure 13.50: CodePipeline—Add source stage

Step 3: Add build stage

Here, first off, you need to choose the **AWS CodeBuild** as the **Build provider**. Alternatively, Jenkins can be chosen as the **Build provider**. The region **US West (Oregon)** is chosen. Next, choose the build **Project name** that you have already created in the **CodeBuild** console, or create a build project in the CodeBuild console. The **Create project** link will redirect you to the CodeBuild console. In the **Build type** section, you need to choose **Single build** as it is expected to trigger a single build.

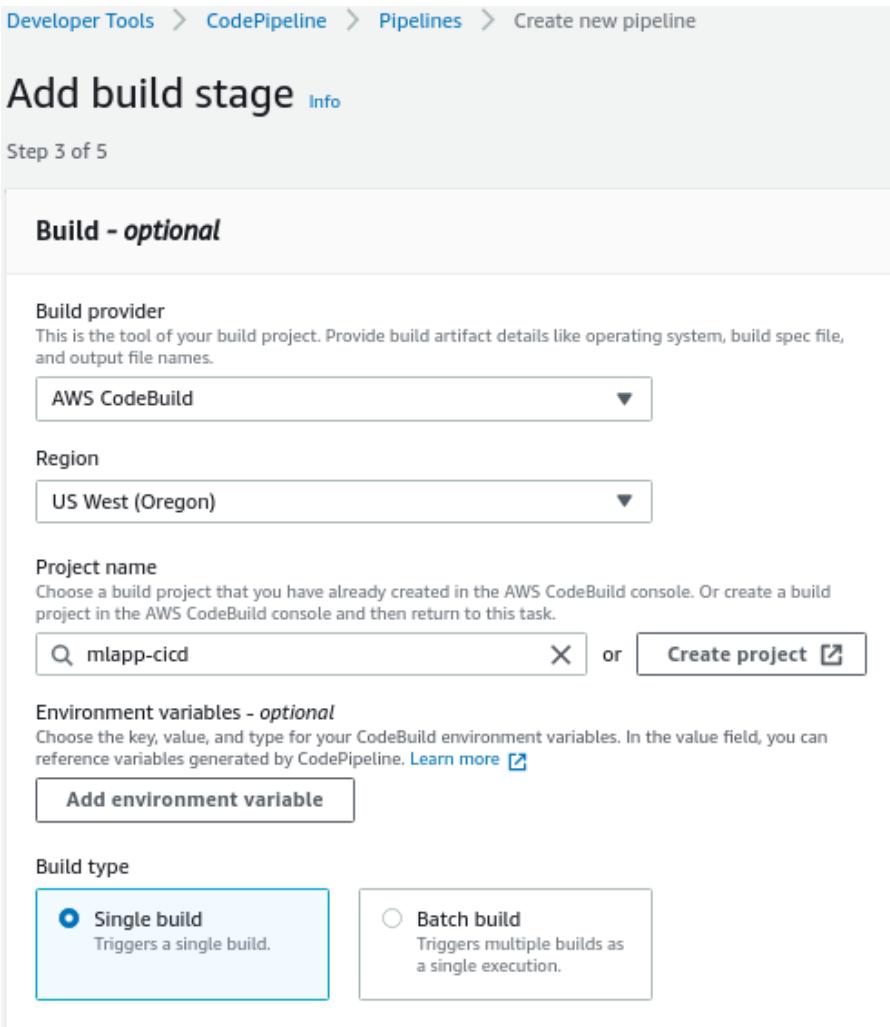


Figure 13.51: CodePipeline—Add build stage

Step 4: Add the deploy stage

In this step, provide deployment details where the application needs to deploy. This step depends on the previous step. Once the build is complete, this step will deploy the application to Elastic Container Service (ECS) cluster.

First, choose the deploy provider as **Amazon ECS**. The region is **US West (Oregon)**. Then, choose the cluster name that you have already created in the Amazon ECS console or create a cluster in the Amazon ECS console and complete this step. After

that, specify the cluster and service name that was created in the previous section. Refer to the following figure for step 4 configuration.

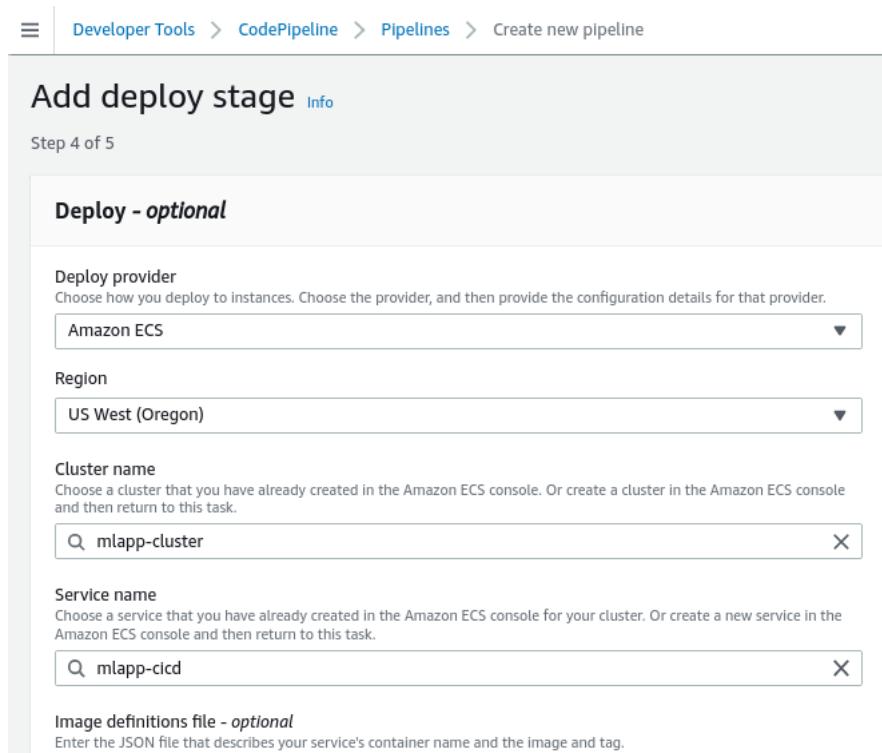


Figure 13.52: CodePipeline—Add deploy stage

Step 5: Review

In this step, review the configuration of the previous steps.

If the configuration is fine, hit **Create pipeline**.

Run CodePipeline

By default, it will start building the pipeline for the first time soon after its creation. When any updates are pushed to the source, that is, AWS CodeCommit, pipeline execution will begin. You can see the progress and status of the pipeline on the main page of CodePipeline. Go to pipeline **mlapp-cicd**, and you can see the status of each phase. It displays the link to that service. For instance, the progress of the build can be seen through the link under the **Build** step.

Go to the details in the **Build** section; you can see the source and submitter details. In the following figure, who started the build job and the source version are displayed. In the first row, the build job was started by CodePipeline, and in the next row, the

build job was started manually by the IAM user.

Build run	Status	Build number	Source version	Submitter	Duration
mlapp-cicd:5c3cb12caeda-45a7-a479-2ff667b0922b	Succeeded	11	arn:aws:s3:::codepipeline-us-west-2-945654230632/mlapp-cicd/SourceArtifacts/VxXO13G	codepipeline/mlapp-cicd	2 minutes 49 seconds
mlapp-cicd:cbe2999b-8a00-4f7d-bab5-8c8dd875afc8	Succeeded	8	refs/heads/main	suhas	3 minutes 0 seconds

Figure 13.53: CodePipeline—Build status

The main page of CodePipeline is shown in the following figure. It displays details like the latest status of the pipeline and the latest source revision.

Developer Tools > CodePipeline > Pipelines

Pipelines Info

< 1 >

Name	Most recent execution	Latest source revisions
mlapp-cicd	Succeeded	Source - ea9c31db: Update

Figure 13.54: Running a CodePipeline—Succeeded

To access the app, you can use the DNS name shown on the main page of the load balancer. A DNS name will not change even if a new task is created or if the public

IP of the task changes. The following figure shows the prediction after passing the input data using FastAPI UI.

```

Request URL
http://mlapp-lb-700497658.us-west-2.elb.amazonaws.com/predict_status

Server response
Code Details
200 Response body
{
  "status": "Approved"
}

```

Figure 13.55: Running a CodePipeline–ML app prediction using DNS of ALB

Monitoring

Monitoring is an essential part of the deployment. It is recommended to monitor ECS clusters and other integrated services to ensure the high availability of applications. This enables you to check the overall health of ECS containers. First, create a plan for the resources and services that need to be monitored. Next, decide the action to be taken if a specified event is detected. Decide who should be notified of the event or error, where to monitor metrics and events, and so on.

ECS provides cluster-level statistics, such as:

- **CPU Utilization**—Current% of CPU utilized by the ECS cluster
- **Memory Utilization**—Current% of Memory utilized by the ECS cluster

Amazon ECS metric data is automatically sent to CloudWatch in 1-minute periods that get captured for 2 weeks. **Amazon CloudWatch Container Insights** allows you to aggregate and analyze metrics and logs from containerized applications. It helps to monitor utilization metrics such as CPU, memory, disk, and network. It also provides diagnostic information about container restart failures.

You can create an AWS Lambda function that will get triggered if a specified event is detected and will send the alert to the slack channel. This way, team members and developers will be notified about the alert.

Thus, you have learned to create a simple CI/CD pipeline using CodePipeline to deploy the ML app on the Amazon ECS Fargate cluster. However, you can modify it as per business and application requirements. Reference figures in this chapter are from classic UI. If you see a new UI for creating and managing services, you can switch back to the classic UI, or you can continue with the new UI. The underlying flow and services will remain the same for the new UI, but you might see selection

changes. Alternatively, you can use AWS CloudFormation or Terraform to automate the creation of infrastructure or services.

Conclusion

In this chapter, you created an Amazon Elastic Container Service (ECS) cluster with Fargate to deploy the ML app. First off, you created an AWS account and created a codebase including a *buildspec.yaml* file for Amazon Elastic Container Service (ECS) deployment. You added the Continuous Training stage in CI/CD pipeline and accuracy test cases in the pytest file. Next, you pushed the codebase and dependencies to AWS CodeCommit. After that, you created AWS Elastic Container Registry (ECR) to store and manage Docker container images, and you used AWS CodeBuild to build and push the Docker images to AWS Elastic Container Registry (ECR) using *buildspec.yaml*. You created an Application Load Balancer (ALB) with a security and target group to distribute incoming traffic across the Fargate tasks, created ECS Service with a security group, and added an Application Load Balancer (ALB) to it. Finally, you created AWS CodePipeline and integrated these services to automate ML app deployment.

In the next chapter, you will study the drift in ML models and the monitoring process for the deployed model.

Points to remember

- Amazon Elastic Container Service (ECS) offers deployment models: EC2 and Fargate.
- Application Load Balancer (ALB) can be specified at the time of service creation only.
- Application Load Balancer (ALB) checks the health of its registered targets (by default, the root path is set to '/'; however, it can be changed to a different path of application) and routes the traffic to healthy targets only.
- It is recommended to install and configure the latest version of AWS CLI and Docker on the local machine.
- Amazon ECS cluster is a logical grouping of tasks or services.
- Create one or more IAM users with the required permissions to run the services and avoid using root user to run the services directly.

Multiple choice questions

1. _____ is serverless pay-as-you-go compute for containers.
 - a) EC2
 - b) Fargate
 - c) CloudWatch
 - d) Amazon Lightsail
2. The target group for load balancing _____
 - a) takes the request from Application Load Balancer (ALB)
 - b) routes the request to specified targets
 - c) handles dynamic IP of the tasks
 - d) all of the above

Answers

1. b
2. d

Questions

- What is a task in the Amazon ECS cluster?
- What is the use of an Application Load Balancer (ALB)?
- What are ECS services and their role?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 14

Monitoring and Debugging

Introduction

So far, you have learned various techniques to deploy ML models in production. However, deploying a model into production is not the end; monitoring is the next important step. This chapter talks about concepts and techniques of model monitoring. Model monitoring is not limited to the final endpoint where the model is deployed; you should also integrate it into the intermediate stages wherever required. The static ML model is trained offline on historical data. However, this model may not deliver consistent performance forever. The main reasons could be changes in input data, business requirements, and model degradation over time.

Structure

The following topics will be covered in this chapter:

- Importance of Monitoring
- Fundamentals of ML monitoring
- Metrics for monitoring your ML system
- Drift in ML - Types and detection techniques
- Operational monitoring using Prometheus and Grafana
- ML model monitoring with whylogs and WhyLabs

Objectives

After studying this chapter, you can start monitoring ML models. You should be able to understand the importance and fundamentals of monitoring, decide on the metrics to monitor for your project and understand the concept of drift in ML, its types, and techniques to detect the drift. You should also be able to monitor drift in the data using whylogs, an open-source lightweight library, and WhyLabs, an observability platform. You should be able to integrate Prometheus and Grafana with FastAPI, and operational monitoring with open-source tools, that is, Prometheus and Grafana.

Importance of monitoring

Once an ML model is deployed in production, it is essential to monitor it in order to ensure that the model's performance stays up to the mark and it continues to deliver reliable output seamlessly. As a matter of fact, there are many reasons for failure in ML apps or services, such as pipeline failure, model degradation over time, change in model input data, system or server failure, and change in the schema.

Multiple teams are involved while deploying models to production, which usually includes a team of Data Scientists, Data Engineers, and DevOps. If prediction errors increase, who will be held responsible? Who is the owner of the model in production?

Due to COVID-19, banks' ML model's performance was heavily affected. The model's predictions were far away from the actual figures. This is the case of a shift in input data. This should give you an overview of the challenges post-deploying models into production and the necessity of monitoring ML models.

Monitoring is essential when it comes to comparing new and old models' performance and their predictions over time. There could be latency issues, that is, delayed output. For tracking and investigating latency issues, efficient monitoring is needed. You need to monitor input data to ensure that production input data is being processed like the training data. To track and handle extreme values, out of the range values, or special cases before passing them to the models, monitoring is required. Last but not least is model security, that is, monitoring is required to track any external attack on the model or system.

The objectives of ML monitoring are as follows:

- To detect the issues or failures at early stages so that necessary steps can be taken
- To keep a track of resource usage and model prediction to evaluate model and system performance in the production environment

- To detect the change in distribution, schema, and anomalies in input data that may cause an error in the predictions
- To ensure the availability of model predictions and that they are explainable
- To track and store metrics in specified storage or database

Model monitoring helps data scientists to reduce model failures, avoid downtime, and ensure reliable outcomes for users.

Fundamentals of ML monitoring

ML monitoring refers to tracking the performance, errors, metrics, and such of deployed models and sending alerts (when required) to ensure models continue performing above an acceptable threshold. It is not limited to tracking input data or model degradation. However, it should take all the things into account that may affect model performance directly or indirectly. ML monitoring helps decide whether an existing model needs to be updated.

The following are the essential steps to consider:

- **Monitoring is the key:** Monitoring is essential once a model is deployed in production. Monitoring enables you to track and fix issues or errors in a faster manner. Monitoring can be broadly divided into two types:
 - **Functional monitoring:** In the case of ML, functional monitoring refers to tracking metrics, errors, and performance related to ML models, such as accuracy and outliers.
 - **Operational monitoring:** Operational monitoring refers to tracking system-specific metrics, such as CPU and RAM utilization, uptime, and throughput.
- **Scalable integration:** Monitoring should be easy to integrate with your existing infrastructure and workflow. The monitoring system should be seamless and scalable to integrate. It should be able to track multiple models if there is more than one model. The monitoring solution should be platform agnostic so that it can be used for different types of deployment, having different tech stacks.
- **Metrics tracking:** Tracking accuracy is not enough. Monitoring systems should be able to track all the metrics that can affect model and system performance over time. A centralized monitoring system should use multiple performance metrics to give the overall status of the solution. Use different metrics for different types of features. For instance, min, max, mean, standard deviation, and outliers can be used for numerical data. The metrics records and logs need to be stored in a database so that they can be analyzed later on.

- **Alert system for important events:** You cannot track 100+ metrics manually 24x7; hence, the alert system should be part of the monitoring solution. Not everything needs an alert. You may be tracking 20+ metrics; however, only a few metrics need high attention. For instance, if input data drift is exceeding the threshold, then the alert system should send the notification to the responsible team or Data Scientists. The whole purpose of the alert system is to notify concerned teams or individuals so that issues or errors can be avoided or can be solved at the earliest to ensure consistency in ML model performance.
- **Root cause analysis and debugging:** Once you get the alert, you may need to act on it. You can start the root cause analysis to determine the issue and fix it by debugging. For instance, if model accuracy is going below the threshold, it could be because of a change in the distribution of input data or anomalies.

To design an efficient monitoring system, you need to consider existing pipelines and infrastructure. There are mainly three pillars of monitoring:

- **Processing and Storage:** It processes and stores the critical metrics in a database or data source with a timestamp. You can query the metrics when required.
- **Graphs and Dashboard:** It will query or fetch the monitoring metrics from a connected database or data source. You can choose the visualization as per the type of metrics. Here, you have to decide the metrics that need to be displayed. It should summarize overall monitoring with intuitive graphs.
- **Alert:** Finally, it should send an alert to the concerned team or data scientists to take immediate action, if required. This helps to prevent future failures or mitigate the impact.

The following figures show the three pillars of the monitoring system for effective monitoring:

Pillars of Monitoring System

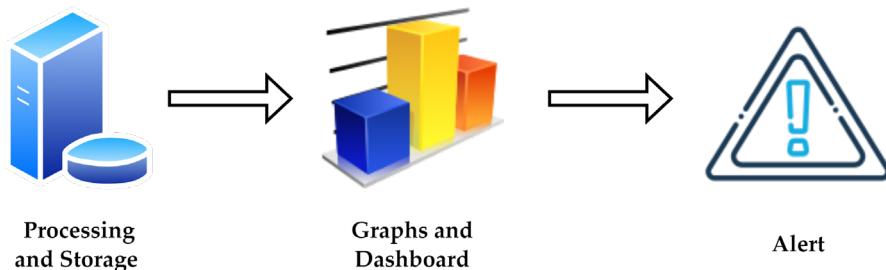


Figure 14.1: Pillars of the monitoring system

Before setting up any monitoring system, here is the list of questions that need to be answered:

- What do you plan to monitor?
- What tools and language are you using?
- What platform or library will be used for monitoring?
- How do you plan to integrate the monitoring setup with the existing environment and tools?
- What threshold is to be used for alerts?
- What action should be taken after detecting an issue or failure?

Finally, as per the model life cycle, monitoring should complete the feedback loop, that is, after detecting an issue or failure, it should send a notification to the concerned team or data scientists so that they can take necessary action, such as retraining the ML model.

Metrics for monitoring your ML system

You need to decide which type of metric to monitor. Operational metrics will enable developers to track system or server-related failures or warnings, such as high resource usage that can increase the latency, which will affect end-user experience. Tracking operational metrics is essential because the server should be in good condition to run the ML model and other tasks. Another factor is the cost. Operational teams have to set limits on the cost and need to track which resource or service causes higher costs.

Here are a few important operational metrics:

- System or Server
 - Resource usage
 - Availability
 - Latency
 - Throughput
- Cost
 - Infrastructure cost
 - Storage cost
 - Additional service (if any)

Once the server is up and running without any issues, you can focus on functional or ML model metrics. You will study the detection of model monitoring metrics and ways to address them in this chapter.

The following are a few important model metrics:

- Input data
 - Model version
 - Data drift
 - Outliers
- ML model
 - Model version
 - Model hyperparameters
 - Metadata
 - Predictions or Output
 - Classification model evaluation metrics
 - Accuracy
 - Confusion Matrix
 - ROC-AUC Score
 - Precision and Recall Scores
 - F1-Score
 - Regression model evaluation metrics
 - Root Mean Square Error (RMSE)
 - R-Squared and Adjusted R-Square
 - Mean Absolute Error (MAE)
 - Mean Absolute Percentage Error (MAPE)
- Prediction drift

You can decide which metrics to track closely and which metrics need alerts.

Drift in ML

Model drift means a change in the behavior of ML models or that the model is not performing as expected or per the Service Level Agreement (SLA). Model performance may degrade after deploying it to production, as the model can receive data that was not introduced during model training.

Types of drift in ML

There are mainly three types of drift in ML:

- Data drift
- Prediction drift
- Concept shift

Data drift

Data drift is when the distribution or characteristics of input features change with reference to training data. It is also known as feature drift, population drift, or covariate shift. If the distribution of input data changes, then its predictions might get affected as the model is not prepared for it.

If a new category gets added to the feature post-deployment, then it may cause an error while making the prediction, as it was not there at the time of model training.

Another example could be, suppose there is a feature called **Credit Rating** in training data, and it has high weightage, which means a change in this feature may cause a major change in model output. Now, the business decided to go with Moody's credit rating instead of S&P. This will cause a change in input data. For instance, S&P's AA-is equivalent to Moody's Aa3 rating.

The following equation shows that the distribution of training data does not match the distribution of reference (production) data.

Mathematically, data drift can be defined as follows:

$$P(X) \neq Pref(X)$$

Where $P(X)$ denotes the input data probability distribution.

Prediction drift

As input data changes with data drift, this may cause a change in the target or prediction variable. It refers to a change in predictions over time. It is also named a prior probability shift, label drift, or unconditional class shift. This can also occur due to the removal or addition of new classes. Retraining the model can help mitigate the model degradation owing to prediction drift.

Mathematically, prediction drift can be defined as follows:

$$P(Y) \neq Pref(Y)$$

Where $P(Y)$ denotes the prior probability distribution of target labels.

Concept shift

A concept shift occurs when the relationship between independent variables and dependent or target variables changes. It is also known as posterior class shift, conditional change, or real concept drift. It refers to changes in the relationship

between the input variables and the target variables. If you detect any significant concept shift, it is very likely that your model's predictions are unreliable. A **Concept** in Concept shift refers to the relationship between independent and dependent variables.

For instance, a car insurance company changes its claim policy that a claim for a specific part of the car will be rejected. Suppose that part of the car got damaged in an accident, the customer's claim for it will be rejected. In this scenario, mapping between the input feature and target feature changes even though the distribution of input data remains the same.

Mathematically, concept shift can be defined as follows:

$$P(Y|X) \neq \text{Pref}(Y|X)$$

Where $P(Y|X)$ denotes the posterior probability distribution of the target labels.

Following are the patterns of concept shift:

Sudden or abrupt



Gradual



Recurring or cyclic



Incremental



Temporary



Figure 14.2: Different patterns of concept shift

Techniques to detect the drift in ML

Statistical distance measurement using distance metrics between two distributions is useful for detecting drift in ML.

If there are many independent variables in the dataset, then you can use dimensionality reduction techniques, such as PCA. Tracking many features can increase the load on the monitoring system and sometimes it becomes difficult to mitigate drift by targeting specific features.

Basic statistical metrics, such as mean value, standard deviation, correlation, and minimum and maximum values comparison, can be used for calculating the drift between training and current independent variables.

Distance measures like Population Stability Index (PSI), Characteristic stability index (CSI), Kullback–Leibler divergence (KL-Divergence), Jensen–Shannon divergence (JS-Divergence), and Kolmogorov-Smirnov (KS) statistics can be used for continuous features.

The cardinality checks, Chi-squared test, and entropy can be used for categorical variables.

Control charts and histogram intersections can be used to detect a drift in data.

Finally, there are several platforms for model monitoring, such as WhyLabs, and libraries, such as deepchecks, and alibi-detect. They come with easy integrations and a ready framework for drift detection. The best part about this is that most of them provide a drift detection framework, storage for logs and historical data, intuitive monitoring dashboards, and alert mechanisms for critical events.

Addressing the drift in ML

Once you detect the drift in the ML model, it can be addressed in the following ways:

Data quality issues

If there is an issue with input data, then it can be easily fixed. For instance, high-resolution images were provided for training face recognition models; however, low-resolution images were passed to the deployed model.

Retraining the model

After detecting the data or concept shift, retraining the model with recent data can improve its performance. Sometimes production data is not sufficient to train the model. In that case, you can combine historical data with recent production data and give more weight to recent data.

Here are four strategies for retraining the model:

- **Periodically retraining:** Scheduling it at a fixed time, for instance, every Monday at 10 PM
- **Data or event-driven:** When new data is available
- **Model or metric driven:** When accuracy is lower than a threshold or SLA
- **Online learning:** Where the model continuously learns in real-time or near real-time on the latest data

Rebuilding or tuning the model

If retraining the model doesn't work, then you may need to consider rebuilding it or tuning it on recent data. You can automate this using a pipeline.

Operational monitoring with Prometheus and Grafana

Prometheus is an open-source system used for event monitoring and alerting. It scrapes the real-time data from instrumented jobs and stores it with a timestamp in the database. The word **instrument** refers to the use of a client library that allows Prometheus to track and scrape its metrics and store them locally. Prometheus offers client libraries that can be used to instrument your application. In this scenario, you will be using the Prometheus Python client in the FastAPI application to expose its metrics that need to be tracked.

A Prometheus server scrapes and stores metrics from instrumented jobs in time series format. This data can be fetched using PromQL - query language, and it can be used to visualize the metrics. It comes with an alert manager to handle alerts.

According to GrafanaLabs, Grafana is an open-source interactive visualization and monitoring platform that enables users to visualize metrics, logs, and traces collected from deployed applications. Grafana is easy to integrate with most common databases, such as Prometheus, Influx DB, ElasticSearch, MySQL, and PostgreSQL.

As Grafana is an open source tool, hence you can write an integration plugin from scratch to connect with multiple data sources. The Grafana dashboard fetches the data from connected data sources and allows you to pick up the visualization type from plenty of visualization options, such as heat maps, bar charts, and line graphs. You can easily run the query, visualize metrics, and set up alerts for critical events.

In this scenario, you will be using Grafana and Prometheus. Both Prometheus and Grafana are open-source tools. As a matter of fact, Prometheus and Grafana are popular combinations in the industry for monitoring systems. Grafana dashboard will be used for visualization and alert management. It will fetch the data from the Prometheus database for querying metrics and will display the intuitive visualization on the dashboard.

Prometheus and Grafana can be separately installed and configured on a local machine or a remote server. However, you will be using docker images of Prometheus and Grafana by executing the *docker-compose.yaml* file.

To maintain consistency, similar files from the previous chapters with minor modifications will be used. The files required for Prometheus and Grafana dashboards are added as shown in the following file structure:

```
.  
└── .gitignore  
└── config.monitoring  
└── datasource.yml
```

```
├── docker-compose.yaml
├── Dockerfile
├── fastapi-dashboard.json
├── LICENSE
├── main.py
├── prometheus.yml
├── pytest.ini
├── README.md
├── requirements.txt
├── runtime.txt
├── start.sh
├── test.py
├── tox.ini
└── app
    └── src/
        ├── MANIFEST.in
        ├── README.md
        ├── requirements.txt
        ├── setup.py
        ├── tox.ini
        └── prediction_model/
            ├── pipeline.py
            ├── predict.py
            ├── train_pipeline.py
            ├── VERSION
            ├── __init__.py
            ├── config/
            │   ├── config.py
            │   └── __init__.py
            ├── datasets/
            │   ├── test.csv
            │   ├── train.csv
            │   └── __init__.py
```

```
└── processing/
    ├── data_management.py
    ├── preprocessors.py
    └── __init__.py
└── trained_models/
    ├── classification_v1.pkl
    └── __init__.py
└── tests/
    └── pytest.ini/
        └── test_predict.py
```

config.monitoring

The extension of this file is *.monitoring*. In this file, configure the admin password and user permission for sign up.

1. GF_SECURITY_ADMIN_PASSWORD=pass@123
2. GF_USERS_ALLOW_SIGN_UP=false

The Docker container currently allows new Grafana users to sign up, so anyone can create an account and view the dashboard. Therefore, for security purposes, the admin password and allowing new users to sign up should be set to false.

prometheus.yml

This file contains Prometheus configurations, such as *scrape_configs* and *scrape_intervals*.

1. # Global config
2. global:
3. scrape_interval: 15s
4. evaluation_interval: 15s
5. external_labels:
6. monitor: "app"
- 7.
8. rule_files:
- 9.
10. scrape_configs:
11. - job_name: "prometheus"

```
12.  
13. static_configs:  
14.     - targets: ["localhost:9090"]  
15.  
16.     - job_name: "app"  
17. dns_sd_configs:  
18.     - names: ["app"]  
19.         port: 8000  
20.         type: A  
21.         refresh_interval: 5s
```

In this file, the scrape interval is set to 15 seconds (**15s**) in the global config, which means it will scrape the metrics data after every 15 seconds. It is targeting an app, that is, the FastAPI app here. Port is the app's port, that is, **8000**, and the refresh interval is set to 5 seconds (**5s**).

Dockerfile

This *Dockerfile* will install the dependencies, set the working directory, expose ports, and finally, run the ML app.

```
1. FROM python:3.7-slim-buster  
2.  
3. ENV PYTHONPATH "${PYTHONPATH}:src/"  
4.  
5. WORKDIR /app/  
6.  
7. COPY . .  
8.  
9. RUN chmod +x /app  
10.  
11. COPY ./start.sh /start.sh  
12.  
13. RUN chmod +x /start.sh  
14.  
15. # Expose the port that uvicorn will run the app on
```

```
16. ENV PORT=8000
17. EXPOSE 8000
18.
19. RUN pip install --upgrade pip
20.
21. RUN pip install --upgrade -r requirements.txt --no-cache-dir
22.
23. CMD ["/start.sh"]
```

docker-compose.yaml

This docker-compose file will set up and run three services:

- The app, that is, FastAPI app: For the ML model. It will run on port **8000**.
- Prometheus server: It will run on port **9090**.
- Grafana dashboard: It will run on port **3000**.

In the following file, you can see that the Grafana service depends on the Prometheus service:

```
1. version: "2.2"
2.
3. services:
4.   app:
5.     build: .
6.     restart: unless-stopped
7.     container_name: app
8.     ports:
9.       - 8000:8000
10.    networks:
11.      example-network:
12.        ipv4_address: 172.16.238.10
13.
14.    prometheus:
15.      image: prom/prometheus:latest
16.      restart: unless-stopped
```

```
17.   container_name: prometheus
18.   ports:
19.     - 9090:9090
20.   volumes:
21.     - ./prometheus.yml:/etc/prometheus/prometheus.yml
22.   command:
23.     - "--config.file=/etc/prometheus/prometheus.yml"
24.   networks:
25.     example-network:
26.       ipv4_address: 172.16.238.11
27.
28. grafana:
29.   image: grafana/grafana:latest
30.   restart: unless-stopped
31.   user: "472"
32.   container_name: grafana
33.   depends_on:
34.     - prometheus
35.   ports:
36.     - 3000:3000
37.   volumes:
38.     - ./datasource.yml:/etc/grafana/provisioning/datasource.yml
39.   env_file:
40.     - ./config.monitoring
41.   networks:
42.     example-network:
43.       ipv4_address: 172.16.238.12
44.
45. networks:
46. example-network:
47. # name: example-network
```

```
48.     driver: bridge
49.     ipam:
50.       driver: default
51.       config:
52.         - subnet: 172.16.238.0/24
```

main.py

This file contains code for the FastAPI app. In this code, add the following two lines to instrument the FastAPI app. In Prometheus, the instrumentation refers to the use of a library in application code (in this scenario, it is FastAPI) so that Prometheus can scrape metrics exposed by the app.

First, add **prometheus-fastapi-instrumentator** with version **5.7.1** to the *requirements.txt* file. Then, import the Prometheus FastAPI instrumentator as follows:

```
1. from prometheus_fastapi_instrumentator import Instrumentator
```

Then, at the end of the file, add the following line to instrument FastAPI app to the Prometheus FastAPI instrumentator:

```
1. Instrumentator().instrument(app).expose(app)
```

With this single line, FastAPI will be instrumented and all Prometheus metrics used in the FastAPI app can be scraped via the added **/metrics** endpoint.

You have already studied the rest of the files in the previous chapters. Refer to the previous chapters for files mentioned in the preceding directory structure.

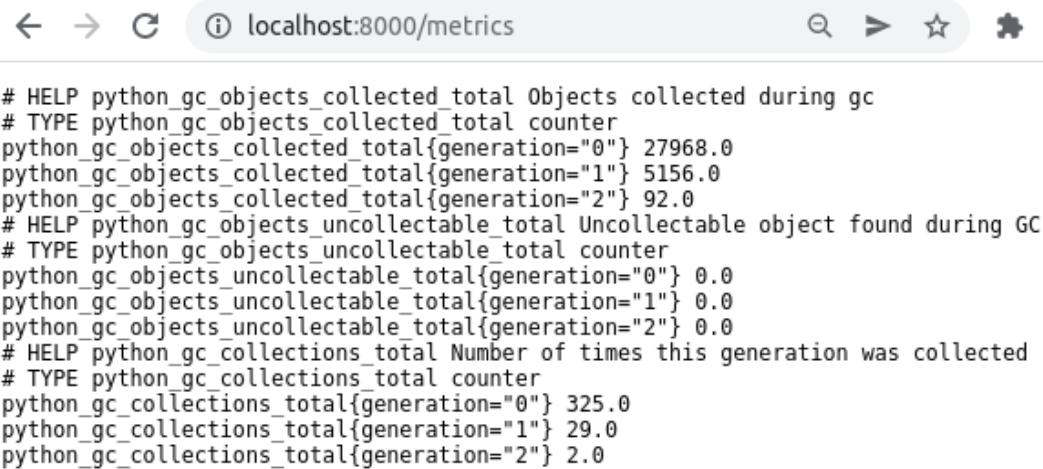
You will need to run the docker containers by running the following docker-compose command in the terminal:

docker-compose up

Now you have access to three containers and their respective ports:

- **FastAPI**: Running at <http://localhost:8000/>
- **Prometheus**: Running at <http://localhost:9090/>
- **Grafana dashboard**: Running at <http://localhost:3000/>

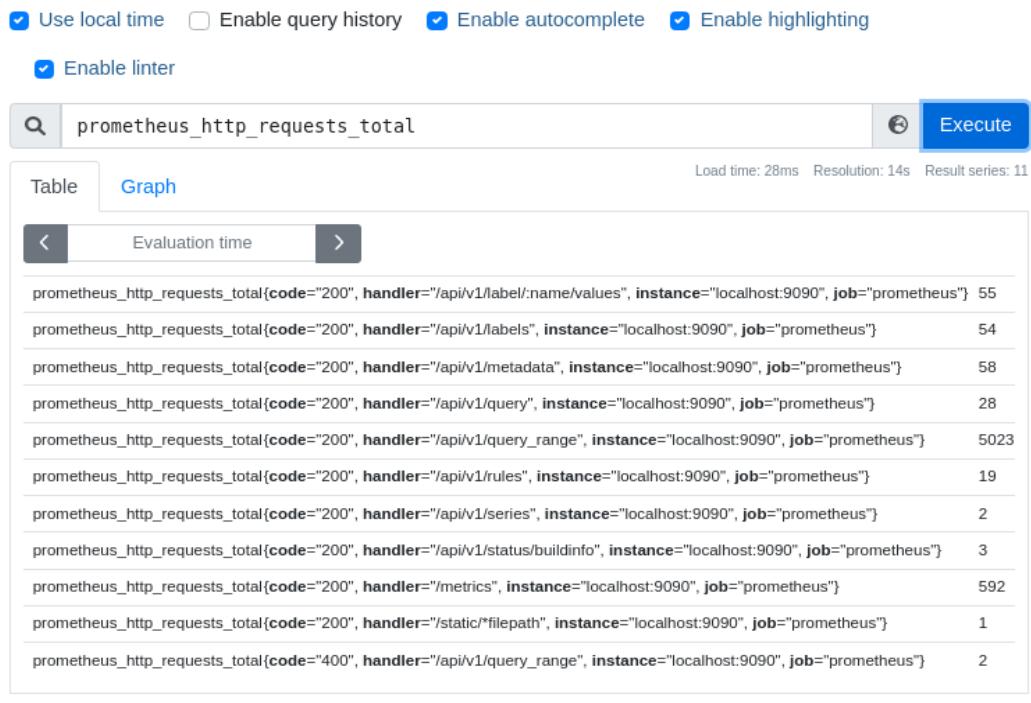
On the FastAPI app, you can access the **/metrics** endpoint to view the data that Prometheus is scraping from it, as shown in the following figure:



```
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 27968.0
python_gc_objects_collected_total{generation="1"} 5156.0
python_gc_objects_collected_total{generation="2"} 92.0
# HELP python_gc_objects_uncollectable_total Uncollectable object found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
# HELP python_gc_collections_total Number of times this generation was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 325.0
python_gc_collections_total{generation="1"} 29.0
python_gc_collections_total{generation="2"} 2.0
```

Figure 14.3: FastAPI metrics

You can access Prometheus UI on port **9000** and run the PromQL query. To get the total number of requests, execute **prometheus_http_requests_total**. You will get the HTTP status code, handler, instance, job name, and count, as shown in the following figure:



Use local time Enable query history Enable autocomplete Enable highlighting
 Enable linter

Execute

		Load time: 28ms Resolution: 14s Result series: 11
<input checked="" type="radio"/> Table	<input checked="" type="radio"/> Graph	
Evaluation time < >		
<pre>prometheus_http_requests_total{code="200", handler="/api/v1/label/:name/values", instance="localhost:9090", job="prometheus"} 55 prometheus_http_requests_total{code="200", handler="/api/v1/labels", instance="localhost:9090", job="prometheus"} 54 prometheus_http_requests_total{code="200", handler="/api/v1/metadata", instance="localhost:9090", job="prometheus"} 58 prometheus_http_requests_total{code="200", handler="/api/v1/query", instance="localhost:9090", job="prometheus"} 28 prometheus_http_requests_total{code="200", handler="/api/v1/query_range", instance="localhost:9090", job="prometheus"} 5023 prometheus_http_requests_total{code="200", handler="/api/v1/rules", instance="localhost:9090", job="prometheus"} 19 prometheus_http_requests_total{code="200", handler="/api/v1/series", instance="localhost:9090", job="prometheus"} 2 prometheus_http_requests_total{code="200", handler="/api/v1/status/buildinfo", instance="localhost:9090", job="prometheus"} 3 prometheus_http_requests_total{code="200", handler="/metrics", instance="localhost:9090", job="prometheus"} 592 prometheus_http_requests_total{code="200", handler="/static/*filepath", instance="localhost:9090", job="prometheus"} 1 prometheus_http_requests_total{code="400", handler="/api/v1/query_range", instance="localhost:9090", job="prometheus"} 2</pre>		

Remove Panel

Figure 14.4: Prometheus UI—Query execution

Prometheus allows you to view these query results in graphical format by switching to the **Graph** tab in Prometheus web UI.

You have seen Prometheus web UI, and PromQL is fetching the metric data as well. Now, in the next part, you will connect Prometheus to the Grafana dashboard as a data source. Metrics data fetched from the Prometheus server will be visualized on the Grafana dashboard. Go to Grafana dashboard web UI and click on the **add your first data source option**, as shown in the following figure:

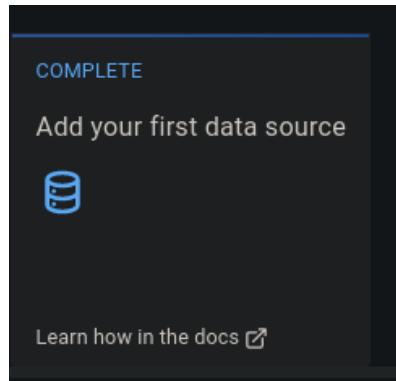


Figure 14.5: Grafana—Adding data source

Next, select the **Prometheus** database from the **Time series databases**, as shown in the following figure:

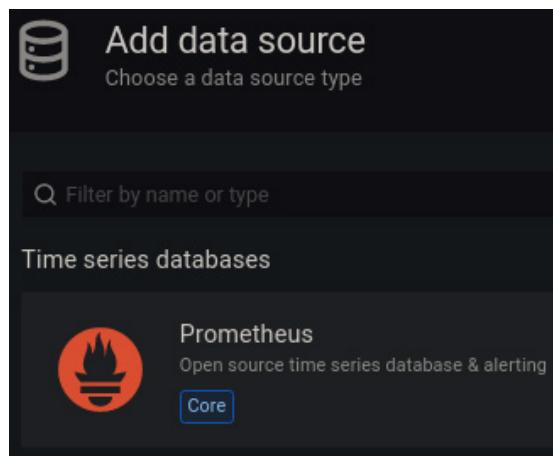


Figure 14.6: Grafana—Prometheus data source

In the settings, provide **Prometheus** as a name in the **Name** field. Then, mention **http://prometheus:9090** in the URL field, as shown in the following figure, as Prometheus service is running on port **9090**:

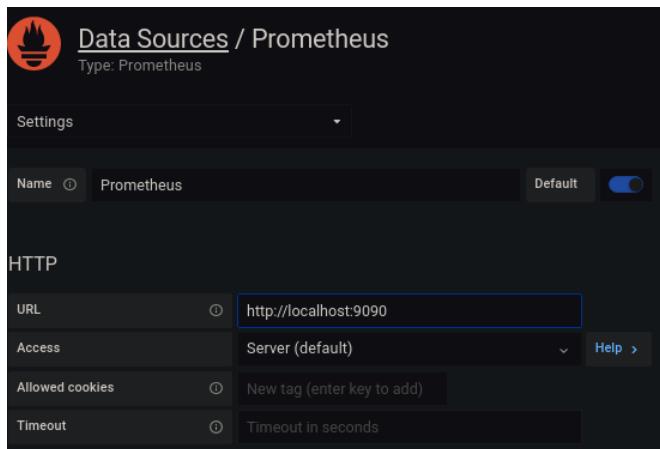


Figure 14.7: Grafana–Prometheus configuration

Finally, proceed with the basic configuration; however, you can also configure other fields. Scroll to the bottom of the page and hit the **Save & test** button, as shown in the following figure:

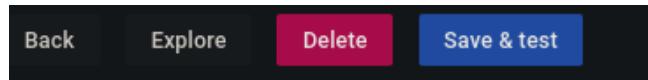


Figure 14.8: Grafana–saving and testing Prometheus configuration

If it is configured properly, you should see the integration status success message after you save and test the configuration.

Once Prometheus integration is done, create a new dashboard with the **New Dashboard** option in Grafana web UI, as shown in the following figure:

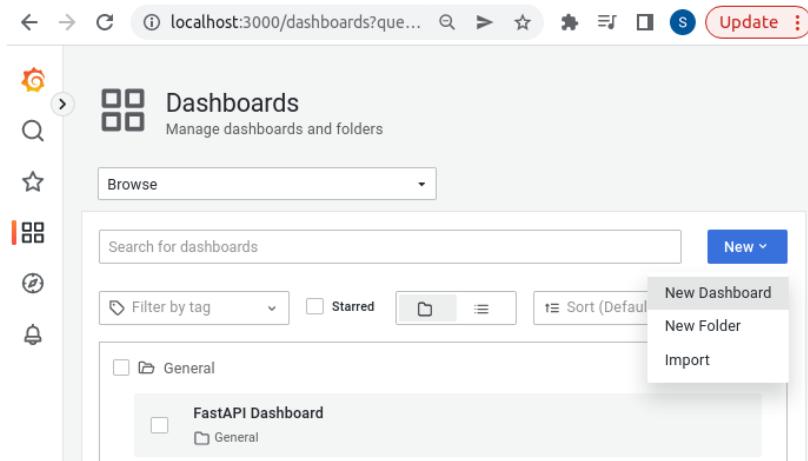


Figure 14.9: Grafana–creating a new dashboard

The Grafana dashboards can be created from scratch or by importing JSON files from storage. Grafana also lets you import the dashboard via [Grafana.com](https://grafana.com), as shown in the following figure:

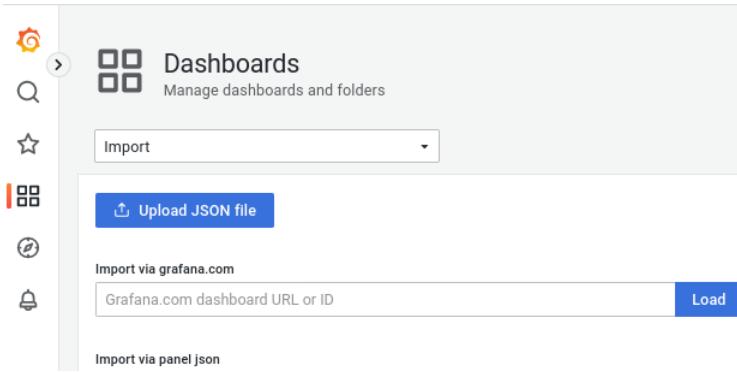


Figure 14.10: Grafana—uploading JSON file

In this scenario, import a JSON file from the directory. You can refer to one of the original public Grafana instances, hosted by Grafana Labs, using the following link:

<https://play.grafana.org/d/000000012/grafana-play-home?orgId=1>

For various dashboard visualizations and sources, refer to the following link <https://grafana.com/grafana/dashboards/>.

Next, provide the dashboard name and Prometheus source, as shown in the following figure:

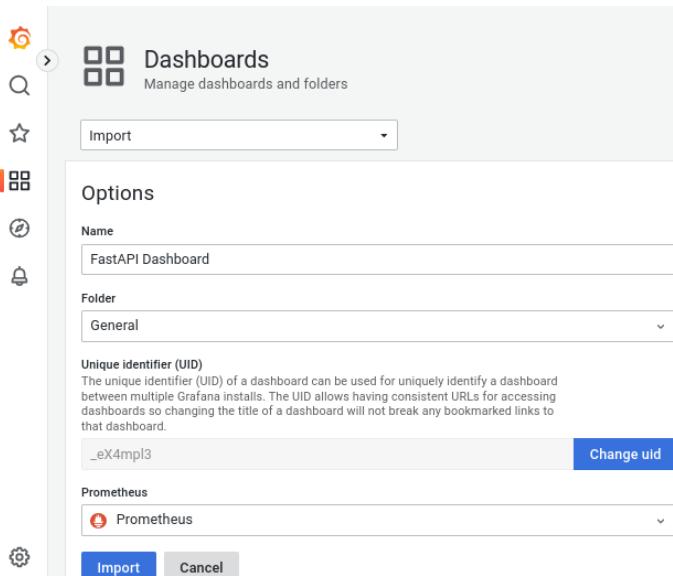


Figure 14.11: Grafana—Prometheus as a data source

Finally, open the newly created dashboard from the web UI. After importing the dashboard from the JSON file, you can customize the dashboard by changing the chart type, as shown in the following figure.

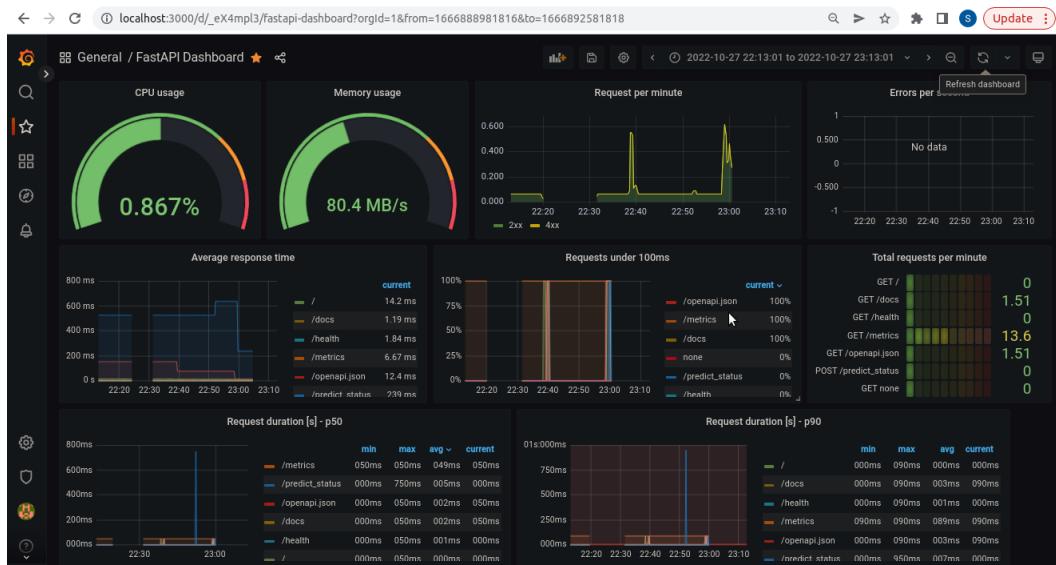


Figure 14.12: Grafana–Dashboard

The next step is to send alerts for critical events. Grafana includes built-in support for the Prometheus Alert manager. Grafana provides mainly three strategies for setting alerts:

- **Cloud Alert manager:** Cloud Alert manager runs in Grafana Cloud. It supports alerts from Grafana, Mimir, and Loki.
- **Grafana Alert manager:** Grafana Alert manager is an internal Alert manager. It is pre-configured and supports alerts from Grafana; however, it cannot receive alerts outside Grafana.
- **An external Alert manager:** You can configure Grafana to use an external alert manager. This is useful when you are looking for a centralized alert manager that can receive alerts from different sources.

You can add the alert rules, labels, and notification policy in Grafana. The following figure shows the Alert UI in Grafana:

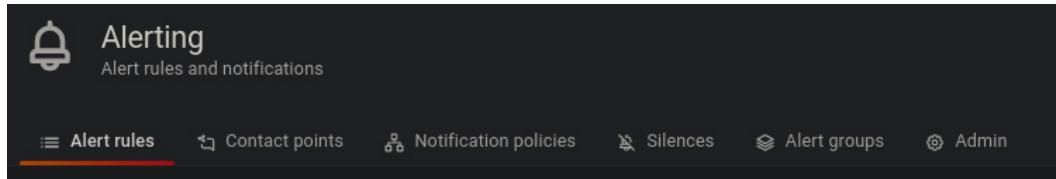


Figure 14.13: Grafana—Alerting UI

ML model monitoring with whylogs and WhyLabs

WhyLabs is a model monitoring and observability platform built to monitor data drift, model performance degradation, data quality, model evaluation metrics, and sending alerts to teams or individuals. It is built on top of an open-source package **whylogs**. You can monitor whylogs profiles continuously with the WhyLabs Observability Platform.

Following are the salient features of WhyLabs:

- **Minimal setup time:** You can start by installing lightweight open-source library whylogs with `pip install whylogs`. In no time, a summary of the dataset can be generated, that is, whylogs profiles, which helps you do the following:
 - Track changes in datasets
 - Visualize the summary statistics of datasets
 - Check the data quality by defining data constraints
- **Seamless integration:** It can be integrated with the existing data pipelines, model life cycle, model framework, and MLOps Ecosystem easily.
- **Data privacy:** WhyLabs works with a statistical summary of the data created by the whylogs agent, so your actual data is not getting transferred or stored in the WhyLabs platform.
- **Scalability:** It can be easily scaled up to handle large amounts of data and multiple projects.
- **Centralized monitoring:** With the WhyLabs platform, you can monitor multiple projects under a single umbrella. It allows you to create multiple isolated projects within a single account, and it also keeps track of input and output data, model performance, and such.

whylogs

Without further delay, let's start using the whylogs logging functionality. First, install whylogs using `pip install whylogs` if it's not installed already. Next, import a loan prediction dataset and start logging it with whylogs.

```
1. # Import packages
2. import numpy as np
3. import pandas as pd
4. from sklearn.model_selection import train_test_split
5. pd.set_option("display.max_columns", None) #To show all columns
   in DataFrame
6. import whylogs as why
7.
8. df = pd.read_csv("https://gist.githubusercontent.
com/suhas-ds/a318d2b1dda8d8cbf2d6990a8f0b7e8a/
raw/9b548fab0952dd12b8fdf057188038c8950428f1/loan_dataset.csv")
9.
10. profile1 = why.log(df)
11. profile_view1 = profile1.view()
12. profile_view1.to_pandas()
```

To view the statistical summary, that is, the whylogs profile, convert it to the pandas DataFrame. The pandas DataFrame of the profile contains the following metrics:

- cardinality/est
- cardinality/lower_1
- cardinality/upper_1
- counts/n
- counts/null
- distribution/max
- distribution/mean
- distribution/median
- distribution/min
- distribution/n
- distribution/q_01

- distribution/q_05
 - distribution/q_10
 - distribution/q_25
 - distribution/q_75
 - distribution/q_90
 - distribution/q_95
 - distribution/q_99
 - distribution/stddev
 - frequent_items/frequent_strings
 - ints/max
 - ints/min
 - type
 - types/boolean
 - types/fractional
 - types/integral
 - types/object
 - types/string

This covers pretty much all the information required about the input data, such as **cardinality**, **types**, **frequent items**, **distribution**, and **counts**.

Constraints for data quality validation

With whylogs, you can use constraints for performing data validation. This feature can be used in unit tests or CI/CD pipelines so that further errors can be avoided. You can set constraints on profile metrics, such as count, distribution, and type. The **whylogs** also supports user-defined custom metrics. You need a whylogs profile to set the constraints on data.

Setting a new constraint is simple; you need to assign values to the following:

- **name**: It can be any string to describe the constraint.
 - **condition**: It is a lambda expression.
 - **metric selector**: It is the type of metric to be measured or validated.

```
3.                                     MetricsSelector,
4.                                     MetricConstraint)
5.
6. # Function with Constraints for Data Quality Validation
7. def validate_feat(profile_view, verbose=False, viz=False):
8.     builder = ConstraintsBuilder(profile_view)
9.     # Define a constraint for data validation
10.    builder.add_constraint(MetricConstraint(
11.        name="Credit_History == 0 or == 1",
12.        condition=lambda x: x.min == 0 or x.max == 1,
13.        metric_selector=MetricsSelector(metric_name='distribution',
14.                                         column_name='Credit_History')
15.    ))
16.
17. # Build the constraints and return the report
18. constraints: Constraints = builder.build()
19.
20. if verbose:
21.     print(constraints.report())
22.
23. # return constraints.report()
24. return constraints
25. # Call the function
26. const = validate_feat(profile_view1, True)
```

In the current scenario, constraints are set to ensure that the **Credit_History** feature should contain either **0** or **1**. If it receives any value other than **0** or **1**, it will fail. You can add multiple constraints to validate multiple conditions at once. The following is the output of the preceding code:

```
[('Credit_History == 0 or == 1', 1, 0)]
```

This can be read as **[('Constraint Name', Pass, Fail)]**. It can be visualized with the whylogs visualization functionality in the notebook.

```

1. from whylogs.viz import NotebookProfileVisualizer
2. visualization = NotebookProfileVisualizer()
3. visualization.constraints_report(const, cell_height=300)

```

Figure 14.14 depicts the constraints report, that is, the output of the preceding code, which visualizes the outcome of the constraint:

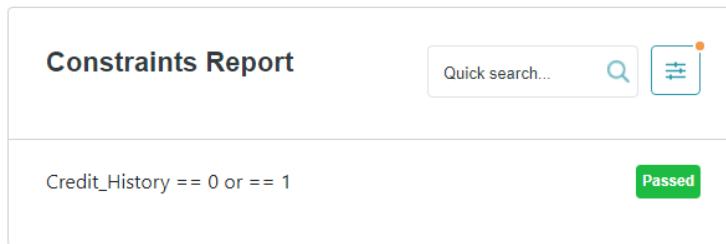


Figure 14.14: whylogs—Constraints Report

In order to validate all constraints in one go, you can simply use `validate()` on top of the constraint's outcome.

```

1. # check all constraints for passing:
2. constraints_valid = const.validate()
3. print(constraints_valid)

```

As it is validating only one condition, it will check whether it is true. As data contains **0** or **1** in the **Credit_History** feature, the test is passed and **True** is printed.

True

Next, with minimal code, you can generate a drift report. For this, you need two whylogs profiles of the data: the target profile and the reference profile.

```

1. from whylogs.viz import NotebookProfileVisualizer
2.
3. visualization = NotebookProfileVisualizer()
4. visualization.set_profiles(target_profile_view=profile_view1,
   reference_profile_view=profile_view2)
5.
6. visualization.profile_summary()
7.
8. visualization.summary_drift_report()

```

With `summary_drift_report()`, you get an easy-to-read drift report. It is an interactive feature-by-feature comparison of two profiles. This report contains the following:

- **Target**: Histogram of the target profile
- **Reference**: Histogram of the reference profile
- **p-value**: Ranges from 0 to 1
- **Total count**: Count of records or instances in the dataset
- **Mean**: Mean of the numeric feature

WhyLabs

The next step is to start uploading these profiles to the WhyLabs platform.

The first step is to sign up for an account at <https://whylabs.ai/whylabs-free-sign-up>. If you have already done this, then log in to the account. Next, create a new project from the **Project Management** section. Depending on the requirement, there are different types of projects available to choose from, such as an ML model and a data pipeline. In this scenario, choose a Classification, that is, an ML project so that you can analyze the model's performance and input-output data.

As shown in the following figure, provide the **Project Name** as `ML_monitor` (it can be updated later on) and **Type** as **Classification**:

The screenshot shows the 'Create Project' page of the WhyLabs Project Management interface. At the top, there is a dark header bar with the text 'Project Management' on the left and 'Account' on the right. Below the header, the title 'Create Project' is centered. A descriptive text states: 'Use the form to create projects for your organization:' followed by two bullet points: 'IDs will be created for each project' and 'Project names can be edited from the active project table'. There are two input fields: 'Project Name' containing 'ML_monitor' and 'Type optional' with a dropdown menu set to 'Classification'. Below these fields is a section titled 'Bulk create projects?' with a note: 'Use this option if you need to set up many projects. Project names will be appended automatically and can be edited later. For now, you can only create projects that have models.' A checkbox labeled 'Enable bulk set up' is present. At the bottom of the form is a large orange button labeled 'Create project'.

Figure 14.15: WhyLabs—Project Management

Then, you need three things to start uploading your whylogs profiles to the WhyLabs platform:

- A WhyLabs **API Key**
- The organization ID
- A different **Dataset or Model ID** for each project

Create a new access token from the **Access Tokens** tab, as shown in the following figure. Save the access token and org ID as it can be used multiple times while accessing the project in WhyLabs.

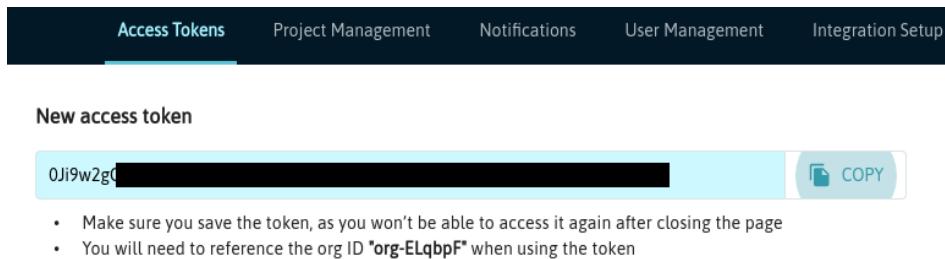


Figure 14.16: WhyLabs—Access Token

WhyLabs allow you to set the notification workflow for important events via the following:

- Emails
- Slack
- PagerDuty

The following figure shows the notification options offered by WhyLabs under the **Notifications** page:

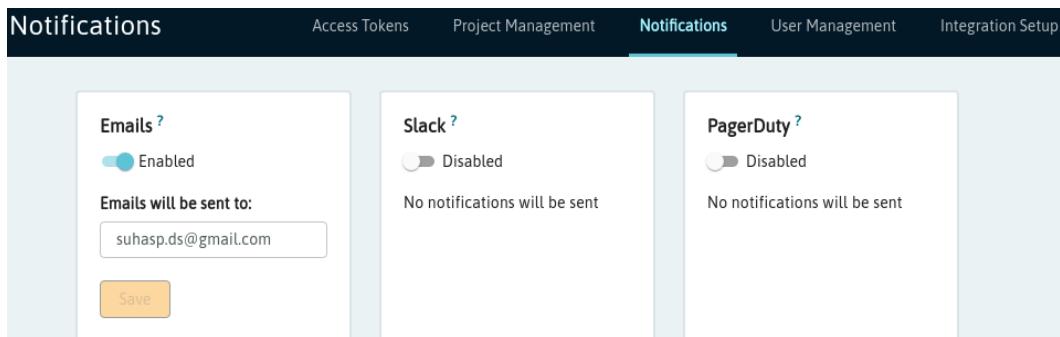


Figure 14.17: WhyLabs—Notifications

In the **User Management** tab, you can see the team members and their roles. You can also control the privilege and access of the users. In this scenario, one user with an **Admin** role is shown in the following figure:



Figure 14.18: WhyLabs–User management

Next, ensure that the target project is selected from the **Select project** dropdown. You can update the default values and selections of the monitor. You can also set the threshold, trailing window, action, severity, and so on, as shown in the following figure:

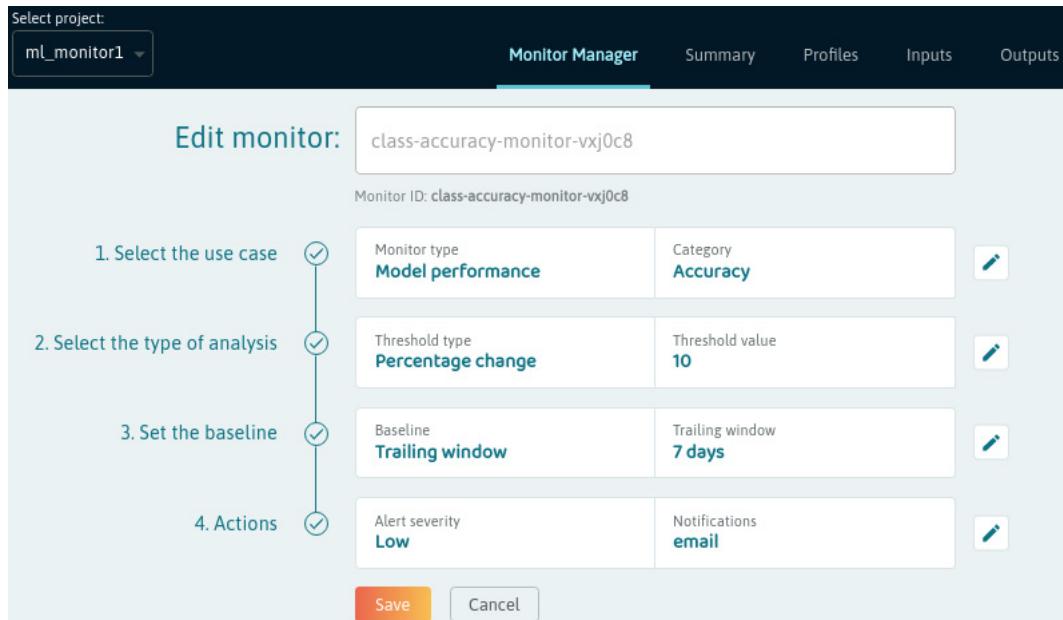


Figure 14.19: WhyLabs–Monitor Manager

WhyLabs offer preset monitors to target common detection. You can use these easy-to-use preset monitors for your project rather than setting them up from the scratch. Once you configure it, you can enable it with a single click. WhyLabs offers the following preset monitors:

- Drift
 - Data drift in all discrete model inputs compared to a trailing 7 days baseline
 - Data drift in all non-discrete model inputs compared to a trailing 7 days baseline
- Data quality
 - Missing values
 - Unique values
 - Data types
 - Percentage change
- Model performance
 - F1 score
 - Precision
 - Recall
 - Accuracy
- Integration health
 - Data availability

The following figure shows the presets available in WhyLabs to check input data quality:

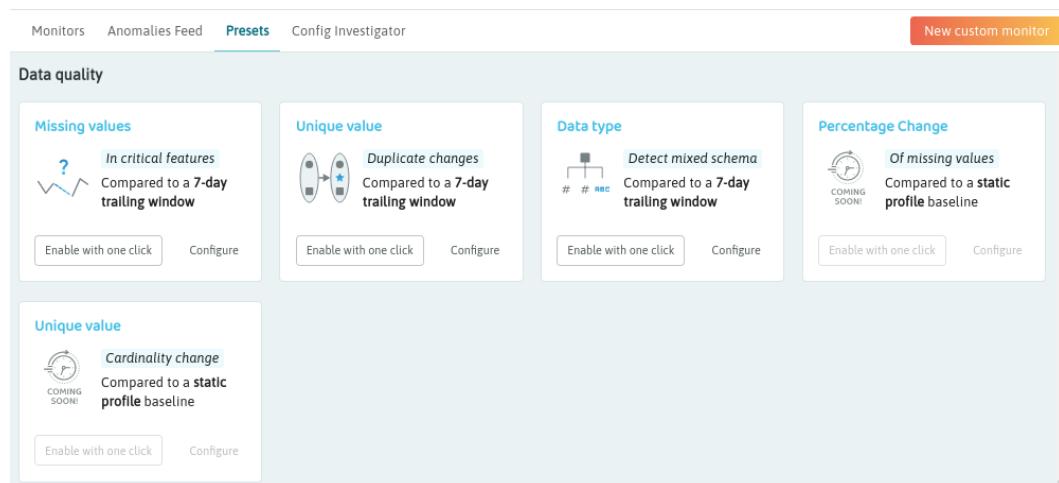


Figure 14.20: WhyLabs–Data quality presets

Now, let's install whylogs, if it's not already installed, using the following command, or simply install whylogs using **pip**:

```
pip install -q "whylogs[whylabs]"
```

First, declare variables with org id, access key, and project id. Next, load the data and build the model.

```
1. whylabs_org = 'org-ELqbpF'
2. whylabs_key = '0Ji9w2gC7q.
1QQ4pxgQBEUdx*****D0vvDrL'
3. whylabs_project = 'model-5'
4.
5. # Importing the required packages
6. import numpy as np
7. import pandas as pd
8. from sklearn.linear_model import LogisticRegression
9.
10. from sklearn import preprocessing
11. from sklearn.model_selection import train_test_split,
GridSearchCV
12. from sklearn import metrics
13. import whylogs as why
14. import os
15. from whylogs.api.writer.whylabs import WhyLabsWriter
16. import datetime as dt
17. writer = WhyLabsWriter()
18. os.environ["WHYLABS_DEFAULT_ORG_ID"] = whylabs_org # ORG-ID is
case sensitive
19. os.environ["WHYLABS_API_KEY"] = whylabs_key
20. os.environ["WHYLABS_DEFAULT_DATASET_ID"] = whylabs_project
21.
22. # Read the data
23. data = pd.read_csv("https://gist.githubusercontent.
com/suhas-ds/a318d2b1dda8d8cbf2d6990a8f0b7e8a/
raw/9b548fab0952dd12b8fdf057188038c8950428f1/loan_dataset.csv")
```

Here, the baseline model is built as the focus is on model monitoring. After building the model, it delivered 80.9% accuracy on test data. Now, create additional columns in the existing DataFrame to store model predictions and scores.

```
1. # Make Prediction  
2. y_pred = lr_model.predict(X)  
3. # Predict probability  
4. y_pred_prob = lr_model.predict_proba(X)  
5.  
6. # Get a maximum probability value of class  
7. y_pred_prob_max = [max(p) for p in y_pred_prob]  
8.  
9. # Output column name should contain the 'output' word  
10. data.rename(columns={'Loan_Status':'output_loan_status'},  
inplace=True)  
11. data['output_prediction'] = y_pred  
12. data['output_score'] = y_pred_prob_max
```

With this, you should have a DataFrame with input data and output of the predictions. Here, consider the probability of the class that is predicted by the model. In order to show the output of daily batches, divide that DataFrame into four DataFrames. Subtract 1 day in every for-loop iteration and assign that time stamp to a given dataset.

```
1. # Splitting the final DataFrame into four  
2. df1, df2, df3, df4 = np.array_split(data, 4)  
3.  
4. daily_batches = [df1, df2, df3, df4]  
5.  
6. for i, data_frame in enumerate(daily_batches):  
7.     date_time = dt.datetime.now(tz=dt.timezone.utc) -  
dt.timedelta(days=i)  
8.  
9.     df = data_frame  
10.    print("logging data for date {}".format(date_time))  
11.    results = why.log_classification_metrics(
```

```

12.         df,
13.         target_column = "output_loan_status",
14.         prediction_column = "output_prediction",
15.         score_column="output_score"
16.     )
17.
18.     profile = results.profile()
19.     profile.set_dataset_timestamp(date_time)
20.
21.     print("writing profiles to whylabs...")
22.     results.writer("whylabs").write()

```

Since this is a classification example, use `log_classification_metrics()` to log the classification model metrics and pass the following four arguments as to `why.log_classification_metrics()`:

- **input DataFrame:** Entire DataFrame `df`
- **target_column:** Actual target column
- **prediction_column:** Predicted target column
- **score_column:** score

WhyLabs will identify a column in data as an output column if the column header contains `output` text.

`WhyLabsWriter()` will publish the summary on the WhyLabs platform. Go to the **Summary** tab in WhyLabs. It offers a compact and single view for multiple widgets like **Data profiles**, **Input Health**, **Model Health**, **Model Performance**, and **alerts**, as shown in the following figure:

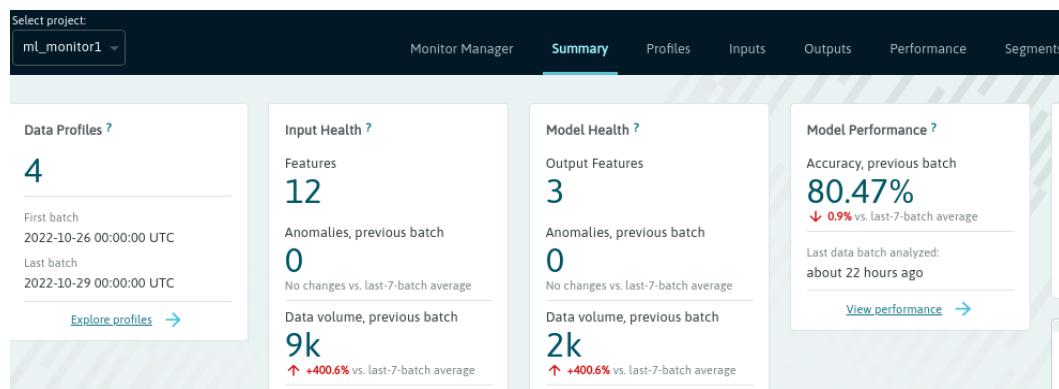


Figure 14.21: WhyLabs—Summary

Under the **Profiles** tab, an overall data summary is displayed. It includes metrics like histogram, feature count, feature type, frequency of items, and null values, as shown in the following figure:

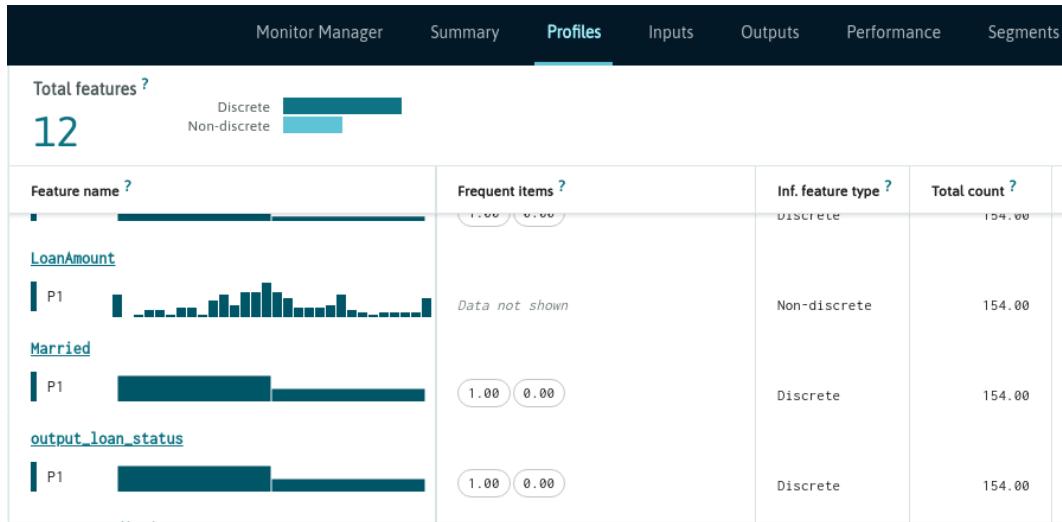


Figure 14.22: WhyLabs—Profiles

Similarly, you can see input features metrics, such as data type, total project anomalies, and drift distance, in the **Inputs** tab, as shown in the following figure:

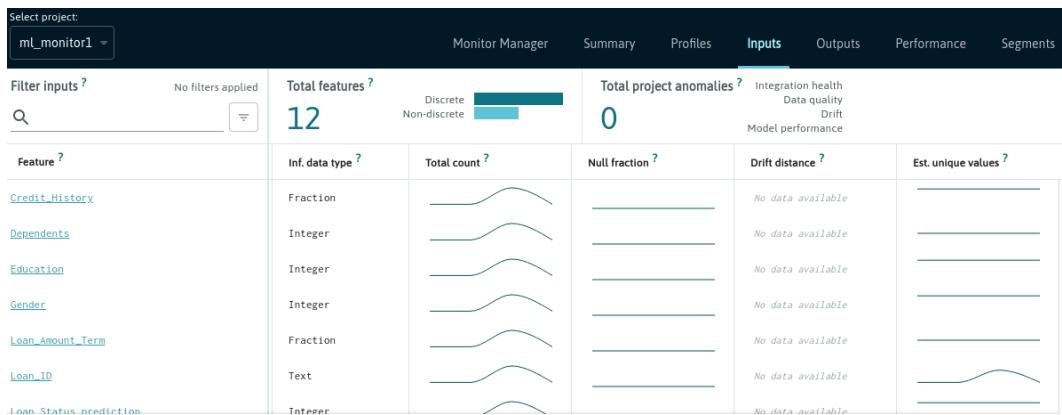


Figure 14.23: WhyLabs—Inputs

Here, you can see the total count over time, hence the line graph is displayed.

Under the **Output** tab, you can see similar metrics, but for output columns.

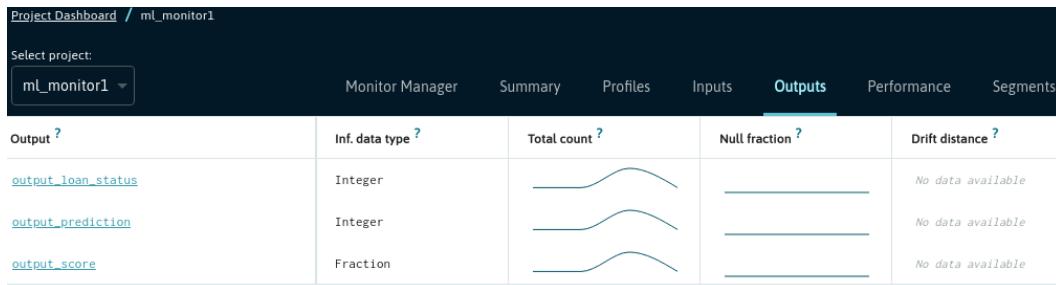


Figure 14.24: WhyLabs—Outputs

Finally, in the **Performance** tab, you can see the model metrics across different time periods. In the current scenario, you have passed the data on a daily basis. Therefore, you can see a change in accuracy on a daily basis. It also displays other model evaluation metrics for classification models, such as ROC, Precision-Recall, Confusion matrix, and f1 score.

In the following figure, you can see recent model accuracy with timestamps and plots for accuracy, ROC, Precision-Recall, and Confusion matrix:

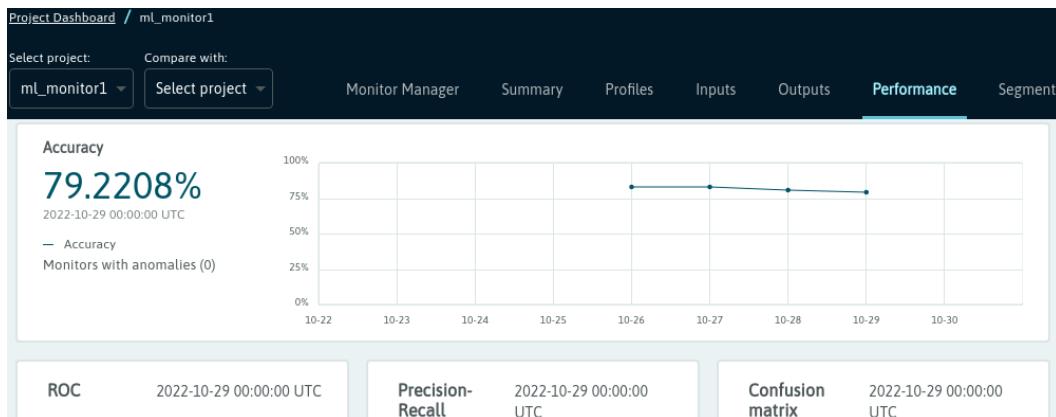


Figure 14.25: WhyLabs—Performance

The **Performance** tab also displays the total input and output data count on a daily basis, as shown in the following figure:

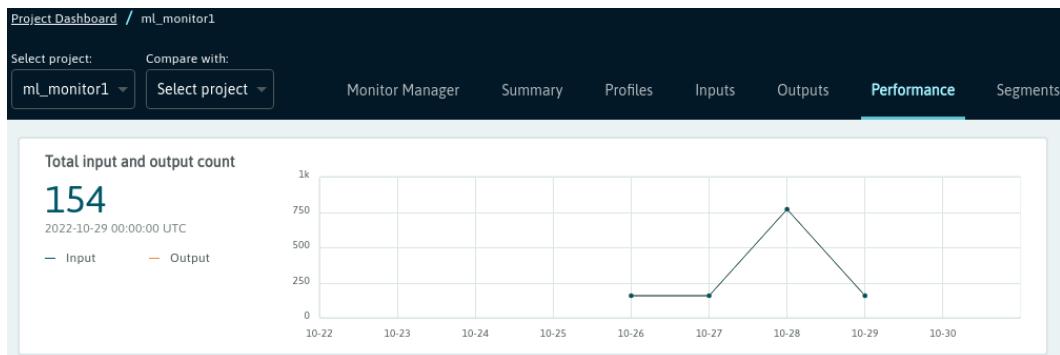


Figure 14.26: WhyLabs–Total input and output count

You can refer to the whylogs code in Google Colab notebook at the following link:

<https://colab.research.google.com/gist/suhas-ds/8bc0c895b7aa95839eaa5a5df5492a42/whylogs-and-whylabs.ipynb>

Overall, WhyLabs supports integration with existing pipelines and tools in the MLOps environment. It requires low or zero maintenance as it runs on minimal configuration, and it is dynamic, meaning it can handle changes in data properties and characteristics.

Conclusion

In this chapter, you learned the importance of monitoring in ML and the fundamentals of ML monitoring. You also studied various techniques to detect and address drift in ML, and you explored different metrics to be tracked for operational monitoring and functional monitoring. Next, you learned to integrate Prometheus and Grafana with FastAPI for operational monitoring. Further on in the chapter, you implemented an ML model monitoring solution with whylogs and WhyLabs. And finally, you covered the alert system to complete the feedback loop of the ML life cycle.

In the next chapter, you will study the steps to follow after deploying an ML model in production. You will also learn different types of ML attacks and model security.

Points to remember

- Make sure the column name contains the word **output** in order to show output or prediction data under the **Output** tab in the WhyLabs platform.

- ML monitoring refers to tracking the performance, errors, metrics, and such of the deployed model and sending alerts (when required) to ensure that the model continues performing above the acceptable threshold.
- Once an ML model is deployed into production, it is essential to monitor it in order to ensure that its performance is up to the mark and that it continues to deliver reliable output seamlessly.
- A concept shift occurs when the relationship between independent variables and dependent or target variables changes.
- Monitoring should complete the feedback loop, that is, after detecting an issue or failure, it should send a notification to the concerned team or data scientists so that they can take the necessary actions.

Multiple choice questions

1. A _____ occurs when the relationship between independent variables and dependent or target variables changes.
 - a) Concept shift
 - b) Data drift
 - c) Covariate Shift
 - d) Label drift
2. Monitoring infrastructure cost is a type of _____.
 - a) Model monitoring
 - b) Recurring monitoring
 - c) Operational monitoring
 - d) Recurring monitoring

Answers

1. a
2. c

Questions

1. What is the concept of shift?
2. What is the role of monitoring in the ML life cycle?
3. What are the different techniques to detect a drift in ML?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 15

Post-Productionizing ML Models

Introduction

Till now, you have studied different stages of the ML life cycle and ways to package, deploy, and monitor ML models. You have also learned to implement tests to ensure the integrity and working of modules. You have created local applications out of ML models that can work on Windows and Android devices; deployed models in popular cloud platforms like Microsoft Azure, GCP, and AWS; and built monitoring solutions for operational and ML model monitoring.

However, the next part is to get business value out of it. After deployment, when a new model is ready, you will have to decide whether the new model is outperforming the existing model in order to deliver reliable predictions. Model security is also essential after deploying models into production.

Structure

In this chapter, the following topics will be covered:

- Bridging the gap between the ML models and the creation of business value
- Model security
- A/B testing

- MLOps is the future

Objectives

After studying this chapter, you should be able to mitigate the risk of different types of attacks in the ML life cycle. In this chapter, you will learn the importance of converting MLOps solutions to business value and making better decisions while deploying an ML model by comparing two or more ML models with the help of A/B testing and Multi-Armed Bandit (MAB) testing. You will also study the future scope of MLOps in the industry.

Bridging the gap between the ML model and the creation of business value

Business users leverage the ML model predictions to build the business strategy and make decisions. Achieving good accuracy doesn't mean having a good business impact. Sometimes, less accurate models increase a company's revenue. You need to fill the gap between building ML solutions and creating business value out of them. Data scientists should be able to convert model predictions into easy-to-understand actions. For instance, with a classification model, you can create five buckets based on the probability score of classes, and then rank them such that the highest probability will have the highest ranking.

After deploying the ML model into the production environment with a monitoring system, the next step is to make sure the MLOps solution will benefit the business users. You can get feedback from the stakeholders, business users, or customers as they are consuming the model output. For instance, you can ask business users if any feature has become more important than the others after deploying the models so that you can give higher weightage to that feature. This way, you can tweak the model to deliver predictions that are more useful for business users.

Data scientists should be able to explain the working of the model (on a high level) to business users, as this will help them to trust the model outputs because many of them are not aware of ML terms. Business users or customers prefer to consume the output in a readable form, such as interactive dashboards or chatbots. You can also use Business Intelligence (BI) tools, such as Tableau or Power BI, to create a dashboard.

Model security

Model security is an essential part of MLOps. While processing the data, it might be important to protect the sensitive information it may contain. Attacks can take

place in the model training stage or the production stage. In this chapter, you will get familiar with different types of attacks so that you can implement adequate solutions to prevent them.

Following are the terms related to model security:

- **Poisoning:** Passing malicious data to the training process aiming to change the model output
- **Extraction:** Reconstructing a new model from the target model that will function the same as the targeted model
- **Evasion:** Attempting to change the label to a particular class by making small variations in input
- **Inference:** Figuring out whether a specific dataset was part of the training data
- **Inversion:** Getting information about training data from the trained model by reverse engineering

Adversarial attack

It is a method of generating hostile data inputs. Attackers intentionally send malicious data to the model so that the model makes incorrect predictions. As the model learns the new data patterns, this attack causes an error in the predictions of the model. This malicious input data may seem normal to humans; however, small changes in input data may have a large impact on model predictions.

Adversarial attacks can be classified into two main categories based on the goal of attackers:

- **Targeted attacks:** Attackers attempt to change the label to a particular target. For this, they can change the input data source to a specific target. It requires more time and effort.
- **Non-targeted attacks:** Attackers do not have any specific target that the model should predict. However, they change the label without any specific target. It requires less time and effort.

Attackers can use the following methods for adversarial attacks:

- **Black-box method:** In this method, attackers can send the input data to the model and get the output based on it.
- **White-box method:** In this method, the attacker is aware of almost everything about the ML model, such as training data and the weights assigned to features.

Data poisoning attack

In a data poisoning attack, attackers have access to input or source data. They change the input data in such a way that the model will make incorrect predictions that are unreliable for making any business decisions or taking any action. Attackers can add some noise to the original data to cause alterations in the prediction of the model. This attack targets the training data and modifies it intelligently.

Distributed Denial of Service attack (DDoS)

It refers to passing complex data to models that will take more time to make predictions. This type of attack limits the use of models for the users. For this, attackers can inject some malware to control the system or server.

Data privacy attack

Data privacy refers to the confidentiality of **Personally Identifiable Information (PII)** and **Personal Health Information (PHI)**. Attackers attempt to learn this type of sensitive information through this type of attack. It can be information about the model or training data. ML models like Support Vector Machines (SVMs) may leak the information, as support vectors are data points from training data.

Data privacy attacks can be broadly classified into the following categories:

- **Membership inference attack:** The goal of this attack is to determine whether input X is part of the training data. Mostly, shadow models are used in this type of privacy attack. Shadow model training uses a shadow dataset in order to imitate the target model. The output of the shadow models is then passed as an input to the meta-model. Finally, the output of the meta-model is used to extrapolate properties of training data or model. Over-fitted models are prone to data privacy attacks.
- **Input inference attack:** It is also known as model inversion or data extraction attack. It is the most common type of attack. The goal of this attack is to extract information from the training dataset by reverse engineering the model. It can also target learning the statistical properties of input data, such as probability distribution. Attackers can attempt to learn the features that are not encoded explicitly while training the model.
- **Model extraction attack:** It is also known as a parameter inference attack. The goal of this attack is to learn the hyper-parameters of the model and then reconstruct the model that behaves like the targeted model. Interestingly, over-fitted models are difficult to extract due to high prediction errors based on test data.

Mitigate the risk of model attacks

The ML pipeline can be broadly divided into two phases: the training phase and the test phase. You can address various ML model attacks based on these phases.

Training phase

A data scientist performs activities like data gathering, data cleansing, feature engineering, choosing suitable algorithms, hyperparameter tuning, and model building. Attackers target this phase via attacks like data poisoning. If a model is trained on poisoned data, you cannot rely on its predictions.

You can implement the following techniques to mitigate the risk of attacks during the training phase:

- Data encryption
- Protect the integrity of training data
- Robust statistics
- Data sanitization

Test phase

In this phase, attackers target ML models. Model extraction attacks are common among attackers. They attempt to determine whether input X is part of training data or try to steal model parameters defined during the training phase.

The following techniques can be implemented to mitigate the risk during the test phase:

- Adversarial training
- Autoencoders
- Distillation
- Ensemble techniques
- Limit the number of requests per user

You can use a Python library **Adversarial Robustness Toolbox (ART)** and a command-line tool Counterfit for ML model security. This library will defend against the most common types of ML attacks. It also supports popular ML frameworks, libraries, and data types.

A/B testing

A/B testing is widely used in marketing, website designs, and email campaigns to learn and understand user preferences. The goal of A/B testing is to increase the conversion rate, success rate, revenue, and so on.

A/B testing involves splitting the audiences or customers from the population into equal sets. These sets will route to the control version and the experimental version. The control version is the existing version, whereas the experimental version is the new or challenger version. First off, define the problem statement for the A/B test, the null hypothesis, and the alternative hypothesis for the problem statement. Next, design the experiment to track and analyze the metric, and then run and validate the experiment. After that, compare the statistics of the output. Finally, make the decision based on the results. If the A/B test is not performed correctly, then its output will be unreliable.

Data scientists perform an offline evaluation of ML models by dividing data into training and validation sets. However, the A/B test allows you to perform the online evaluation of ML models by measuring business metrics or success rates. The A/B test can be implemented when you perform operations on training data, such as scaling and normalizing or applying different algorithms, and hyper-parameters while model building. An MLOps engineer or a data scientist can deploy multiple models simultaneously to test models in production. Google Cloud (GCP) and Amazon SageMaker allow the deployment of multiple models into the production behind the same endpoint to decide which model is performing better from the business point of view.

A Multi-Armed Bandit (MAB) is an advanced version of A/B testing. It is more complex than the A/B test, as it uses ML algorithms while dynamically allocating more traffic to the better-performing version and less traffic to the underperforming version based on the data.

MLOps is the future

The field of Machine Learning is booming. Industries are making ML a critical part of the business development process, and many new challenges are being addressed by the ML system.

MLOps can handle the complexity and scalability of ML models. Hence, the demand for MLOps is increasing across the industry. There is a shortage of MLOps engineers in the industry, as MLOps is an intersection of Machine Learning and software development. Companies need to set up separate teams for MLOps engineers, or they can upskill their data scientists to execute MLOps tasks. MLOps cut costs and reduce the manual efforts of the overall ML life cycle. This allows data scientists and ML engineers to focus on other productive tasks.

MLOps is getting more popular than DevOps. Your model may be performing well in the local environment; however, if it is not reaching end users or customers, then its business impact is low. More than 85% of ML models are not deployed into the production environment. MLOps engineers can fill the gap between research

environments and the production environment. This is a highly demanding field, as data scientists alone are not enough to convert ML models to business value.

Conclusion

In this chapter, you learned the importance of model security and studied the various attacks in the ML life cycle. Building ML models and MLOps solutions is not enough; data scientists should work on delivering business value from them. A/B testing and Multi-Armed Bandit (MAB) play crucial roles while comparing existing models against newer models in order to improve the overall outcome and business metric. Finally, you learned how MLOps demand is increasing in the industry.

Points to remember

- Model security is an essential part of MLOps.
- A/B testing enables you to perform an online evaluation of ML models.
- Attackers can use the Black-box method or the White-box method while attempting an adversarial attack.
- A Python library **Adversarial Robustness Toolbox (ART)** and a command-line tool **Counterfit** can be used for ML model security.
- A **Multi-Armed Bandit (MAB)** is an advanced version of A/B testing. It is smart enough to decide which model should get more traffic by evaluating multiple models.

Multiple choice questions

1. In _____, attackers intentionally send malicious data to the model so that the model makes incorrect predictions.
 - a) phishing
 - b) an adversarial attack
 - c) shoulder surfing
 - d) piggybacking

2. _____ refers to getting information about training data from the trained model by reverse engineering.
- a) Extraction
 - b) Evasion
 - c) Inversion
 - d) Monitoring

Answers

- 1. b
- 2. c

Questions

- 1. What is a data poisoning attack?
- 2. What is A/B testing?
- 3. What are the different techniques that can be implemented to mitigate the model security risk during the training phase?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

Symbols

`__init__` method 9

A

A/B testing 423, 424

Activate Cloud Shell icon 313

Adversarial Robustness Toolbox
(ART) 423

Amazon CloudWatch Container
Insights 378

Amazon Elastic Compute Cloud
(EC2) 327

Amazon Elastic Container Registry
(ECR) 327, 336, 337

Amazon Elastic Container Service
(ECS) 327, 346

cluster 347

container 347

service 347

task 346

task definition 346, 349

Amazon Elastic Kubernetes

Service (EKS) 327

Amazon SageMaker 328

Amazon Simple Storage
Service (S3) 328

Amazon Virtual Private Cloud
(Amazon VPC) 351, 352

Amazon Web Services (AWS) 326

Anaconda

installing 2

Application Load Balancer
(ALB) 352, 358-362

array 4

AWS account

setting up 329, 330

AWS Application Load Balancer
(ALB) 327

- AWS CodeBuild 339-344
container registry access, attaching to service role 345, 346
- AWS CodeCommit 331-335
- AWS CodePipeline 371, 372
build stage, adding 374
deploy stage, adding 375, 376
monitoring 378
running 376-378
settings 372
source stage, adding 373, 374
- AWS Command-Line Interface (CLI) 330
- AWS compute services 326, 327
- AWS ECS deployment models 347
EC2 instance 347, 348
Fargate 348, 349
- AWS Fargate 328
- AWS Lambda 328
- Azure 250, 251
deployment, with GitHub Actions 251-253
infrastructure setup 263
- Azure account
setting up 251
- Azure App Service
configuring, for using GitHub Actions for CD 274-278
creating 267-270
- Azure Container Instance (ACI) 289, 292, 297
- Azure Container Registry (ACR) 252, 263
creating 264-267
- Azure Kubernetes Service (AKS) 250, 292
- Azure Machine Learning (AML) service 249, 279
- Azure Machine Learning Studio experiments 280
features 279
runs 280-289
workspace 280
- C**
- CD pipeline
configuring 292-298
- CI/CD pipeline
building 189
codebase, developing 189-196
creating, with Jenkins 202
for ML 184, 185
GitHub Actions and Heroku, using 227
Personal Access Token (PAT), creating on GitHub 196
using, Cloud Build 317, 318
using, CodePipeline 369, 370
webhook, creating on GitHub repository 197, 198
- CI/CD pipeline, with Jenkins 202
deployment-status-email 210-217
GitHub-to-container 203-205
testing 207-210
training 205-207
- CI pipeline
configuring 290-292
- class 9
- Cloud Alert manager 401
- Cloud Build 309-311
trigger, creating 318-322
- Cloud Source Repositories 304-308, 317
- Cloudwatch logs 345

- CodeBuild 339
 code editor
 installing 3
 Container Registry 311, 312
 Continuous Delivery (CD) 186
 Continuous Deployment 186
 Continuous Integration (CI) 185, 186,
 339
 Continuous Training (CT) 186, 336
 control statements and loops
 for loop 6, 7
 if...else 6
 pass statement 8
 while loop 7
 CORS (Cross-Origin Resource
 Sharing) 191, 229, 254
 CSV files
 loading 15
 saving 15
- D**
- data privacy attacks
 input inference attack 422
 membership inference attack 422
 model extraction attack 422
 data structures 4
 array 4
 dictionary 4
 list 4
 set 5
 string 5
 tuple 5
 dictionary 4
 Distributed Version Control
 System (DVCS) 18
 Docker 112
 detached mode 118
- environment, setting up 113
 Hello World example 116
 installing 113
 old versions, uninstalling 113, 114
 Docker commands 127
 Docker compose 115
 Docker container
 ML model, deploying 122-127
 running 120, 121
 Docker Engine
 installing 114
 Dockerfile
 creating 119, 120
 Docker Hub rate limit 337-339
 Docker image
 building 120
 Docker objects
 Docker container networking 118
 Docker containers 118
 Dockerfile 117
 Docker image 117
 Domain Name System (DNS) 370
 drift, in ML 386
 addressing 389
 concept drift 387, 388
 data drift 387
 data quality issues 389
 detecting techniques 388, 389
 model, rebuilding 389
 model, retaining 389
 model, tuning 389
 types 386
 DRY (Don't Repeat Yourself) principles
 41
- E**
- EC2 (Elastic Compute Cloud) 347

- ECS service
 - auto-scaling 367
 - configuration, reviewing 367-369
 - configuring 363, 364
 - creating 363
 - network, configuring 365-367
- Elastic Container Registry (ECR) 370
- environment variables
 - package, building 71
 - package, installing 71, 72
 - package usage, with example 73, 74
 - paths, adding 70
 - setting up 70
- Excel files
 - loading 15
 - saving 15
- external Alert manager 401
- F**
 - FastAPI 131-139
 - Flask 143-150
 - for loop 6, 7
 - Function as a Service (FaaS) 328
 - functions 8
- G**
 - GCP account
 - setting up 303, 304
 - GCP cloud shell
 - using, for deployment 316
 - GCP platform
 - URL 303
 - Git 17, 18
 - configuration 22
 - file, adding 23-26
 - GUI clients 20
 - installing 20
- installing, for all platforms 20
- installing, in Linux 20
- status, checking 22
- workflow 19
- Git commands
 - changes 21, 22
 - new repository 21
 - revert 22
 - setup 21
 - update 21
- GitHub 17, 19
- GitHub account
 - creating 20, 21
- GitHub Actions 225-270
 - configuration 226
 - service principal 273
- Git repository
 - initializing 22
- GNU Privacy Guard (GPG) 114
- Google Cloud Platform (GCP) 302
 - services 302, 303
- Google Kubernetes Engine (GKE) 302, 313, 314
 - deployment.yaml 314
 - service.yaml 315
- Grafana 390
- Grafana Alert manager 401
- Graphical User Interface (GUI) 157
- Gunicorn 150
- H**
 - Hello World! example 3
- Heroku 219-221
 - deployment, with Container Registry 225
- deployment, with GitHub
 - repository integration 222

pipeline flow 224
 setting up 221
 Heroku deployment
 PRODUCTION 224
 REVIEW APPS 223
 STAGING 223
 Heroku Git
 deploying with 222

I

IAM management console 345
 if...else statement 6
 iloc 14, 15
 instrument 390

J

Jenkins 187
 configuring 199-202
 installation 187-189

K

KivyMD app 173-178
 Kubernetes 312
 Kubernetes Engine API 313

L

list 4
 load balancing 352
 Application Load Balancer (ALB) 358-362
 security group 356-358
 target group 353-356
 loc 14

M

method 9
 MICE algorithm 59
 Microsoft Azure 250
 ML as a Service (MLaaS) 250

ML-based app
 building, with kivy and kivyMD 170-173
 building, with Tkinter 160-162
 MLflow 78
 components 82
 environment, setting up 79
 installing 79
 Miniconda installation 79-81
 scenarios 78, 79
 MLflow projects 94-97
 MLflow models 97-100
 MLflow registry 100, 101
 features 102, 103
 MLflow server, starting 103-108
 MySQLdb module, installing 102
 MySQL server, setting 101
 MLflow tracking 83
 artifacts 84
 log data, into run 84-94
 metric 83
 parameters 83
 source 83
 start and end time 83
 ML life cycle 30
 business impact 31
 data collection 31, 32
 data preparation 32
 feature engineering 32, 33
 model building 33
 model deployment 34
 model evaluation 34
 model testing 34
 model training 33
 monitoring 35
 optimization 35

- ML model
 - deployment, with Azure DevOps and Azure ML 278, 279
 - de-serializing 48
 - gap bridging between business value creation 420
 - post-productionizing 419
 - requirements file 47, 48
 - serializing 48
 - virtual environments 46, 47
 - ML model attacks
 - risk mitigation 423
 - ML monitoring
 - constraints, for data quality validation 404-407
 - functional monitoring 383
 - fundamentals 383-385
 - metrics 385, 386
 - metrics tracking 383, 384
 - operational monitoring 383
 - scalable integration 383
 - with whylogs and WhyLabs 402
 - MLOps 39, 40
 - automation 41
 - benefits 40
 - efficient management 41
 - features 424
 - reproducibility 41
 - tracking and feedback loop 42
 - ML packages
 - business problem 52
 - classification_v1.pkl 65
 - configuration module 55, 56
 - data 52
 - data_management.py 57
 - developing 51, 53
 - __init__.py module 54
 - MANIFEST.in file 54, 55
 - ML model, building 53
 - pipeline.py 62
 - predict.py module 63
 - preprocessors.py 58-62
 - pytest.ini 68
 - requirements.txt 64
 - sdist 70
 - setup.py 66
 - test_predict.py 68-70
 - train_pipeline.py 65
 - version 68
 - wheel 70
 - model deployment 35
 - batch predictions 35
 - mobile and edge devices 36, 37
 - real-time predictions 37
 - web service/REST API 36
 - model deployment challenges, in production environment
 - data-related challenges 38
 - portability 38
 - robustness 38, 39
 - scalability 38
 - security 39
 - team coordination 37, 38
 - model security 420, 421
 - adversarial attack 421
 - data poisoning attack 422, 423
 - data privacy attack 422
 - Distributed Denial of Service attack (DDoS) 422
 - monitoring 382, 383
 - Multi-Armed Bandit (MAB) 424
- N
- Network Load Balancer (NLB) 327

- NGINX 150-154
- NumPy 10
- NumPy array 10
reshaping 12
- O**
- object 9
- Object Oriented Programming (OOP) 9
class 9
`__init__` method 9
method 9
object 9
`self` 9
- operational monitoring
with Prometheus and Grafana 390-401
- P**
- Pandas 12
- Pandas DataFrame
using 12, 13
- pass statement 8
- Personal Access Token (PAT) 196
- Personal Health Information (PHI) 422
- Personally Identifiable Information
(PII) 422
- Platform as a Service (PaaS) 219, 250
- prod.workflow.yml 270-272
- Prometheus 390
- pytest fixtures 49
- Python
installing 2
installing, on Mac OS 2
installing, on Linux 2
installing on Windows 2
- Python 101 1
- Python app
converting, into Android app 179, 180
- Python-based Tkinter app
converting, into Windows
EXE file 167-169
- Python code
testing, with pytest 48, 49
- Python file
executing 3
- Python Package Index (PyPI) 3
- Python packaging
and dependency management 49
modular programming 50
module 50
package 50, 51
- R**
- Representational State Transfer (REST)
130
- Rest APIs 130
- risk mitigation, ML model attacks
test phase 423
training phase 423
- S**
- Service Principal (automatic) 282
- set 5
- src directory
`docker-compose.yml` 234, 259
`Dockerfile` 234, 259
`.gitignore` 229, 253
`main.py` 229-232, 254-257
`production.yml` 239-243
`pytest.ini` 235, 260
`requirements.txt` 258
`requirements.txt` file 233
`runtime.txt` 235, 260
`start.sh` 235, 260
`test.py` 235, 236, 260, 261
`tox.ini` 237, 262, 263

workflow.yml 238
workflow.yml workflow 244-246

Staging 241

Streamlit 139-143

string 5

T

task definition 349, 350
task, running with 350
Tiny Machine Learning (TinyML) 36
Tkinter 158
 features 160
 Hello World app 159
 ML-based app, building 160-162
Tkinter app 163-167
Tool Command Language (TCL) 158
tuple 5

V

Virtual Machines (VM) 250
Virtual Private Cloud (VPC) 303

W

Web Server Gateway Interface (WSGI)
 143
while loop 7
WhyLabs 402, 407
 access token 408
 features 402
 Notifications page 408
 output tab 415, 416
 profile tab 414
 project management 407, 408
 user management 409-413
whylogs 403, 404

Machine Learning in Production

DESCRIPTION

'Machine Learning in Production' is an attempt to decipher the path to a remarkable career in the field of MLOps. It is a comprehensive guide to managing the machine learning lifecycle from development to deployment, outlining ways in which you can deploy ML models in production.

It starts off with fundamental concepts, an introduction to the ML lifecycle and MLOps, followed by comprehensive step-by-step instructions on how to develop a package for ML code from scratch that can be installed using pip. It then covers MLflow for ML life cycle management, CI/CD pipelines, and shows how to deploy ML applications on Azure, GCP, and AWS. Furthermore, it provides guidance on how to convert Python applications into Android and Windows apps, as well as how to develop ML web apps. Finally, it covers monitoring, the critical topic of machine learning attacks, and A/B testing.

With this book, you can easily build and deploy machine learning solutions in production.

KEY FEATURES

- Explore several ways to build and deploy ML models in production using an automated CI/CD pipeline.
- Develop and convert ML apps into Android and Windows apps.
- Learn how to implement ML model deployment on popular cloud platforms, including Azure, GCP, and AWS.

WHAT YOU WILL LEARN

- Master the Machine Learning lifecycle with MLOps.
- Learn best practices for managing ML models at scale.
- Streamline your ML workflow with MLFlow.
- Implement monitoring solutions using whylogs, WhyLabs, Grafana, and Prometheus.
- Use Docker and Kubernetes for ML deployment.

WHO THIS BOOK IS FOR

Whether you are a Data scientist, ML engineer, DevOps professional, Software engineer, or Cloud architect, this book will help you get your machine learning models into production quickly and efficiently.



BPB PUBLICATIONS

www.bpbonline.com

ISBN 978-93-5551-810-1



9 789355 518101