





Join a community dedicated to learning open source

The Red Hat® Learning Community is a collaborative platform for users to accelerate open source skill adoption while working with Red Hat products and experts.



Network with tens of thousands of community members



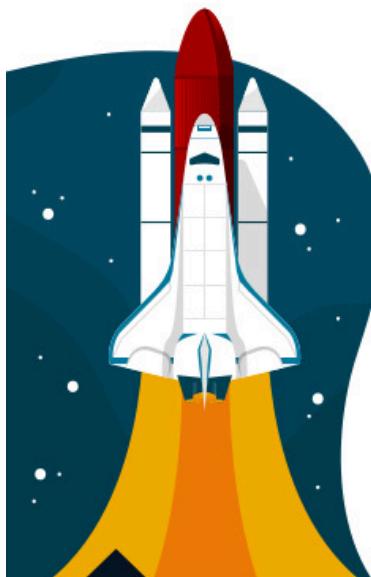
Engage in thousands of active conversations and posts



Join and interact with hundreds of certified training instructors



Unlock badges as you participate and accomplish new goals



This knowledge-sharing platform creates a space where learners can connect, ask questions, and collaborate with other open source practitioners.

Access free Red Hat training videos

Discover the latest Red Hat Training and Certification news

Connect with your instructor - and your classmates - before, after, and during your training course.

Join peers as you explore Red Hat products

Join the conversation learn.redhat.com



Copyright © 2020 Red Hat, Inc. Red Hat, Red Hat Enterprise Linux, the Red Hat logo, and Ansible are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Developing and Deploying AI/ML Applications on Red Hat OpenShift AI



RHOAI 2.8 AI267

Developing and Deploying AI/ML Applications on Red Hat

OpenShift AI

Edition 3 20240801

Publication date 20240801

Authors: Alejandro Serna-Borja Lencina, Guy Bianco IV,

Jaime Ramírez Castillo, Rafael Ruiz Hernández

Course Architect: Ravishankar Srinivasan

DevOps Engineers: Richard Allred, Zachary Guterman

Editor: Sam Ffrench

© 2024 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are © 2024 Red Hat, Inc.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of Red Hat, Inc.

This instructional program, including all material provided herein, is supplied without any guarantees from Red Hat, Inc. Red Hat, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details contained herein.

If you believe Red Hat training materials are being used, copied, or otherwise improperly distributed, please send email to training@redhat.com [mailto:training@redhat.com] or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

Red Hat, Red Hat Enterprise Linux, the Red Hat logo, JBoss, OpenShift, Fedora, Hibernate, Ansible, RHCA, RHCE, RHCSA, Ceph, and Gluster are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle American, Inc. and/or its affiliates.

XFS® is a registered trademark of Hewlett Packard Enterprise Development LP or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is a trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack word mark and the Square O Design, together or apart, are trademarks or registered trademarks of OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. Red Hat, Inc. is not affiliated with, endorsed by, or sponsored by the OpenStack Foundation or the OpenStack community.

All other trademarks are the property of their respective owners.

Contributors: Jaime Yagüe Mayans, Ted Singdalson, Pablo Solar Vilariño

Document Conventions	ix
Admonitions	ix
Inclusive Language	x
Introduction	xi
Developing and Deploying AI/ML Applications on Red Hat OpenShift AI	xi
Orientation to the Classroom Environment	xiii
Performing Lab Exercises	xix
1. Introduction to Red Hat OpenShift AI	1
Introduction to Red Hat OpenShift AI	2
Architecture	7
Summary	13
2. Data Science Projects	15
Data Science Projects	16
Guided Exercise: Data Science Projects	20
Workbenches	23
Guided Exercise: Workbenches	36
Data Connections	39
Guided Exercise: Data Connections	42
Summary	46
3. Jupyter Notebooks	47
Introduction to Jupyter Notebooks	48
Guided Exercise: Introduction to Jupyter Notebooks	56
Collaboration with Jupyter Notebooks	58
Guided Exercise: Collaboration with Jupyter Notebooks	68
Summary	82
4. Installing Red Hat OpenShift AI	83
Red Hat OpenShift AI Installation	84
Guided Exercise: Red Hat OpenShift AI Installation	94
Summary	99
5. Manage Users and Resources	101
Users, Projects, and Permissions	102
Guided Exercise: Users, Projects, and Permissions	107
Managing Resources	110
Guided Exercise: Managing Resources	115
Summary	120
6. Custom Notebook Images	121
Create and Import a Custom Notebook Image	122
Guided Exercise: Create a Custom Notebook Image	126
Summary	130
7. Introduction to Machine Learning	131
Machine Learning Concepts	132
Quiz: Machine Learning Concepts	139
Machine Learning Workflow	143
Summary	146
8. Training Models	147
Training ML Models	148
Guided Exercise: Training ML Models	162
Training Models with Custom Workbenches	164
Guided Exercise: Training Models with Custom Workbenches	168
Summary	171

9. Enhancing Model Training with RHOAI	173
Inspecting Workbench Resources	174
Guided Exercise: Inspecting Workbench Resources	178
Scaling Data Loading	180
Guided Exercise: Scaling Data Loading	183
Monitoring the Training Process	185
Guided Exercise: Monitoring the Training Process	195
Software Engineering Principles for Data Science	197
Guided Exercise: Software Engineering Principles for Data Science	202
Summary	205
10. Introduction to Model Serving	207
Concepts and Key Aspects of Serving AI Models	208
Quiz: Concepts and Key Aspects of Serving AI Models	211
Saving Trained Machine Learning Models	215
Guided Exercise: Saving Trained Machine Learning Models	224
Serving Models as Stand-Alone Applications	226
Guided Exercise: Serving Models as Stand-alone Applications	229
Summary	233
11. Model Serving in Red Hat OpenShift AI	235
Using Model Servers to Deploy Models	236
Guided Exercise: Using Model Servers to Deploy Models	244
Consuming the Model Serving API	248
Guided Exercise: Consuming the Model Serving API	254
Creating and Using Model Servers	258
Guided Exercise: Creating and Using Model Servers	261
Summary	266
12. Introduction to Data Science Pipelines	267
Concepts of Data Science Pipelines	268
Quiz: Concepts of Data Science Pipelines	274
Creating a Pipeline Server	278
Guided Exercise: Creating a Pipeline Server	281
Summary	283
13. Elyra Pipelines	285
Creating Pipelines with Elyra	286
Guided Exercise: Creating Pipelines with Elyra	295
Summary	302
14. Kubeflow Pipelines	303
Creating Pipelines with Kubeflow Pipelines	304
Guided Exercise: Creating Pipelines with Kubeflow Pipelines	310
Summary	316

Document Conventions

This section describes various conventions and practices that are used throughout all Red Hat Training courses.

Admonitions

Red Hat Training courses use the following admonitions:



References

These describe where to find external documentation that is relevant to a subject.



Note

Notes are tips, shortcuts, or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on something that makes your life easier.



Important

Important sections provide details of information that is easily missed: configuration changes that apply only to the current session, or services that need restarting before an update applies. Ignoring these admonitions will not cause data loss, but might cause irritation and frustration.



Warning

Do not ignore warnings. Ignoring these admonitions will most likely cause data loss.

Inclusive Language

Red Hat Training is currently reviewing its use of language in various areas to help remove any potentially offensive terms. This is an ongoing process and requires alignment with the products and services that are covered in Red Hat Training courses. Red Hat appreciates your patience during this process.

Introduction

Developing and Deploying AI/ML Applications on Red Hat OpenShift AI

Developing and Deploying AI/ML Applications on Red Hat OpenShift AI (AI267) provides students with the fundamental knowledge about using Red Hat OpenShift for developing and deploying AI/ML applications. This course helps students build core skills for using Red Hat OpenShift AI to train, develop and deploy machine learning models through hands-on experience.

This course is based on Red Hat OpenShift® 4.14, and Red Hat OpenShift AI 2.8.

Course Objectives

- Describe the main features, architecture, and components of Red Hat OpenShift AI, and the challenges that it solves
- Install Red Hat OpenShift AI by using the web console and the CLI
- Upgrade Red Hat OpenShift AI components
- Manage Red Hat OpenShift AI users, and controlling access
- Create and configure custom notebook images
- Describe basic ML concepts, and use Red Hat OpenShift AI to implement model training workflows
- Describe the concepts and components required to export, share, and serve trained machine learning models
- Describe the concepts and components required to automate AI/ML workflows
- Create CI/CD pipelines by using JupyterLab
- Create CI/CD pipelines by using Python code

Audience

- Data scientists and AI practitioners who want to use Red Hat OpenShift AI to build and train ML models
- Developers who want to build and integrate AI/ML enabled applications
- MLOps engineers responsible for installing, configuring, deploying, and monitoring AI/ML applications on Red Hat OpenShift AI

Prerequisites

- Experience in Python development is required, or completion of the *Python Programming with Red Hat* (AD141) course

Introduction

- Experience in Red Hat OpenShift is recommended, or completion of the *Red Hat OpenShift Developer II: Building and Deploying Cloud-native Applications (DO288)* course
- Basic experience in the AI, data science, and machine learning fields is recommended

Orientation to the Classroom Environment

In this course, the main computer system that is used for hands-on learning activities (exercises) is **workstation**.

The **workstation** machine has a standard user account, **student** with **student** as the password. Although no exercise in this course requires you to log in as **root**, if you must, then the **root** password on the **workstation** machine is **redhat**.

From the **workstation** machine, you type the **oc** commands to manage the Red Hat OpenShift Container Platform (RHOC) cluster, which comes preinstalled as part of your classroom environment. The preinstalled RHOC cluster runs the Red Hat OpenShift AI (RHOAI) product.

Also from the **workstation** machine, you run the commands that are required to complete the exercises for this course.

If exercises require you to open a web browser to access any application or website, then you must use the graphical console of the **workstation** machine and use the Firefox web browser from there.



Note

During the initial start of your classroom environment, the RHOC cluster takes more time to become fully available. The **lab** command at the beginning of each exercise checks and waits as required.

If you try to access your cluster by using either the **oc** command or the web console without first running a **lab** command, then your cluster might not yet be available. If the cluster is not available, then wait a few minutes and try again.

Log in to OpenShift from the Shell

To access your RHOC cluster from the **workstation** machine, use <https://api.ocp4.example.com:6443> as the API URL, for example:

```
[student@workstation ~]$ oc login -u admin -p redhatocp \
https://api.ocp4.example.com:6443
```

Besides the **admin** user, who has cluster administrator privileges, your OpenShift cluster also provides a **developer** user, with **developer** as the password, with no special privileges.

Accessing the OpenShift Web Console

If you prefer to use the OpenShift web console, then open a Firefox web browser on your **workstation** machine and access the following URL:

<https://console-openshift-console.apps.ocp4.example.com>

Click **htpasswd_provider** and provide the login credentials for either the **admin** or the **developer** user.

Accessing the RHOAI Web Console

To use the RHOAI web console, open a Firefox web browser on your **workstation** machine and access the following URL:

<https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com>

Click **htpasswd_provider** and provide the login credentials for the **admin** user.

The Classroom Environment

Every student gets a complete remote classroom environment. As part of that environment, every student gets a dedicated RHOCP cluster.

This environment contains all the necessary resources for the course.

The classroom environment runs entirely as virtual machines in a large Red Hat OpenStack Platform cluster, which is shared among many students.

Red Hat Training maintains many OpenStack clusters, in data centers across the globe, to provide lower latency to students from many countries.

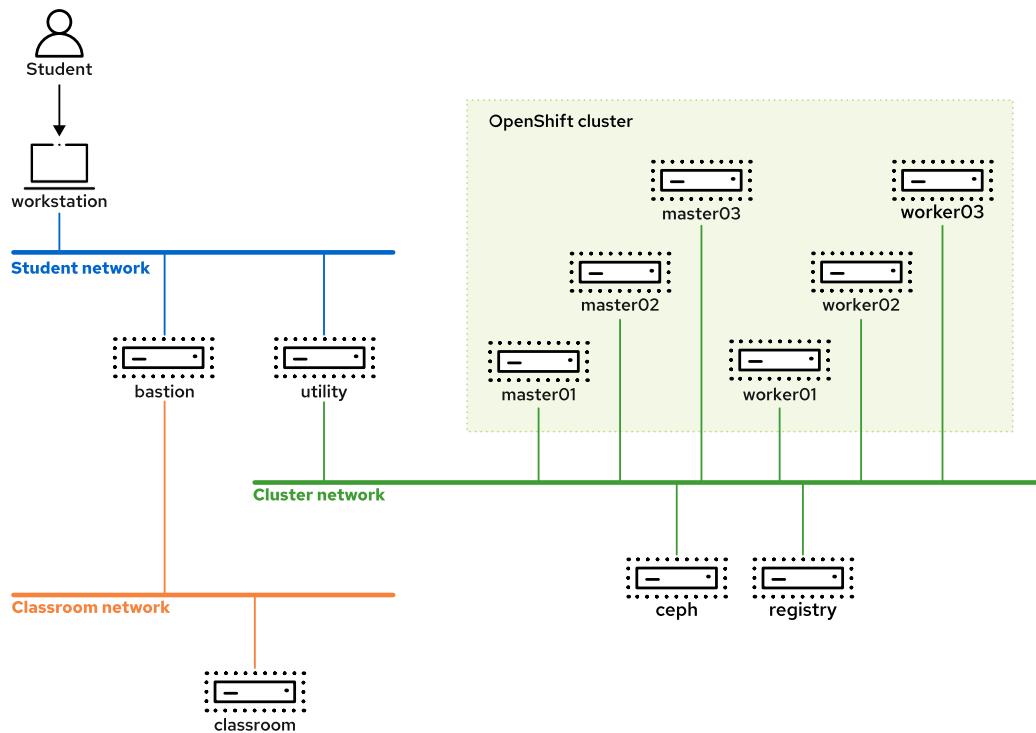


Figure 0.1: AI267 classroom architecture

All machines on the Student, Classroom, and Cluster networks run Red Hat Enterprise Linux 9 (RHEL 9), except those machines that are nodes of the OpenShift cluster. These cluster nodes run RHEL CoreOS.

The **bastion**, **utility**, **registry**, and **classroom** systems must always be running. These systems provide infrastructure services that the classroom environment and its OpenShift cluster require. For most exercises, you are not expected to interact with any of these services directly.

Introduction

Usually, the `lab` commands from the exercises access these machines to set up your environment for the exercise, and require no further action from you.

All systems in the *Student* network are in the `lab.example.com` DNS domain, and all systems in the *Classroom* network are in the `example.com` DNS domain.

The `masterXX` and `workerXX` systems are nodes of the RHOC cluster that is part of your classroom environment.

All systems in the *Cluster* network are in the `ocp4.example.com` DNS domain.

Classroom Machines

Machine name	IP addresses	Role
<code>workstation.lab.example.com</code>	172.25.250.9	Graphical workstation for system administration
<code>classroom.example.com</code>	172.25.254.254	Router to link the Classroom network to the internet
<code>bastion.lab.example.com</code>	172.25.250.254	Router to link the Student network to the Classroom network
<code>utility.lab.example.com</code>	172.25.250.253	Router to link the Student and Cluster networks
<code>ceph.ocp4.example.com</code>	192.168.50.30	Server with a Red Hat Storage Ceph preinstalled cluster
<code>registry.ocp4.example.com</code>	192.168.50.50	Server with Quay and GitLab
<code>master01.ocp4.example.com</code>	192.168.50.10	Control plane node
<code>master02.ocp4.example.com</code>	192.168.50.11	Control plane node
<code>master03.ocp4.example.com</code>	192.168.50.12	Control plane node
<code>worker01.ocp4.example.com</code>	192.168.50.13	Compute node
<code>worker02.ocp4.example.com</code>	192.168.50.14	Compute node
<code>worker03.ocp4.example.com</code>	192.168.50.15	Compute node

The Dedicated OpenShift Cluster

The Red Hat OpenShift Container Platform 4 cluster inside the classroom environment is preinstalled by using the Pre-existing Infrastructure installation method. All nodes are treated as bare metal servers, even though they are virtual machines in an OpenStack cluster.

OpenShift cloud provider integration capabilities are not enabled, and some features that depend on that integration, such as machine sets and autoscaling of cluster nodes, are not available.

Your OpenShift cluster is in the state from running the OpenShift installer with default configurations, except for some day-2 customizations:

Introduction

- The cluster provides a default storage class that is backed by a Network File System (NFS) storage provider.
- The cluster provides storage classes that are backed by Ceph, to provide S3 buckets.

Controlling Your Systems

You are assigned remote computers in a Red Hat Online Learning (ROLE) classroom. Self-paced courses are accessed through a web application that is hosted at rol.redhat.com [<http://rol.redhat.com>]. Log in to this site with your Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

The virtual machines in your classroom environment are controlled through web page interface controls. The state of each classroom virtual machine is displayed on the **Lab Environment** tab.

The screenshot shows a web-based interface for managing a lab environment. At the top, there are tabs for 'Table of Contents', 'Course', and 'Lab Environment'. Below the tabs, there's a section titled '▶ Lab Controls' with instructions: 'Click CREATE to build all of the virtual machines needed for the classroom lab environment. This may take several minutes to complete. Once created the environment can then be stopped and restarted to pause your experience.' It also notes that deleting the lab will remove all virtual machines and lose progress. Below this is a button bar with 'DELETE' (red), 'STOP' (blue), and an information icon. The main area is a table listing five virtual machines:

Virtual Machine	State	Action	Open Console
bastion	active	ACTION -	OPEN CONSOLE
classroom	active	ACTION -	OPEN CONSOLE
servera	building	ACTION -	OPEN CONSOLE
serverb	building	ACTION -	OPEN CONSOLE
workstation	active	ACTION -	OPEN CONSOLE

Figure 0.2: An example course lab environment management page

Machine States

Virtual machine state	Description
building	The virtual machine is being created.
active	The virtual machine is running and available. If the virtual machine just started, then it might still be starting services.
stopped	The virtual machine is shut down. On starting, the virtual machine boots into the same state that it was in before shutdown. The disk state is preserved.

Classroom Actions

Button or action	Description
CREATE	Create the ROLE classroom. Creates and starts all the necessary virtual machines for this classroom.
CREATING	The ROLE classroom virtual machines are being created. Creation can take several minutes to complete.
DELETE	Delete the ROLE classroom. Deletes all virtual machines in the classroom. All saved work on those systems' disks is lost.
START	Start all virtual machines in the classroom.
STARTING	All virtual machines in the classroom are starting.
STOP	Stop all virtual machines in the classroom.

Machine Actions

Button or action	Description
OPEN CONSOLE	Connect to the system console of the virtual machine in a new browser tab. You can log in directly to the virtual machine and run commands, when required. Normally, log in to the workstation virtual machine only, and from there, use ssh to connect to the other virtual machines.
ACTION > Start	Start (power on) the virtual machine.
ACTION > Shutdown	Gracefully shut down the virtual machine, and preserve disk contents.
ACTION > Power Off	Forcefully shut down the virtual machine, and still preserve disk contents. This action is equivalent to removing the power from a physical machine.
ACTION > Reset	Forcefully shut down the virtual machine and reset the associated storage to its initial state. All saved work on that system's disks is lost.

At the start of an exercise, if you are instructed to reset a single virtual machine node, then click **ACTION > Reset** for that specific virtual machine only.

At the start of an exercise, if you are instructed to reset all virtual machines, then click **ACTION > Reset** on every virtual machine in the list.

To return the classroom environment to its original state at the start of the course, you can click **DELETE** to remove the entire classroom environment. After the lab is deleted, click **CREATE** to provision a new set of classroom systems.

**Warning**

The **DELETE** operation cannot be undone. All completed work in the classroom environment is lost.

The Auto-stop and Auto-destroy Timers

The Red Hat Online Learning enrollment entitles you to a set allotment of computer time. To help to conserve your allotted time, the ROLE classroom uses timers, which shut down or delete the classroom environment when the appropriate timer expires.

To adjust the timers, locate the two + buttons at the bottom of the course management page. Click the auto-stop + button to add another hour to the auto-stop timer. Click the auto-destroy + button to add another day to the auto-destroy timer. Auto-stop has a maximum of 11 hours, and auto-destroy has a maximum of 14 days. Be careful to keep the timers set while you are working, so that your environment is not unexpectedly shut down. Be careful not to set the timers unnecessarily high, which could waste your subscription time allotment.

Performing Lab Exercises

You might see the following lab activity types in this course:

- A *guided exercise* is a hands-on practice exercise that follows a presentation section. It walks you through a procedure to perform, step by step.
- A *quiz* is typically used when checking knowledge-based learning, or when a hands-on activity is impractical for some other reason.
- An *end-of-chapter lab* is a gradable hands-on activity to help you to check your learning. You work through a set of high-level steps, based on the guided exercises in that chapter, but the steps do not walk you through every command. A solution is provided with a step-by-step walk-through.
- A *comprehensive review lab* is used at the end of the course. It is also a gradable hands-on activity, and might cover content from the entire course. You work through a specification of what to accomplish in the activity, without receiving the specific steps to do so. Again, a solution is provided with a step-by-step walk-through that meets the specification.

To prepare your lab environment at the start of each hands-on activity, run the `lab start` command with a specified activity name from the activity's instructions. Likewise, at the end of each hands-on activity, run the `lab finish` command with that same activity name to clean up after the activity. Each hands-on activity has a unique name within a course.

The syntax for running an exercise script is as follows:

```
[student@workstation ~]$ lab action exercise
```



Important

This training is a compilation of other Red Hat OpenShift AI courses (AI26*). It uses a single lab environment and multiple lab script libraries.

Each exercise requires a specific library, which corresponds to the original course that the exercise belongs to. To ensure that you use the correct library, every exercise starts with a `lab install -u AI26*` command.

Additionally, note that the `workstation` machine resets the installed library to `AI262` on startup.

The `action` is a choice of `start`, `grade`, or `finish`. All exercises support `start` and `finish`. Only end-of-chapter labs and comprehensive review labs support `grade`.

start

The `start` action verifies the required resources to begin an exercise. It might include configuring settings, creating resources, checking prerequisite services, and verifying necessary outcomes from previous exercises. You can perform an exercise at any time, even without performing preceding exercises.

Introduction

grade

For gradable activities, the `grade` action directs the `lab` command to evaluate your work, and shows a list of grading criteria with a PASS or FAIL status for each. To achieve a PASS status for all criteria, fix the failures and rerun the `grade` action.

finish

The `finish` action cleans up resources that were configured during the exercise. You can perform an exercise as many times as you want.

The `lab` command supports tab completion. For example, to list all exercises that you can start, enter `lab start` and then press the Tab key twice.

Chapter 1

Introduction to Red Hat OpenShift AI

Goal

Identify the main features of Red Hat OpenShift AI and describe the architecture and components of Red Hat AI.

Sections

- Introduction to Red Hat OpenShift AI
- Architecture

Introduction to Red Hat OpenShift AI

Objectives

- Describe the main features of Red Hat OpenShift AI.

A Brief Introduction to Artificial Intelligence

Artificial Intelligence (AI) is an area of computer science that is focused on solving tasks that usually require human reasoning or cognitive abilities, such as the following examples:

- Spam detection
- Recommendation systems
- Speech recognition
- Text sentiment analysis
- Image recognition
- Disease diagnosis
- Fraud detection
- Price prediction

More recently, with the advent of large language and diffusion models, AI systems have evolved to generate text, images, audio, and act as expressive chat bots, among other abilities.

The AI field encompasses multiple, broad disciplines that are closely related and typically overlap significantly. The developments and research on AI not only require expertise from technical fields, such as mathematics or engineering, but also knowledge and counseling from other areas such as life sciences, philosophy, or sociology. The following diagram is a high-level, noncomprehensive, opinionated overview of the AI field, and the common problems and methods that each subdiscipline tackles.

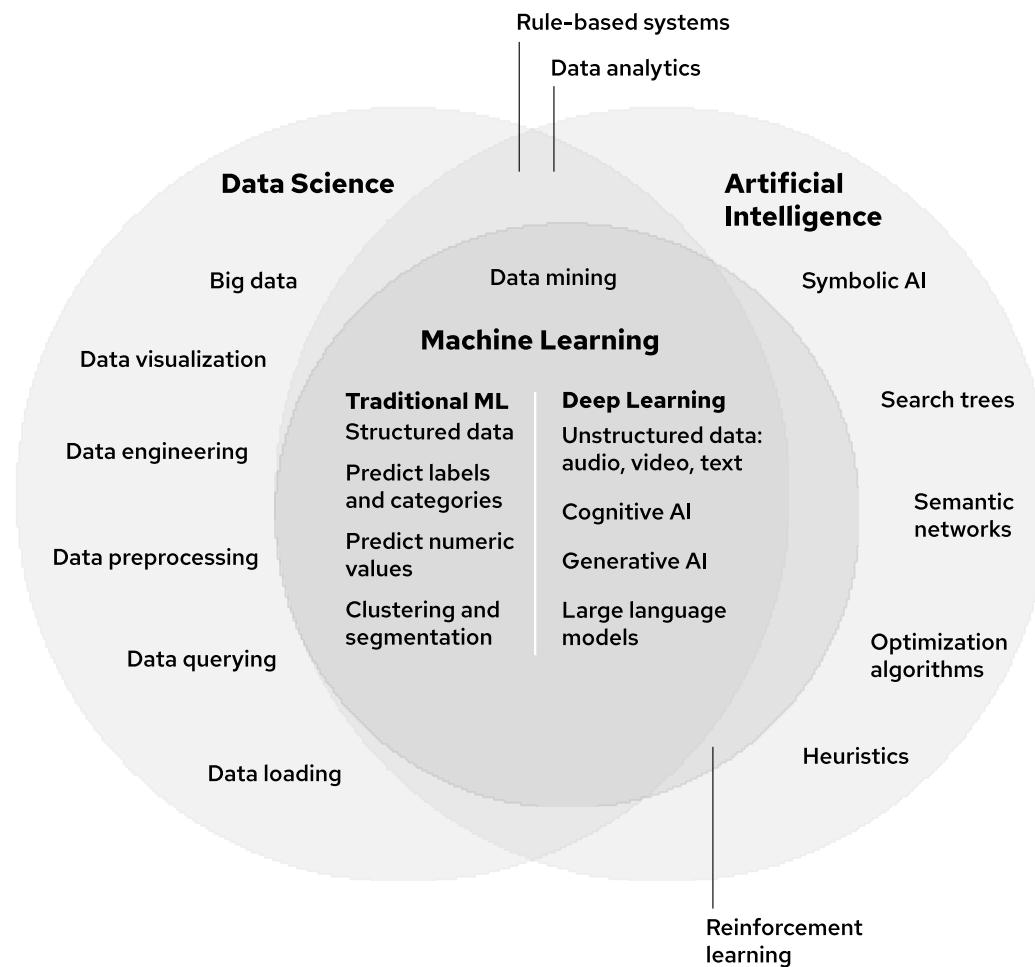


Figure 1.1: A high-level, opinionated overview of the AI field

Data Science

Data Science (DS) is a broad and interdisciplinary field that focuses on analyzing data and extracting knowledge from that data. Data science is essential in the creation of AI applications. The reason for this dependency is that, currently, AI is data-centric. Most of the techniques used to create intelligent applications require data and rely on learning patterns from such data.

Data science challenges range from capturing and gathering data, to querying, describing, analyzing and visualizing the data to extract knowledge and answer questions. Being such a diverse field, data science integrates techniques from many other areas, such as algebra, calculus, statistics, probability, data and software engineering, and Big Data.

Machine Learning (ML)

Machine Learning (ML) is a subset of data science that aims to train computers to learn patterns from data and produce generalized responses. These trained, intelligent systems are called *ML models*, or *models*. In general terms, an ML model is a mathematical function that takes a set of inputs and produces an output. When you train a model, you are in fact applying an algorithm that learns the operations that this function must compute to produce the desired output. The function learns these operations by using the patterns discovered in the training data set.

Generalization is one of the most important features of ML models, and refers to the ability of a model to produce satisfactory responses on unseen data. For example, assume that you train a model to recognize cats by using a training data set of 1000 images. You can consider that the model generalizes well if it detects cats that were not initially included in the initial 1000 training images.

The ML techniques are diverse and suitable for different purposes.

- Traditional ML techniques provide best results in structured data, such as tabular data sets. These methods are suitable for predicting numerical values, categorizing, or segmenting data automatically. For example, you can apply these techniques to predict house pricing.
- Deep learning techniques are powerful and provide best results in artificial cognition and generation, typically by using unstructured data such as text, audio, and images.

**Note**

Artificial cognition is a term that covers the subset of AI applications that simulate human cognitive abilities, such as image and speech recognition, or language understanding.

- *Reinforcement Learning (RL)* is arguably more specialized, and not always considered a DS technique. Rather than using existing data for training, RL simulates actions and outcomes for learning. A notable example of the use of Reinforcement Learning is training models to play video games or classic strategy games, such as Chess and Go.

Deep Learning

Deep Learning (DL) is a type of machine learning, characterized by the use of large neural networks. A *Neural Network* is an ML model used to learn complex patterns in data, by using an architecture inspired in biological neurons.

Research on deep learning and its use in generative and cognitive AI has been particularly prolific in the last decade. These methods have proven to be effective with unstructured data, such as text, images, sound, or videos. Examples of deep learning are artificial vision, speech recognition, and natural language processing. The latest trends in DL include techniques such as *Diffusion Models* and *Large Language Models (LLMs)*, which are capable of generating multimedia content, as well as producing well structured text narratives, writing code, and maintaining coherent conversations.

One downside of deep learning models is their size and complexity. These models typically need large data sets for training, and have high memory and compute requirements. The use of Graphical Processing Units (GPU) is common in deep learning to speed up computations.

Non-ML Techniques

Despite the popularity of deep learning and the suitability of ML techniques for creating intelligent applications, many AI applications rely on other forms of AI. These alternative forms do not extract knowledge from data, and instead use other techniques to perform intelligent tasks. For example, some of these models use conditional logic rules gathered from human experts to make explainable decisions. Another example is the use of shortest-path optimization algorithms in navigation systems.

**Note**

For a more detailed explanation of Machine Learning, refer to the *Creating Machine Learning Models with Red Hat OpenShift AI* (AI264) course.

Red Hat OpenShift AI

One of the main difficulties that AI projects face is the gap between data science and engineering. Engineering and operations teams are not always familiar with the complexities of AI, data science, and machine learning. Likewise, data scientists sometimes lack the engineering resources and expertise to build, deploy, and monitor models in modern cloud environments.

Machine learning operations (MLOps) help organizations solve this challenge. MLOps is inspired by DevOps principles and brings many of the benefits of DevOps to AI projects, streamlining the whole lifecycle of ML models.

Red Hat OpenShift AI (RHOAI) is a platform for data scientists, AI practitioners, developers, machine learning engineers, and operations teams to prototype, build, deploy, and monitor AI models. By using an MLOps perspective, RHOAI helps teams overcome key challenges:

Operationalize the ML lifecycle

Many organizations struggle to push their AI models beyond the training stages because the teams that train these models lack the tools to deploy, deliver, and maintain AI-based systems in production. To solve this problem, RHOAI provides tools such as stable working environments, continuous integration and continuous deployment (CI/CD) pipelines, and model serving frameworks.

With these tools, users can operationalize their ML workflows and achieve reproducibility, boost automation, better collaboration, cost savings, and improved governance and compliance.

Complex ML working environments

AI practitioners often struggle to integrate and maintain the myriad of tools, libraries, and versions required for data science and machine learning, including the significant effort required to configure GPU support and drivers and keep those updated. RHOAI provides data scientists with ready-to-use working environments that are preconfigured with standard AI/ML tools and libraries.

RHOAI Key Features

RHOAI turns Red Hat OpenShift into a full-fledged AI platform, which includes the common tools and features that the AI ecosystem requires. The following features are some key capabilities of RHOAI:

Jupyter notebooks

Arguably one of the most popular tools for data science, research, and experimentation. Jupyter Notebooks provide interactive and executable live documents and shells on the web browser. A notebook can contain executable code, rich text, and multimedia content.

Portable working environments

Each working environment is available as a container image. Users can create these environments based on the specific container image that they need. RHOAI provides preconfigured images, but users can create custom ones.

GPU-ready environments

RHOAI provides stable working environments that are preconfigured for *Compute Unified Device Architecture (CUDA)* and NVIDIA GPUs, which are typically required for deep learning.

Cloud-first Model Serving

RHOAI ships with model servers, such as OpenVino, which enable users to serve their models right after training them, without having to develop specific servers or APIs.

Pipelines

Users can automate the typical ML workflows of data gathering, model training, evaluation, and deployment, instead of performing these tasks manually.

Model monitoring

You can monitor the performance of AI models in production with Prometheus and Grafana dashboards.



References

Understanding AI/ML use cases

<https://www.redhat.com/en/topics/cloud-computing/ai-ml-use-cases>

What is machine learning?

<https://www.redhat.com/en/topics/ai/what-is-machine-learning>

What is deep learning?

<https://www.redhat.com/en/topics/ai/what-is-deep-learning>

What is generative AI?

<https://www.redhat.com/en/topics/ai/what-is-generative-ai>

What is MLOps?

<https://www.redhat.com/en/topics/ai/what-is-mlops>

For more information about the RHOAI features, refer to the *Product features* chapter in the Red Hat OpenShift AI Self-managed 2.8 *Introduction to Red Hat OpenShift AI* documentation at

https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/introduction_to_red_hat_openshift_ai/index#product-features_intro

Architecture

Objectives

- Describe the architecture and components of Red Hat OpenShift AI.

Architecture of Red Hat OpenShift AI

Red Hat OpenShift AI (RHOAI) is designed as an MLOps modular platform for managing the complete lifecycle of AI/ML projects in cloud environments. Available as an OpenShift operator, RHOAI gathers together a portfolio of tools for handling data science and machine learning processes, automating ML workflows, and serving and monitoring models. The core components of RHOAI use open source projects, such as Project Jupyter and Kubeflow.

Additionally, Red Hat actively partners with leading AI vendors, such as Anaconda, Intel, IBM, and NVIDIA, to make their tools available for data scientists as RHOAI components. You can optionally activate these components to expand the RHOAI capabilities. Depending on the component that you want to activate, you might need to install an operator, use the Red Hat Ecosystem, or add a workbench image.

The following diagram depicts the general architecture of a RHOAI deployment, including the most important concepts and components:

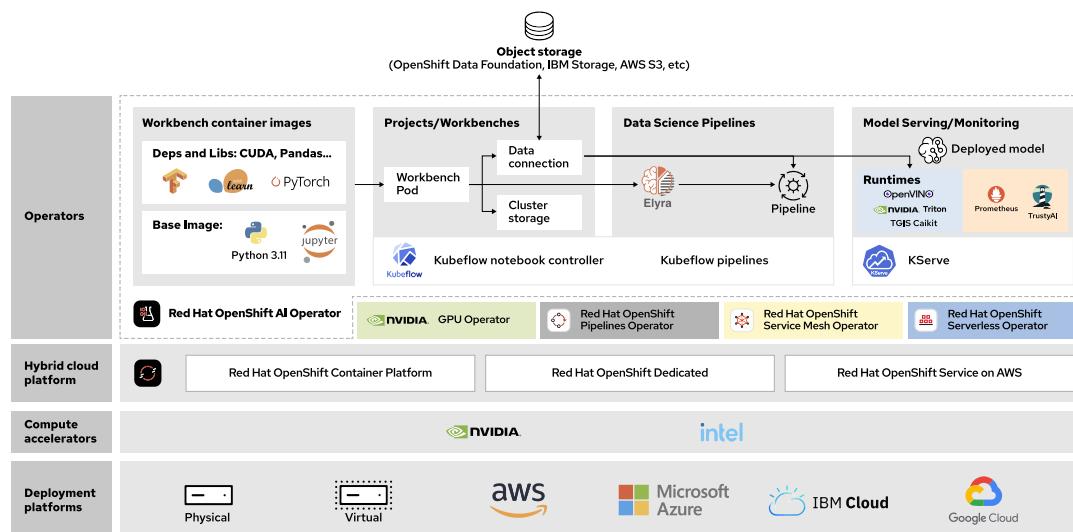


Figure 1.2: High-level RHOAI component architecture



Note

The TrustyAI feature for model monitoring has been removed, starting with RHOAI 2.7.

Workbenches

In RHOAI, a *workbench* is a containerized, cloud-native working environment for data scientists. Workbenches run as OpenShift pods and are designed for machine learning and data science. To this end, workbenches include the JupyterLab notebook execution environment, standard data science libraries, such as TensorFlow, and GPU acceleration capabilities, among other features. RHOAI ships with preconfigured workbench images, which provide data scientists with stable working environments with little setup. The implementation of workbenches relies on the Kubeflow upstream project.



Note

The RHOAI GPU capabilities require the NVIDIA GPU operator.

Workbench images

A *workbench image* is a container image that RHOAI uses to create workbenches. RHOAI includes a number of images ready to use for multiple common data science stacks. Each image contains a different set of libraries and versions. `Minimal Python`, `Standard Data Science`, `PyTorch`, and `TensorFlow` are examples of default images included in RHOAI.



Note

You can create custom workbench images and import them into RHOAI, if the default workbenches do not fit your needs.

Cluster storage

Workbenches run as containers, and containers are designed to be stateless. However, users must be able to retain their work after logging out or if the workbench restarts. To make workbenches stateful, RHOAI uses *cluster storage*, which is a Persistent Volume Claim (PVC) mounted in a specific directory of the workbench container. Every workbench includes, at least, one cluster storage mounted in the root directory of the workbench.

Data Connections

In RHOAI, a *data connection* is a set of configuration parameters for connecting workbenches to S3-compatible storage services. When you associate a data connection to a workbench, RHOAI injects the values of the data connection as environment variables into the workbench.

Data Science Pipelines

In RHOAI, a *data science pipeline* is a workflow that executes scripts or Jupyter notebooks in an automated way. Data Science Pipelines are a key feature in an AI project, considering the experimental nature of data science and machine learning. Pipelines can help data scientists operationalize the execution of their experiments and organize the results. For example, a typical pipeline might automate an experiment run by gathering data, cleaning data, training a model, evaluating the model, and uploading the model and its evaluation report to S3.



Note

To use Data Science Pipelines, you must first install the Red Hat OpenShift Pipelines operator in your cluster.

For a more detailed explanation of pipelines, refer to the *Automation using Data Science Pipelines (AI266)* course.

The implementation of Data Science Pipelines relies on the Tekton and Kubeflow upstream projects. You can create pipelines visually with the Elyra GUI.

Model

An *AI model*, *machine learning model*, or just *model* is the main artifact that results from the training phase of a machine learning workflow. Each data science or ML framework uses its own format to export models. An exported model is typically saved as one or multiple files. In most cases, you must use the same framework for both training and using the model. Often, developers also need an HTTP API, or similar, to expose the model to the general public.

Model Serving

In RHOAI, a *model server* is a component that automatically deploys models. The model server uses a data connection to download the model files from an S3 model store. After downloading the model server files into the container, the model server exposes the model via standard REST or gRPC APIs. These APIs are often called *inference APIs* as they provide the inferences captured in the model.

RHOAI uses KServe as the Model Serving platform, and supports model runtimes such as OpenVINO, Triton, Text Generation Inference Server (TGIS) and Caikit.



Note

To deploy a model by using Model Serving, first you must save the model in an S3 model store.

Additionally, certain model serving capabilities require the Red Hat Service Mesh and Serverless operators.

For a more detailed explanation of Model Serving, refer to the *Deploying Machine Learning Models with Red Hat OpenShift AI* (AI265) course.

Model Monitoring

Administrations and engineers might want to observe Prometheus metrics such as the CPU or memory use of your model server. Data scientists can inspect potential drifts in the predictions and decisions of the deployed models.

Open Data Hub

RHOAI is based on the *Open Data Hub (ODH)* upstream project. ODH is an open source, community-driven AI platform, designed for the hybrid cloud, and built on popular AI open source tools. RHOAI is a subset of ODH, delivered both as a cloud service or self-managed. RHOAI can be optionally deployed with additional independent software vendor (ISV) features. The following table summarizes the major differences between RHOAI and ODH:

	Open Data Hub	Red Hat OpenShift AI
Type	Community project	Red Hat product
Offerings	Self-managed	Self-managed, managed cloud service
Components	Community projects	Community projects, Red Hat Ecosystem Catalog items, Partner integrations

	Open Data Hub	Red Hat OpenShift AI
GUI	Open Data Science UI	UI Integrated in Red Hat OpenShift

Machine Learning Workflow in RHOAI

RHOAI augments Red Hat OpenShift to be an MLOps platform. RHOAI provides data scientists, engineers, administrators, and developers with a consistent and collaborative platform for AI/ML projects. The following diagram describes how you can streamline a common AI/ML workflow in RHOAI:

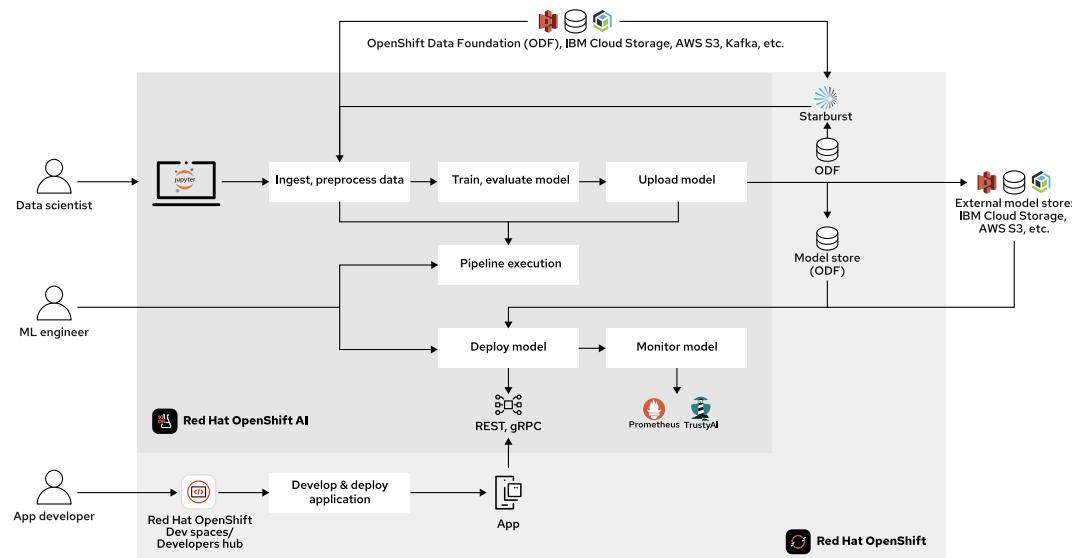


Figure 1.3: Example of an ML workflow in Red Hat OpenShift

Ingest data

Data scientists start their work by loading data into a workbench. They can, for example, upload files from their local machines to workbenches. Likewise, they can download data into a workbench by pulling files from S3-compatible storage, making queries to databases, or reading data streams.

One common way to load data is by using the Pandas Python library, which RHOAI includes in most of the default workbenches. Pandas offers functions to load data from different sources, such as CSV, JSON, or SQL. You can also install your preferred libraries or include them in custom workbenches.

Additionally, RHOAI can add more data ingestion features by integrating with certified ISV ecosystem apps from the Red Hat Marketplace. Starburst and Cloudera are examples of these integrations.

Preprocess and explore data

Data scientists clean, prepare, analyze, and visualize the data. The data scientist carries out such tasks within a workbench, often by using Jupyter notebooks and libraries such as Matplotlib, Pandas, and NumPy. Most of the default workbench images include these libraries.

Train model

Data scientists use the preprocessed, clean data to train the model in a workbench. RHOAI provides workbench images for training models with commonly used libraries, such as

TensorFlow, PyTorch, and Scikit-learn. Some of these images also include ready-to-use GPU support, to enable faster training.

Evaluate model

After training, data scientists evaluate the performance of the trained model on testing and validation subsets of the data. Data scientists use these reserved portions of the data to verify that the trained models generalize and perform well on unseen samples.

Typically, data scientists repeat the *Preprocessing-Training-Evaluation* cycle until they are satisfied with the model evaluation metrics.

Export and upload model

When the model is trained and evaluated, the data scientists use the data connection to upload the files to a model store. This step might also involve the conversion of the model into a suitable format for serving.

Pipeline execution

In collaboration with data scientists, ML engineers build data science pipelines to automate the execution of Preprocessing-Training-Evaluation cycles. This activity is particularly important as it automates training models, running experiments consistently, and saving the results of each experiment in a model store.

Serve model

ML engineers can create model servers that fetch a trained model from the model store, and expose the model via a REST or a gRPC interface. The model server uses a data connection to download the model files.

Monitor model

Users can monitor the performance of a model in production to detect drift. For example, ML engineers might want to focus on aspects such as speed or scalability, and data scientists on model accuracy. When the performance model degrades, the team can decide how to react, for example, by triggering a training cycle with updated data.

Develop and deploy applications

After the model is available in production, application developers can develop and deploy intelligent applications that use the deployed models, by consuming the REST/gRPC interfaces of the model server.



References

Open Data Hub

<https://opendatahub.io/>

Kubeflow

<https://www.kubeflow.org/>

KServe

<https://kserve.github.io/website>

Top 5 ways developers and data scientists can collaborate

<https://www.redhat.com/en/resources/top-5-ways-developers-collaborate-checklist>

For more information, refer to the *Overview of OpenShift AI* chapter in the Red Hat Red Hat OpenShift AI Self-managed 2.8 *Introduction to Red Hat OpenShift AI* documentation at

https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/introduction_to_red_hat_openshift_ai/index#overview-of-openshift-ai_intro

Summary

- Data Science and Machine Learning (ML) are currently the most important disciplines for the creation of AI applications.
- MLOps is inspired by DevOps and helps organizations operationalize and streamline the lifecycle of ML projects.
- Red Hat OpenShift AI (RHOAI) augments Red Hat OpenShift to be a complete MLOps platform.
- RHOAI provides data scientists with stable workbenches that require little setup.
- RHOAI provides administrators and ML engineers with tools for ML workflows automation, model serving, and model monitoring.
- RHOAI is available as an operator and offers optional integrations with partners and independent software vendors.

Chapter 2

Data Science Projects

Goal

Organize code and configuration by using data science projects, workbenches, and data connections

Sections

- Data Science Projects (and Guided Exercise)
- Workbenches (and Guided Exercise)
- Data Connections (and Guided Exercise)



Data Science Projects

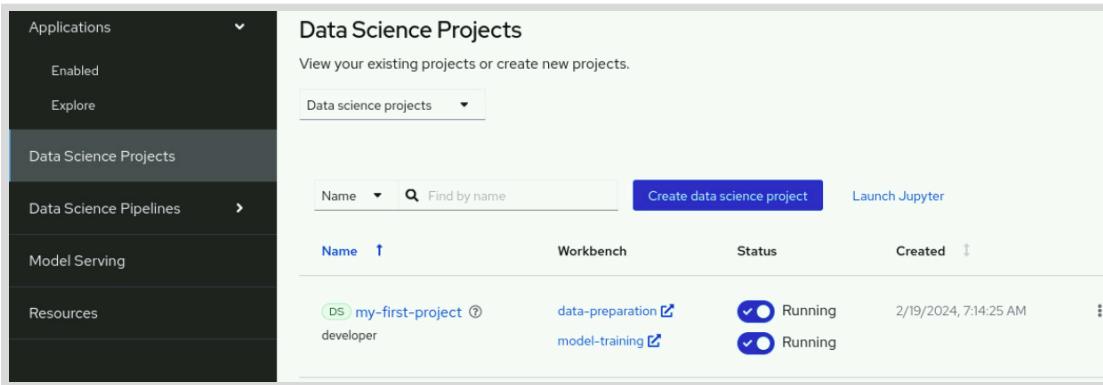
Objectives

- Organize code and configuration by using data science projects.

Working with Data Science Projects

In Red Hat OpenShift AI (RHOAI), a data science project is the preferred way to organize resources when working on an AI/ML application. Similarly to how you use projects in OpenShift for other workloads, you can use data science projects to organize the different components that you need for your AI applications.

You can create and manage data science projects by using the RHOAI dashboard.



The screenshot shows the Data Science Projects page in the RHOAI dashboard. On the left, there is a sidebar with a dropdown menu set to 'Enabled' and options like 'Explore', 'Data Science Projects' (which is selected and highlighted in blue), 'Data Science Pipelines', 'Model Serving', and 'Resources'. The main content area has a title 'Data Science Projects' and a sub-instruction 'View your existing projects or create new projects.' Below this is a search bar with 'Data science projects' typed in. A button 'Create data science project' is visible. The main table lists existing projects: 'my-first-project' (developer) with workbenches 'data-preparation' and 'model-training', both marked as 'Running'. The table includes columns for Name, Workbench, Status, and Created. A 'Launch Jupyter' link is also present.

Figure 2.1: Data Science Projects page

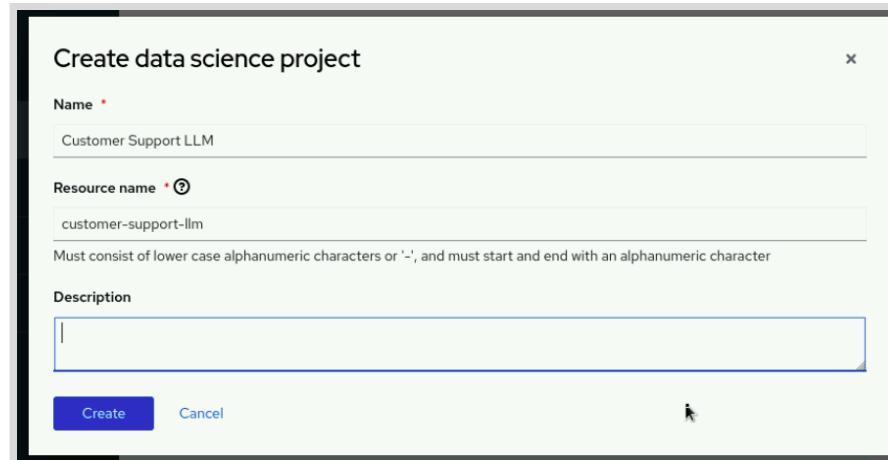
The preceding image shows an example project that contains several workbenches.

Creating a Data Science Project

To create a data science project from the RHOAI dashboard, click **Create data science project** in the Data Science Projects page.

In the modal window that opens, you can enter the project name, the name of the OpenShift project resource, and an optional description. When you enter the project name, the creation modal window uses that name to generate a name that follows OpenShift project naming rules.

You can change the OpenShift resource name to anything you like, but you cannot change this value after project creation.



The screenshot shows a 'Create data science project' dialog box. It has fields for 'Name' (Customer Support LLM), 'Resource name' (customer-support-llm), and 'Description' (empty). There are 'Create' and 'Cancel' buttons at the bottom.

Figure 2.2: Data science project creation

After you create the project, RHOAI opens the project dashboard page.

From this page, you can manage the project components and the permissions.

 **Note**

By default, only the project owner and administrator users can access a data science project. Permissions for data science projects are covered in the *Red Hat OpenShift AI Administration (AI263)* course.

Deleting a Data Science Project

To delete a data science project from the RHOAI dashboard, you can locate the project to delete, click its : button, and then click **Delete project**. Finally, type the project name to confirm the deletion.

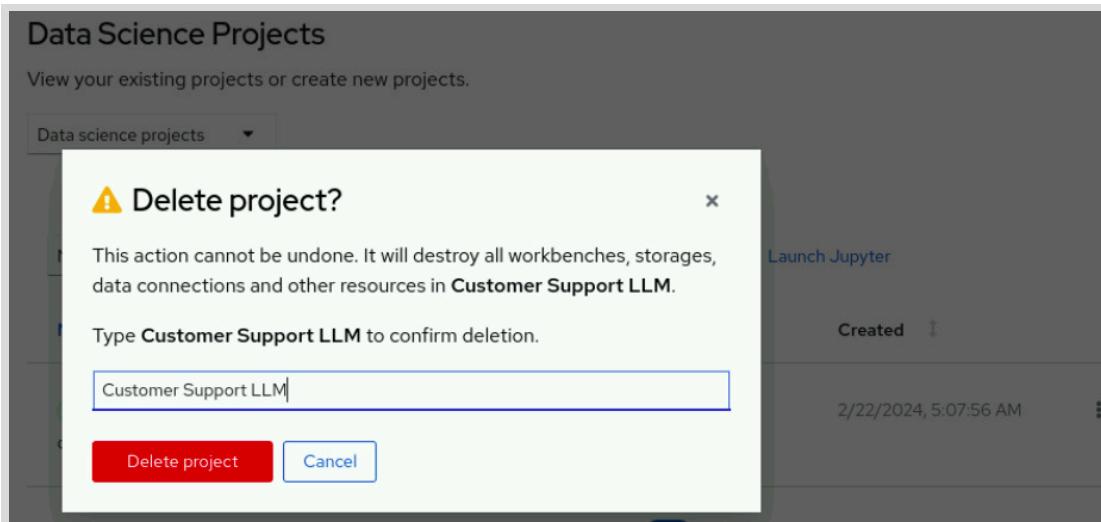


Figure 2.3: Data science project deletion

The Data Science Project Resource

Internally, a data science project is an OpenShift project that contains specific labels to identify the project as a RHOAI-specific project. The following is an example project YAML definition:

```
kind: Project  
apiVersion: project.openshift.io/v1  
metadata:  
  name: my-ai-project  
  labels:  
    kubernetes.io/metadata.name: my-ai-project  
    modelmesh-enabled: 'true'  
    opendatahub.io/dashboard: 'true' ❸  
    ...output omitted...
```

- ❶ A data science project is still an OpenShift project.
- ❷ The name uniquely identifies the data science project.
- ❸ This label identifies the OpenShift project as a data science project.



Note

When creating a data science project outside of the RHOAI dashboard, be sure to include the `opendatahub.io/dashboard: 'true'` label in the project definition. Otherwise, the project is not available in the RHOAI dashboard.

Data Science Project Components

A data science project can include the following components:

- Workbenches
- Cluster storage
- Data connections
- Pipelines
- Models and Model servers



Note

Data Science Projects are OpenShift projects, so they can contain, and usually do, other resources such as ConfigMaps, Secrets, or Services.

You can manage the project components by selecting a project from the **Data Science Projects** section of the RHOAI dashboard. After clicking one of the projects, you can access the data science main page where you have access to the project components, permissions, and settings.

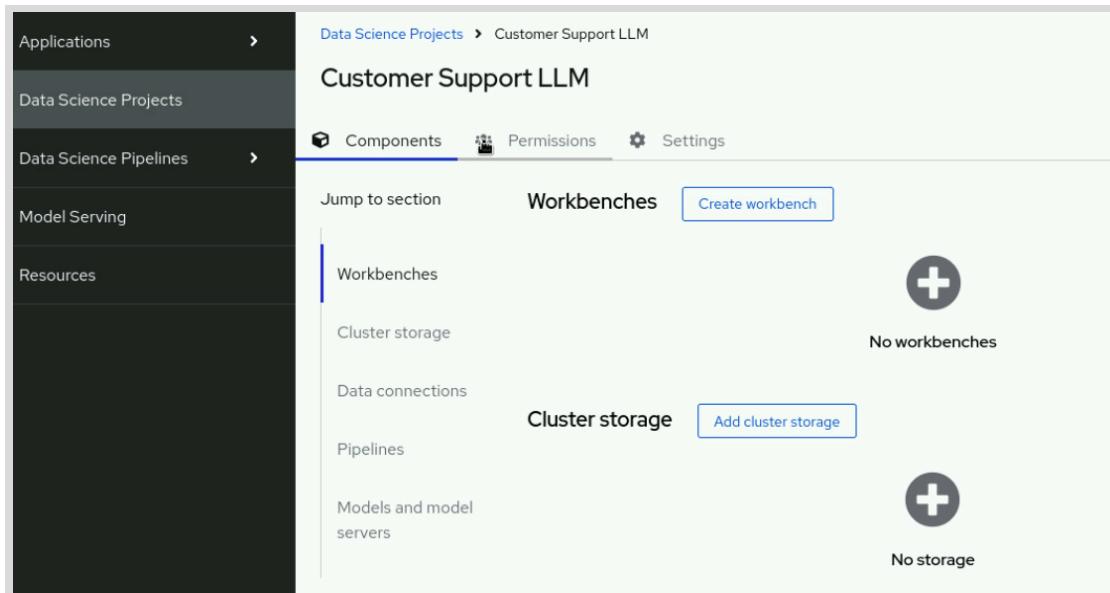


Figure 2.4: RHOAI data science project main page

These components will be covered later in the course.



References

For more information about data science projects, refer to the *Using Data Science Projects* section in the *Working on Data Science Projects* chapter in the Red Hat OpenShift AI Self-managed 2.8 *Working on Data Science Projects* documentation at
https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/working_on_data_science_projects/index#using_data_science_projects

► Guided Exercise

Data Science Projects

Organize code and configuration by using data science projects.

Outcomes

- Create a data science project by using the Red Hat OpenShift AI (RHOAI) dashboard.
- List and find projects available to your user.
- Delete a project.

Before You Begin

As the student user on the **workstation** machine, use the **lab** command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI262 && \
lab start projects-concept
```

Instructions

- ▶ 1. Open the RHOAI dashboard.
 - 1.1. Open the web browser and navigate to <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com>.
 - 1.2. If RHOAI requests you to log in, then click **Log in with OpenShift**. Next, click **htpasswd_provider**. Enter **developer** as both the username and the password.
 - ▶ 2. Create a data science project named **project-ds**.
 - 2.1. Click **Data Science Projects** in the left navigation pane, and then click **Create data science project**.
 - 2.2. In the modal window that opens, enter the **project-ds** as the project name and click **Create**.
- After you create the project, RHOAI opens the project dashboard page. From this page you can manage the project components and the permissions.

- 3. Examine the data science project labels in the `project-ds` project.
- 3.1. Open a terminal, and run the `oc login` command by using `developer` as both the username and password.

```
[student@workstation ~]$ oc login -u developer -p developer \
https://api.ocp4.example.com:6443
Login successful.

...output omitted...
```

- 3.2. Use the `oc describe` command to get the project metadata. Verify that the `opendatahub.io/dashboard=true` label shows in the output.

```
[student@workstation ~]$ oc describe project project-ds
Name:                  project-ds
Created:                2 hours ago
Labels:                 kubernetes.io/metadata.name=project-ds
                        opendatahub.io/dashboard=true
...output omitted...
```

- 4. Find the `project-ds` in the projects available for the `developer` user.
- 4.1. Click **Data Science Projects** to navigate to the view with the list of projects that your user can access.
 - 4.2. Verify that you see `project-ds` in the results table. Optionally you can start typing `project-ds` in the **Find by name** search box to filter the projects.

- 5. Delete the `project-ds` project.
- 5.1. Locate the `project-ds` project in the projects table, click the `:` button, and click **Delete project**.
 - 5.2. Type the project name in the modal window to confirm the deletion.
 - 5.3. Verify that the `project-ds` does not show in the **Data Science Projects** view.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish projects-concept
```

Workbenches

Objectives

- Set up workbenches and create Jupyter Notebooks.

Workbench Definition

In Red Hat OpenShift AI (RHOAI), a *workbench* is a containerized working environment that runs as a pod. Workbenches typically include a collection of common AI/ML libraries and a JupyterLab server. When you create a workbench, RHOAI creates all the OpenShift resources for the workbench to work. For example, RHOAI exposes the JupyterLab URL through an OpenShift route so that you can use and interact with the workbench via the web browser.

Workbenches are stateful, meaning that you can stop and restart them without losing your work.

Data science projects can include more than one workbench. This is useful when you need to run different experiments for a project that requires different tooling. For example, you might want to train a simple model with Scikit-learn and another model with PyTorch. Also, you could have one workbench for data exploration and another workbench for model training.

Creating Workbenches

To create a workbench from the RHOAI dashboard, navigate to your project, and then click **Create workbench** to open the workbench creation page.

Create workbench

Configure properties for your workbench.

Jump to section	Name * my-first-workbench
Name and description	Description <input type="text"/>
Notebook image	Notebook image
Deployment size	Image selection * Minimal Python
Environment variables	Version selection * 2023.2 (Recommended)
Cluster storage	<small>Hover an option to learn more information about the packages included.</small>
Data connections	View package information

In the **Create workbench** section, you can define the following workbench properties.

Name

This name identifies your workbench. The OpenShift resources that RHOAI creates for your workbench use this name as a prefix by default.

Description

An optional description for your workbench.

Image selection

These are the container images that embed the runtimes and libraries. RHOAI provides default images tailored to provide a ready-to-work experience for data scientists out of the box. Notebook images are supported for a minimum of one year.

Administrators can provide additional custom images to expand the default ones.

Version selection

This field shows after you select a workbench image, and is populated with the image available versions. Earlier versions might be available to support projects that require legacy compatibility.

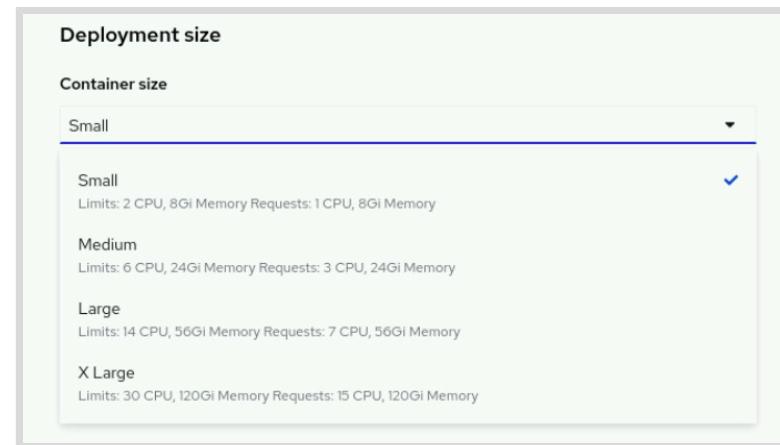
You can view more details about the selected image by clicking **View package information**.

The screenshot shows the 'Create workbench' configuration page. On the left, under 'Notebook image', the 'Image selection' dropdown is set to 'PyTorch'. Below it, the 'Version selection' dropdown is set to '2023.2 (Recommended)'. A tooltip for 'Hover an option to learn more info' is visible. A blue link 'View package information' is present. On the right, a modal window titled 'Package information' displays the following details:

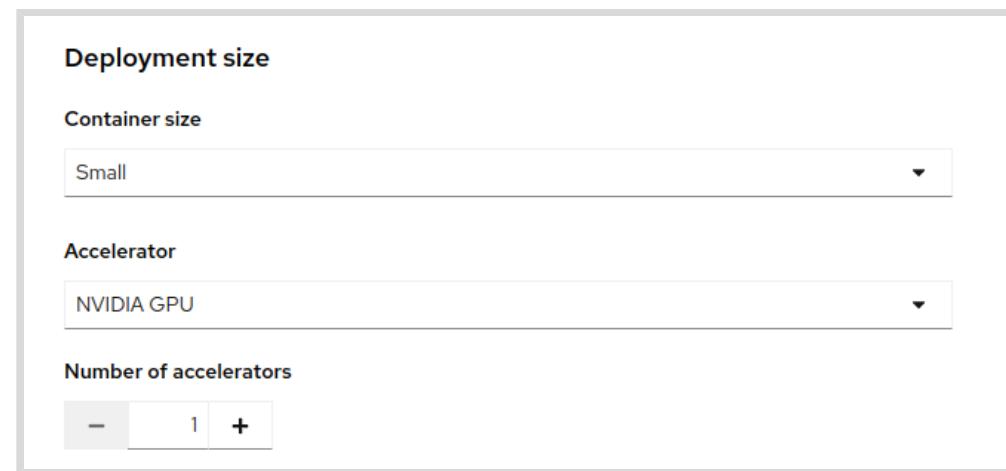
Package information	
Jupyter notebook image with PyTorch libraries and dependencies to start experimenting with advanced AI/ML notebooks.	
Packages included	
PyTorch v2.0 Tensorboard v2.13 Boto3 v1.28 Kafka-Python v2.0 Kfp-tekton v1.5 Matplotlib v3.6 Numpy v1.24 Pandas v1.5 Scikit-learn v1.3 Scipy v1.11 Elyra v3.15 PyMongo v4.5 Pyodbc v4.0 Codeflare-SDK v0.12 Sklearn-onnx v1.15 Psycopg v3.1 MySQL Connector/Python v8.0	

Deployment size

Various container sizes are available to support small to extra large projects. Each size provides an associated number of CPU and memory resources, which are listed in the drop-down menu.



Clusters that have GPU support show an **Accelerator** option where you can select the type and the number of accelerators.



Environment variables

You configure a workbench with environment variables by using OpenShift secrets or configuration maps.

Cluster storage

You can create new persistent storage by providing a name and storage size. This creates an OpenShift persistent volume claim (PVC) to reference that storage. You can also reuse an existing persistent storage.

Cluster storage

Cluster storage will mount to /

Create new persistent storage
This creates storage that is retained when logged out.

Name *
my-first-workbench

Description

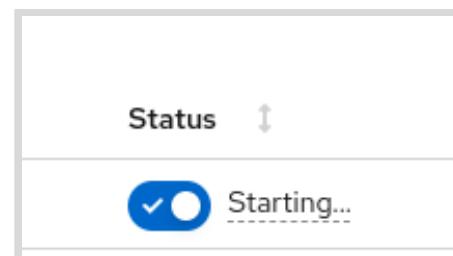
Persistent storage size
- 20 + GiB

Use existing persistent storage
This reuses a previously created persistent storage.

Data Connections

You can provide a workbench with configuration values to connect to storage systems. RHOAI creates the data connection as a Kubernetes Secret, and injects its values into the workbench as environment variables.

When you click **Create workbench**, RHOAI triggers the creation of the workbench and redirects you to the data science project page. While RHOAI is provisioning the workbench, the workbench displays with the **Starting...** status.



If the workbench creation is successful, then you should see the workbench running in the project dashboard. The dashboard should also display any persistent storage associated with the workbench.

The screenshot shows the 'Workbenches' section of the OpenShift interface. A workbench named 'my-first-workbench' is listed, showing it's running and connected to persistent storage. There are buttons for creating a new workbench and adding cluster storage.

Workbench Environment Configuration

Environment variables are the recommended way to inject configuration into software applications. In this way, the application, or in this case the RHOAI workbench, is decoupled from the working environment.

To add environment configuration, you can click **Add variable** in the **Environment variables** section of the workbench form. OpenShift provides a way to provide configuration to your workbench by using secrets or configuration maps.

Each **ConfigMap** or **Secret** can contain multiple key/value items. To add more items to a **ConfigMap** or a **Secret**, click **Add another key / value pair**.

The screenshot shows the 'Environment variables' form for a 'Secret'. It has a dropdown for selecting the secret type ('Secret') and a dropdown for 'Key / value'. A key is defined with the value 'external_service_api_key' and several asterisks as the value. Buttons for adding more keys and variables are at the bottom.

Figure 2.13: Environment variable as a secret



Note

By default OpenShift secrets are base64 encoded but not encrypted. This means that anyone who can access your data science project, also has access to the secret information.

Alternatively, you can upload a file that includes multiple key/value pairs by uploading a YAML file with the **Secret** or **ConfigMap** definition.



Figure 2.14: Multiple environment variables in a ConfigMap definition

You can access these environment variables from your workbench, either from the workbench terminal, or from a Jupyter notebook.

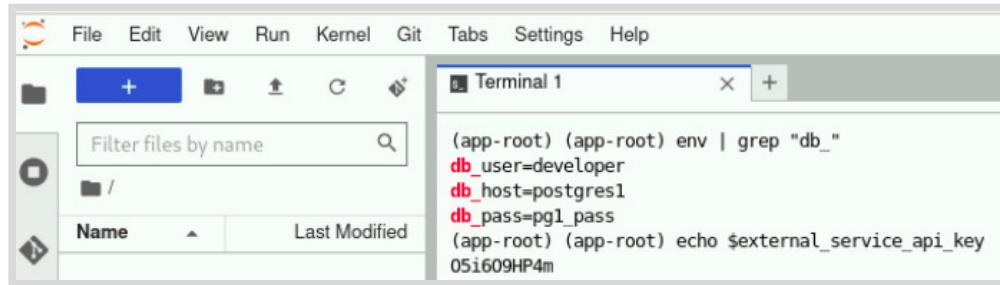


Figure 2.15: Accessing environment variables from the workbench terminal

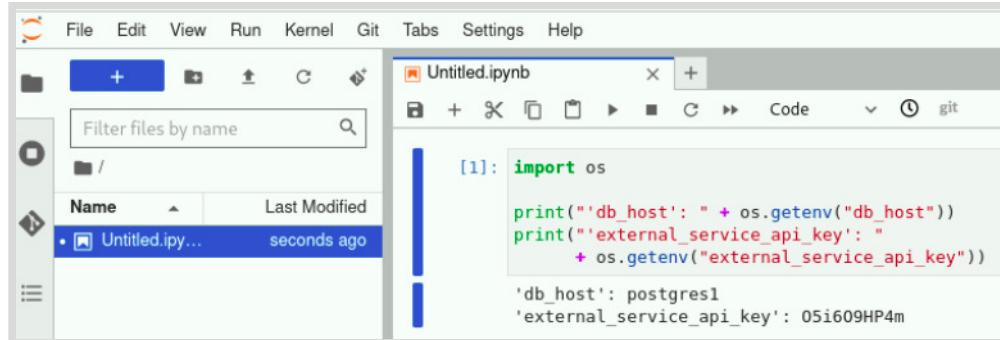


Figure 2.16: Accessing an environment variable from a Jupyter notebook

Cluster Storage

Workbenches require persistent storage to ensure that your progress is not lost when you stop or remove the workbench. Cluster storage uses an OpenShift storage resource called persistent volume claim (PVC). Note that cluster storage is different from data connections in RHOAI. Data connections provide workbenches with configuration values that allow access to S3 storage services. Data connections are covered in another section in the course.

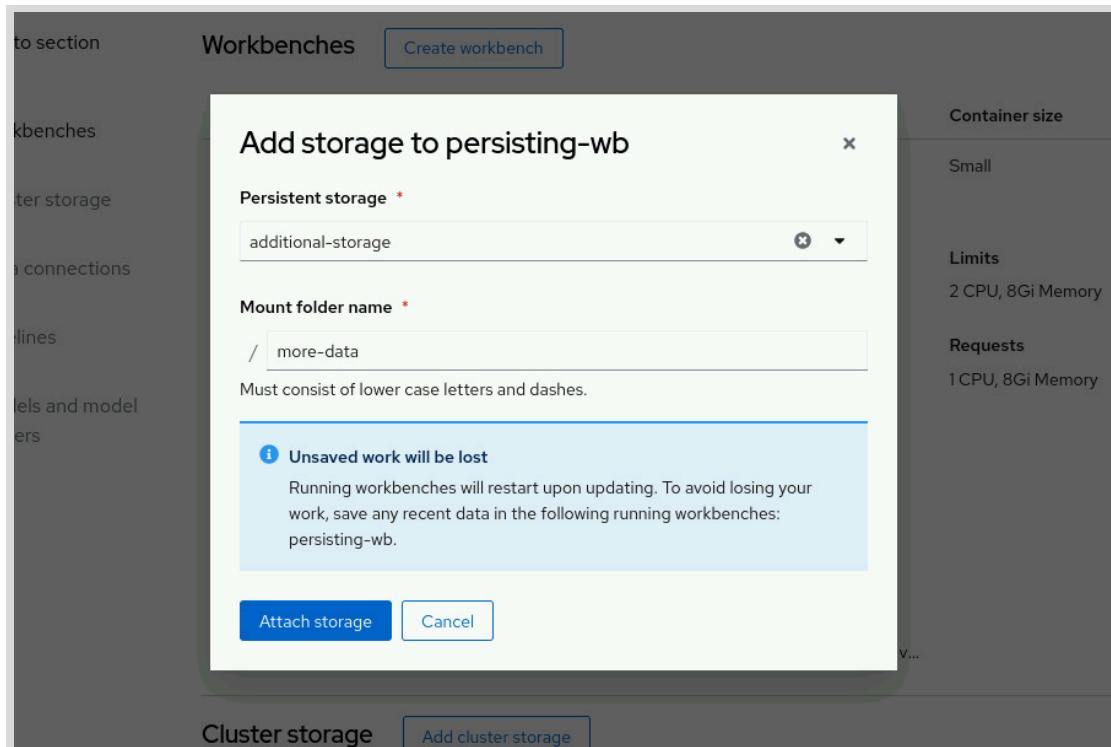
When creating a workbench, you can create new cluster storage or use an existing cluster storage that is not associated to another workbench. To create cluster storage, you can accept the

defaults from the **Create new persistent storage** option in the **Cluster storage** section. The default values use the workbench name for the name of the storage and a storage size of 20 GiB.

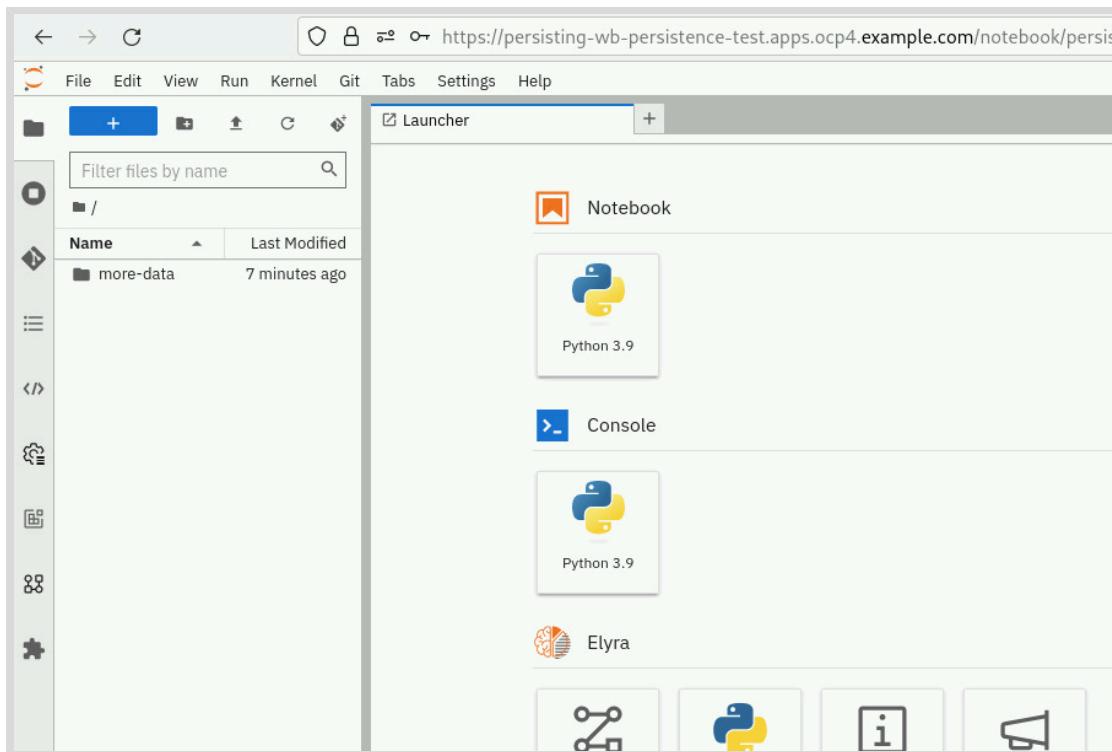
If you recreate the workbench, and you want to keep your previous work, then you must select the **Use existing storage** option. Then select the name of the persistent storage that contains your work.

RHOAI mounts storage in the `/opt/app-root/src` directory of the container. This directory is treated as the root directory (`/`) of the JupyterLab workspace.

You can also create and attach additional persistence volumes to a workbench. For example, you might attach a volume called **additional-storage** to a workbench and specify the location as `/more-data`.



In this case, the volume is attached to `/opt/app-root/src/more-data` within the container. However, in the workspace, it simply shows up as a subdirectory called `more-data`.

**Note**

You cannot decrease the storage size of storage that is associated to a workbench.

Additionally, if you delete the PVC of a storage assigned to a running workbench, then the PVC switches to **Terminating** state until you stop the workbench.

Accessing the Workbench

In the data science project page, you can click the > icon to view more details about the workbench. The card that slides down displays the workbench details, including storage usage. You can use the **Add storage** button to add more storage mount points to the workbench.

A screenshot of a data science project page. A card for a workbench named 'my-first-workbench' is displayed. The card includes sections for 'Workbench storages' (showing 0.04Gi used of 20Gi), 'Packages' (JupyterLab v3.5, Notebook v6.5), 'Limits' (1CPU, 1Gi Memory), and 'Requests' (500m CPU, 1Gi Memory). There is also a 'Add storage' button. The top of the card shows the workbench's status as 'Running' with an 'Open' button.

To open a workbench you can click **Open**. RHOAI requires that you enter your credentials to authenticate to the workbench's JupyterLab. Finally, the OAuth container in your workbench requires that you grant permissions for your workbench.

Authorize Access

Service account my-first--worbench in project rhods-intro-s2 is requesting permission to access your account

Requested permissions

user:info
Read-only access to your user information (including username, identities, and group membership)

user:check-access
Read-only access to view your privileges (for example, "can I create builds?")

You will be redirected to <https://my-first--worbench-rhods-intro-s2.apps.rhods-internal.61tk.p1.openshiftapps.com/oauth/callback>

Allow selected permissions **Deny**

You can click **Allow selected permissions** and you are redirected to JupyterLab.

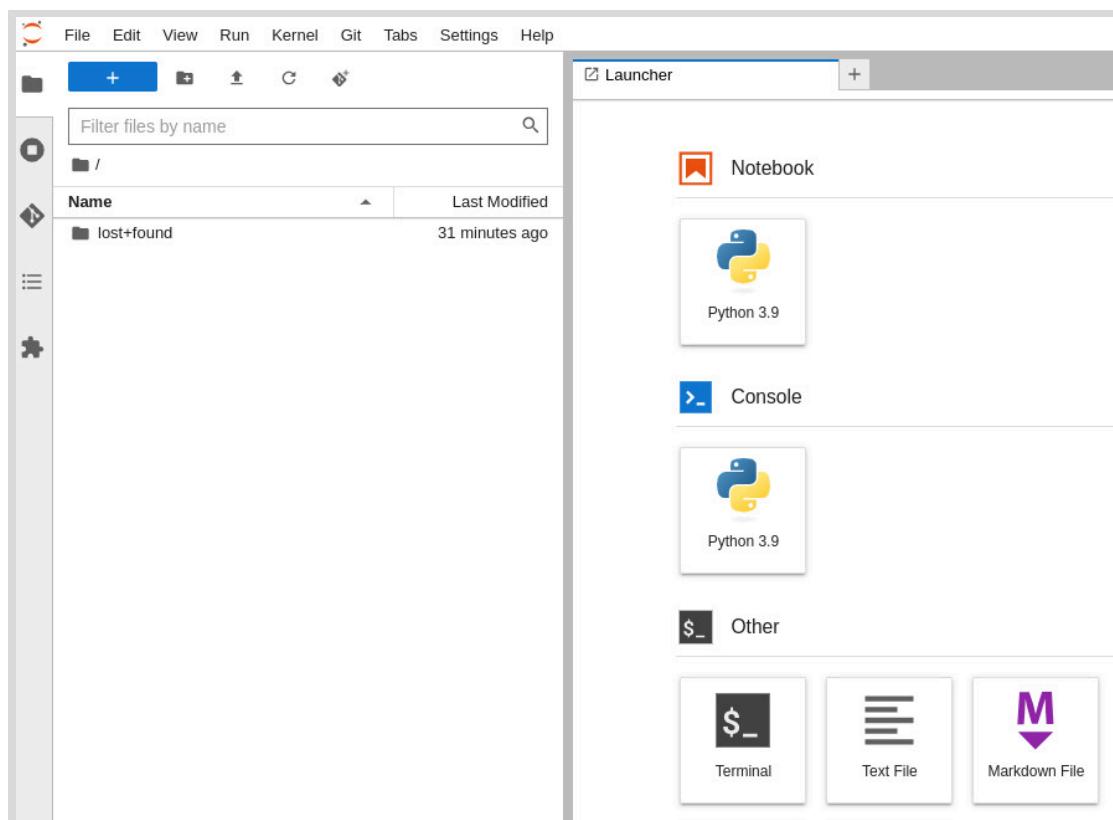


Figure 2.21: JupyterLab home page

Starting and Stopping Workbenches

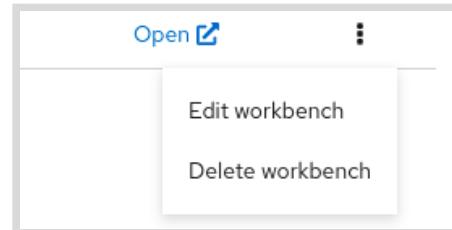
From the RHOAI data science project page, you can click the **Status** switch of the workbench to stop it. The workbench status changes to **Stopping....**

After the workbench is stopped, you can toggle the switch to start the workbench again. The workbench status changes to **Starting....** When the workbench is running, you can switch to the JupyterLab browser tab. If you see a message indicating that the server is unavailable, then you can refresh the page.

Editing a Workbench

When working in a data science project, you might want to change the workbench base image to use different libraries, or you might want to change the container size to increase the hardware limits.

To edit a workbench, you can click the workbench : button, and then click **Edit workbench**, from the RHOAI data science project page.



Warning

Make sure that you save your work at JupyterLab before updating a workbench.

When you edit a workbench, RHOAI restarts the workbench, so any unsaved work will be lost.

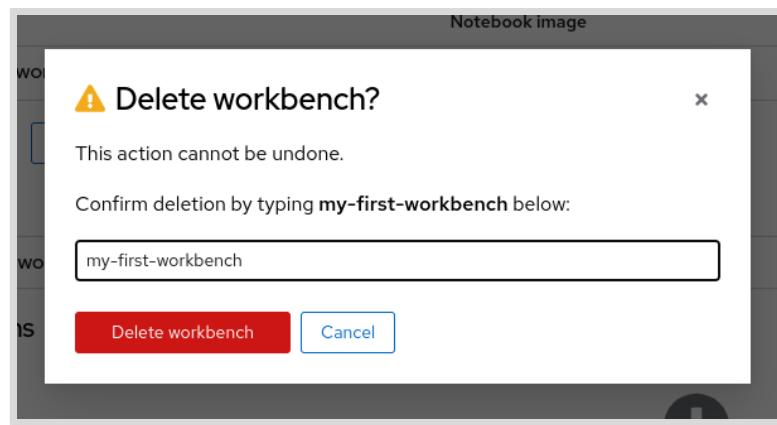


Note

You cannot edit a workbench while the workbench is starting or stopping.

Deleting a Workbench

From the project dashboard, click the workbench : button, and then click **Delete workbench**. In the delete window, type the workbench name to confirm the operation:



Note

Deleting a workbench does not delete the associated persistent storage. This means that you can create another workbench and restore the data that you were using in the previous workbench.

OpenShift Workbench Resources

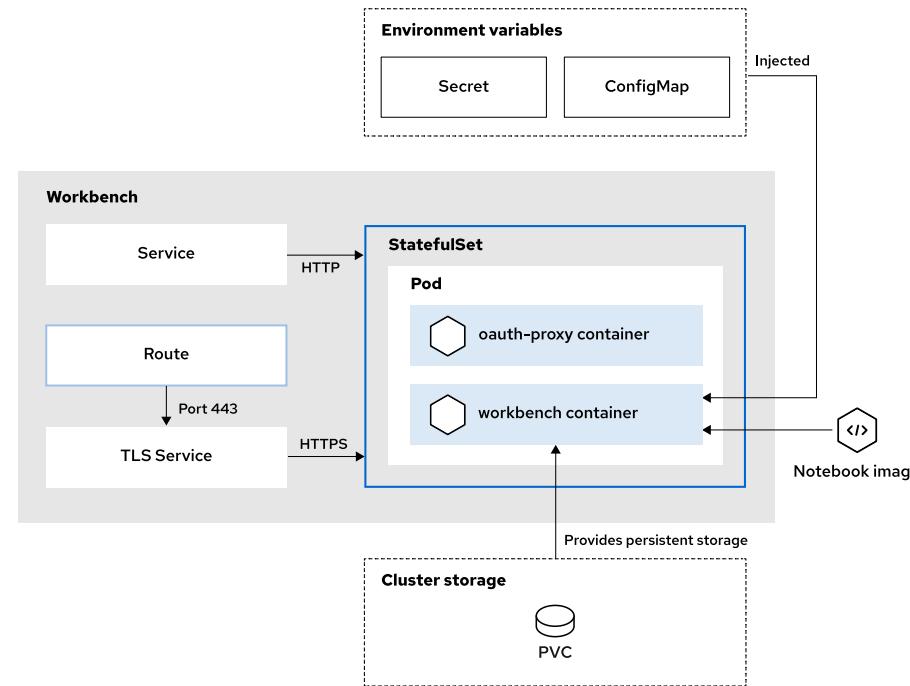
RHOAI internally defines workbenches as stateful applications. When you create a workbench, RHOAI creates a `StatefulSet` object in the OpenShift namespace that corresponds to your data science project. The `StatefulSet` manages a pod that includes two containers:

- The workbench container, which contains JupyterLab and other packages. You interact with this container when you use the JupyterLab web interface.
- The `oauth-proxy` container, which provides authorization and authentication to access JupyterLab.

RHOAI also creates a service and a route to enable HTTPS traffic into the workbench container. The OpenShift route redirects HTTP traffic to HTTPS. If you list the resources of the OpenShift namespace that corresponds to your data science project, you should see something similar to this:

NAME	READY	STATUS	...
<code>pod/my-first-workbench-0</code>	2/2	Running	...
<hr/>			
NAME	TYPE	...	
<code>service/modelmesh-serving</code>	ClusterIP	...	
<code>service/my-first-workbench</code>	ClusterIP	...	
<code>service/my-first-workbench-tls</code>	ClusterIP	...	
<hr/>			
NAME	READY	...	
<code>statefulset.apps/my-first-workbench</code>	1/1	...	
<hr/>			
NAME		...	
<code>route.route.openshift.io/my-first-workbench</code>		...	

Creating a new persistent storage in RHOAI results in the creation of a PVC in the corresponding OpenShift project. The following diagram represents the OpenShift resources involved in a workbench:



Workbench Limits

If your memory, CPU, or GPU requirements are high, then RHOAI might not be able to allocate workbench resources. You might see a message similar to the following:

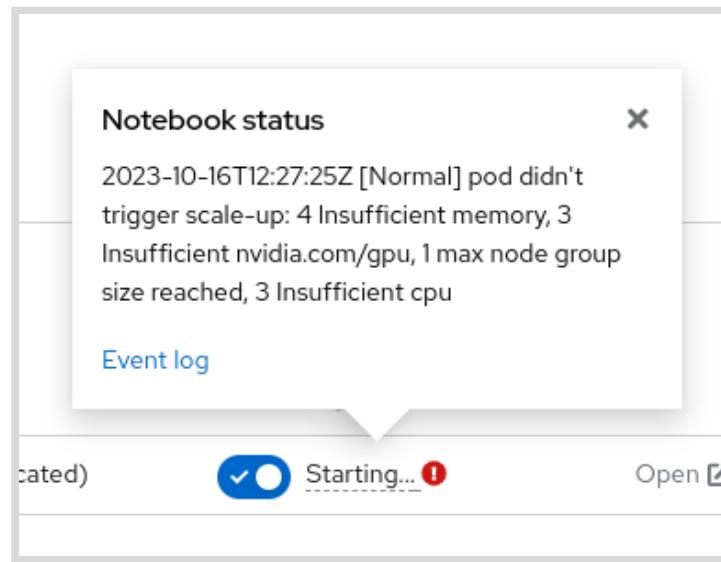


Figure 2.25: OpenShift resources for a RHOAI workbench

In this case, try to decrease your resource requirements by editing the workbench, or contact your RHOAI administrator.



References

For more information about workbenches, refer to the *Using Project Workbenches* section in the *Working on Data Science Projects* chapter in the Red Hat OpenShift AI Self-managed 2.8 *Working on Data Science Projects* documentation at https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/working_on_data_science_projects/index#using_project_workbenches

For more information about notebook images, refer to the *Notebook Images for Data Scientists* section in the *Creating and Importing Notebooks* chapter in the Red Hat OpenShift AI Self-managed 2.8 *Working on Data Science Projects* documentation at https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/working_on_data_science_projects/index#notebook-images-for-data-scientists_notebooks

For more information about working with accelerators, refer to the *Overview of Accelerators* section in the *Working with Data Science Pipelines* chapter in the Red Hat OpenShift AI Self-managed 2.8 *Working on Data Science Projects* documentation at https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/working_on_data_science_projects/index#working-with-accelerators_accelerators

► Guided Exercise

Workbenches

Set up workbenches and create Jupyter Notebooks.

Outcomes

- Create and access a workbench within Red Hat OpenShift AI (RHOAI).
- Set and retrieve an environment variable within a workbench container.
- Verify that cluster storage attached to a workbench persists without the workbench.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI262 && \
lab start projects-workbenches
```

Instructions

- ▶ 1. Switch to the `projects-workbenches` data science project and navigate to the project dashboard page.
 - 1.1. In a web browser, navigate to the RHOAI dashboard at `https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com` and authenticate as the `developer` user by using the `developer` password.
 - 1.2. Navigate to the project details by clicking **Data Science Projects** and then the `projects-workbenches` project.
- ▶ 2. Create a workbench called `projects-workbenches-wb`.
 - 2.1. Click **Create workbench**, enter `projects-workbenches-wb` as the name of the workbench, and select `Minimal Python` for the image.
 - 2.2. Scroll down to the **Environment variables** section and click **Add variable**. Select `ConfigMap` and `Key / value`. Enter `DATA_FILE` as the key and `data.csv` as the value.
 - 2.3. Scroll down to the **Cluster storage** section. Ensure that the **Create new persistent storage** option is selected and that the **Name** field is set to `projects-workbenches-wb`.
 - 2.4. Set the **Persistent storage size** field to `2Gi` and then click **Create workbench**. RHOAI triggers the creation of the workbench and redirects you to the data science project page.

- 2.5. Inspect the workbench by clicking the > icon next to the name of the workbench to view more details. The expanded workbench information section shows details about attached storage, installed packages, and resource limits.
- ▶ 3. Access the `projects-workbenches-wb` workbench and view the configured environment variable.
 - 3.1. After the workbench shows a status of **Running**, click **Open** to open a new browser tab to the workbench.
 - 3.2. Similar to accessing the RHOAI dashboard, use the username **developer** and password **developer** to authenticate to the workbench's JupyterLab instance.
 - 3.3. Accept the default selected permissions by clicking **Allow selected permissions**. The JupyterLab interface displays.
 - 3.4. In the **Launcher** tab, click the **Terminal** item to open a new terminal.
 - 3.5. In the terminal, verify that the **DATA_FILE** environment variable is set within the workbench:

```
(app-root) (app-root) echo $DATA_FILE  
data.csv
```

- ▶ 4. Create a new file within the workbench container to verify that it persists after restarting the workbench.
 - 4.1. Create a file called `hello.txt` with `hello world` as the contents.
 - 4.2. From the browser tab containing the RHOAI data science project page, click the toggle within the **Status** to disable it and stop the workbench. In the dialog that shows, click **Stop workbench** to confirm.
The workbench status changes to **Stopping....**
 - 4.3. After the workbench changes to **Stopped**, toggle the switch back to enabled.
The workbench status changes to **Starting....**
 - 4.4. After the workbench is running again, switch to the JupyterLab browser tab.



Note

You might see a message indicating that the server is unavailable. If so, use the browser refresh button or press F5 to refresh the page.

- 4.5. Use the terminal as before to verify that the `hello.txt` file still exists.

```
(app-root) (app-root) cat hello.txt  
hello world
```

- ▶ 5. Delete and recreate the workbench with the existing storage.

- 5.1. Close the JupyterLab browser tab and return to the RHOAI data science project page.
- 5.2. In the row containing the `projects-workbenches-wb` workbench, click the menu button (:), and then click **Delete workbench**.
- 5.3. In the delete confirmation dialog, type the workbench name `projects-workbenches-wb` and click **Delete workbench** to confirm.
- 5.4. Create a similar workbench by clicking **Create workbench**, entering `projects-workbenches-wb` as the name, and selecting `Minimal Python` as the image. However, in the `Cluster storage` section, select **Use existing persistent storage** and select the `projects-workbenches-wb` option.
- 5.5. Click **Create workbench** and wait for the workbench to start.
- 5.6. Open the new workbench by clicking **Open**, and log in to JupyterLab.
- 5.7. From the left-pane navigation, open the `hello.txt` file by double-clicking it, and verify that its contents remain the same.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish projects-workbenches
```

Data Connections

Objectives

- Associate data sources to workbenches by using data connections.

Data Connections

Within artificial intelligence (AI), machine learning (ML), and data science environments, data availability is a key factor for the success of a project. For projects with small amounts of data, it can be enough to load data by uploading files directly to the workbench or by making database queries from notebooks.

However, complex projects involve larger datasets. Depending on the dataset size, uploading files or querying databases might not be feasible. Storing the data directly in code repositories is also not recommended, as this can overload the repository. Many source control systems include file size limits to prevent such consequences.

Storing data in a dedicated storage system can handle larger datasets. When needed, download the data from the data store into your workbench.

Similarly, because of their potential size, trained models must be stored in a dedicated storage system.

Data Connections Are Secrets

Red Hat OpenShift AI (RHOAI) uses *data connections* to define a set of configuration values that facilitate connecting workbenches to storage systems. Data connections contain a set of configuration values used to connect to storage systems.



Note

Data connections differ from cluster storage. By using `PersistentVolumeClaim` (PVC) objects, cluster storage provides persistent storage for workbenches and persists progress within workbenches.

RHOAI creates data connections as `Secret` objects standard to Red Hat OpenShift Container Platform (RHOC). These secrets are prefixed with `aws-connection-` and stored in the RHOC namespace corresponding to your data science project.

Data connection secrets include the following keys:

Key	Meaning
<code>AWS_S3_ENDPOINT</code>	A connection string to connect to the S3 API server.
<code>AWS_ACCESS_KEY_ID</code>	The access key for the S3 server. This is usually a username.

Key	Meaning
AWS_SECRET_ACCESS_KEY	The password for the S3 user.
AWS_DEFAULT_REGION	The S3 region containing the bucket.
AWS_S3_BUCKET	The name of the S3 bucket containing the data.

Although these variables suggest the use of S3, the values are only environment variables. It is up to the user to decide how to use them. They are often used to connect to a system that implements the S3 connection protocol.

You can set these values via the RHOAI dashboard. After doing so, RHOAI creates a data connection secret object similar to the following example definition:

```
apiVersion: v1
kind: Secret ①
type: Opaque
data: ②
  AWS_ACCESS_KEY_ID: bXlhY2Nlc3NrZXk=
  AWS_DEFAULT_REGION: ""
  AWS_S3_BUCKET: ""
  AWS_S3_ENDPOINT: aHR0cHM6Ly9zMy5leGFtcGx1LmNvbQ==
  AWS_SECRET_ACCESS_KEY: bXlzzWNyZXRrZXk=
metadata:
  annotations:
    opendatahub.io/connection-type: s3 ③
    openshift.io/display-name: example-data-connection
    ...output omitted...
  name: aws-connection-example-data-connection ④
  namespace: example-project
```

- ① Declares the object as a `Secret`.
- ② Defines the key-value pairs within the secret.
- ③ The type of data connection. The current version of RHOAI only supports S3.
- ④ The name of the data connection within RHOAI.

Because the data connection is a regular RHOCP Secret, the values are stored in base64 format. To retrieve the original value, use the `base64` tool with the `-d` option. For example, the following command retrieves and decodes the `AWS_S3_ENDPOINT` value within a RHOAI data connection secret called `example-data-connection`:

```
[student@workstation ~]$ oc get secret aws-connection-example-data-connection \
-o jsonpath='{.data.AWS_S3_ENDPOINT}'|base64 -d
https://s3.example.com
```

Storing values within secrets in base64 format makes it harder for someone to glance at the values, but provides no additional security.

The base64 format is not a form of encryption. Anybody with a value in this format can retrieve the decoded value. As in the preceding example, no keys are necessary to do so.

Exercise caution and follow standard security practices to ensure that only authorized users can access the Secret objects.

Using the Data Connection Environment Variables

When you associate a data connection to a workbench, RHOAI injects the key-value pairs of the data connection as environment variables into the workbench container.

RHOAI does not automatically open a connection from the workbench to AWS S3. You are responsible for reading and using the environment variables of the data connection in the notebook. Within the workbench, retrieve the variables and use them to configure a connection to the storage system.

For example, the following workbench code snippet uses the built-in os Python library to read the connection environment variables:

```
import os

key_id = os.getenv("AWS_ACCESS_KEY_ID")
secret_key = os.getenv("AWS_SECRET_ACCESS_KEY")
region = os.getenv("AWS_DEFAULT_REGION")
endpoint = os.getenv("AWS_S3_ENDPOINT")
bucket_name = os.getenv("AWS_S3_BUCKET")
```

You can connect to S3 storage by using the boto3 Python library and the data connection values. When connected to S3, you can download the required files for data exploration or model training.

Although each workbench can only have one data connection, the same data connection can be used across multiple workbenches.

Similarly, store your trained models in the S3 bucket specified by the data connection. Storing your models via the data connection enables the model serving features within RHOAI. Model serving uses the data connection to download model files.



References

For more information, refer to the *Using Data Connections* section in the *Working on Data Science Projects* chapter in the Red Hat Red Hat OpenShift AI Self-managed 2.8 *Working on Data Science Projects* documentation at https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/working_on_data_science_projects/index#using_data_connections

For more information on RHOCP Secrets, refer to the *Providing sensitive data to pods by using secrets* chapter in the *OpenShift Container Platform* documentation at <https://docs.openshift.com/container-platform/4.14/nodes/pods/nodes-pods-secrets.html>

► Guided Exercise

Data Connections

Associate data sources to workbenches by using data connections.

Outcomes

- Create a data connection in Red Hat OpenShift AI (RHOAI) that stores information to connect to an S3 bucket.
- Attach a data connection to a workbench in RHOAI.
- Use the data connection values to upload and download objects to the S3 bucket within a Python Jupyter notebook.

Before You Begin

As the **student** user on the **workstation** machine, use the **lab** command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI262 && \
lab start projects-data
```

Instructions

- 1. Deploy and configure the MinIO server. This provides the S3 bucket for the data connection.
- 1.1. In a new command line terminal window, authenticate **oc** with the cluster and switch to the **projects-data** project.

```
[student@workstation ~]$ oc login -u developer -p developer \
https://api.ocp4.example.com:6443
...output omitted...
Login successful.
...output omitted...
[student@workstation ~]$ oc project projects-data
...output omitted...
```

- 1.2. Apply the provided **minio.yaml** manifest file. This creates several Kubernetes resources, including the MinIO pod, service, and routes.

```
[student@workstation ~]$ MANIFEST_HOST="https://raw.githubusercontent.com"
no output expected
[student@workstation ~]$ wget \
$MANIFEST_HOST/RedHatTraining/AI26X-apps/main/intro/projects-data/minio.yaml
...output omitted...
[student@workstation ~]$ oc apply -f minio.yaml
...output omitted...
```

- 1.3. In a web browser, navigate to the MinIO UI by using the address from the route.

```
[student@workstation ~]$ oc get route minio-ui -o jsonpath='{.spec.host}'
minio-ui-projects-data.apps.ocp4.example.com
```

- 1.4. Log into the UI by using the `minio` username with password `minio123`.
 - 1.5. Because no bucket exists, the UI prompts you to create one. Click **Create a Bucket**, provide the name `projects-data-bucket`, and click **Create Bucket**. Leave all other form values as their default.
- ▶ 2. In the `projects-data` Data Science Project, create a new data connection called `projects-data-connection`.
- 2.1. In a web browser, navigate to `https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com`. Log in as the `developer` user by using the `developer` password.
 - 2.2. From the left navigation pane, click **Data Science Projects** and click the project called `projects-data`.
 - 2.3. From the **Data connections** section, click **Add data connection**.
 - 2.4. Use the following values to fill out and submit the form.

Field	Value
Name	<code>projects-data-connection</code>
Access key	<code>minio</code>
Secret key	<code>minio123</code>
Endpoint	<code>https://minio-api-projects-data.apps.ocp4.example.com</code>
Region	<code>us-east-1</code>
Bucket	<code>projects-data-bucket</code>

- ▶ 3. Create a new workbench called `projects-data-wb`.
- 3.1. From the **Workbenches** section, click **Create workbench**.
 - 3.2. Enter `projects-data-wb` as the name and select **Standard Data Science** as the image. Do not submit the form yet.

- 3.3. At the bottom of the form, enable the `Use a data connection` checkbox, select `Use existing data connection`, and then select `projects-data-connection`. Leave the rest of the fields as their default and click `Create workbench`.

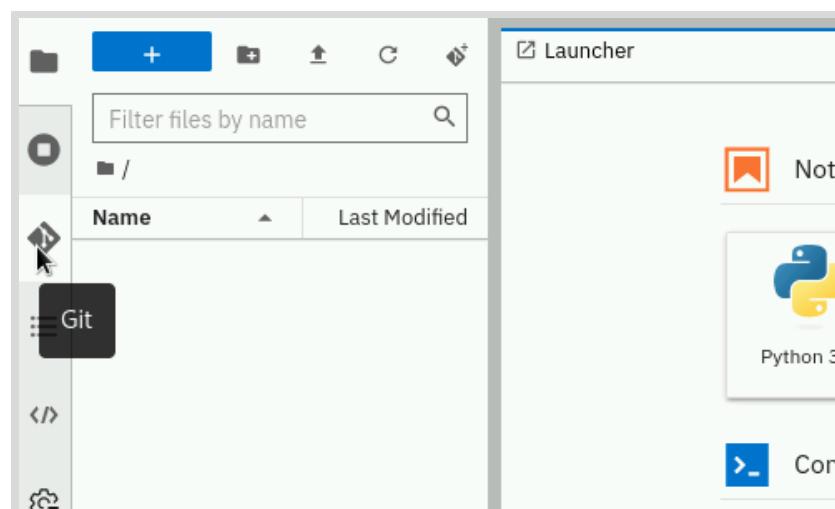
Wait for the `projects-data-wb` workbench to finish starting. The `Status` column for the workbench says `Running` when it is ready.

- 3.4. After the workbench has started, open it by clicking `Open` within the same row.

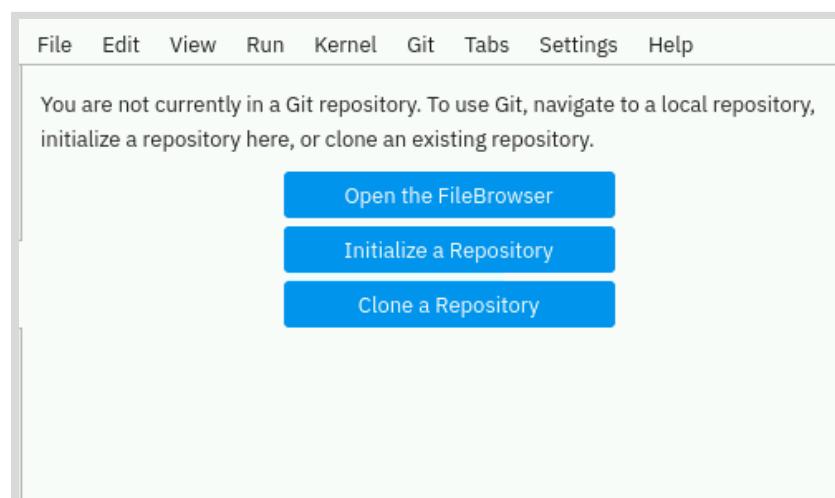
- 3.5. Authenticate with the `htpasswd_provider` option, as the `developer` user by using password `developer`. Accept permissions by clicking `Allow selected permissions`.

▶ 4. Clone the Git repository to access the notebook.

- 4.1. In the JupyterLab interface, click the `Git` icon from the left tools pane.



- 4.2. Click `Clone a repository`.



- 4.3. Enter `https://github.com/RedHatTraining/AI26X-apps` as the repository, and click `Clone`.

- 4.4. In the JupyterLab file browser, verify the new content by navigating to the `AI26X-apps/intro/projects-data` directory.

- 5. Read and follow the instructions within the `projects-data-nb.ipynb` notebook for the remainder of this exercise.

**Note**

Keep in mind the following tips when working with Jupyter Notebooks:

- Variables defined within one cell are available within other cells of the notebook.
- Execute the active cell of a notebook by pressing **Shift+Enter** or clicking **Run > Run Selected Cells**.
- Cells are run and re-run according to the order that you execute them. In the corresponding exercise notebook, be sure to run the cells in the provided order, but feel free to modify and re-run individual cells.
- When a cell runs, the output is the last-executed statement within the cell. Use this to quickly print variable values without calling the Python `print` function explicitly.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish projects-data
```

Summary

- A Red Hat OpenShift AI (RHOAI) project is an OpenShift project that helps you organize resources when working on an AI/ML application.
- A workbench is a containerized working environment that runs a JupyterLab server and a collection of common AI/ML libraries.
- RHOAI internally defines a workbench as a stateful application by creating a **StatefulSet**. The **StatefulSet** manages a pod with a JupyterLab container and an OAuth proxy container.
- RHOAI creates a route and the required services to expose JupyterLab.
- You can add cluster storage from your data science project detail view, which creates a persistent volume claim that you can later attach to a workbench.
- RHOAI uses data connections to define a set of configuration values that ease connecting workbenches to storage systems. This configuration is stored in OpenShift secrets.
- Workbenches typically use a connection to a storage system when loading data or when saving a trained model.
- When you associate a data connection to a workbench, RHOAI injects the key-value pairs of the data connection as environment variables into the workbench container. You can use the data connection environment variables to access the storage from within the workbench.

Chapter 3

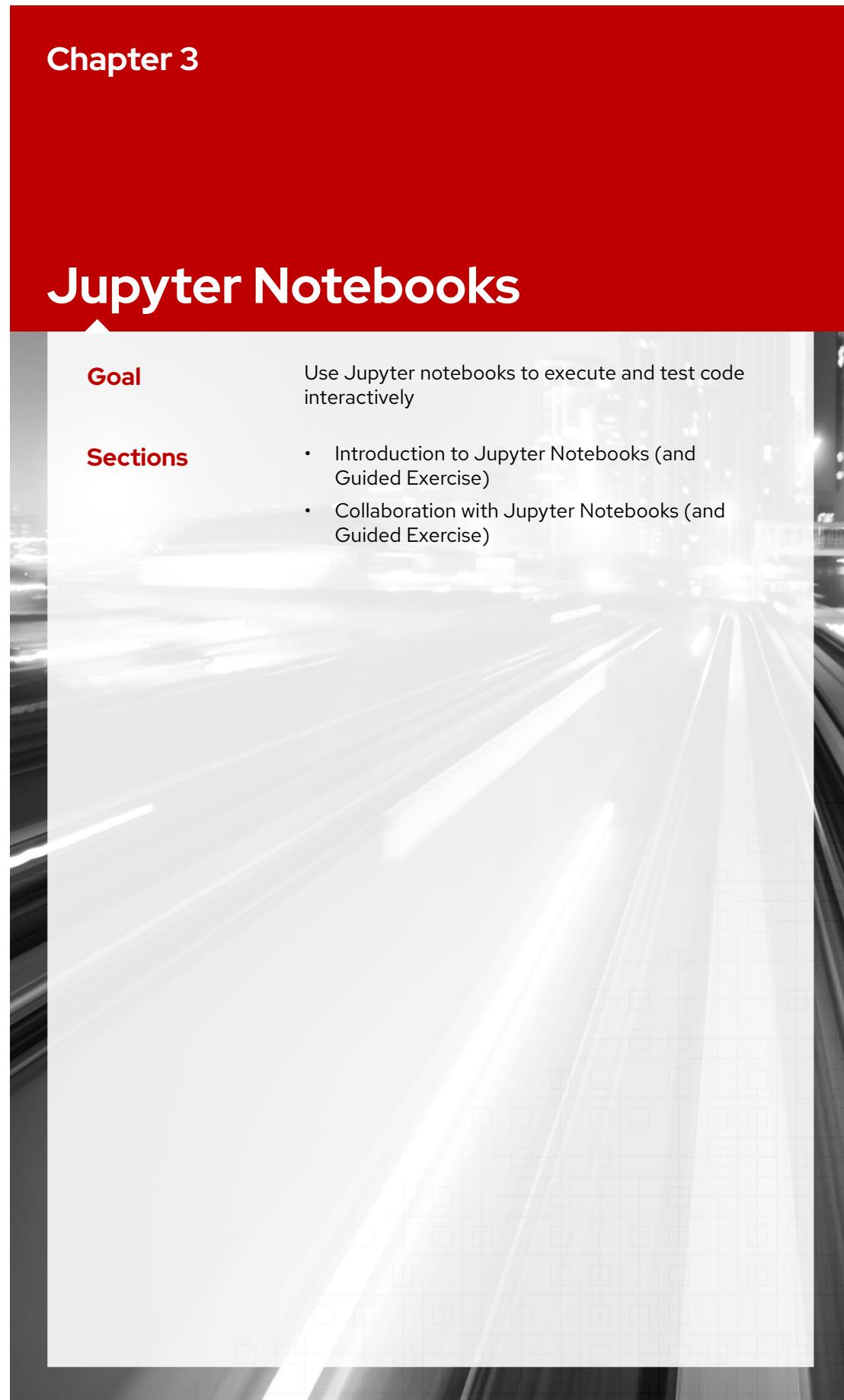
Jupyter Notebooks

Goal

Use Jupyter notebooks to execute and test code interactively

Sections

- Introduction to Jupyter Notebooks (and Guided Exercise)
- Collaboration with Jupyter Notebooks (and Guided Exercise)



Introduction to Jupyter Notebooks

Objectives

- Create interactive documents that contain narrative text, executable code, and visualizations.

Introduction

A Jupyter notebook is an interactive, live document that can contain rich text, executable code, multimedia-content and visualizations. Jupyter notebooks have become a popular tool for data scientists who work in data science and machine learning projects.

To fully understand the Jupyter ecosystem, you must understand the following three concepts:

Notebook file

A notebook file, also called *Jupyter notebook*, or just *notebook*, is a file that contains code and multimedia content, including text, images, and audio. These files typically use the `.ipynb` extension, which stands for *IPython notebook*. The IPython notebook format uses a JSON representation to encode the content of a notebook.

A notebook contains cells. These cells can contain either Markdown or executable code.

JupyterLab

JupyterLab is the execution environment that runs the notebook files. It exposes a web interface for you to visualize, edit and execute the notebooks via a web browser.



Note

The JupyterLab predecessor was called *Jupyter Notebook*. It also provided a notebook execution environment and a web interface.

Note that the naming of the classic Jupyter Notebook environment can lead to confusion, because the *Jupyter notebook* term can also refer to a notebook file.

There are many ways to run an execution environment for notebooks:

- You can install JupyterLab with `pip` and run it on your computer.
- You can use the JupyterLab instances included in the Red Hat OpenShift AI (RHOAI) workbenches.
- You can use online notebook execution environments, such as Binder, Kaggle, or Google Colab.

Project Jupyter

This is the umbrella open source project that defines the notebook file format and develops and maintains the JupyterLab and the Jupyter Notebook environments.

All the default workbench images of RHOAI include JupyterLab.

Jupyter notebooks are popular because they provide an interactive and flexible working environment for experimentation, training, documentation, and data analysis.

Chapter 3 | Jupyter Notebooks

The versatility of Jupyter notebooks makes them ideal for a wide range of users. For data scientists and researchers, these features enable them to analyze data, create visualizations, and to create reproducible and shareable work and experiments. For educators, Jupyter notebooks are a powerful tool to create hands-on content, which can combine narrative with executable examples.

Kernels

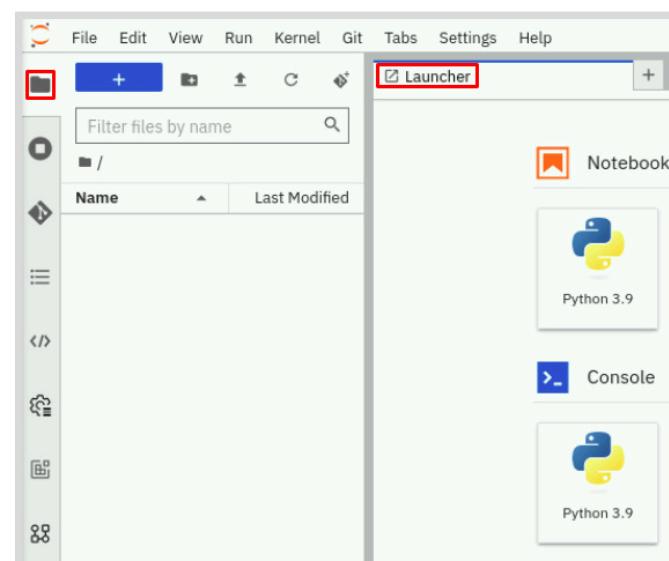
In the context of JupyterLab (and Jupyter Notebook), a *kernel* is the process that runs the code cells in a notebook. A kernel determines the programming language for the notebook. By default, JupyterLab includes the `ipykernel` Python kernel, which depends on IPython, an interactive Python shell.

You can also configure JupyterLab to use kernels for other programming languages. Many of these kernels are actively maintained by the Jupyter community. The list of kernels is available at <https://github.com/jupyter/jupyter/wiki/Jupyter-kernels>.

You can interact with kernels without JupyterLab to create your own notebook environments. For example, the VSCodium Python extension supports Jupyter notebooks without installing JupyterLab. Instead, VSCodium interacts with `ipykernel`.

Working with Jupyter Notebooks

When you open a new RHOAI workbench, a JupyterLab instance opens and you are presented with the file browser section and a launcher tab.



The Launcher

The Launcher window of JupyterLab is an area for you to open items such as workbenches, consoles, or files. To open a launcher, you have multiple options:

- By clicking the + blue button, near the top left of the JupyterLab window.
- By clicking File > New Launcher.
- By using the **Ctrl+Shift+L** keyboard shortcut. After you open the launcher, create a notebook by clicking one of the available notebook options, such as **Python 3.9**.

Directory Navigation

With the file browser tab, you can navigate the directory tree by clicking the directories in the file listing, and navigate back by clicking the path elements underneath the search bar.

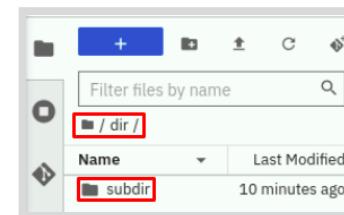


Figure 3.2: Directory navigation

Creating Notebooks

In JupyterLab, you can create notebooks in multiple ways.

Open the **File > New** menu from the top menu bar, and then click **Notebook**. After creating the notebook file, JupyterLab prompts you to select the kernel that you want to use. Depending on your workbench, you might have more than one kernel available. You can also create a notebook from the launcher tab. In RHOAI, the default workbench images include Python 3.9 as the kernel.

You can also create a notebook from the file browser, by right clicking the file browser canvas and clicking **New Notebook**.

When you create a notebook, JupyterLab creates the notebook file with the `Untitled.ipynb` name. When you save the notebook for the first time, JupyterLab prompts you to enter a name for the newly created notebook.



Important

Before creating a notebook, verify that you are at the directory where you want to create the notebook file. Use the JupyterLab file browser to navigate to the right directory.

You can also rename a notebook at any time by right clicking the notebook file in the file browser, and then clicking **Rename**. Alternatively, you can select the file and press **F2**.

Notebook Execution

To execute the code in a notebook, you can run individual cells, a group of selected cells, or the complete notebook:

- To run a single cell, select the cell and press **Ctrl+Enter**.
- To run a group of cells, select the cell at the beginning, press **Shift+Down** to expand the selection and then **Ctrl+Enter** to execute.
- To run the complete notebook you might want to restart the kernel to avoid errors. You can click the **Restart Kernel and Run All Cells** button, as highlighted in the following screen capture.

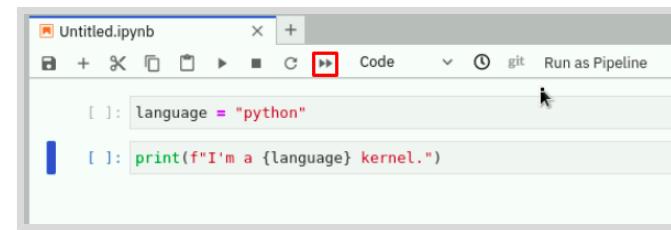


Figure 3.3: Restart kernel and run all cells

When you restart the kernel, you delete the notebook state, deleting variable values or function definitions for example. It is a good practice to restart the kernel and run all cells to verify that you can reproduce your notebook's results.

The output that code execution produces forms part of the notebook itself.



Figure 3.4: Cell output

Saving and Exporting a Notebook

When you save a notebook, JupyterLab saves the cells and the current execution output to the notebook file.

To save a notebook you can click the **Save** button from the toolbar.

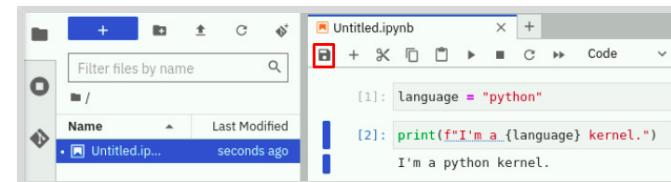


Figure 3.5: Saving a notebook

If you are working with a new notebook, then JupyterLab asks you to name the file before saving it.

After you finish your notebook, you might want to export it to share your work. You can export your notebook to different formats by clicking **File > Save and Export Notebook As...** in the top menu bar. As you can see in the following image, you can choose from different export formats, such as HTML or PDF.

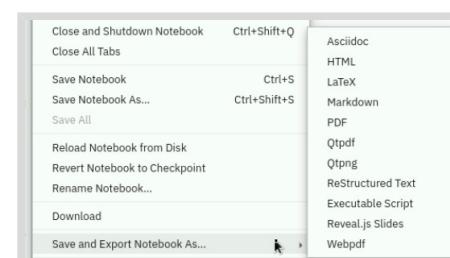


Figure 3.6: Notebook export formats

Working with Notebook Cells

A notebook is made of blocks, called *cells*. Cells can be of two types:

Markdown cells

These cells contain Markdown content and are intended to add rich text to the notebook.

Code cells

These cells contain executable code. JupyterLab includes support for cells that run Python code, but you can use other kernels to add support for different programming languages.

You can change the cell type by pressing **Esc** to exit edit mode, and then pressing **m** to change to markdown type, or **y** to change to code type.

Executing Cells

To update the cell contents, you enter edit mode by clicking the cell. Then, you can exit edit mode by pressing **Esc**, or you can execute the cell by pressing **Ctrl+Enter**. Executing a markdown cell renders the cell's contents. Executing a code cell interprets the code and prints any output or errors that result from the code execution.

When you execute a code cell, the effect of that code is available in the current notebook.

Because of this, imports, variable assignments, and functions that you define in a cell that you execute are available in other cells in the same notebook. To remove this state you must restart the kernel process, by using one of the options for restarting the kernel in the **Kernel** option from the top menu bar.



Figure 3.7: Restart kernel menu

While a cell is executing, the brackets to the left of the cell show an asterisk. This means that the kernel process is busy doing the computation. You can stop a long running cell by clicking the **Interrupt the kernel** button. This interrupts the cell execution, maintaining the kernel state.

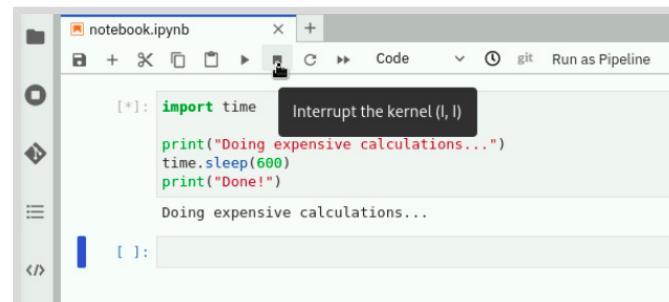
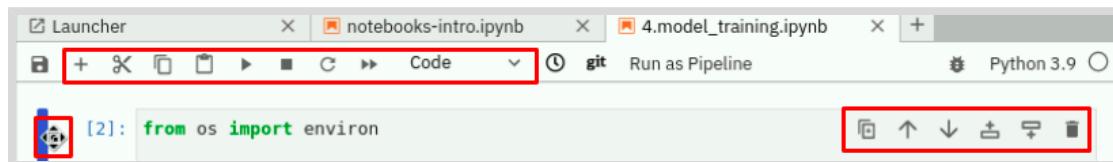


Figure 3.8: Stopping the kernel process

Cell User Interface

The JupyterLab interface has two graphical toolbars to help you work with cells. You can also reorder cells with the mouse by clicking the blue left margin from the selected cell, and dragging the cell to the desired position.



You can find each button action by hovering over each icon with the mouse.

You can also find other cell actions by right-clicking the cell to open the contextual menu.

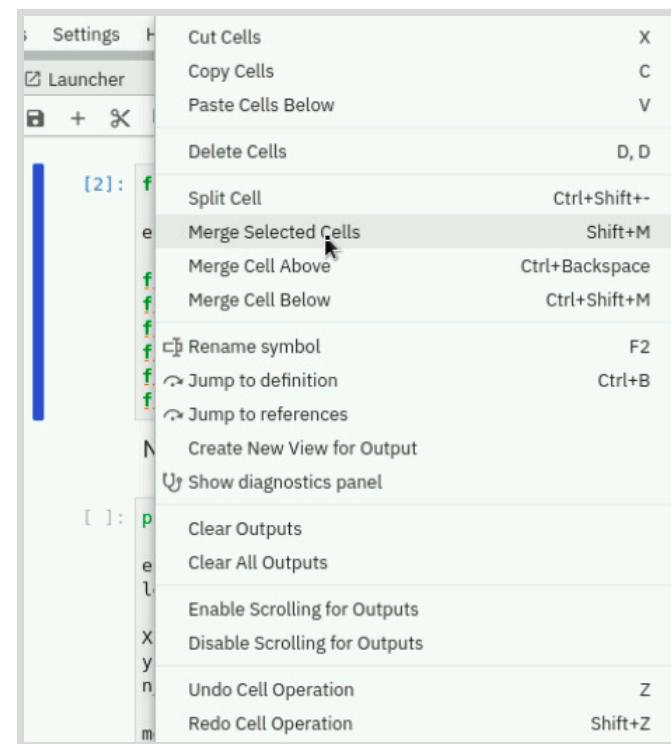


Figure 3.10: Cell contextual menu

Shell and Magic Commands

You can run shell commands from a code cell by using an exclamation mark (!) at the beginning of a line.

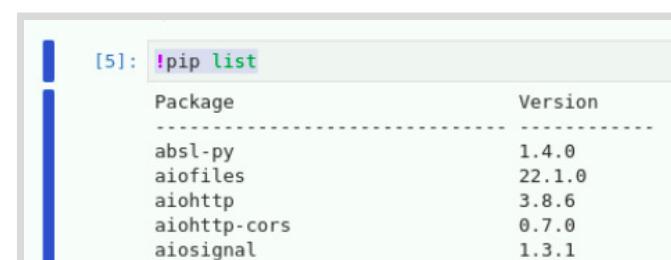


Figure 3.11: Shell command

Chapter 3 | Jupyter Notebooks

JupyterLab has built-in commands called `magic` commands, which you can use by starting a line with either one or two percentage signs (%).

Magic commands that start with a single percentage (%) are line commands, that take the arguments from the same line. Magic commands that start with a double percentage (%%) are cell commands that take the following lines as arguments.

For example, you can use the `%time` command to time a single line or the `%%time` command to time the following lines.

```
[1]: from random import random
%time rand = [random() for i in range(0, 100)]
CPU times: user 12 µs, sys: 2 µs, total: 14 µs
Wall time: 16.5 µs

[2]: %%time
for i in range(0,100):
    random()
CPU times: user 10 µs, sys: 1 µs, total: 11 µs
Wall time: 13.4 µs
```

Figure 3.12: Magic commands

Note that, for running a shell command (!), IPython spawns a shell in a separate subprocess. Shell commands are useful for performing system-wide or file system tasks.

In contrast, magic commands act at the kernel process level. The kernel that runs the notebook directly runs magic commands, without spawning new processes. Magic commands use an IPython/Jupyter API.

Working with Kernels

When you open a notebook, a kernel starts in the background and keeps running until you manually close it, or until it crashes.



Note

Typically, the Python kernel does not crash due to exceptions raised when you run a cell, such as a `FileNotFoundException`. The kernel tries to capture exceptions, format them, and display the error as part of the output.

A much more common scenario is a kernel crash caused by memory exhaustion, for example.

To see the list of running kernels, click **Running Terminals and Kernels**. There you can see the kernels from your open kernel tabs and the kernels from the tabs that you closed recently. You can stop individual kernels by hovering and clicking **x**, or you can close all the kernels by clicking **Shutdown All**.

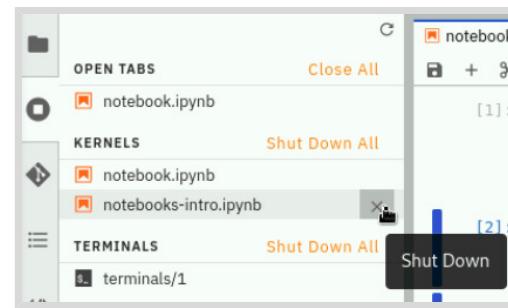
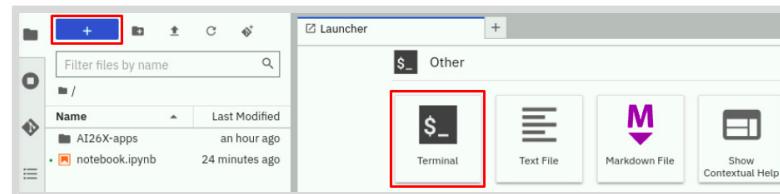


Figure 3.13: Running terminals and kernels

Working with Terminals

You can also open a terminal to have access to the workbench container via JupyterLab. To do this, you can open a new launcher tab and click the **Terminal** card.



This opens a terminal tab in your workbench container working directory.

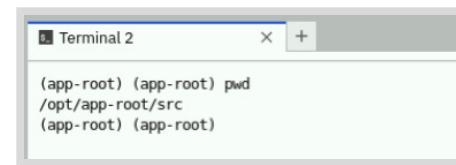


Figure 3.15: Terminal tab



References

Jupyter Project Home Page

<https://jupyter.org>

JupyterLab Documentation

<https://jupyterlab.readthedocs.io/en/3.6.x/index.html>

ipykernel

<https://docs.jupyter.org/en/latest/projects/kernels.html#term-ipynotebook>

Kernels List

<https://github.com/jupyter/jupyter/wiki/Jupyter-kernels>

Built-in magic commands

<https://ipython.readthedocs.io/en/stable/interactive/magics.html>

► Guided Exercise

Introduction to Jupyter Notebooks

Create interactive documents that contain narrative text, executable code, and visualizations.

Outcomes

- Work with Jupyter notebook cells.
- Debug source code in a code cell.
- Export a notebook to a read-only format.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

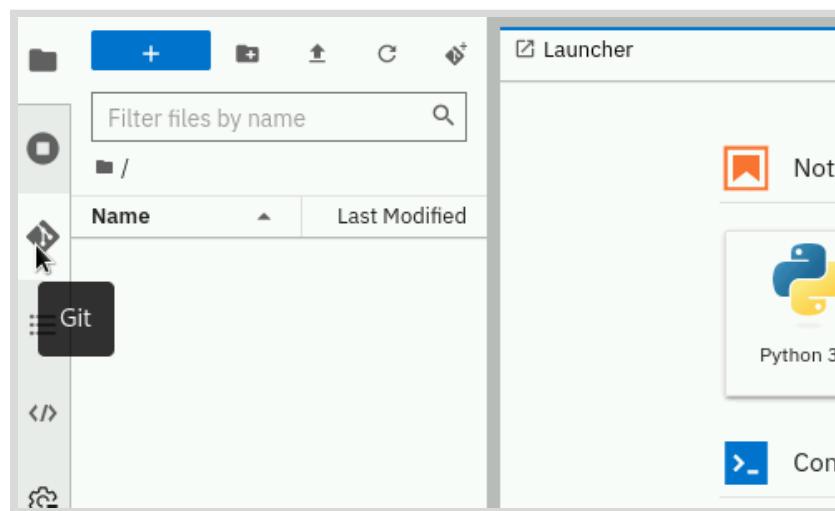
```
[student@workstation ~]$ lab install -u AI262 && \
lab start notebooks-intro
```

Instructions

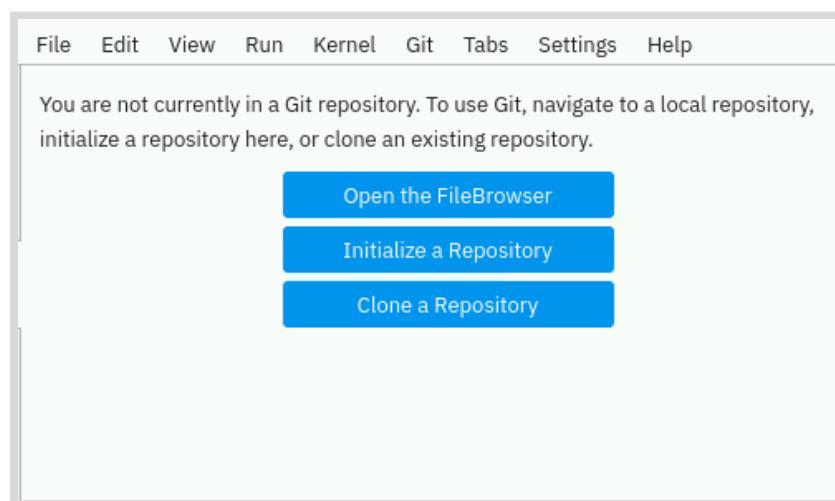
- ▶ 1. Create a workbench called `notebooks-intro-wb`.
 - 1.1. Open the web browser and navigate to <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com>.
 - 1.2. From the Red Hat OpenShift AI (RHOAI) dashboard, click **Data Science Projects**. Then, click the `notebooks-intro` project, and then click **Create workbench** from the **Workbenches** section.
 - 1.3. Enter `notebooks-intro-wb` as the name and select **Standard Data Science** in the **Image selection** menu. Leave the rest of the fields as their default and click **Create workbench**.
 - 1.4. In the **Workbenches** section, wait for the `notebooks-intro-wb` workbench to start. The status column should show **Running**.
- ▶ 2. Open the RHOAI workbench.
 - 2.1. Open the workbench after it starts by clicking **Open** within the same row.
 - 2.2. Log in by clicking `htpasswd_provider` and by using `developer` as both the username and the password.
Click **Allow selected permissions** to allow access.
- ▶ 3. Clone the Jupyter notebook for this exercise.

Chapter 3 | Jupyter Notebooks

- 3.1. In the JupyterLab interface, click the Git icon from the left tools pane.



- 3.2. Click Clone a Repository.



- 3.3. Enter <https://github.com/RedHatTraining/AI26X-apps> as the repository, and click Clone.

- 3.4. In the JupyterLab file browser, navigate to the AI26X-apps/intro/notebooks-intro directory.

► 4. Read and follow the instructions within the notebook for the remainder of this exercise.

Finish

On the workstation machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish notebooks-intro
```

Collaboration with Jupyter Notebooks

Objectives

- Create, import, execute, and share notebooks by using a Git repository.

Collaborating with Git

Developers have extensively used version control systems in software engineering as a crucial component for collaboration and continuous integration and deployment (CI/CD). Similarly, AI and ML projects can introduce version control in their workflow to ease cooperation and enable DevOps activities such as MLOps and GitOps.

Version control systems, and in particular, Git, foster collaboration in the following ways:

- Multiple contributors can work on the same code base simultaneously.
- Contributors can isolate their work in features or fixes by using branches, therefore reducing the probability of conflicts with the main development branch.
- After the work in a branch is complete, contributors can propose, review, and discuss the integration of their changes into the main development branch.
- Users can track changes and revert them if a new change causes a problem.

In Red Hat OpenShift AI (RHOAI), the default workbench images include JupyterLab and *jupyterlab-git*, a Git extension for JupyterLab. This extension adds graphical controls to the JupyterLab interface for interacting with Git repositories.



Note

You can also interact with Git repositories with the `git` CLI, by opening a terminal in JupyterLab. Using the CLI can be useful for advanced Git commands, for users who are familiar with the Git command syntax, or for those who prefer a command-line interface.

Although this section focuses on the Git extension, you can add both options, the CLI and the extension, to your tool set, and use the one that better adjusts to your needs at each moment.

Git Workflows

The collaboration and collective experience of developers, organizations, and open source communities has resulted in several frameworks of good practices and guidelines, called *Git workflows*. These workflows define how teams should work together and coordinate by using common features across Git platforms, such as branches and pull requests. The following workflows are some of the most common:

- GitHub Flow [<https://docs.github.com/en/get-started/quickstart/github-flow>]
- Git Flow [<https://nvie.com/posts/a-successful-git-branching-model/>]
- Trunk-based Development [<https://trunkbaseddevelopment.com/>]

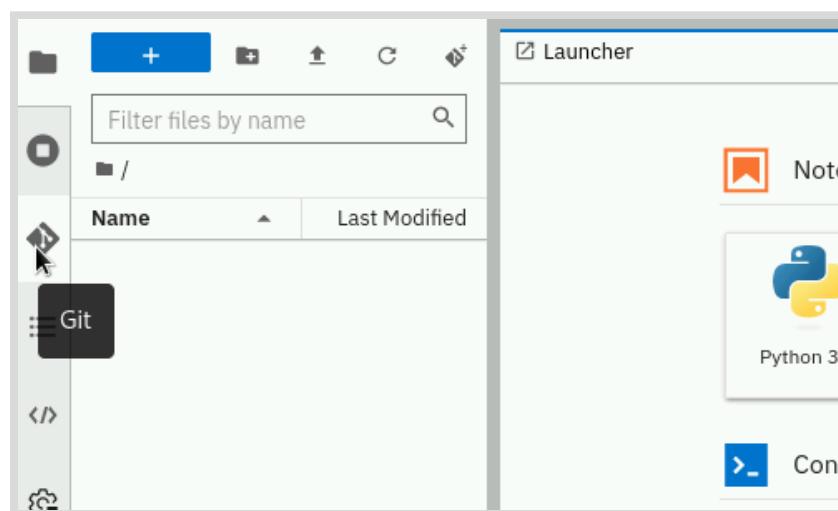
**Note**

Each Git workflow promotes its own but valid approaches to collaboration. Pick the workflow that you find the most suitable for you and your organization, or adjust these workflows to your needs.

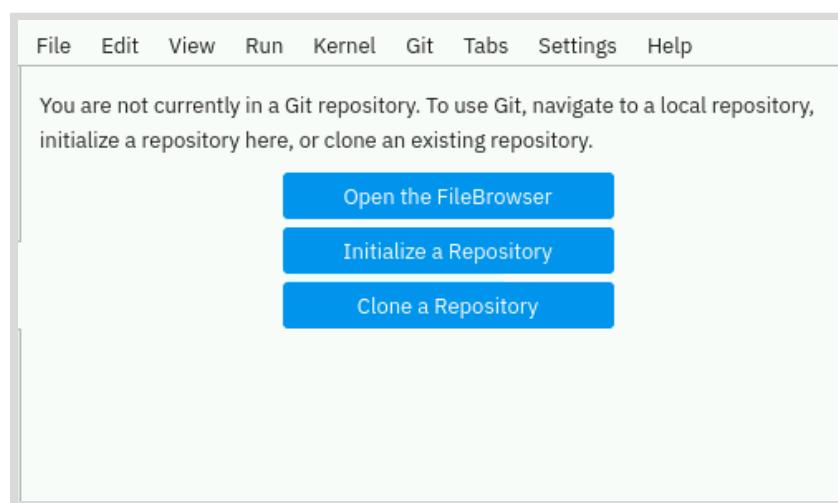
ML/AI and data science projects typically require a high degree of experimentation, especially in the data analysis and model training phases. Arguably, you might want to explore workflows that make it easier to collaboratively develop and track experiments.

Importing a Notebook from Git

To import a notebook from a Git repository, you must clone the repository first. To clone repository, click the Git icon from the left tool bar of JupyterLab.



In the window that opens at the left pane, you can clone a repository, initialize a new repository or return to the file browser to navigate to the directory where you want to clone the repository.



When you click **Clone a repository**, a modal window opens. Enter the URI of the repository and then click **Clone**. JupyterLab clones the repository in your current directory.

Chapter 3 | Jupyter Notebooks

Finally, use the file browser to navigate to the cloned repository and find the notebook that you want to work on.

Adding a Notebook to a Git Repository

To share a new notebook with the other repository contributors, you must first include the notebook in the repository.

Before you add the notebook to the repository, make sure that you have saved the changes. If you have previously opened the notebook, then select the notebook tab at the right pane of JupyterLab, and press **Ctrl+S**. Alternatively, click **File > Save Notebook**.

Click the Git icon in the left tool bar to open the Git extension. Git displays the notebook in the **Untracked** area.

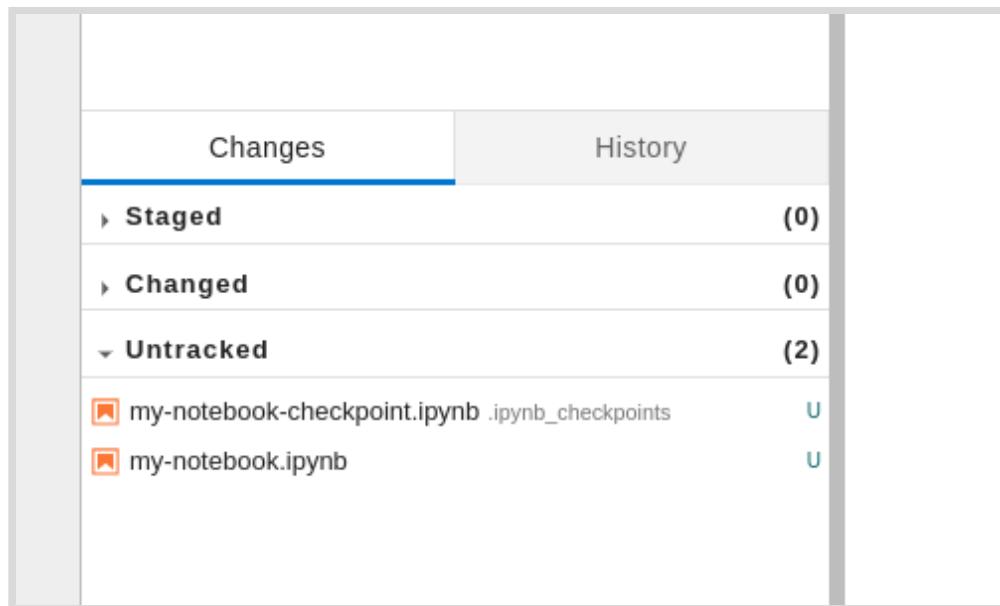


Figure 3.20: Git untracked area



Important

Do not add the `.ipynb_checkpoints` directories to the repository. These directories contain local checkpoint files that JupyterLab creates when you save a notebook. Checkpoints are relative to your own workbench and might create conflicts with the local checkpoints of other contributors.

As a best practice, ignore these files by creating a `.gitignore` file in the root of your repository, if it does not exist, and add the `.ipynb_checkpoints` line to this file.

To start tracking an untracked file, right click the file and then click **+ Track**. Tracking a file means adding the file to the **Git Staging area**. The staging area is a preparation area where you add all the changes that you want to include in your next commit.

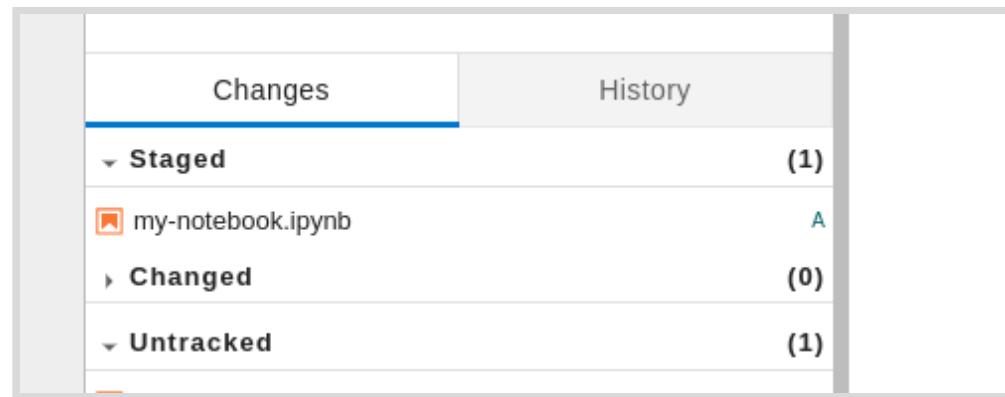


Figure 3.21: Git staged area

After the notebook is in the staging area, commit the staged changes to the repository. At the bottom of the Git extension pane, add a commit message in the **Summary** field and optionally add a description. Click **COMMIT** to commit the changes.

**Note**

Git asks you for your name and email the first time you create a commit in a repository.

To verify the commit, you can use the **History** tab of the Git extension. The information that the history tab provides is similar to the output of the `git log` command.

To finish sharing your work with your peers, push the commit to the remote repository. Click the up-arrow cloud icon at the top of the Git pane. You can also use the **Git > Push to Remote** option at the top menu bar.

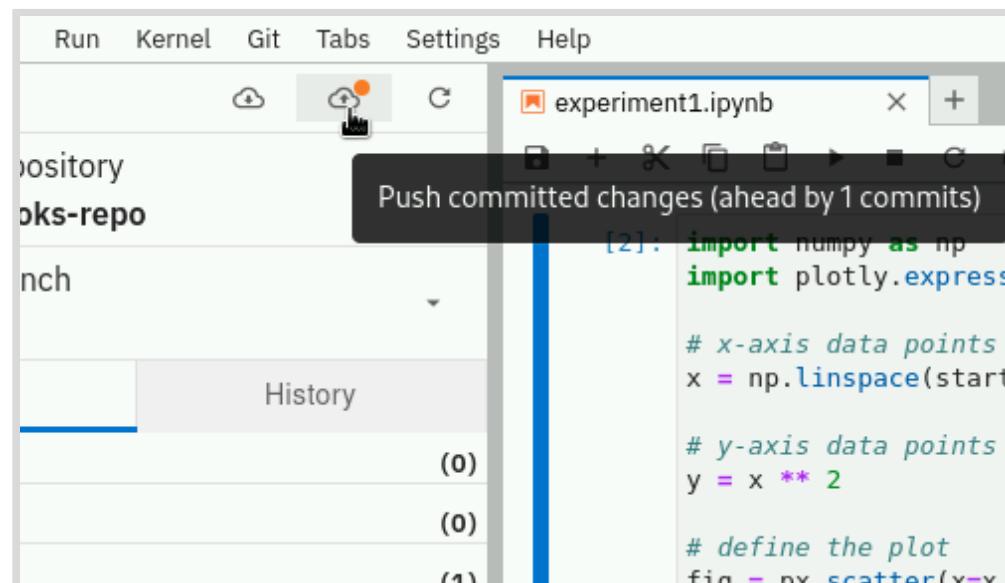


Figure 3.22: Git pull/push/refresh buttons

**Note**

An orange dot at the pull and push buttons indicates that there are pending changes to be pulled or pushed. You can click the refresh icon to verify the pending changes.

To push changes to a remote repository, Git requires your credentials. In JupyterLab, the Git extension opens a modal window for you to enter your Git username and password.

**Note**

If you prefer to authenticate with SSH keys, then you must include your key in the workbench. To inject the key into the workbench container, you have multiple options, which are out of the scope of this section:

- Create a specific SSH key inside your workbench and add to your Git server.
- Upload an SSH key that you already use for Git authentication into the workbench.
- Store your SSH key in a Kubernetes Secret, and then mount the Secret into your workbench. You might require support from the cluster administrator for this option.

After you commit a new notebook to the repository, the repository tracks the changes that you make to that notebook. For files that are already included in the repository, the Git extension displays the changes under the **Changed** section of the **Changes** area. Similarly to newly created files, you must stage the changes before committing them.

Branching

You can create, delete, and switch branches with the Git extension, as you would do with the Git CLI. Click the **Current Branch** selector from the Git pane to view the list of branches.

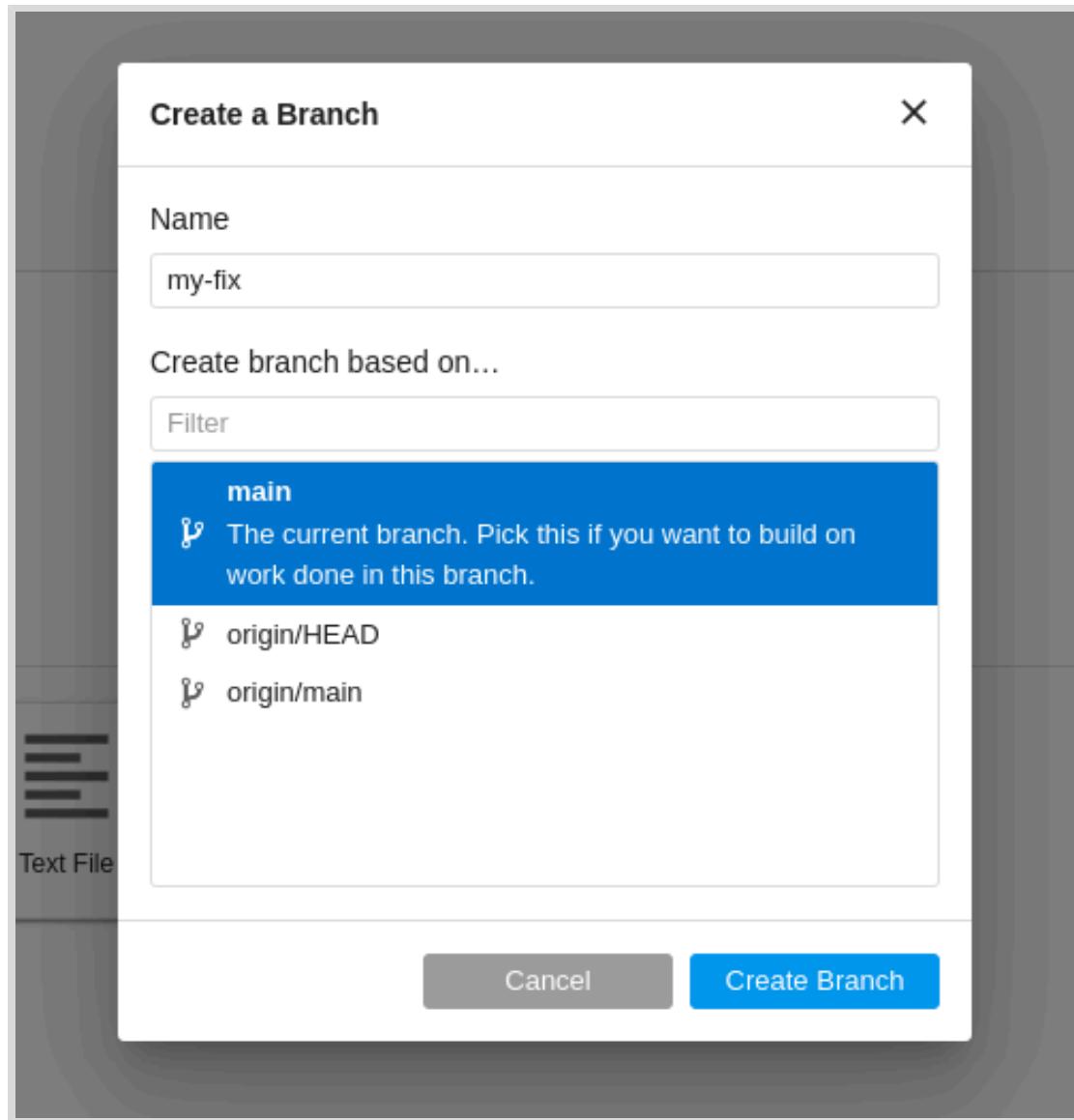
The screenshot shows the Git pane in JupyterLab. At the top, it says "Current Branch" followed by "main". Below this is a tab bar with "Branches" (which is active) and "Tags". A "Filter" input field and a "New Branch" button are also present. The main list contains three entries: "main" (selected), "origin/HEAD", and "origin/main".

Branch
main
origin/HEAD
origin/main

Figure 3.23: Git branches area

Chapter 3 | Jupyter Notebooks

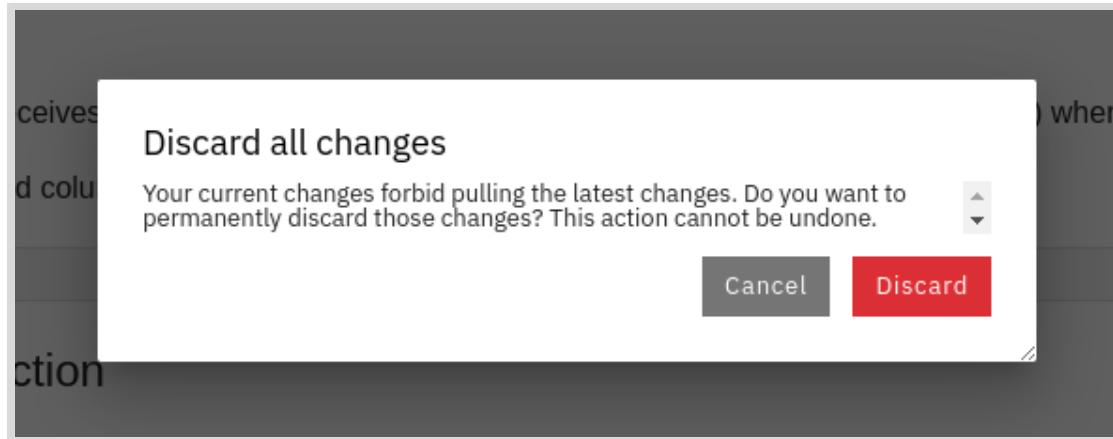
You can use the branches area for viewing branches, tags, and creating branches. You can also switch to a different branch by clicking the branch name from the list. To create a branch, click **New Branch**, then enter the name of the new branch and select the branch that you want to base the new branch on.



Pulling Changes

To work with the latest version of the code, you should regularly pull changes from the remote repository. To pull changes from a remote repository, you can use the **down-arrow cloud icon** at the top of the Git pane, or the **Git > Pull from Remote** option from the top menu bar.

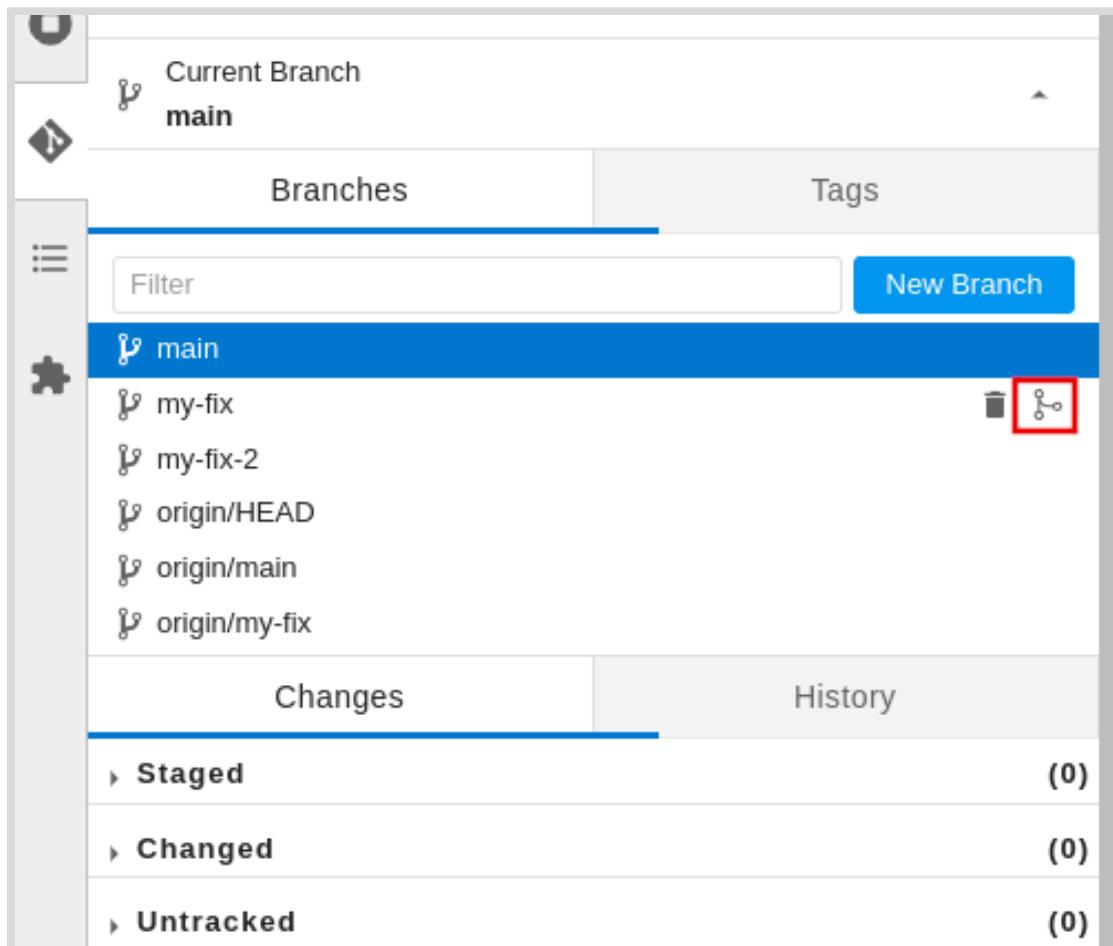
If you have uncommitted changes in your workbench, then you might see a message indicating that your changes forbid pulling the changes, as follows:



To solve this problem, commit your changes in the workbench before pulling.

Merging Branches

You can merge two branches by using the JupyterLab Git extension. First, change to the branch that you want to merge the other branch into, such as `main`. This branch is your current one. Then, hover over the branch that you want to merge into the current branch, and click the **Merge icon**, as displayed in the following screen capture:



Chapter 3 | Jupyter Notebooks

You can also merge two branches by using the **Git > Merge Branch...** option from the top menu bar.

Conflicts

Conflicts happen when you merge two branches that have diverged. For example, a divergence can occur when you or other contributors change the same line, in the same file, in two branches.

When you try to merge two branches and Git detects conflicts, JupyterLab displays an error message at the bottom right of the window.



Note

Clicking the **SHOW** button does not provide more information about the conflict. JupyterLab displays only an internal server error.

If Git conflicts are the cause of the error, then JupyterLab displays the conflicts under the **Changes** area, in the **Conflicted** section.

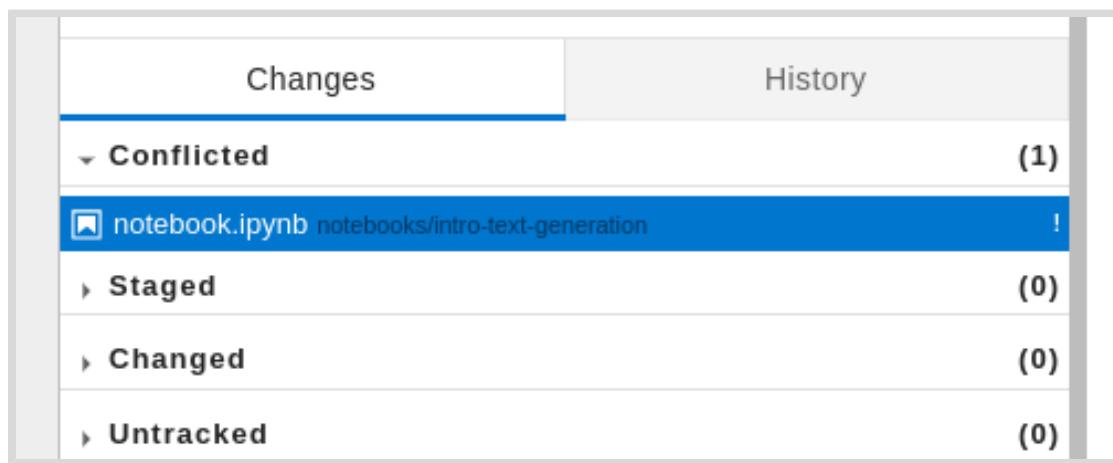


Figure 3.28: Git conflicts area

To fix the conflict, you might want to use the *Diff view*. To open this view, you can hover over the conflicting file and click the **Diff** icon. Alternatively, you can right click the file and click **Diff**, or use the **Git > Double click opens diff** menu item from the top menu bar.

The Notebook Diff View

Solving notebook conflicts can be challenging. In notebooks, conflicts can occur at different elements:

- Notebook metadata, such as kernel information, or Python version.
- Cells, including code and markdown.
- Cell outputs, including text and embedded multimedia content.
- Cell operations, such as creating or removing cells.

The `ipynb` Jupyter notebook format is JSON-based, so one valid approach is to solve conflicts by opening the notebook with a text editor, inspecting, and editing the JSON code. This approach can quickly become error-prone and time consuming, if the notebook contains complex content.

An alternative to JSON-based conflict fixing is the use of a visual editor. The `jupyterlab-git` extension uses the `nbdime` Python package to provide the Diff view feature for solving notebook conflicts in a visual way. The Diff view of a notebook is similar to a table view, as the following screen capture shows:

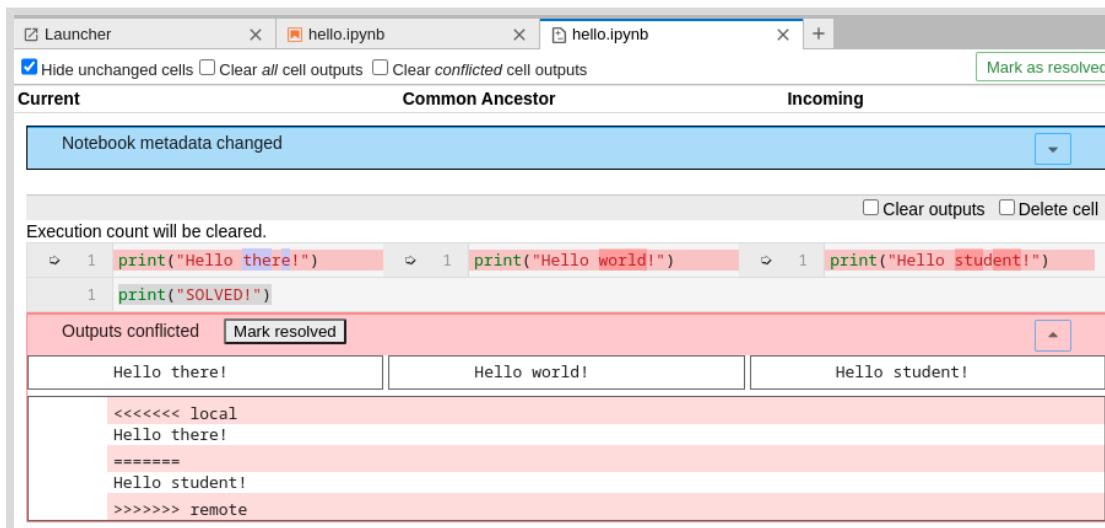


Figure 3.29: Diff view

The columns of this view correspond the current branch, the common ancestor, and the incoming branch. The view includes three major sections of rows:

- The first section with the blue background corresponds to the notebook metadata, which is initially collapsed and displays `Notebook metadata changed`. If you expand this section, then two rows display. The first row displays the metadata, in JSON, of each branch. The second row is a text field where you can edit the metadata to solve the conflict.
- The second section is for cell conflicts. The first row of this section displays the conflicting cell at each branch. The second row is an editable text field where you can solve the conflicts.
- The third section with the red background is for outputs. Similarly to the other sections, the first row displays output at each branch. In the second row, you can solve the output changes by dragging and removing output line by line.

After you have solved the conflicts, click the `Mark as resolved` button at the top right. Git then moves the file to the `Staged` area, so you can commit the merge.

**Note**

Solving output conflicts can be difficult, for example, when the conflicts affect graphs or multimedia content. Instead of using the Diff view to solve output conflicts, you can fix the cell conflicts, mark the notebook as resolved, and then re-execute the notebook cells to regenerate the outputs.

For more information about how to use the Diff view, refer to *nbdime* in the references section.

**References****JupyterLab Documentation**

<https://jupyterlab.readthedocs.io/en/3.6.x/index.html>

jupyterlab-git: A JupyterLab extension for version control using Git

<https://github.com/jupyterlab/jupyterlab-git>

nbdime: diffing and merging of Jupyter Notebooks

<https://nbdime.readthedocs.io/en/stable/>

► Guided Exercise

Collaboration with Jupyter Notebooks

Create, import, execute, and share notebooks by using a Git repository.

Outcomes

- Create notebooks.
- Stage, commit, push and pull notebooks in Git repositories.
- Solve Git conflicts in notebooks.

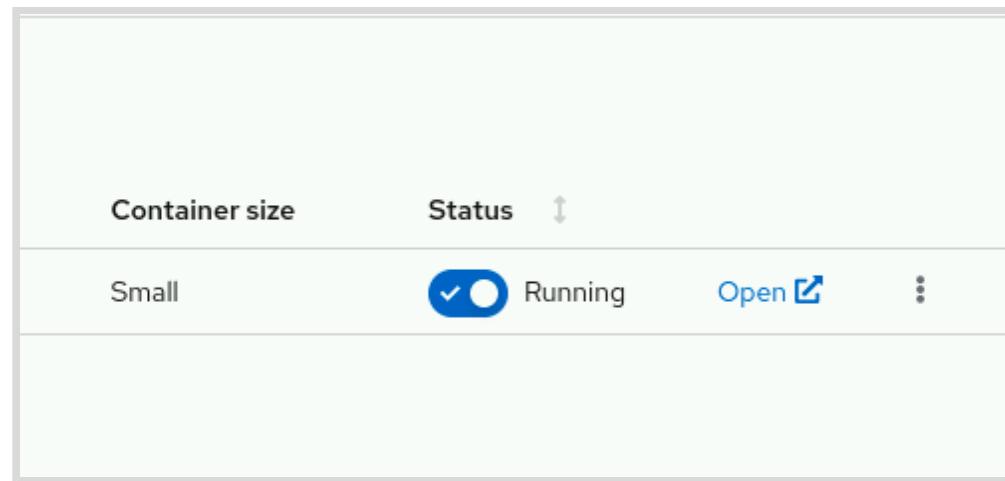
Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

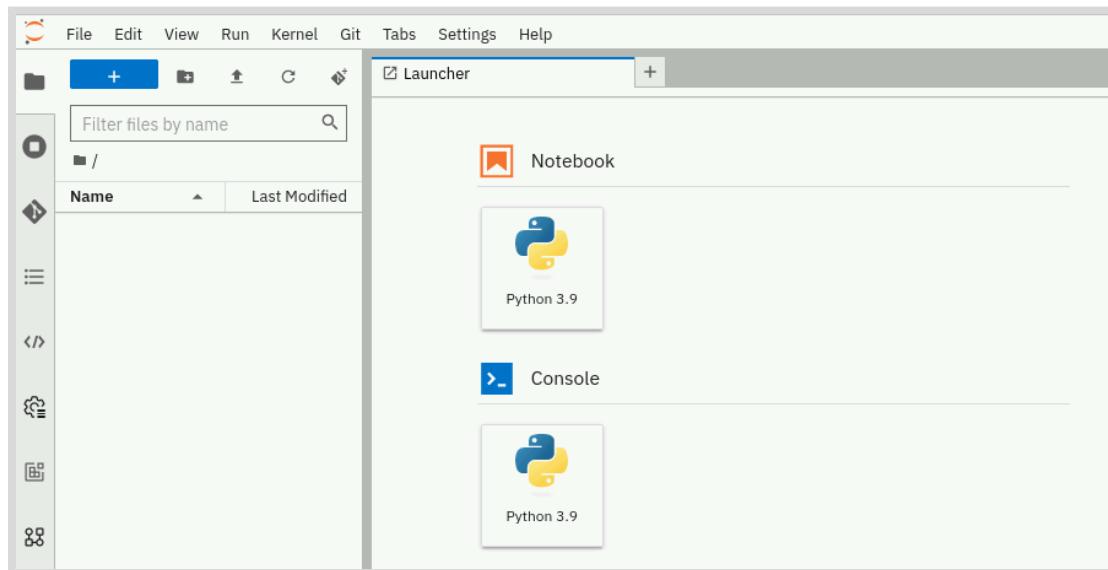
```
[student@workstation ~]$ lab install -u AI262 && \
lab start notebooks-collaboration
```

Instructions

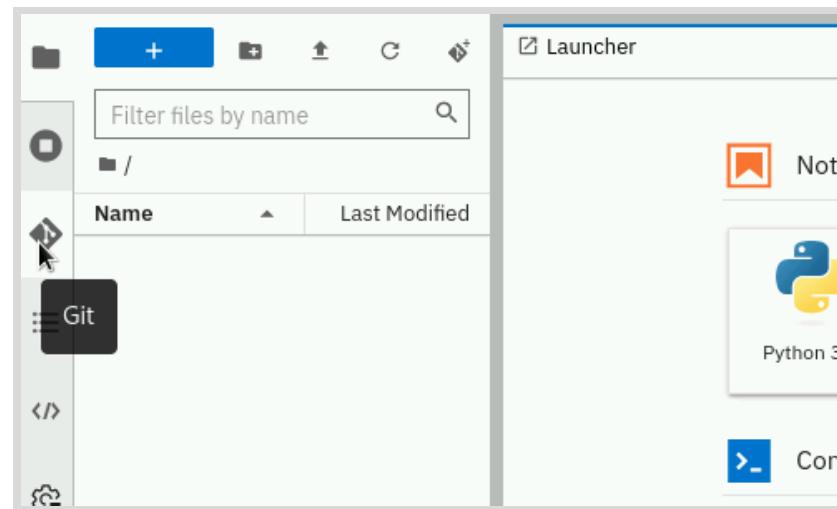
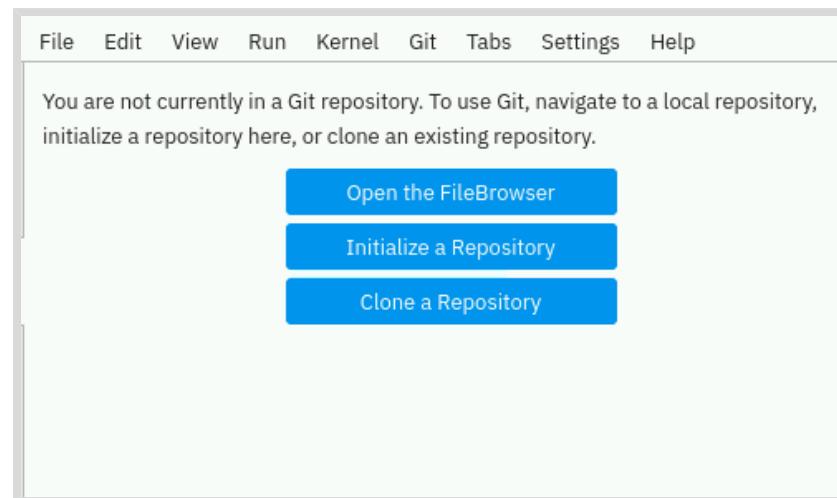
- 1. Create and open a standard data science workbench.
 - 1.1. Open the web browser and navigate to <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com>.
 - 1.2. If RHOAI requests you to log in, then click **Log in with OpenShift**. Next, click **htpasswd_provider**. Enter **developer** as both the username and the password.
 - 1.3. In the left navigation pane, click **Data Science Projects**.
 - 1.4. In the list of projects, click **notebooks-collaboration**.
 - 1.5. In the project dashboard, click **Create workbench**.
 - 1.6. Enter the following values in the form and leave the rest of the values unchanged.
 - Name: **git-experiments**
 - Notebook image > image selection: **Standard Data Science**
 - 1.7. Click **Create workbench**.
 - 1.8. Wait until the **git-experiments** workbench is in the **Running** status and then click **Open**.



- 1.9. Log in into the workbench by using the `htpasswd_provider` option. Use `developer` as the username and the password. In the **Authorize Access** page, click **Allow selected permissions**.
- 1.10. Verify that the JupyterLab interface is visible.



- 2. Clone a Git repository and open a notebook.
- 2.1. In the JupyterLab interface, click the **Git** icon from the left tools pane.

Chapter 3 | Jupyter Notebooks**2.2. Click Clone a Repository.**

- 2.3. Enter <https://github.com/RedHatTraining/AI26X-apps> as the repository, and click **Clone**.
- 2.4. In the JupyterLab file browser, navigate to the `AI26X-apps/intro/notebooks-collaboration` directory and double click the `hello.ipynb` notebook to open it.
- 2.5. Verify that JupyterLab opens the notebook.

A screenshot of the JupyterLab notebook interface. The notebook tab bar shows 'hello.ipynb'. The main area displays a single code cell with the following Python code:

```
import numpy as np
import matplotlib.pyplot as plt
```

The cell has been run, and the output shows:

```
[1]: X = np.arange(0, 10, step=0.1)
```

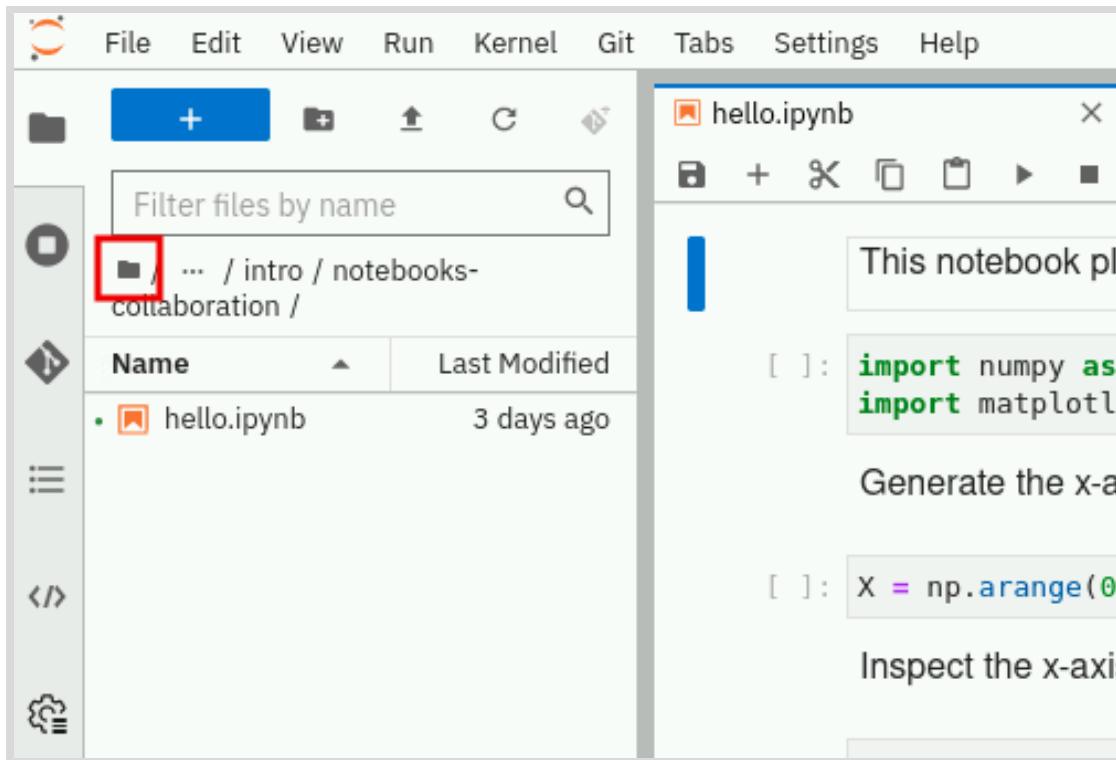
Below the code, there is explanatory text: 'Generate the x-axis points.' and 'Inspect the x-axis data points.'

```
[2]: y = np.sin(X)
```

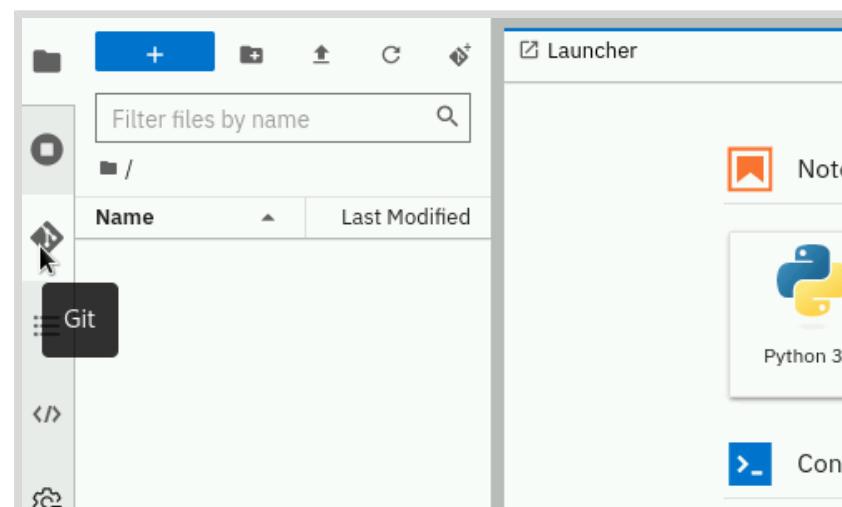
Below the code, there is explanatory text: 'Pass X to the function, to get the function result for the y-axis.'

Chapter 3 | Jupyter Notebooks

- 3. In a different repository, create a notebook and push the notebook to the repository.
- 3.1. Open a new browser tab and navigate to <https://github.com>. If you are not logged in to GitHub, then click **Sign in** in the upper-right corner. If you do not have a GitHub account, then create one by clicking **Sign up**.
 - 3.2. Create a public Git repository called **my - notebooks - repo**. In the upper-right corner of GitHub, click **+**, and then click **New repository**. Enter **my - notebooks - repo** as the repository name and select **Public**. Select **Add a README file** to initialize the repository and click **Create repository**.
 - 3.3. Return to JupyterLab, and navigate to the root directory of the workbench by clicking the directory icon.

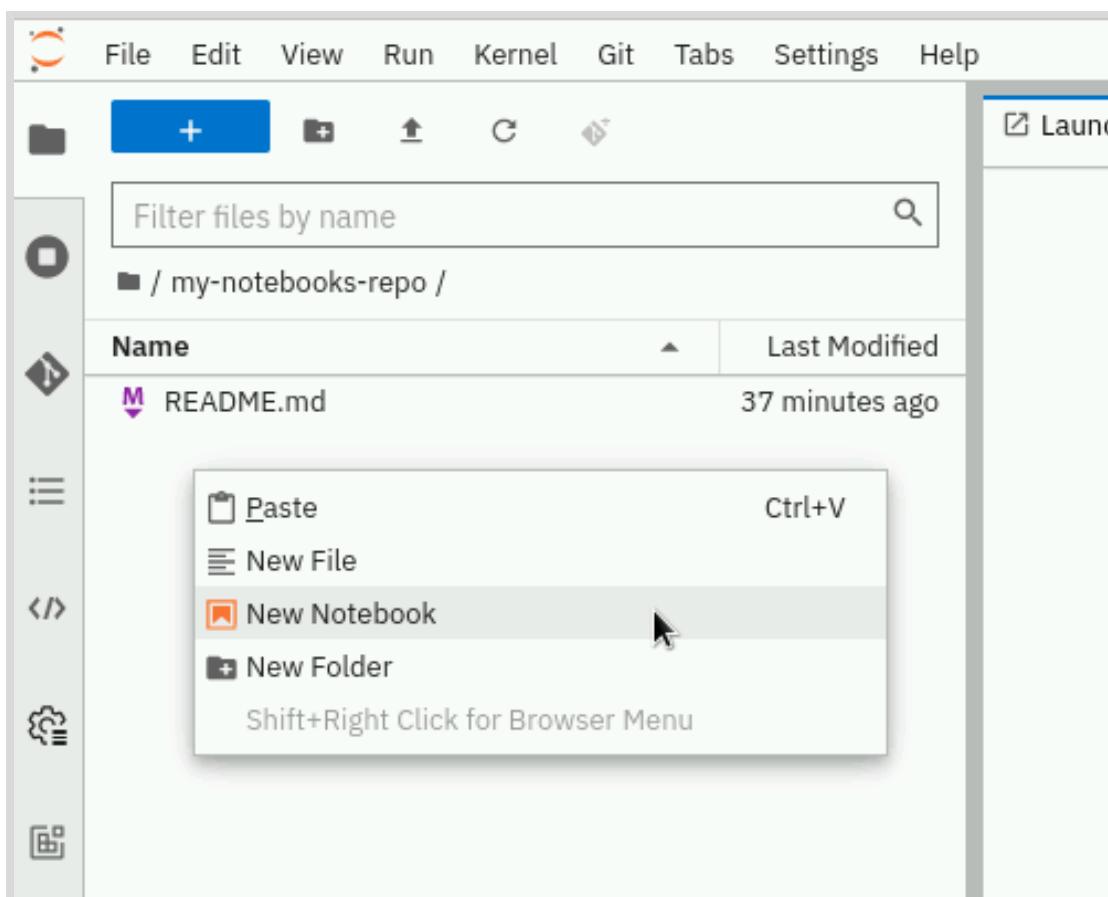


- 3.4. Click the **Git** icon from the left tools pane.



Chapter 3 | Jupyter Notebooks

- 3.5. Click **Clone a Repository**.
- 3.6. Enter https://github.com/YOUR_GITHUB_USERNAME/my-notebooks-repo.git and click **Clone**.
- 3.7. Navigate to the **my-notebooks-repo** in the JupyterLab file browser.
- 3.8. Right click the empty canvas of the file browser, and then click **New Notebook**.



- 3.9. In the window that opens, select Python 3.9 as the kernel and click **Select**.
- 3.10. Enter the following code in the first cell of the notebook:

```
import numpy as np
import plotly.express as px

# x-axis data points
x = np.linspace(start=-50, stop=50)

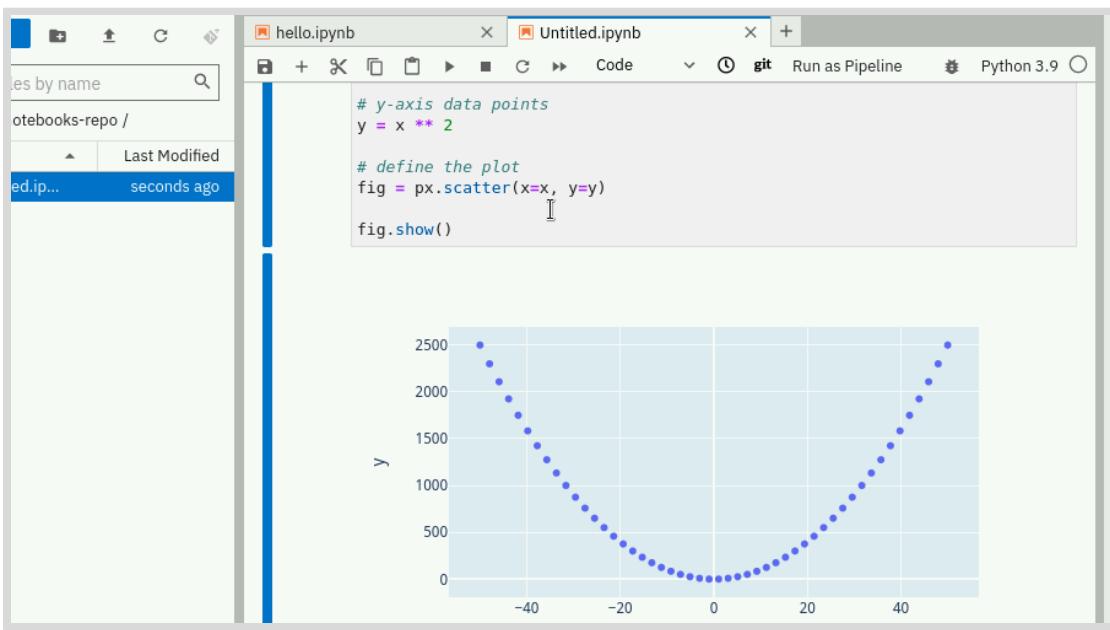
# y-axis data points
y = x ** 2

# define the plot
fig = px.scatter(x=x, y=y)

fig.show()
```

Chapter 3 | Jupyter Notebooks

- 3.11. Execute the cell by pressing **Ctrl+Enter**.



The screenshot shows a Jupyter Notebook interface. On the left, there's a file browser pane showing a folder named 'notebooks-repo' with a file 'hello.ipynb' selected. The main workspace contains two code cells. The first cell contains Python code to generate a scatter plot of $y = x^2$:

```
# y-axis data points
y = x ** 2

# define the plot
fig = px.scatter(x=x, y=y)

fig.show()
```

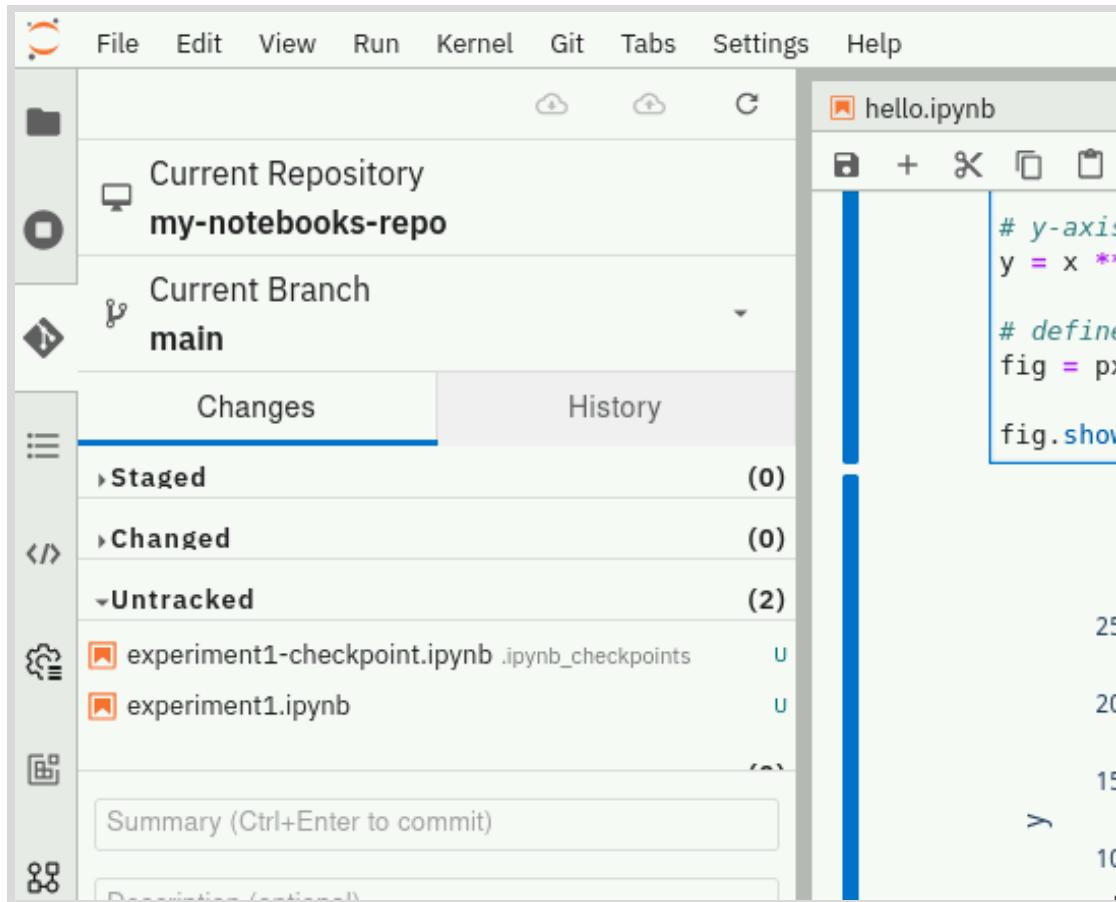
The second cell displays a scatter plot with a parabolic curve, representing the function $y = x^2$. The x-axis ranges from -50 to 50, and the y-axis ranges from 0 to 2500. The data points are blue dots.

- 3.12. Press **Ctrl+S** to save the notebook. Alternatively, you can click **File > Save Notebook**.

A modal window opens to rename the notebook file. Enter **experiment1.ipynb** and click **Rename**.

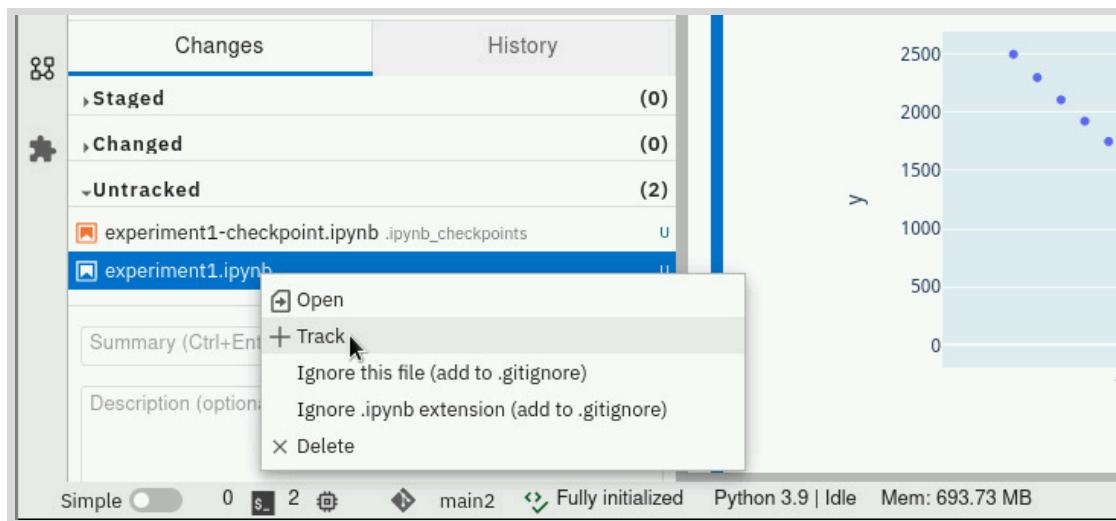
- 4. Push the notebook to the remote repository.

- 4.1. Click the **Git** icon from the left pane. Git displays the **experiment1.ipynb** file as untracked.

**Note**

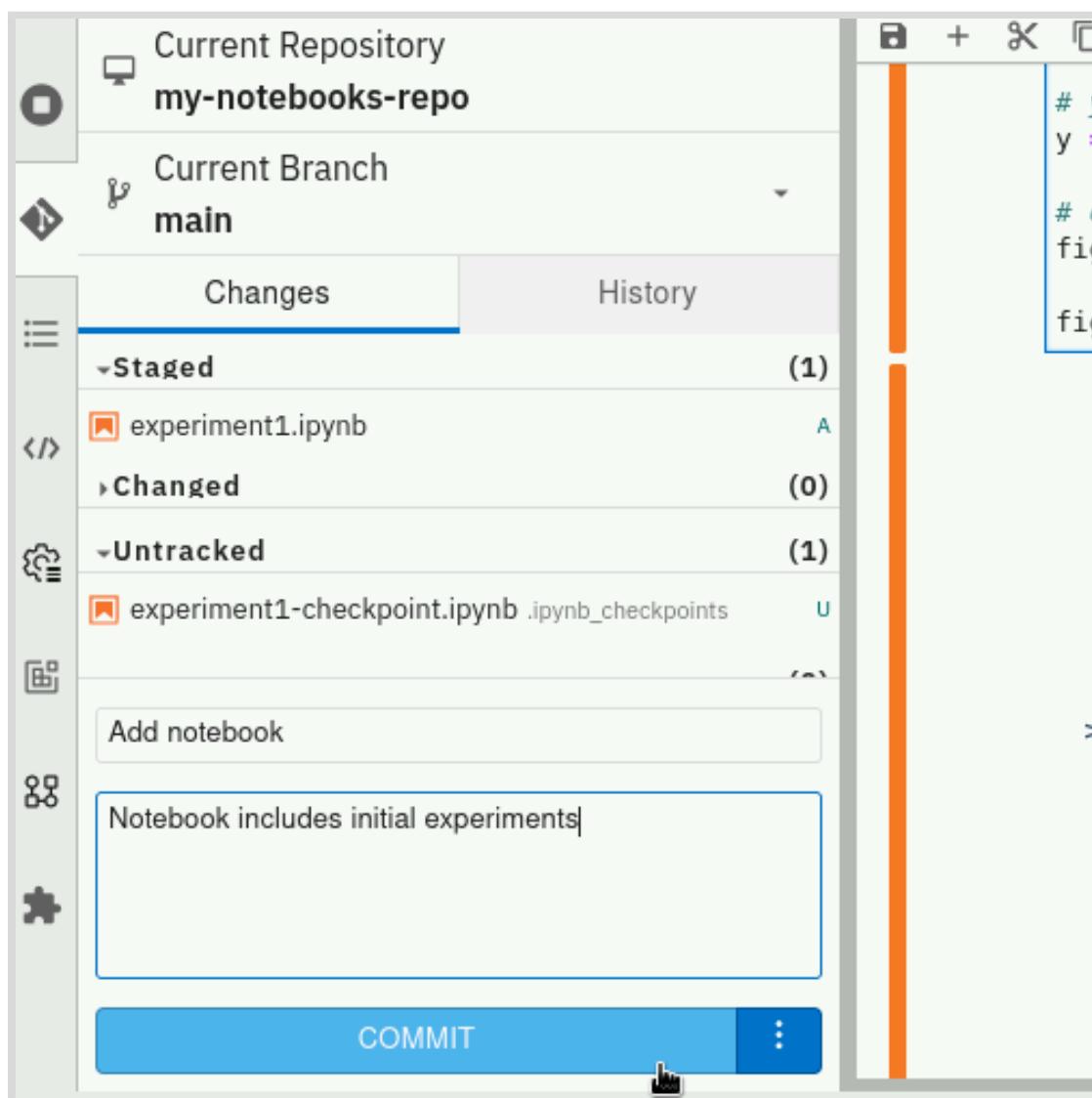
You can ignore the `experiment1-checkpoint.ipynb` file. JupyterLab creates these checkpoints when you save a notebook. Typically, you should not include checkpoint files in the repository.

- 4.2. Add the `experiment1.ipynb` file to the Git staging area. Right click the file and click Track.



Chapter 3 | Jupyter Notebooks

- 4.3. Enter a summary and a description of the changes at the bottom of the Git area, and click **COMMIT**.



- 4.4. At this point, JupyterLab prompts for the committer information. Enter your name and email, and click **OK**.
- 4.5. Verify that Jupyter lab displays the newly created commit. Click the **History** tab, which displays the commit that you just created.

The screenshot shows the Jupyter Notebook interface. On the left, there's a sidebar with icons for file operations, a current repository named 'my-notebooks-repo' under 'main', and a 'Changes' tab showing a recent commit by 'Jaime' (commit ID e4db13e) made 55 seconds ago. Below this is an 'Add notebook' button. The main area has a tab for 'hello.ipynb' containing Python code for generating a scatter plot. A vertical orange bar is visible on the right side of the screen.

```
# y-axis data
y = x**2

# define the figure
fig = px.scatter(x=x, y=y)

fig.show()
```

- 4.6. Push your changes to the remote repository. Click the up-arrow cloud icon at the top of the Git pane.

The screenshot shows the Jupyter Notebook interface with a modal window titled 'Push committed changes (ahead by 1 commits)' overlaid. The modal contains the message 'Push successful'. The background shows the 'experiment1.ipynb' notebook with code for generating a scatter plot. The Git pane on the left shows a repository named 'books-repo' under 'main'. The 'History' tab in the Git pane shows a commit history with '(0)' entries.

```
[2]: import numpy as np
import plotly.express as px

# x-axis data points
x = np.linspace(start=0, end=10, num=100)

# y-axis data points
y = x ** 2

# define the plot
fig = px.scatter(x=x, y=y)

fig.show()
```

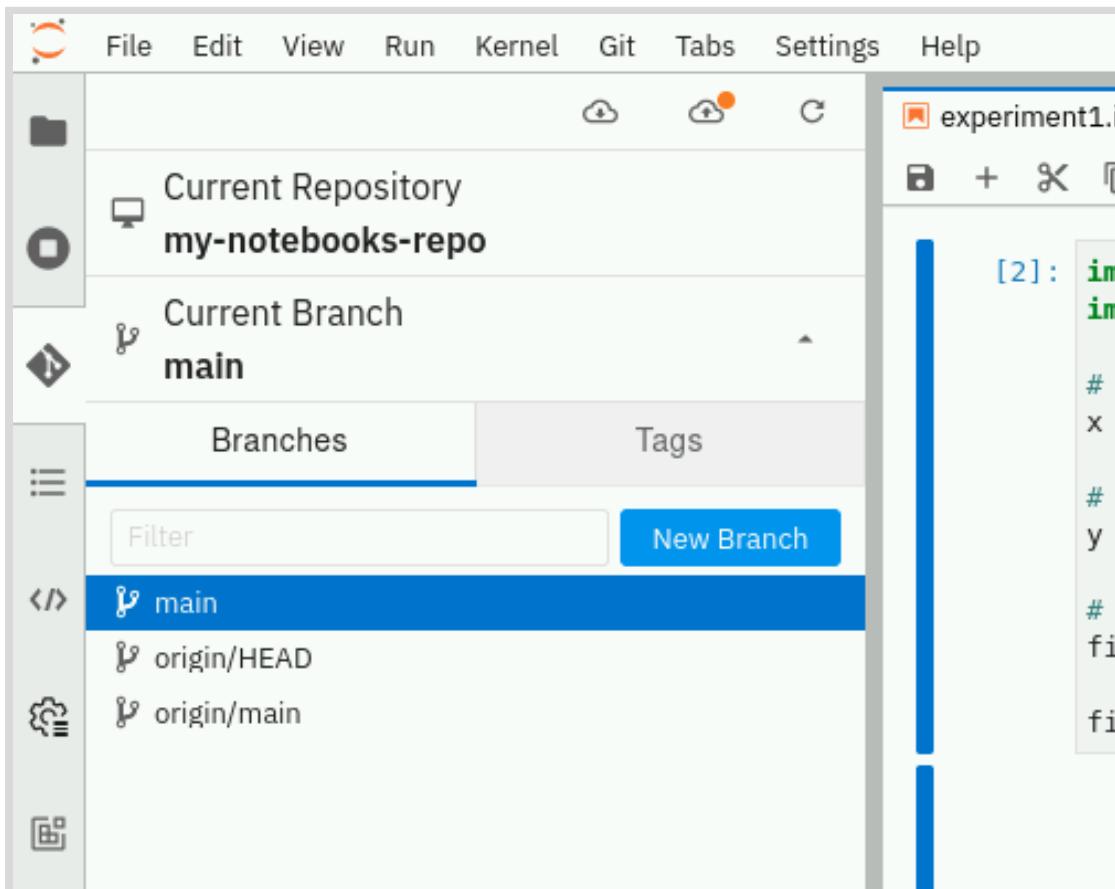
- 4.7. In the modal window that opens, provide the credentials to authenticate to GitHub.



Important

GitHub does not allow the use of passwords for pushing code to repositories. To push code, you must create a personal access token and use it as the password, when Git prompts you for credentials. To create a token, navigate to <https://github.com/settings/tokens>. Ensure that the token includes the repo scope.

- 4.8. Navigate to your GitHub repository at `https://github.com/YOUR_GITHUB_USERNAME/my-notebooks-repo`. Verify that your repository contains the `experiment1.ipynb` file. Notice that GitHub can render the notebook in view-only mode.
- ▶ 5. Create a branch, change the notebook, and commit the changes.
 - 5.1. In the Git pane, click the **Current Branch** selector, which displays `main` as the current branch.
 - 5.2. In the branches area that opens, click **New branch**.

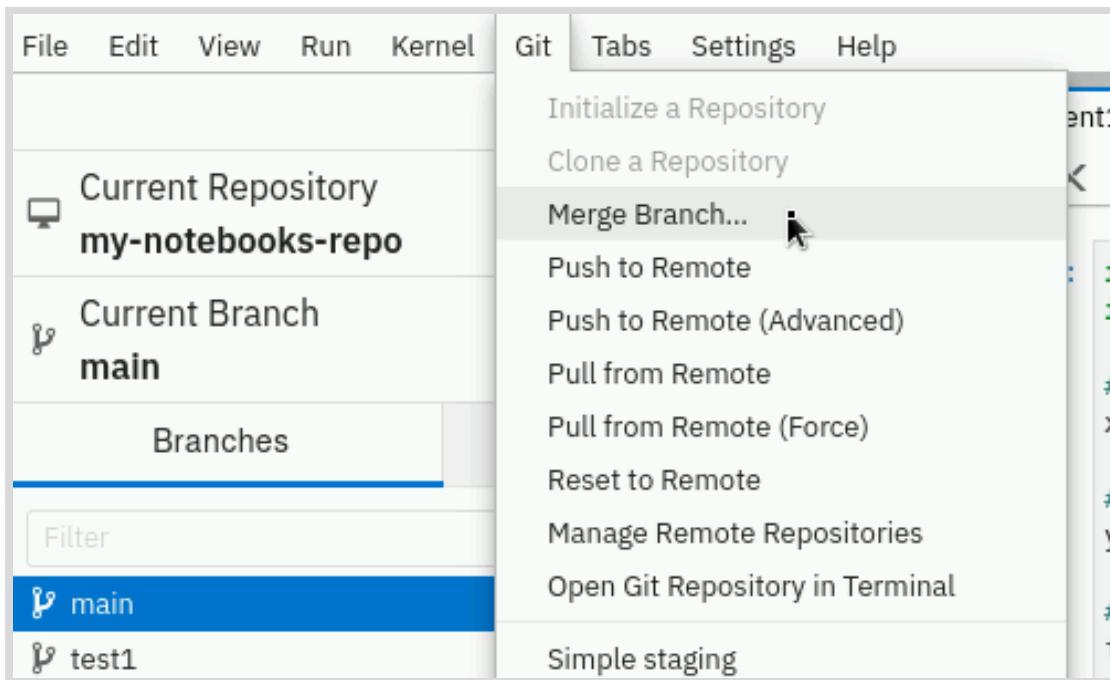


- 5.3. In the **Create a Branch** dialog that opens, enter `test1` as the branch name. Select `main` in the **Create branch based on...** selector and click **Create branch**.
- 5.4. Ensure that the **Current Branch** selector shows `test1`.
- 5.5. Return to the `experiment1.ipynb` notebook. If the notebook is not open, then click in the file browser icon from the left toolbar and reopen the notebook.

Make the following change to the `experiment1.ipynb` notebook.

```
...code omitted...
# y-axis data points
y = x ** 3
...code omitted...
```

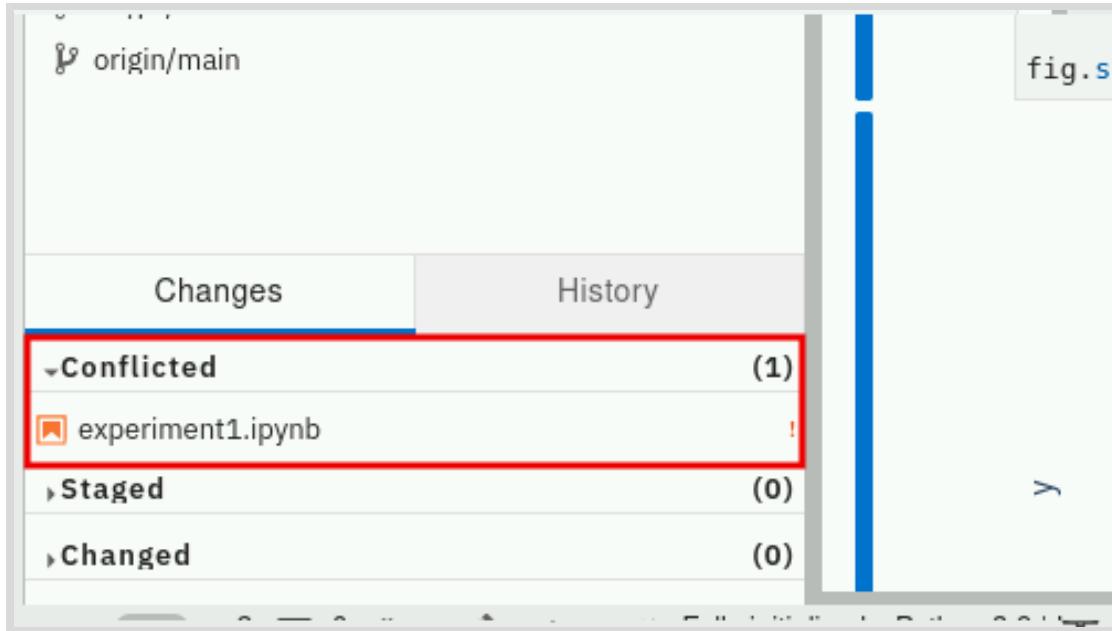
- 5.6. Press **Ctrl+Enter** to run the cell.
 - 5.7. Press **Ctrl+S** to save the notebook.
 - 5.8. Stage the changes. In the Git pane, under the **Changed** section, right-click the `experiment1.ipynb` file and click **Stage**.
 - 5.9. Enter `Change function` as the summary message and click **COMMIT**.
- ▶ 6. Modify the notebook in the `main` branch.
- 6.1. From the Git tab, switch back to the `main` branch. From the **Current Branch** selector, select `main`.
 - 6.2. After you have switched to the `main` branch, make this change to the notebook:
- ```
...code omitted...
y-axis data points
y = x ** 5
...code omitted...
```
- 6.3. Press **Ctrl+Enter** to run the cell.
  - 6.4. Press **Ctrl+S** to save the notebook.
  - 6.5. Stage the changes. In the Git pane, under the **Changed** section, right-click the `experiment1.ipynb` file and click **Stage**.
  - 6.6. Enter `Switch to x ** 5` as the summary message and click **COMMIT**.
- ▶ 7. Merge the `test1` branch into the `main` branch and solve the Git conflict.
- 7.1. From the top menu, click **Git > Merge Branch....**



- 7.2. In the Merge Branch window, select test1 to merge the test1 branch into main. Click Merge.
- 7.3. Verify that the merge fails.



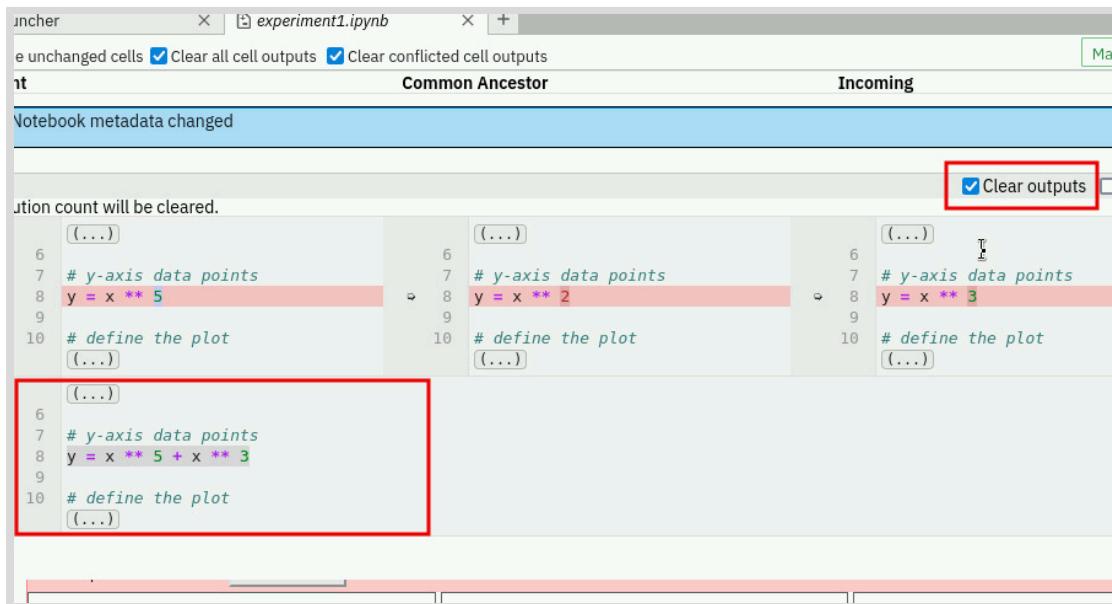
- 7.4. Verify that Git displays the conflicting files, and double click experiment1.ipynb to open the Diff view.



- 7.5. Fix the conflict. In the second row of cells, specify the conflict resolution for the conflicting cell. Change the cell code as follows:

```
y = x ** 5 + x ** 3
```

Select Clear outputs.



- 7.6. Click Mark as resolved.



- 7.7. If the `experiment1.ipynb` notebook is open, then close it. Next, reopen the notebook from the file browser.
- 7.8. Click the first cell of the `experiment1.ipynb` notebook and press `Ctrl+Enter` to regenerate the output.
- 7.9. Press `Ctrl+S` to save the notebook.

## Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish notebooks-collaboration
```

# Summary

---

- A *notebook*, also called Jupyter Notebook, is a file that contains code and multimedia content, including text, images, and audio. When you save a notebook with outputs from code executions, you also save the execution outputs to the file.
- *JupyterLab* is the execution environment that runs the notebook files. It exposes a web interface for you to visualize, edit and execute the notebooks via a web browser.
- In the context of JupyterLab, a kernel is the process that runs the code cells in a notebook. A kernel determines the programming language for the notebook.
- A notebook is made of blocks, called *cells*, which can be *Markdown cells* or *code cells*. Markdown cells contain Markdown content and are intended to add rich text to the notebook. Code cells contain executable code in a programming language that the kernel supports.
- AI and ML projects can introduce Git version control in their workflow to ease cooperation and enable DevOps activities such as MLOps and GitOps.
- In Red Hat OpenShift AI (RHOAI), the default workbench images include JupyterLab and *jupyterlab-git*, a Git extension for JupyterLab, which provides a graphical user interface (GUI) to interact with Git.

## Chapter 4

# Installing Red Hat OpenShift AI

### Goal

Installing Red Hat OpenShift AI by using the web console and the CLI, and managing Red Hat OpenShift AI components.

### Sections

- Red Hat OpenShift AI Installation (and Guided Exercise)

# Red Hat OpenShift AI Installation

---

## Objectives

- Installing Red Hat OpenShift AI by using the web console and the CLI.

## Red Hat OpenShift AI Deployment Options

You can install Open Data Hub, which is the upstream project of Red Hat OpenShift AI (RHOAI), on any OpenShift deployment option. Red Hat offers a enterprise-grade supported Red Hat OpenShift AI (RHOAI) service, based on Open Data Hub, in two different deployment options:

- As a self-managed service running on Red Hat OpenShift Container Platform (RHOCP).
- As a managed cloud service running on Red Hat OpenShift Dedicated, or Red Hat OpenShift Service on AWS (ROSA).

If you use the RHOAI managed cloud service, then you do not need to perform the RHOAI components installation. This course focuses on the RHOAI self-managed platform. To learn more about the RHOAI cloud service, see the *Red Hat OpenShift AI Service Definition* article in the references section.

## RHOAI Components for Installation

The following diagram shows the components of a RHOAI service running on different hybrid cloud platforms.

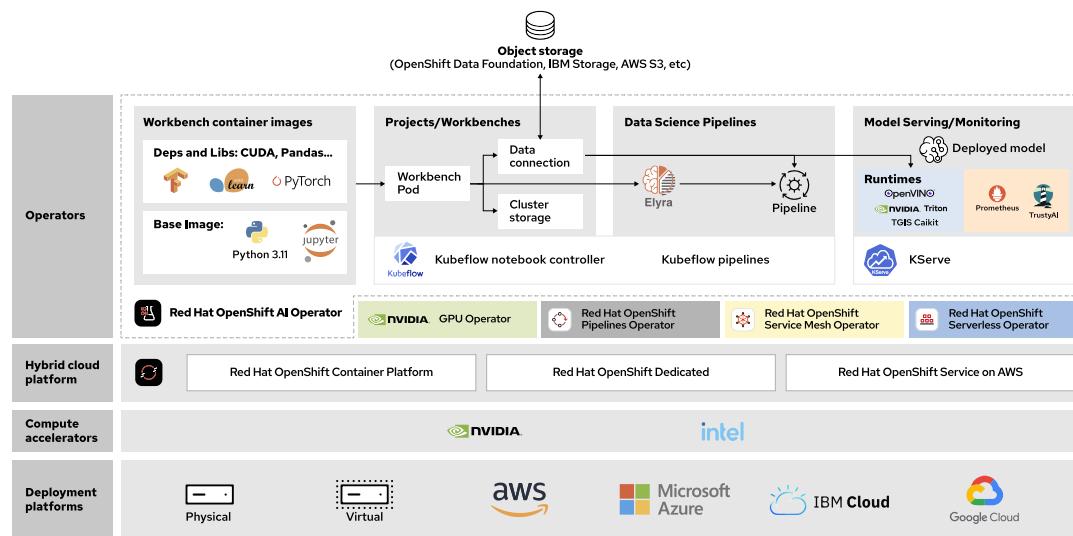


Figure 4.1: RHOAI Components



### Note

The TrustyAI feature for model monitoring has been removed, starting with RHOAI 2.7.

Different OpenShift operators manage all the technical integration between the models, workbenches, storage, data connections, pipelines, notebooks, and all the AI/ML objects and tools that RHOAI provides.

### RHOAI operator

The RHOAI operator is the main component of RHOAI, and deploys and manages all the dependant operators and components included in RHOAI. From the administration perspective, this operator provides three Kubernetes objects, or custom resource definitions (CRDs):

- The FeatureTracker CRD
- The DSCInitialization CRD
- The DataScienceCluster CRD (DSC)

The DataScienceCluster object creates the FeatureTracker and the DSCInitialization objects. The DSCInitialization object verifies that all the Kubernetes objects needed to exist, such as namespaces, configmaps, network policies, services, and roles.

FeatureTracker objects maintain references to OpenShift resources created by the operator, to facilitate the garbage collection of unused objects. As a RHOAI administrator you only create DataScienceCluster objects.

You can define only one DataScienceCluster object in an OpenShift cluster.

When creating a DataScienceCluster you can choose which RHOAI components to install, by defining the state of each component with Removed, or Managed. The following example is the YAML definition of a DataScienceCluster:

```
kind: DataScienceCluster
apiVersion: datasciencecluster.opendatahub.io/v1
metadata:
 name: rhods
 labels:
 app.kubernetes.io/name: datasciencecluster
 app.kubernetes.io/instance: rhods
 app.kubernetes.io/part-of: rhods-operator
 app.kubernetes.io/managed-by: kustomize
 app.kubernetes.io/created-by: rhods-operator
spec:
 components:
 codeflare: ①
 managementState: Removed
 dashboard: ②
 managementState: Managed
 datasciencepipelines: ③
 managementState: Managed
 kserve: ④
 managementState: Removed
 serving:
 ingressGateway:
 certificate:
 type: SelfSigned
 managementState: Removed
 name: knative-serving
 kueue: ⑤
 managementState: Removed
```

```
modelmeshserving: ⑥
 managementState: Managed
 ray: ⑦
 managementState: Removed
 workbenches: ⑧
 managementState: Managed
```

- ① *Codeflare* is an IBM software stack for developing and scaling machine-learning and Python workloads. It uses and needs the *Ray* component. The *Codeflare* component on RHOAI is a Technology Preview feature.
- ② Provides the RHOAI dashboard.
- ③ Enables you to build portable machine learning workflows. Requires the OpenShift Pipelines Operator to be present before enabling the data science pipelines.
- ④ *KServe* is a Kubernetes based serverless framework. RHOAI uses *Kserve* to serve large language models that can scale based on demand. Requires the OpenShift Serverless and the OpenShift Service Mesh operators to be present before enabling the component.
- ⑤ *Kueue* is a set of APIs and controllers for job queueing.
- ⑥ *KServe* also offers a component for general-purpose model serving, called *ModelMesh Mesh Serving*. Activate this component to serve small and medium size models.
- ⑦ Component to run the data science code in a distributed manner. The *Ray*, or *KubeRay*, component on RHOAI is a Technology Preview feature.
- ⑧ Workbenches are containerized and isolated working environments for data scientists to examine data and work with data models. Data scientists can create workbenches from an existing notebook container image to access its resources and properties. Workbenches are associated to container storage to prevent data loss when the workbench container is restarted or deleted.

To learn more about each component, see the references section.

### NVIDIA GPU and Node Feature Discovery operators

You need to use these operators to use NVIDIA graphical processing units (GPUs). The NVIDIA GPU operator is supported by NVIDIA. The Node Feature Discovery operator manages the hardware features detection and labels the OpenShift nodes with hardware information. To learn more about these operators, see the references section.

### Red Hat OpenShift Pipelines operator

The data science pipelines RHOAI component uses the CI/CD platform that this operator provides. To learn more about the Red Hat OpenShift Pipelines operator installation, see the references section.

### Red Hat OpenShift Serverless and Red Hat OpenShift Service Mesh operators

If you serve large models, then you need all components of the RHOAI model serving platform. The *KServe* CRD manages and orchestrates the lifecycle of deployment objects, storage accesses, and networking setup. The Red Hat OpenShift Service Mesh operator provides a network of microservices. It is based on the Istio open source project. Red Hat OpenShift Serverless provides a serverless runtime based on the Knative open source project. To learn more about Red Hat OpenShift Service Mesh and Red Hat OpenShift Serverless, see the references section.

## RHOAI Requirements

The following are the requirements for the RHOAI operator installation.

- A product subscription to RHOAI Self-Managed.
- Access to a RHOCOP cluster in version 4.12 or later, with *cluster-admin* privileges.

The RHOCOP cluster must have the following resources:

- Two worker nodes with at least 8 CPUs and 32 GiB of RAM available for the RHOAI operator.
- A default storage class with at least `ReadWriteOnce` (RWO) mode. RHOAI uses the OpenShift default storage class for dynamically provisioning persistent volumes for resources such as workbenches.
- An identity provider. RHOAI uses the same identity provider as the OpenShift cluster.
- All the operators that your installation needs. As explained previously, if you plan to use some RHOAI features, then you must install the operators before installing the RHOAI operator. The operator versions compatible with RHOAI are the versions compatible for the RHOCOP cluster. For example, at the time of writing this lecture, you can find the most recent version of the Red Hat OpenShift Service Mesh operator in the RHOCOP 4.14 documentation. To see the complete list of specific versions, see the *Red Hat OpenShift AI: Supported Configurations* article in the references section.

Note that these requirements do not include the resources needed for the other operators, or for the models that data scientists deploy on RHOAI.

## Installation Steps for RHOAI Self-Managed

### Installation by using the RHOCOP administration console

1. Log in to the RHOCOP administrator console by using a user that has the `cluster-admin` role assigned.
2. Navigate to **Operators > OperatorHub**, and search for **Red Hat OpenShift Data Science**.
3. Click the Red Hat OpenShift Data Science operator.

The screenshot shows the Red Hat OpenShift OperatorHub interface. On the left, there's a sidebar with a navigation menu and a search bar. The main area is titled "OperatorHub" and displays a list of operators categorized under "All Items". One operator, "Red Hat OpenShift AI", is highlighted. A modal window for "Red Hat OpenShift AI" is open, showing its details: version 2.8.0 provided by Red Hat. The modal includes sections for "Channel" (set to "stable"), "Version" (set to "2.8.0"), and "Capability level" (checkboxes for "Basic Install", "Seamless Upgrades", "Full Lifecycle", and "Deep Insights" are checked). To the right of the modal, descriptive text explains that Red Hat OpenShift AI is a complete platform for the entire lifecycle of AI/ML projects, mentioning tools like PyTorch, tensorflow, and CUDA.

Then, click **Install** to open the operator's installation view.

4. In the **Install Operator** page you can customize some parameters:

The screenshot shows the "Install Operator" page for the Red Hat OpenShift AI operator. The top navigation bar includes the Red Hat OpenShift logo and user information. The main content area has a title "Install Operator" and a subtitle "Install your Operator by subscribing to one of the update channels to keep the Operator up to date. The strategy determines either manual or automatic updates." Below this, there are three configuration sections: "Update channel" (set to "stable"), "Version" (set to "2.8.0"), and "Installation mode" (radio button selected for "All namespaces on the cluster (default)"). To the right, there's a summary section for "Red Hat OpenShift AI" which includes the provider information "provided by Red Hat" and a "Provided APIs" table. The table lists the "Data Science Cluster" API, marked as required, with a note that "DataScienceCluster is the Schema for the datascienceclusters API". At the bottom of the summary section, there's a link "DSCI DSC Initialization".

#### **The Update channel used to search for new operator versions**

There are five update channels:

- **fast**: Intended for production use. One month of full support and a release cycle of one month.

## Chapter 4 | Installing Red Hat OpenShift AI

- **stable**: Intended for production use. Three months of full support and a release cycle of three months.
- **stable-x.y**: Intended for production use. Seven months of full support and a release cycle of three months. It allows to plan and execute the upgrade to the next stable release while keeping the deployments under full support.
- **eus-x.y**: Intended for enterprise-grade environments that cannot upgrade within a seven month window. Seven months of full support followed by Extended Update Support for eleven months and a release cycle of nine months.
- **alpha**: Intended for development use.

### The Installed Namespace where the RHOAI operator is installed

Red Hat recommends to use the default name for this operator.

### The Update approval for the RHOAI operator

If you select **Manual** then the RHOAI operator updates need manual approval.

After selecting the parameters, click **Install**.

The operator finishes the installation after a few minutes, and the cluster is ready to create the main RHOAI operator object, the DSC.

5. Click **Create DataScienceCluster**, and configure it.

```
1 apiVersion: datasciencecluster.opendatahub.io/v1
2 kind: DataScienceCluster
3 metadata:
4 name: rhods
5 labels:
6 app.kubernetes.io/name: datasciencecluster
7 app.kubernetes.io/instance: rhods
8 app.kubernetes.io/part-of: rhods-operator
9 app.kubernetes.io/managed-by: kustomize
10 app.kubernetes.io/created-by: rhods-operator
11 spec:
```

Accessibility help | View shortcuts | Show tooltips

DataScienceCluster

Schema

DataScienceCluster is the Schema for the datascienceclusters API.

- apiVersion string

APIVersion defines the versioned schema of this representation of an object. Servers should convert

By default, you see the YAML editor for the DSC object to create. The YAML definition for the DSC configures as **Managed** to every component that is a General Availability component, and as **Removed** only to the components that are Technical Preview features.

Finally, click **Create** to instantiate the DSC object on the RHOCP cluster.

### Installation by using the OpenShift CLI

1. Log in to the RHOCP cluster by using a user that has the `cluster-admin` role assigned, and create the namespace for the RHOAI operator.

```
[user@host ~]$ oc login -u adminuser -p thepassword https://openshift-api-url:6443
Login successful.
...output omitted...
[user@host ~]$ oc create namespace redhat-ods-operator
namespace/redhat-ods-operator created
```

2. Create the OperatorGroup and the Subscription.

Create the `rhoai-operator-group.yaml` file with the following YAML definition:

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
 name: rhods-operator
 namespace: redhat-ods-operator
```

Then, create the operator group:

```
[user@host ~]$ oc create -f rhoai-operator-group.yaml
operatorgroup.operators.coreos.com/rhods-operator created
```

3. Create the subscription.

Create the `rhoai-subscription.yaml` file with the following YAML definition:

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
 name: rhods-operator
 namespace: redhat-ods-operator
spec:
 name: rhods-operator
 channel: stable
 source: redhat-operators
 sourceNamespace: openshift-marketplace
```

Then, create the subscription:

```
[user@host ~]$ oc create -f rhoai-subscription.yaml -n redhat-ods-operator
subscription.operators.coreos.com/rhods-operator created
```

At this point, the RHOAI operator uses the following three projects on the RHOCP cluster:

- `redhat-ods-applications`
- `redhat-ods-monitoring`
- `redhat-ods-operator`

Right after this step, the `redhat-ods-operator` project is the only one that contains running resources. The `redhat-ods-applications` and `redhat-ods-monitoring` namespaces do not start any pod until the DSC object is created.

4. Create the DSC object.

Create the `default-dsc.yaml` file with the following YAML definition:

```
apiVersion: datasciencecluster.opendatahub.io/v1
kind: DataScienceCluster
metadata:
 name: default-dsc
spec:
 components:
 codeflare:
 managementState: "Removed"
 dashboard:
 managementState: "Managed"
 datasciencepipelines:
 managementState: "Removed"
 kserve:
 managementState: "Removed"
 modelmeshserving:
 managementState: "Managed"
 ray:
 managementState: "Removed"
 workbenches:
 managementState: "Managed"
```

Then, create the DSC:

```
[user@host ~]$ oc create -f default-dsc.yaml -n redhat-ods-operator
datasciencecluster.datasciencecluster.opendatahub.io/default-dsc created
```

After a few minutes, the DSC object creates a new `rhods-notebooks` namespace, and the needed OpenShift resources. The `redhat-ods-applications` namespace contains the Kubernetes Deployment objects. The number of deployments in this namespace depends on the components that you have chosen in the DSC definition:

```
[user@host ~]$ oc get deployments -n redhat-ods-applications
NAME READY UP-TO-DATE AVAILABLE AGE
etcd 1/1 1 1 2m2s
modelmesh-controller 3/3 3 3 2m2s
notebook-controller-deployment 1/1 1 1 2m5s
odh-model-controller 3/3 3 3 2m1s
odh-notebook-controller-manager 1/1 1 1 2m6s
rhods-dashboard 5/5 5 5 2m9s
```

## Uninstallation Steps for RHOAI Self-Managed

To uninstall the RHOAI platform from a RHOC cluster, Red Hat recommends using the `oc` CLI tool. You must preserve the content of data science projects by backing up the contents of all persistent volume claims that contain the data scientists' work.

1. Log in to the RHOC cluster by using a user that has the `cluster-admin` role assigned, and delete the DSC object.

```
[user@host ~]$ oc login -u adminuser -p thepassword https://openshift-api-url:6443
Login successful.
...output omitted...
[user@host ~]$ oc get datasciencecluster
NAME AGE
default-dsc 22m
[user@host ~]$ oc delete datasciencecluster default-dsc
datasciencecluster.datasciencecluster.opendatahub.io "default-dsc" deleted
```

The resources created by the DSC object are deleted, both deployments and namespaces.

2. Delete the DSCInitialization object.

```
[user@host ~]$ oc get dscinitialization
NAME AGE PHASE CREATED AT
default-dsci 34m Progressing 2024-02-02T11:41:45Z
[user@host ~]$ oc delete dscinitialization default-dsci
dscinitialization.dscinitialization.opendatahub.io "default-dsci" deleted
```

3. Delete the RHOAI operator subscription.

```
[user@host ~]$ oc delete subscription rhods-operator -n redhat-ods-operator
subscription.operators.coreos.com "rhods-operator" deleted
```

4. Delete the operator namespaces, and other namespaces created by the RHOAI platform.

```
[user@host ~] oc delete ns redhat-ods-operator
namespace "redhat-ods-operator" deleted

[user@host ~] oc delete ns -l opendatahub.io/generated-namespace
namespace "redhat-ods-applications" deleted
namespace "redhat-ods-monitoring" deleted
namespace "rhods-notebooks" deleted
[user@host ~] oc delete ns -l opendatahub.io/dashboard=true
...output omitted...
```

At this point, you might need to uninstall the other operators that you installed to enable RHOAI to use all the features, such as Red Hat OpenShift Pipelines, Red Hat OpenShift Service Mesh, or Red Hat OpenShift Serverless. See the references section for more information about uninstalling these operators.



## References

### Red Hat OpenShift AI Service Definition

<https://access.redhat.com/support/policy/updates/rhoai/service>

For more information about how to configure distributed workloads, see the *Configuring distributed workloads* section in the *Working on data science projects* guide at

[https://access.redhat.com/documentation/en-us/red\\_hat\\_openshift\\_ai\\_self-managed/2.8/html-single/working\\_on\\_data\\_science\\_projects/index#configuring-distributed-workloads\\_distributed-workloads](https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/working_on_data_science_projects/index#configuring-distributed-workloads_distributed-workloads)

For more information about how to install the KServe component, see the *Serving Large Language Models* section in the *Serving Models* guide at

[https://access.redhat.com/documentation/en-us/red\\_hat\\_openshift\\_ai\\_self-managed/2.8/html-single/serving\\_models/index#serving-large-language-models\\_serving-large-language-models](https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/serving_models/index#serving-large-language-models_serving-large-language-models)

For more information about model monitoring ,see the *Serving models* guide at

[https://access.redhat.com/documentation/en-us/red\\_hat\\_openshift\\_ai\\_self-managed/2.8/html-single/serving\\_models/index#monitoring-model-performance\\_monitoring-model-performance](https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/serving_models/index#monitoring-model-performance_monitoring-model-performance)

For more information about the NVIDIA GPU operator, see the *NVIDIA GPU architecture overview* chapter in the *Architecture* guide from the OpenShift documentation at

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.14/html-single/architecture/index#nvidia-gpu-architecture-overview](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.14/html-single/architecture/index#nvidia-gpu-architecture-overview)

### Node Feature Discovery operator documentation

[https://docs.openshift.com/container-platform/4.14/hardware\\_enablement/psap-node-feature-discovery-operator.html](https://docs.openshift.com/container-platform/4.14/hardware_enablement/psap-node-feature-discovery-operator.html)

### Red Hat OpenShift Service Mesh installation

[https://docs.openshift.com/container-platform/4.14/service\\_mesh/v2x/preparing-ossm-installation.html](https://docs.openshift.com/container-platform/4.14/service_mesh/v2x/preparing-ossm-installation.html)

### Red Hat OpenShift Serverless installation

<https://docs.openshift.com/serverless/1.32/install/install-serverless-operator.html>

### Red Hat OpenShift Pipelines installation

[https://docs.openshift.com/pipelines/1.14/install\\_config/installing-pipelines.html](https://docs.openshift.com/pipelines/1.14/install_config/installing-pipelines.html)

### Red Hat OpenShift AI: Supported Configurations

<https://access.redhat.com/articles/rhoai-supported-configs>

## ► Guided Exercise

# Red Hat OpenShift AI Installation

Perform a Red Hat OpenShift AI Installation

### Outcomes

- Install Red Hat OpenShift AI (RHOAI) by using the web console.
- Configure RHOAI components by creating the Data Science Cluster (DSC) object.

### Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI263 && lab start rhoai-install
```

### Instructions

- ▶ 1. Log in to the Red Hat OpenShift Container Platform (RHOCOP) using `admin` as the user and `redhatocp` as the password.
  - 1.1. Open a web browser and navigate to `https://console-openshift-console.apps.ocp4.example.com`.
  - 1.2. Click `htpasswd_provider` and log in as the `admin` user with `redhatocp` as the password.
- ▶ 2. Install the RHOAI operator.
  - 2.1. Click **Operators > OperatorHub**. In the **Filter by keyword** field, type `openshift ai` to locate the RHOAI operator, and then click **Red Hat OpenShift AI**.
  - 2.2. The web console displays information about the RHOAI operator. Click **Install** to proceed to the **Install Operator** page.

## Chapter 4 | Installing Red Hat OpenShift AI

The screenshot shows the Red Hat OpenShift OperatorHub interface. At the top, there's a navigation bar with the Red Hat OpenShift logo and a user account icon for 'admin'. Below the header, the path 'OperatorHub > Operator Installation' is shown. The main title is 'Install Operator'. A sub-section titled 'Update channel \*' has a dropdown menu set to 'stable'. Another dropdown for 'Version \*' is set to '2.8.0'. Under 'Installation mode \*', the radio button 'All namespaces on the cluster (default)' is selected, with a note that 'Operator will be available in all Namespaces.' The right side of the screen displays information about the 'Red Hat OpenShift AI' operator, provided by Red Hat. It lists 'Provided APIs' such as 'DSC Data Science Cluster' (marked as required) and 'DSCI DSC Initialization'. A note for 'DSC Data Science Cluster' states: 'DataScienceCluster is the Schema for the datascienceclusters API.'

- 2.3. The **Install Operator** page contains installation options. You can use the default options.

The RHOCP cluster is configured to use a mirror registry with only the required operators for the course. In this registry, the RHOAI operator has a single version available in both **fast** and **stable** channels.

Ensure that the **stable** update channel is selected, and click **Install**.

### ► 3. Create the DSC object.

- 3.1. Wait for the RHOAI operator to finish the installation, and then click **Create DataScienceCluster**.

Navigate to the YAML view to read the **DataScienceCluster** custom resource definition (CRD).

The screenshot shows the Red Hat OpenShift YAML editor. The left sidebar shows a navigation tree with 'Administrator', 'Home', 'Operators', 'OperatorHub' (selected), 'Workloads', 'Networking', 'Storage', 'Builds', and 'Distributions'. The main area is titled 'Create DataScienceCluster' with the sub-instruction 'Create by manually entering YAML or JSON definitions, or by dragging and dropping a file into the editor.' Below this, it says 'Configure via: Form view (radio button selected) YAML view'. A code editor window displays the following YAML code:

```

1 apiVersion: datasciencecluster.opendatahub.io/v1
2 kind: DataScienceCluster
3 metadata:
4 name: rhods
5 labels:
6 app.kubernetes.io/name: datasciencecluster
7 app.kubernetes.io/instance: rhods
8 app.kubernetes.io/part-of: rhods-operator
9 app.kubernetes.io/managed-by: kustomize
10 app.kubernetes.io/created-by: rhods-operator
11 spec:

```

At the bottom of the code editor are 'Create' and 'Cancel' buttons, and a 'Download' button to the right. To the right of the code editor is a panel titled 'DataScienceCluster' with a 'Schema' tab. The schema notes: 'DataScienceCluster is the Schema for the datascienceclusters API.' It includes a description of the 'apiVersion' field: 'apiversion string APIVersion defines the versioned schema of this representation of an object. Servers should convert'.

**Chapter 4 |** Installing Red Hat OpenShift AI

- 3.2. Scroll down to the `spec` section of the YAML file, and review what components are configured to install, `Managed`, or to not install, `Removed`.
- 3.3. To avoid installing the RHOAI pipelines component, change the `datasciencepipelines` value to `Removed`.

```
...output omitted...
spec:
 components:
 codeflare:
 managementState: Removed
 dashboard:
 managementState: Managed
 datasciencepipelines:
 managementState: Removed
 kserve:
 managementState: Managed
 serving:
 ingressGateway:
 certificate:
 type: SelfSigned
 managementState: Managed
 name: knative-serving
 kueue:
 managementState: Removed
 modelmeshserving:
 managementState: Managed
 ray:
 managementState: Removed
 workbenches:
 managementState: Managed
```

Then, click **Create**.

- 3.4. After creating the DSC, a view showing the details is displayed.  
Wait until the status of the cluster reads `Phase: Ready`. Then click `default-dsc`.
- 3.5. Inspect the messages in the `Conditions` section of the `default-dsc` DSC view.

**Created at**  
 Feb 19, 2024, 6:15 AM

**Owner**  
No owner

---

**Conditions**

| Type               | Status | Reason                                | Message                                                                                                                                                                                   |
|--------------------|--------|---------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Reconcile Complete | True   | ReconcileCompletedWithComponentErrors | DataScienceCluster resource reconciled with component errors: 1 error occurred:<br>* operator serverless-operator not found. Please install the operator before enabling kserve component |

**Chapter 4 |** Installing Red Hat OpenShift AI

Although the DSC object is ready, the KServe component is not, because KServe depends on the Red Hat OpenShift Serverless operator, which is not installed.

- 4. Install the Red Hat OpenShift Serverless operator by using the RHOPC console.
- 4.1. Click **Operators > OperatorHub**. In the **Filter by keyword** field, type **serverless** to locate the Red Hat OpenShift Serverless operator, and then click **Red Hat OpenShift Serverless operator**.
  - 4.2. The web console displays information about the Red Hat OpenShift Serverless operator.  
Click **Install** to proceed to the **Install Operator** page. In the operator installation page, click **Install** and then wait until the operator is ready for use.
  - 4.3. Return to the DSC page.  
Click **Operators > Installed Operators**. Click **Red Hat OpenShift AI** and then click the **Data Science Cluster** tab. Click **default-dsc** to open the DSC page.  
Verify that, after you install Red Hat OpenShift Serverless, the RHOPC cluster reconciles again the status of the DSC object, and corrects the previously failed KServe component. You might need to wait for about five minutes to see the reconciliation completed.

| Type              | Status | Updated                 | Reason             | Message                                             |
|-------------------|--------|-------------------------|--------------------|-----------------------------------------------------|
| ReconcileComplete | True   | ⌚ Mar 18, 2024, 4:06 AM | ReconcileCompleted | DataScienceCluster resource reconciled successfully |
| Available         | True   | ⌚ Mar 18, 2024, 4:06 AM | ReconcileCompleted | DataScienceCluster resource reconciled successfully |
| Progressing       | False  | ⌚ Mar 18, 2024, 4:06 AM | ReconcileCompleted | DataScienceCluster resource reconciled successfully |
| Degraded          | False  | ⌚ Mar 18, 2024, 4:05 AM | ReconcileCompleted | DataScienceCluster resource reconciled successfully |
| Upgradeable       | True   | ⌚ Mar 18, 2024, 4:06 AM | ReconcileCompleted | DataScienceCluster resource reconciled successfully |

**Figure 4.8: DataScienceCluster and all the components successfully installed**

- 5. Verify the RHAI operator OpenShift resources.
- 5.1. Click **Home > Projects**.  
Select User in the **Filter** field. Observe that there are four projects related to the RHAI operator:
    - **redhat-ods-applications**
    - **redhat-ods-monitoring**
    - **redhat-ods-operator**
    - **rhods-notebooks**
 The **redhat-ods-operator** and **redhat-ods-applications** show consuming CPU and memory from OpenShift resources.  
You can see that the **redhat-ods-monitoring** and the **rhods-notebooks** namespaces are not consuming resources. This is because monitoring is not enabled and the cluster does not have any running notebooks.
  - 5.2. Click **redhat-ods-applications** to see the namespace details.  
Then, click the **Workloads** tab. Observe that the **default-dsc** object is managing one CronJob, one Job, and seven Deployment Kubernetes objects. If you unselect the **Expand** field in **Display options**, then you can see the **dsc-managed** objects grouped by the component that uses each object.

| Workload        | Status | Count |
|-----------------|--------|-------|
| kserve          | Active | 1 D   |
| model-mesh      | Active | 3 D   |
| rhods-dashboard | Active | 1 CJ  |
| workbenches     | Active | 1 D   |
|                 |        | 2 D   |

Figure 4.9: RHOAI components workloads

► 6. Access the RHOAI console.

- 6.1. Click Red Hat OpenShift AI in the upper Red Hat Applications menu.

Figure 4.10: The RHOAI console access

- 6.2. Log in to the RHOAI console by using the same method that you used for the RHOCOP console: Click `htpasswd_provider` and log in as the `admin` user with `redhatocp` as the password.

## Finish

On the workstation machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish rhoai-install
```

# Summary

---

- You can use Red Hat OpenShift AI (RHOAI) as a provided and managed service, or as a self-managed service running on any deployment option of Red Hat OpenShift Container Platform (RHOC).
  - If you use the RHOAI managed service, then you do not need to install the RHOAI operator.
  - You control the installed components by creating and configuring the DataScienceCluster (DSC) custom resource definition.
  - To use the KServe component, RHOAI depends on Red Hat OpenShift Service Mesh and Red Hat Serverless.
  - To use the data science pipelines component, RHOAI depends on the Red Hat OpenShift Pipelines.



## Chapter 5

# Manage Users and Resources

### Goal

Managing Red Hat OpenShift AI users, and resource allocation for Workbenches.

### Sections

- Users, Projects, and Permissions (and Guided Exercise)
- Managing Resources (and Guided Exercise)

# Users, Projects, and Permissions

## Objectives

- Managing Red Hat OpenShift AI users and controlling access.

## Users, Projects, and Permissions

Red Hat OpenShift AI (RHOAI) uses Red Hat OpenShift Container Platform (RHOCOP) authentication as the method for all RHOAI related resources. By default, any user with the ability to authenticate with the RHOCOP cluster can access the RHOAI dashboard.

Users with the `cluster-admin` role have administrator permissions in the RHOAI dashboard. Additionally, any user within the `rhods-admins` default group has administrator permissions.

Administrator access in the RHOAI dashboard allows users to control various settings of RHOAI through the dashboard UI, including the following configurations:

- Notebook images and settings
- Serving runtimes
- Idle notebook culler, which automatically removes inactive notebooks

Use the `oc adm groups` command to add users to the `rhods-admins` group. For example, the following command adds the `my-user1` and `my-user2` users to the group.

```
[user@host ~]$ oc adm groups add-users rhods-admins my-user1 my-user2
...output omitted...
```

## Default User Permissions

You manage the default permissions for the RHOAI dashboard by using the **Settings > User management** section.

The screenshot shows the Red Hat OpenShift AI User management interface. The left sidebar has a dark theme with white text. It includes sections for Applications (Enabled, Explore), Data Science Projects, Data Science Pipelines, Model Serving, Resources, Settings (Notebook image settings, Cluster settings, Serving runtimes, User management), and a 'Save changes' button at the bottom. The 'User management' section in the Settings menu is highlighted with a blue bar. The main content area is titled 'User management' and contains two sections: 'Data Science administrator groups' and 'Data Science user groups'. Both sections have a 'Select the OpenShift groups...' instruction, a dropdown menu with a single item (e.g., 'rhods-admins' or 'system:authenticated'), and a 'View, edit, or create groups in OpenShift under User Management' link below the dropdown.

You can set administrator groups to groups other than default `rhods-admins`. Setting different administrator groups is useful, for example, to synchronize groups to an external system. Additionally, the `cluster-admin` cluster role grants administrator permissions within the dashboard. However, organizations should not rely on the `cluster-admin` role to grant administrator permissions.

The normal Data Science user group can control which users can sign into the dashboard. By default, the dashboard grants access to authenticated users that have the `system:authenticated` role, which can be assigned to a custom group.



### Note

Updating the access in the `User management` section within the RHOAI dashboard only impacts users' ability to access the dashboard. It does not impact permissions granted by regular Kubernetes-based access control.

For example, if the dashboard is restricted to only a certain group, but a user not part of that group has access to the Kubernetes namespace, then they can still create and modify any Data Science related objects via managing the underlying Kubernetes objects.

## Managing Dashboard Permissions with GitOps

Within the `redhat-ods-applications` RHOCP project, the `odh-dashboard-config` object contains the configuration settings available within the `User management` section of the RHOAI

## Chapter 5 | Manage Users and Resources

dashboard, along with other settings. As such, updating settings in one location updates them in the other.

Because it is an RHOCP object, you can view the settings by retrieving its YAML representation, as the following command demonstrates:

```
[user@host ~]$ oc get odhdashboardconfigs odh-dashboard-config -n \
redhat-ods-applications -o yaml
apiVersion: opendatahub.io/v1alpha
kind: OdhDashboardConfig ①
metadata:
 name: odh-dashboard-config
 namespace: redhat-ods-applications
 ...output omitted...
spec:
 ...output omitted...
 groupsConfig: ②
 adminGroups: rhods-admins
 allowedGroups: system:authenticated
 ...output omitted...
```

- ① The object type is a custom resource definition (CRD) called `OdhDashboardConfig`.
- ② The `groupsConfig` section controls which groups are RHOAI administrators groups.

Like many other RHOCP objects, you can export and track this object with a version control system, or combine it with GitOps practices.

## Managing Permissions on a Data Science Project

If self-provisioning is enabled, then the RHOAI dashboard enables users to create new Data Science Projects. Self-provisioning is an OpenShift feature that grants regular users permissions to create their own projects. The user who created the Data Science Project automatically receives the RHOCP administrator role for the namespace.

Because Data Science Projects are just normal RHOCP projects with certain labels, RHOAI project permissions function the same as any other RHOCP project. You manage project user and group permissions by using Kubernetes role-based access control (RBAC). The RHOAI dashboard provides a user interface to assign additional user or group permissions to a project.

The **Permissions** tab within the project's view page of the RHOAI dashboard includes the following capabilities for project administrators:

- Add users to the project
- Add groups to the project
- Grant the `edit` role to users and groups
- Grant the `admin` role to users and groups

**Figure 5.2: The Permissions tab within the RHOAI dashboard**

The Permissions tab only displays users and groups when you grant them permissions directly through the dashboard. If you granted permissions to users and groups by using role bindings on the project, then the user interface does not display those users and groups.

## Manually Creating Data Science Projects

If self-provisioning is disabled, then you must manually create Data Science Projects. For example, to manually create a Data Science Project called `myproject` from the command line, create the namespace and apply the necessary labels:

```
[user@host ~]$ oc create namespace myproject
namespace/myproject created
[user@host ~]$ oc label namespace myproject opendatahub.io/dashboard='true' \
modelmesh-enabled='true'
namespace/myproject labeled
```

Alternatively, apply the following YAML manifest:

```
kind: Namespace
apiVersion: v1
metadata:
 name: myproject
 labels:
 modelmesh-enabled: 'true'
 opendatahub.io/dashboard: 'true'
```

After the Data Science Project is created, configure access to the project for users and groups by using either the command line or a namespace role binding.

**Note**

Namespaces and projects are often used to mean the same thing. Although there are technical differences between these objects, a project is essentially a Kubernetes namespace with some additional features provided by RHOCP. RHOCP creates a corresponding project for every namespace and vice versa.

**References**

For information on configuring user access to RHOCP with various identity providers, refer to the RHOCP documentation at

**Preparing for users**

[https://docs.openshift.com/container-platform/4.14/post\\_installation\\_configuration/preparing-for-users.html](https://docs.openshift.com/container-platform/4.14/post_installation_configuration/preparing-for-users.html)

For information on managing role-based access control within RHOCP, refer to the documentation at

**Adding roles to users**

[https://docs.openshift.com/container-platform/4.14/authentication/using-rbac.html#adding-roles\\_using-rbac](https://docs.openshift.com/container-platform/4.14/authentication/using-rbac.html#adding-roles_using-rbac)

For more information about user management within RHOAI, see the *Managing*

*Users* guide in the *Red Hat OpenShift AI Self-Managed* documentation at

[https://access.redhat.com/documentation/en-us/red\\_hat\\_openshift\\_ai\\_self-managed/2.8/html-single/managing\\_users/index](https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/managing_users/index)

## ► Guided Exercise

# Users, Projects, and Permissions

Managing Red Hat OpenShift AI users and controlling access.

### Outcomes

- Create a new group within Red Hat OpenShift Container Platform (RHOC).
- Grant edit permissions to that group on a Data Science Project within Red Hat OpenShift AI (RHOAI).

### Before You Begin

As the **student** user on the **workstation** machine, use the **lab** command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI263 && lab start manage-useraccess
```

### Instructions

- 1. Observe that the **developer** user does not have access to the **manage-useraccess** Data Science Project.
- 1.1. In a web browser, open the RHOAI dashboard by navigating to <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com>.
  - 1.2. Log in as the **developer** user by using **developer** as the password.
  - 1.3. View the list of Data Science Projects by clicking **Data Science Projects** from the left navigation menu.  
Observe that the **manage-useraccess** project is not in the list.
- 2. As an administrator user, create a group that includes the **developer** user.
- 2.1. Open a new private browser window by clicking the menu in the top-right corner of the browser window and selecting **New private window**. Navigate to <https://console-openshift-console.apps.ocp4.example.com>, click **htpasswd\_provider**, and authenticate with username **admin** and password **redhatocp**.



#### Note

Doing this in a private browser window enables you to log in as both users simultaneously.

- 2.2. From the left-hand navigation, click **User Management > Groups**. Only the **rhods-admins** group displays.

**Note**

Ensure that the **Administrator** perspective is selected in the top of the left-hand navigation. The User Management interface is not available from the **Developer** perspective.

- 2.3. Click **Create Group** and update the YAML in the editor to match the following:

```
apiVersion: user.openshift.io/v1
kind: Group
metadata:
 name: my-team
users:
 - developer
```

Click **Create** to create the **my-team** group.

- ▶ **3.** As an administrator user, add permissions to the **my-team** group to manage the **manage-useraccess** Data Science Project.
  - 3.1. In a new tab within the private browser window, navigate to <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com>.
  - 3.2. Authenticate as the **admin** user by using the **redhatocp** password.
  - 3.3. View the list of Data Science Projects by clicking **Data Science Projects** from the left navigation menu.
  - 3.4. Click the **manage-useraccess** project and then click the **Permissions** tab.
  - 3.5. Click **Add group** and select **my-team** from the selection menu that appears under the **Name** column.
  - 3.6. Verify that the **Edit** permission is selected and then click the checkmark icon to confirm adding the permission.
- ▶ **4.** Verify that the **developer** user has edit access to the **manage-useraccess** Data Science Project.
  - 4.1. From the browser window authenticated as the **developer** user, refresh the RHOAI projects page. The **manage-useraccess** project now appears.
  - 4.2. Click the **manage-useraccess** project to view its details.
  - 4.3. Observe that the **Permissions** tab that you used as the **admin** user does not display for the **developer** user. This is because the **developer** user has edit permissions, but not administrator permissions for the **manage-useraccess** Data Science Project.

## Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish manage-useraccess
```



# Managing Resources

## Objectives

- Manage resource allocations for Workbenches.

## Kubernetes Resources

Many of the objects in Red Hat OpenShift AI (RHOAI) are tied to an underlying Kubernetes resource. Because of this connection, many RHOAI objects can be created, updated, or deleted by performing the respective action on the underlying resource. Although some of these resources use a custom resource definition (CRD), many do not. For example, Data Science Projects are Red Hat OpenShift projects that include specific labels.

The following are resources that you are likely to encounter when working with RHOAI.

### **Data Science Projects** (`projects.project.openshift.io`)

A Data Science Project is synonymous with an OpenShift Project or a Namespace. See the users section for more information on how to create and manage Data Science Projects.

### **Users** (`users.user.openshift.io`)

A user corresponds to a specific user that can have permissions within OpenShift and RHOAI.

### **Groups** (`groups.user.openshift.io`)

A group is a collection of users that, like users, can have permissions within OpenShift and RHOAI.

### **Role Bindings** (`rolebindings.rbac.authorization.k8s.io`)

Role bindings associate roles, such as `edit`, to users and groups, granting them the relevant permissions.

### **Workbenches** (`notebooks.kubeflow.org`)

A workbench is a development environment running in an OpenShift pod that uses the Kubeflow Notebook Controller. Depending on the workbench image, workbenches can run a number of web-based editors, including JupyterLab, Visual Studio Code, and R Studio.

### **Persistent Volume Claims** (`persistentvolumeclaim`)

Workbenches use cluster storage to persist a user environment when the workbench pod is stopped or restarted. When creating a workbench, RHOAI automatically creates a default cluster storage instance to store any code or files within the workbench. Additional cluster storage can be added to the workbench, such as a dedicated persistent volume claim for training data.

### **Data Connections** (`secret`)

A data connection is an OpenShift secret that stores the values required to connect to an S3 bucket.

### **Data Science Pipeline Applications (DSPA)**

### **(`datasciencelinesapplications.datasciencelinesapplications.opendatahub.io`)**

A DSPA creates an instance of a Data Science Pipeline and requires a data connection and an S3 bucket to create the instance. A DSPA is namespace-scoped to prevent leaking data across multiple projects.

**Pipelines (pipelines.tekton.dev)**

Pipelines are executable workflows that can perform various tasks in a sequence, such as building and then deploying a data model. They result in a PipelineRun per execution.

**Pipeline Runs (pipelineruns.tekton.dev)**

A pipeline run represents a specific execution of a pipeline. Although this OpenShift object is a valid Tekton PipelineRun, it should be uploaded to the RHOAI dashboard, and not applied directly to the cluster.

**Models (inferenceservices.serving.kserve.io)**

Models require a data connection and a location where the model file is stored in the S3 bucket.

**Model Servers (servingruntimes.serving.kserve.io)**

Models are associated with a specific model server, which host the model and are used to create endpoints. A single model server can serve multiple models from a single instance.

## Reclaiming Cluster Resources

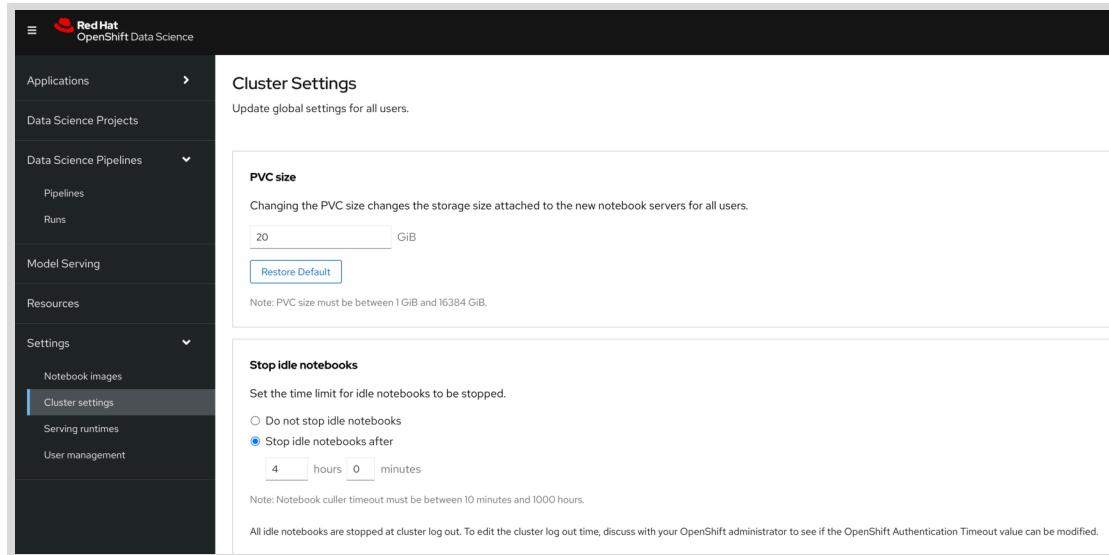
Over time, projects can be abandoned as users leave your organization, move to different initiatives, or otherwise use different OpenShift projects. To free up cluster resources, clean up such unused projects either from the RHOAI dashboard or by using the oc command-line interface.

In addition to manually cleaning up under-utilized resources, RHOAI provides features that can further reduce resource usage.

### Idle Notebook Culling

Notebooks and workbenches can consume many compute resources. Additionally, users might forget to shut them down when they are finished, which consumes cluster resources unnecessarily.

The Idle Notebook Culler helps reduce the number of inactive notebooks running on the cluster. It tracks the last time an action was taken in a notebook and shuts down the pods if the notebook has been inactive for a period of time. A notebook is considered inactive when no user has taken an action inside of the notebook. Actions include executing a cell, creating files, or interacting with the user interface.



**Figure 5.3: Administrators enable notebook culling through the Settings > Cluster settings section of the RHOAI dashboard.**

Alternatively, idle notebook culling is configured by creating a configuration map, such as the following:

```
kind: ConfigMap
apiVersion: v1
metadata:
 name: notebook-controller-culler-config ①
 namespace: redhat-ods-applications ②
 labels:
 opendatahub.io/dashboard: 'true'
data:
 CULL_IDLE_TIME: '240' ③
 ENABLE_CULLING: 'true' ④
 IDLENESS_CHECK_PERIOD: '1' ⑤
```

- ① The name of the configuration map must be `notebook-controller-culler-config`
- ② The namespace containing the RHOAI dashboard
- ③ The amount of time, in minutes, that a notebook can be idle before it is automatically stopped
- ④ An option to enable or disable the automatic culling
- ⑤ How often, in minutes, that the culler checks for idle notebooks

If the cluster is set to end user sessions after a time, then that time overrides the idle notebook time limit.

For example, a cluster is set to time out user sessions after one hour of inactivity, and the idle notebook time limit is set to five hours. In this case, RHOAI stops a notebook if no user activity is detected after one hour, not five.

## Workbench and Model Server Sizes

When launching workbenches or model servers from the RHOAI dashboard, users can select from several default sizes. However, these default options might not suit your organization's needs. In such cases, you can use custom model server and notebook size configurations.

```

apiVersion: opendatahub.io/v1alpha
kind: OdhDashboardConfig
metadata:
 annotations:
 internal.config.kubernetes.io/previousKinds: OdhDashboardConfig
 internal.config.kubernetes.io/previousNames: odh-dashboard-config
 internal.config.kubernetes.io/previousNamespaces: default
 name: odh-dashboard-config 1
 namespace: redhat-ods-applications 2
 labels:
 app.kubernetes.io/part-of: rhods-dashboard
 app.opendatahub.io/rhods-dashboard: 'true'
spec:
 ...output omitted...
 modelServerSizes: 3
 - name: Small
 resources:
 limits:
 cpu: '2'
 memory: 8Gi
 requests:
 cpu: '1'
 memory: 4Gi
 ...output omitted...
 notebookSizes: 4
 - name: Small
 resources:
 limits:
 cpu: '2'
 memory: 8Gi
 requests:
 cpu: '1'
 memory: 8Gi
 ...output omitted...

```

- 1** The name of the dashboard configuration, which must match `odh-dashboard-config`
- 2** The namespace where the RHOAI dashboard is installed
- 3** The default model server configurations
- 4** The default workbench configurations



## References

For more information about culling idle notebooks, see the *Stopping idle notebooks* section in the *Managing Resources* guide in the *Red Hat OpenShift AI Self-Managed* documentation at

[https://access.redhat.com/documentation/en-us/red\\_hat\\_openshift\\_ai\\_self-managed/2.8/html-single/managing\\_resources/index#stopping-idle-notebooks\\_notebook-mgmt](https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/managing_resources/index#stopping-idle-notebooks_notebook-mgmt)

For more information about limit ranges on RHOCP, see the *Restrict resource consumption with limit ranges* section in the *Working with clusters* chapter in the *Nodes* guide in the RHOCP documentation at

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.14/html-single/nodes/index#nodes-cluster-limit-ranges](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.14/html-single/nodes/index#nodes-cluster-limit-ranges)

## ► Guided Exercise

# Managing Resources

Manage resource allocations for Workbenches.

### Outcomes

- Configure Red Hat OpenShift AI (RHOAI) to stop cluster resources from unused AI/ML workbenches.
- Define and manage the workbenches and model server sizes available for AI/ML developers.

### Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI263 && lab start manage-resources
```

### Instructions

- 1. Create a small workbench called `manage-resources-wb`.
- 1.1. In a web browser, navigate to the RHOAI dashboard at <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com> and authenticate as the `admin` user by using the `redhatocp` password.
  - 1.2. Navigate to the `manage-resources` project details by clicking **Data Science Projects** and then the `manage-resources` project.
  - 1.3. Click **Create workbench** and enter `manage-resources-wb` as the name of the workbench. Select **Minimal Python** as the notebook image. Leave all other options with the default value. Then, click **Create workbench**.

The screenshot shows the Red Hat OpenShift AI interface for managing resources. At the top, there's a navigation bar with the Red Hat logo and 'OpenShift AI'. Below it, a breadcrumb navigation shows 'Data Science Projects > manage-resources'. The main area is titled 'manage-resources' and contains several sections:

- Components**: Shows 'Workbenches' (selected), 'Permissions', and 'Settings'.
- Jump to section**: Options include 'Workbenches', 'Cluster storage', 'Data connections', 'Models and model servers', and 'Data connections'.
- Workbenches**: A table with columns 'Name', 'Notebook image', 'Container size', 'Status', and an 'Open' button. It lists one entry: 'manage-resources-wb' (Minimal Python, Small, Running).
- Cluster storage**: A table with columns 'Name', 'Type', and 'Connected workbenches'. It lists one entry: 'manage-resources-wb' (Persistent storage, connected to 'manage-resources-wb').
- Data connections**: A table with a 'Add data connection' button.

Figure 5.4: The created small workbench running.

Do not open the `manage-resources-wb` running workbench.

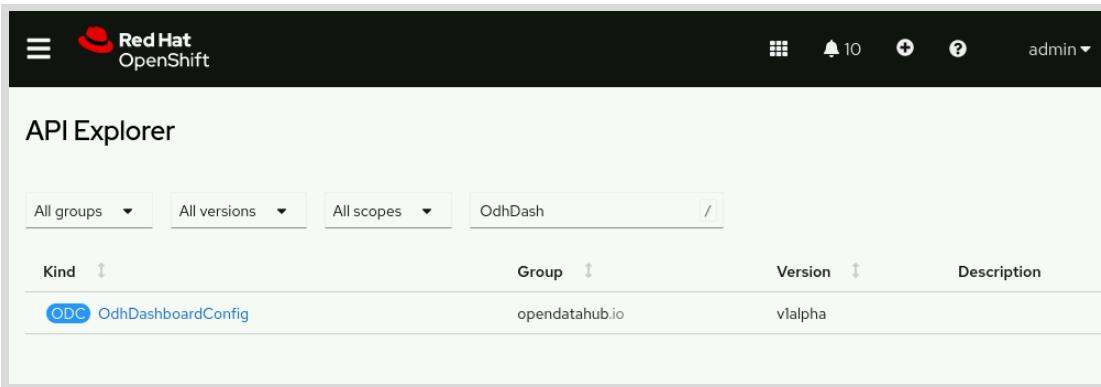
- ▶ 2. Set the culling time to ten minutes by using the RHOAI dashboard, and continue with next steps.
  - 2.1. Navigate to **Settings > Cluster settings**. Scroll down and locate the **Stop idle notebooks** section.
  - 2.2. Select **Stop idle notebooks after**, and change the **4 hours 0 minutes** field to **0 hours 10 minutes**. Scroll down and click **Save changes**.
  - 2.3. In the upper right menu, click **admin > Log out**.
  - 2.4. Open another browser window and navigate to the Red Hat OpenShift Container Platform (RHOCP) console at `https://console-openshift-console.apps.ocp4.example.com` and authenticate as the **admin** user by using the `redhatocp` password.
  - 2.5. Navigate to **Workloads > ConfigMaps**. Use the **Project** selector to select the `redhat-ods-applications` project.
  - 2.6. Observe that the `notebook-controller-culler-config` ConfigMap object has a different time stamp than the others, because the RHOAI operator creates it when you modify the culling time.
  - 2.7. Navigate to **Workloads > Pods**. Use the **Project** selector to select the `manage-resources` project.

There is a running pod with 2 containers inside. This pod contains the notebook server that serves the workbench.

You must wait ten minutes to verify that the culling works. After that time, the RHOAI operator stops this pod. If you reach this step after more than ten minutes, then you

can see that the pod has already stopped. If you reach this step in under ten minutes, then you can continue now with the following steps before verifying that this pod is stopped.

- ▶ 3. Modify the notebook available sizes to add an option called **MediumSmall**, with two CPUs and 16 Gi of memory.
  - 3.1. In the RHOCP console, navigate to **Home > API Explorer**, and filter by **OdhDashboardConfig**.



The screenshot shows the Red Hat OpenShift API Explorer interface. At the top, there is a navigation bar with icons for home, search, and user admin. Below the navigation bar, the title "API Explorer" is displayed. Underneath the title, there are several dropdown menus: "All groups", "All versions", "All scopes", and a search bar containing "OdhDash". A table below lists the "OdhDashboardConfig" object. The table columns are "Kind", "Group", "Version", and "Description". The single entry shows "OdhDashboardConfig" under Kind, "opendatahub.io" under Group, "v1alpha" under Version, and no description provided.

Figure 5.5: The **OdhDashboardConfig** object.

Then, click **OdhDashboardConfig** to access the custom resource definition details. Ensure that you select **All** projects in the project selector.

- 3.2. Click the **Instances** tab, and then click **odh-dashboard-config** to access the object details.

- 3.3. Access the object YAML definition by clicking the **YAML** tab.

Scroll down to the **spec** section, and locate the **notebookSizes** section. Add the following YAML snippet to create a new notebook size for your developers:

```
...output omitted...
notebookSizes:
- name: Small
 resources:
 limits:
 cpu: '2'
 memory: 8Gi
 requests:
 cpu: '1'
 memory: 8Gi
- name: MediumSmall
 resources:
 limits:
 cpu: '2'
 memory: 16Gi
 requests:
 cpu: '1'
 memory: 16Gi
- name: Medium
...output omitted...
```

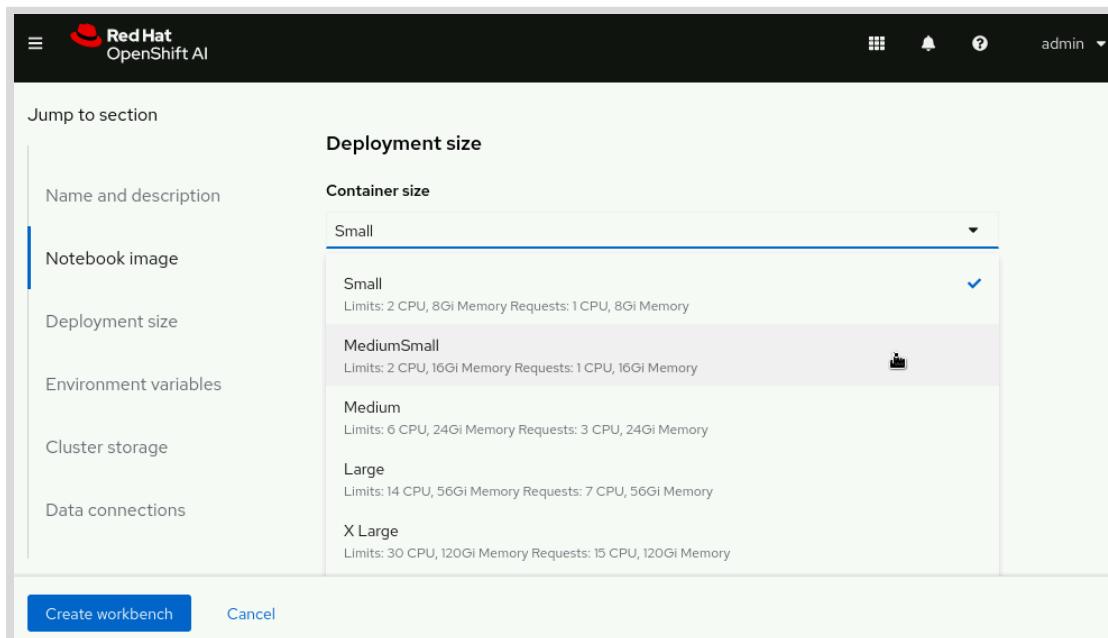
## Chapter 5 | Manage Users and Resources

Click **Save** to persist the `odh-dashboard-config` object modification. A warning displays to inform that you are modifying a resource managed by an operator. Click **Save**. Then click **Reload** for the change to take effect.

### ► 4. Create a `MediumSmall` size workbench.

- 4.1. In another browser tab, navigate to the RHOAI dashboard at <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com> and authenticate as the `admin` user by using the `redhatocp` password.
- 4.2. Navigate to the `manage-resources` project details by clicking **Data Science Projects** and then the `manage-resources` project.
- 4.3. Create a workbench called `manage-resources-mediumsmall-wb`.

Click **Create workbench**, enter `manage-resources-mediumsmall-wb` as the name of the workbench, and select `Minimal Python` for the image. In the `Deployment size` section, select the `MediumSmall` size. Leave all other options with the default value. Then, click **Create workbench**.



**Figure 5.6: The `MediumSmall` size option that you created.**

At this point, the `manage-resources` data science project contains two workbenches: `manage-resources-wb` and `manage-resources-mediumsmall-wb`. If you have spent more than ten minutes on the last steps, then you will see that the `manage-resources-wb` notebook server has stopped:

|                                 | Name           | Notebook image | Container size                      | Status  | Action               |
|---------------------------------|----------------|----------------|-------------------------------------|---------|----------------------|
| manage-resources-mediumsmall-wb | Minimal Python | MediumSmall    | <input checked="" type="checkbox"/> | Running | <a href="#">Open</a> |
| manage-resources-wb             | Minimal Python | Small          | <input type="checkbox"/>            | Stopped | <a href="#">Open</a> |

Figure 5.7: The small workbench stopped after the culling time

- ▶ 5. Verify that the small size notebook is stopped, after the defined culling time.
  - 5.1. In the RHOAI dashboard, navigate to **Data Science Projects** and click **manage-resources**.
  - 5.2. Verify that the idle notebooks feature worked, and, therefore, the **manage-resources-wb** is stopped.
  
- ▶ 6. Revert the culling time to four hours in the RHOAI dashboard.
  - 6.1. Navigate to **Settings > Cluster settings**. Scroll down and locate the **Stop idle notebooks** section.
  - 6.2. Select **Stop idle notebooks after**, and change the **0 hours 10 minutes** field to **4 hours 0 minutes**. Scroll down and click **Save changes**.

## Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish manage-resources
```

# Summary

---

- You can manage authorization and authentication, and cluster settings by using the Red Hat OpenShift AI (RHOAI) dashboard.
- RHOAI utilizes the authorization and authentication infrastructure from the Red Hat OpenShift Container Platform (RHOC) cluster where the RHOAI platform is running. By default, any user with the ability to authenticate with the RHOC cluster can access the RHOAI dashboard.
- The `rhods-admins` OpenShift group is the default group that manages the RHOAI platform, but you can define other administrator groups by using the RHOAI dashboard, or by modifying the `OdhDashboardConfig` object.
- RHOAI resources and components are Kubernetes, or OpenShift, objects.
- You can use the RHOAI dashboard to change some cluster settings, such as the workbenches culling time, the default persistent volume size, or the available workbench templates.

## Chapter 6

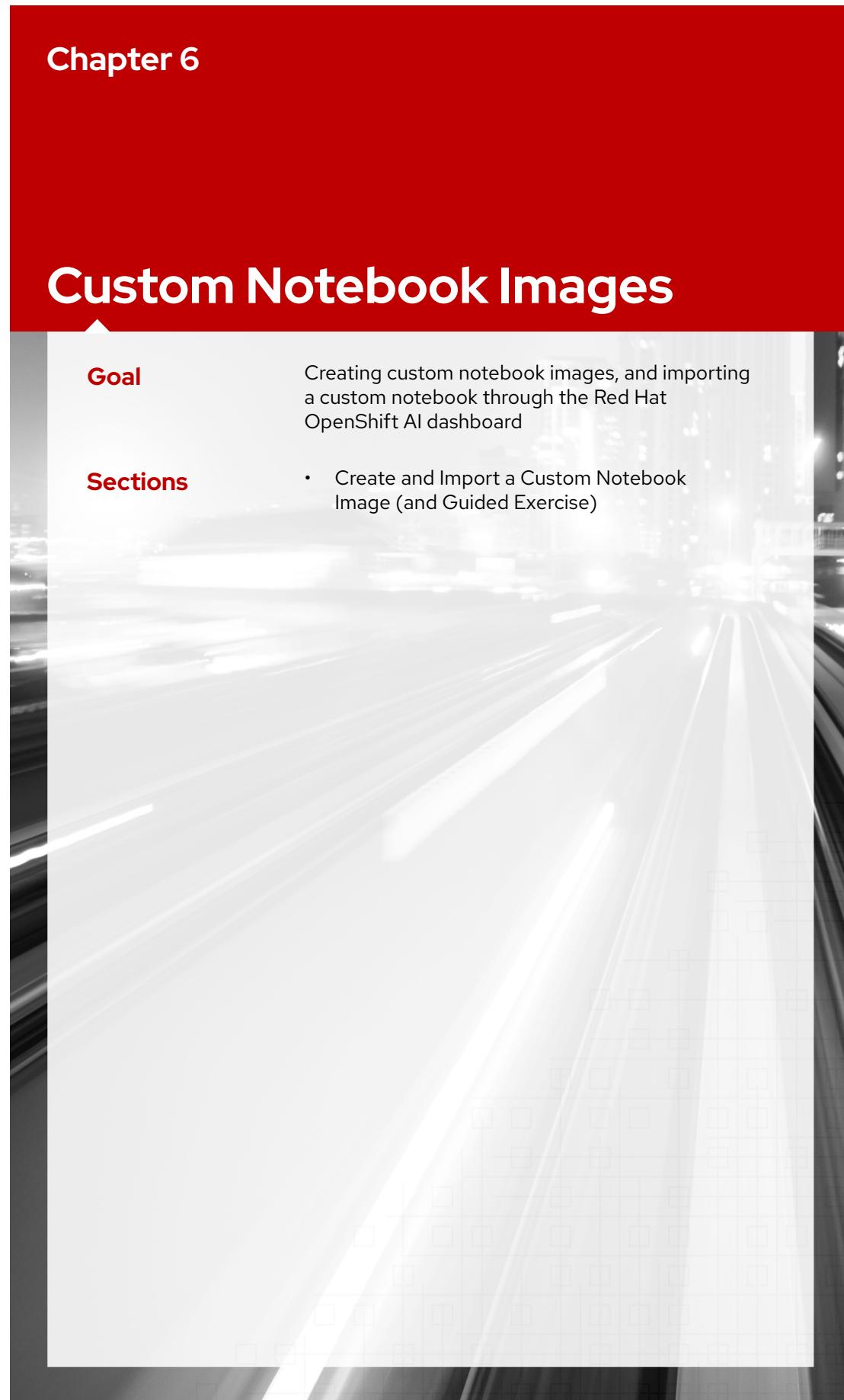
# Custom Notebook Images

### Goal

Creating custom notebook images, and importing a custom notebook through the Red Hat OpenShift AI dashboard

### Sections

- Create and Import a Custom Notebook Image (and Guided Exercise)



# Create and Import a Custom Notebook Image

## Objectives

- Create custom notebook images.
- Importing custom notebook images to Red Hat OpenShift AI.

## Creating a Custom Notebook Image

By using custom notebook images in Red Hat OpenShift AI (RHOAI), you can add packages and libraries to your workbenches. This customization can provide a consistent development environment for your teams and ensures that projects are reproducible between team members.

Workbenches in RHOAI run as containers, just like any other OpenShift workload. To create a custom workbench, you must create a container image that contains the JupyterLab environment and any additional software that you wish to provide. Because RHOAI runs on OpenShift, you can use the typical features and mechanisms provided by OpenShift to create and manage containers and container images.



### Note

*Custom notebook images* and *custom workbench images* are synonyms. You might find both terms used interchangeably in this course and in the documentation.

## Building and Pushing Container Images

Podman is a tool for running and managing containers and container images. As such, it includes functionality to build container images from Containerfiles. Podman runs the instructions within a Containerfile to produce a container image.

For example, given that a Containerfile is in the current working directory, the following command creates a container image with the `my-image:1.0` tag.

```
[user@host ~]$ podman build . -t my-image:1.0
```

Typically, you should use fully-qualified domain names (FQDN) to tag your images, such as `example.com/my-repo/my-image:1.0`. By using such a specific image tag, you reduce the chances of pulling an image from an unintended source.

After a container image is built, you can then push it to a container registry for use outside your local machine. If your image tag is an FQDN, then Podman defaults to pushing it to the registry specified as part of the tag.

For example, the following command attempts to push the container image to a registry hosted at `example.com`.

```
[user@host ~]$ podman push example.com/my-repo/my-image:1.0
```

## Chapter 6 | Custom Notebook Images

Container registries store container images for use by the rest of a team. For example, a Kubernetes or OpenShift cluster, given proper access, can pull and deploy images directly from a container registry.

## User and Group Permissions

In OpenShift, every container within a namespace runs with a user that has a random user ID in a group with ID 0. Therefore, to be modified, directories and files must be accessible by this user. A common way to achieve this is by setting the ownership of the file to user default (1001) and group 0.

When launching a notebook in RHOAI, the working directory of a user is mounted at /opt/app-root/src. This is not configurable, so custom images must use this default location for persisted data.

## Containerfiles

To build a new container image, you create a Containerfile that enumerates instructions for constructing the image. The specification for Containerfiles includes a number of available instructions that take arguments.



### Note

Containerfiles are largely compatible with the Dockerfile syntax. In most cases, the two types of files are interchangeable.

The (DO188) *Red Hat OpenShift Development I: Introduction to Containers with Podman* course covers creating Containerfiles and using Podman extensively.

The following example Containerfile uses several standard instructions to specify how to build a new image.

```
FROM quay.io/modh/odh-generic-data-science-notebook ①
COPY requirements.txt ./ ②
RUN echo "Installing softwares and packages" && \
 pip install micropipenv && \
 micropipenv install && \
 rm -f ./requirements.txt
```

- ➊ The FROM instruction specifies the base image that provides the starting point for the custom image.
- ➋ The COPY instruction copies a file into the container image. Specifically, this example copies a requirements.txt file that contains a list of Python dependencies.
- ➌ The RUN instruction executes a provided command. Because each instance of the RUN instruction results in an additional image layer, reduce the number of layers by combining commands. After installing the dependencies, the requirements.txt file is no longer necessary.

**Note**

The micropipenv command is a lightweight tool to install packages with a requirements.txt file. This tool is an alternative to the standard pip command, and is designed for containerized environments.

## RHOAI Base Images

To build a custom workbench image, use a RHOAI-specific image as the base. Then, add any necessary packages, libraries, and configurations for your particular use case.

You can find the base images in the following Quay.io repositories:

- RHOAI base images: <https://quay.io/modh>
- OpenDataHub (ODH) base images: <https://quay.io/opendatahub/workbench-images>
- ODH community contributed images: <https://quay.io/opendatahub-contrib/workbench-images>

Although the recommendation is to use the RHOAI images, you can use the ODH upstream images if they match your needs more closely.

For more details, refer to the GitHub repositories. The default RHOAI notebook Containerfiles are available in the <https://github.com/opendatahub-io/notebooks> repository. This repository provides a source of images and examples for building custom images.

Additional images are available in the <https://github.com/opendatahub-io-contrib/workbench-images> repository.

## Importing a Custom Workbench Image

RHOAI can import any container image that is accessible by the OpenShift cluster. After importing, users can specify the image when creating workbenches.

To import an image, use the **Settings > Notebook image settings** section of the RHOAI dashboard. Note that only administrator users have access to this section.

Upon importing, you can specify any software and packages that the image provides as well as the version of each. Doing so can help users determine which workbench images they might need.

An imported workbench image is stored as an OpenShift ImageStream object that has specific annotations and labels for RHOAI. The following is an example of the image stream for a custom workbench image:

```
apiVersion: image.openshift.io/v1
kind: ImageStream ①
metadata:
 annotations: ②
 opendatahub.io/notebook-image-creator: admin
 opendatahub.io/notebook-image-desc: ""
 opendatahub.io/notebook-image-name: example-workbench
 opendatahub.io/notebook-image-url: example.com/my-repo/example-workbench:1.0
 opendatahub.io/recommended-accelerators: '[]'
 ...output omitted...
 labels:
 app.kubernetes.io/created-by: byon
 opendatahub.io/dashboard: "true"
```

**Chapter 6 |** Custom Notebook Images

```
opendatahub.io/notebook-image: "true" 3
name: custom-example-workbench-image
namespace: redhat-ods-applications 4
...output omitted...
spec:
...output omitted...
tags:
- annotations:
 opendatahub.io/notebook-python-dependencies: '[]' 5
 opendatahub.io/notebook-software: '[]'
 openshift.io/imported-from: example.com/my-repo/example-workbench:1.0
from:
 kind: DockerImage
 name: example.com/my-repo/example-workbench:1.0
...output omitted...
```

- 1** Workbench images are stored as image streams within the cluster.
- 2** RHOAI stores entered metadata about the imported image as annotations on the image stream itself.
- 3** This label determines whether the custom workbench image is enabled.
- 4** Custom workbench image streams are stored in the `redhat-ods-applications` project.
- 5** Annotations on the image stream tag store metadata such as any provided Python libraries or other dependencies.

After importing a custom workbench image, users can create workbenches with it just like any other workbench. Upon creating their workbench, they select the custom workbench instead of one provided by default.



## References

### **Podman Build Documentation**

<https://docs.podman.io/en/latest/markdown/podman-build.1.html>

### **Podman Push Documentation**

<https://docs.podman.io/en/latest/markdown/podman-push.1.html>

### **Dockerfile Reference**

<https://docs.docker.com/reference/dockerfile/>

For more information on managing image streams and image stream tags, refer to the *Managing image streams* chapter of the OpenShift Images documentation at [https://docs.openshift.com/container-platform/4.14/openshift\\_images/image-streams-manage.html](https://docs.openshift.com/container-platform/4.14/openshift_images/image-streams-manage.html)

## ► Guided Exercise

# Create a Custom Notebook Image

Create custom notebook images.

Importing custom notebook images to Red Hat OpenShift AI.

## Outcomes

- Build a custom workbench image for use in Red Hat OpenShift AI (RHOAI).
- Add the data visualization library `seaborn`, based on `matplotlib`, to a workbench image.
- Import a custom workbench image into RHOAI.
- Create a workbench from a custom workbench image.

## Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI263 && lab start customnotebook-create
```

## Instructions

### ► 1. View and create the manifest files.

- 1.1. In a command-line terminal window, navigate to the materials directory.

```
[student@workstation ~]$ cd ~/AI263/labs/customnotebook-create
...output omitted...
```

- 1.2. Review the contents of the `Containerfile`.

```
[student@workstation customnotebook-create]$ cat Containerfile
FROM registry.ocp4.example.com:8443/opendatahub/workbench-images:jupyter-
datascience-ubi9-python-3.9-2023b-20240219-ffe72a0

COPY requirements.txt ./

RUN echo "Installing softwares and packages" && \
 pip install micropipenv && \
 micropipenv install && \
 rm -f ./requirements.txt

RUN chmod -R g+w /opt/app-root/lib/python3.9/site-packages && \
 fix-permissions /opt/app-root -P
```

**Note**

The Open Data Hub container images are available from Quay in this repository:  
<https://quay.io/repository/opendatahub/workbench-images>

The various images are all under the same name, but use tags to differentiate between the images. Also, note that the URL to view the repo differs from the URL used to access the container images.

- 1.3. Create a `requirements.txt` file that specifies the `seaborn` dependency.

```
[student@workstation customnotebook-create]$ echo "seaborn==0.12.2" >> \
requirements.txt
```

- 1.4. Authenticate Podman with the classroom registry by using the `developer` username and `developer` password.

```
[student@workstation customnotebook-create]$ podman login \
registry.ocp4.example.com:8443 -u developer -p developer
Login Succeeded!
```

- 1.5. Build a new container image with a fully-qualified image tag.

```
[student@workstation customnotebook-create]$ podman build . -t \
registry.ocp4.example.com:8443/student/custom-workbench:1.0
...output omitted...
Successfully tagged registry.ocp4.example.com:8443/student/custom-workbench:1.0
...output omitted...
```

**Note**

Building the container image can take several minutes.

- 1.6. Push the image to the classroom container registry.

```
[student@workstation customnotebook-create]$ podman push \
registry.ocp4.example.com:8443/student/custom-workbench:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

**Note**

If the command fails due to a missing bearer token, then try the command again.

If you still face problems, then you can proceed to the next step by using the prebuilt custom image at `quay.io/redhattraining/ai263-custom-workbench:1.0`.

- ▶ 2. Import the custom workbench image into RHOAI.

**Chapter 6 |** Custom Notebook Images

- 2.1. In a new browser window, navigate to <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com> to access the RHOAI dashboard. Log in as the **admin** user by using the **redhatocp** password.
- 2.2. Within the left side navigation, click **Settings > Notebook images**, and then click **Import new image**.
- 2.3. Within the form, specify **registry.ocp4.example.com:8443/student/custom-workbench:1.0** as the **Image location** and enter **custom-workbench** as the name. Optionally, provide a description explaining that the image provides the **seaborn** Python package.
- 2.4. Scroll to the bottom of the form, select the **Packages** tab, and click **Add packages**.
- 2.5. Specify the **seaborn** package and version **0.12.2**. Click the check button in the right side of the row to confirm.

Providing additional packages provided by the image helps teammates select the correct image when creating workbenches.

**Note**

Specifying additional packages on the image only provides information to users. This does not add anything to the image nor does it verify that the information is correct. It is up to you to verify that this metadata correctly reflects what is provided by the custom image.

- 2.6. Click **Import** to import the custom workbench image.

The custom image displays within the image settings page. Optionally, expand the row in the table to view extra data about the image, such as the specified packages.

**► 3. Create a workbench from the custom image.**

- 3.1. Log out of the RHOAI dashboard and log back in as the **developer** user by using the **developer** password.
- 3.2. From the left side navigation, click **Data Science Projects**, then click the **customnotebook-create** project.
- 3.3. Click **Create workbench** within the **Workbenches** section.
- 3.4. Provide **custom-wb** as the name, and select the **custom-workbench** image from the **Image selection** field. Leave the default values in other fields, and click **Create workbench**.
- 3.5. Wait for the status column to show **Running**. When it is ready, click **Open** to open the workbench in a new browser tab.
- 3.6. Authenticate as the **developer** user with the **developer** password. Accept the default permissions by clicking **Allow selected permissions**.

**► 4. Verify that the workbench has the **seaborn** Python package pre-installed.**

- 4.1. Within the Jupyter Launcher interface, create a new notebook by clicking **Python 3.9** under the **Notebook** section.

**Important**

Be sure to click **Python 3.9** in the **Notebook** section and not one in another section, such as **Console**.

- 4.2. Within the empty notebook cell, enter the following command:

```
pip list | grep seaborn
```

This command uses **pip** to list all available Python packages and then filters the results for the **seaborn** string.

- 4.3. Execute the notebook cell by either pressing **Shift+Enter** or by clicking **Run > Run Selected Cells**.

The cell output shows that the **seaborn** package with version **0.12.2** is installed.

**Note**

Ignore any warnings about upgrading the version of **pip**.

## Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish customnotebook-create
```

# Summary

---

- Custom workbenches provide a way to augment the default workbenches with additional software and resources.
- Because workbenches are provisioned as containers, customization just requires creating a new container image.
- Containerfiles contain the sequence of instructions for building container images.
- After pushing a container image to an accessible registry, an administrator imports it via the Red Hat OpenShift AI dashboard.
- To build a custom workbench image, use one of the RHOAI images as the base.

## Chapter 7

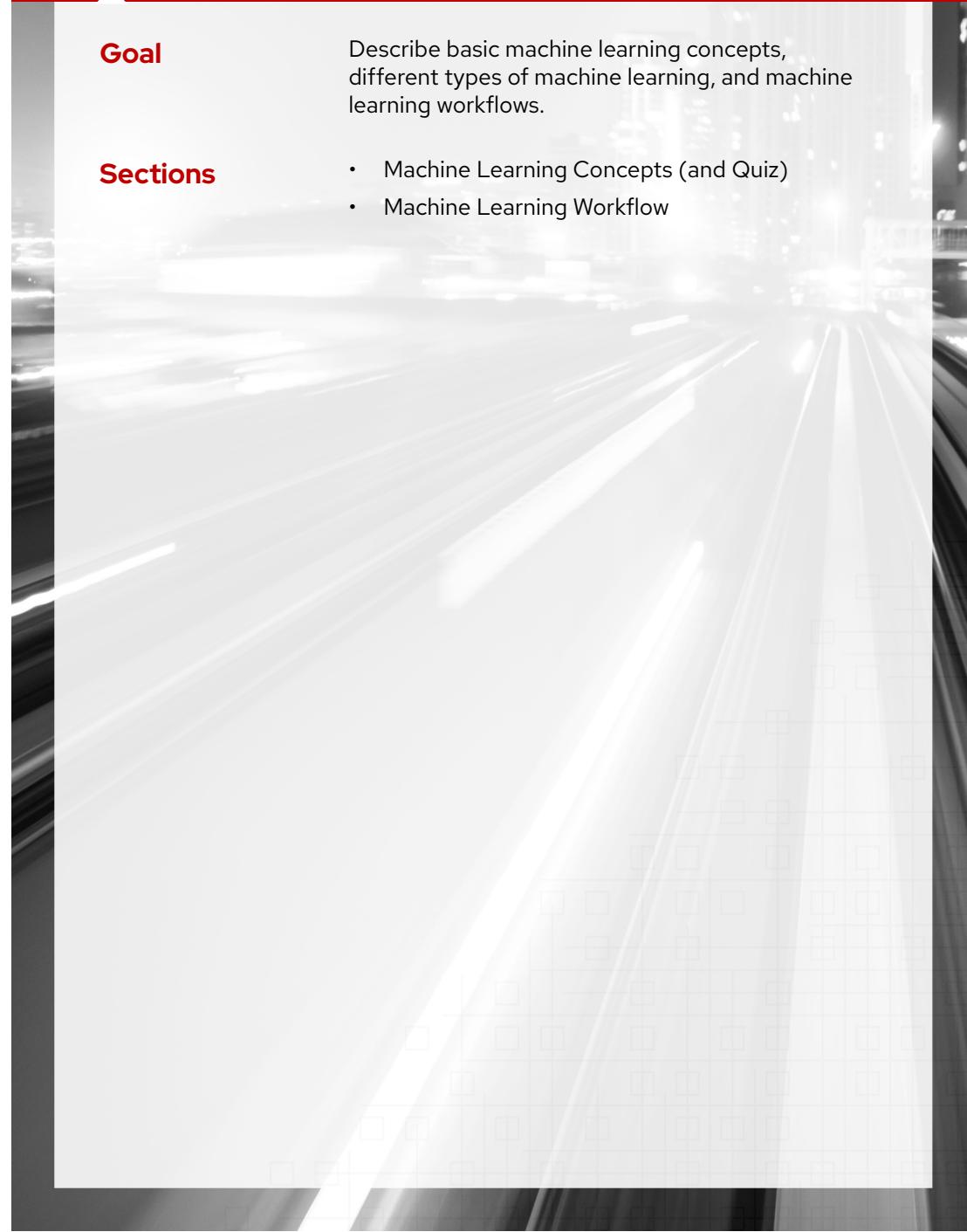
# Introduction to Machine Learning

### Goal

Describe basic machine learning concepts, different types of machine learning, and machine learning workflows.

### Sections

- Machine Learning Concepts (and Quiz)
- Machine Learning Workflow



# Machine Learning Concepts

## Objectives

- Describe basic machine learning concepts and types.

## The Goal of Machine Learning

Current software programming methods can solve a wide range of tasks, especially those that involve clear rules and deterministic outputs. However, other tasks might be more difficult to solve with traditional techniques, such as those in which you cannot define a clear set of steps to find a solution.

For example, consider that you are developing an application that identifies text sentiment. If you use traditional programming techniques, then you would have to write a complex and large set of rules to identify whether a text has a positive or negative sentiment. Developing a good solution to this problem would be difficult or impossible to implement. That implementation would also be difficult to maintain.

Similar examples of this kind are common, such as recognizing objects in images, understanding human speech, predicting the weather, and generating music, among many others. All these tasks are extremely difficult to approach by using traditional programming paradigms.

*Machine Learning (ML)* is the area of computer science that enables computers to learn from data, and can help solve the mentioned tasks. ML techniques use existing data or trial-and-error experience to learn how a real-world task works.



### Note

The goal of machine learning is to create algorithms that learn to solve a problem from data or from experience, without the need for an explicitly programmed set of instructions to solve such problem.

Machine learning is considered mostly a subset of data science, and the main approach for creating artificial intelligence (AI) applications. For more information about AI and data science, refer to the *Introduction to Red Hat OpenShift AI (AI262)* course.

## Use Cases

Scenarios where Machine learning can be a potential solution include the following common properties:

- You cannot describe the procedure or criteria to obtain a solution.
- You cannot develop an optimal solution by using traditional programming methods.
- You need to solve a non-deterministic task under changing, uncontrolled scenarios.
- You have access to data that includes many observations of the task that you want to solve.
- You want to find patterns, correlations, or insights in the data.

Although ML offers powerful mechanisms for solving complex problems, many tasks are still more suitable for traditional programming methods. Machine Learning is not recommended in the following scenarios:

- You can solve a problem by applying a set of rules and logic procedures, such as sorting lists or interacting with databases.
- You do not have access to data that describes the task.
- The data is poor, scarce, or not representative enough.
- You must be able to interpret why the application produces a certain output.

## Basic Concepts

### Model

An *ML model*, or a *model*, is a mathematical function that can recognize patterns in data and use those patterns to produce outputs. Rather than programming the model, you must train it by using learning algorithms and data that describe the problem that you want to solve. There are many families of models, and each one has its own algorithms, advantages, disadvantages, and use cases. Examples of typical models are regression models, decision trees, and neural networks.

### Inference

Inference is the process of executing a model as a software function. The model takes a set of inputs, applies the necessary computations to the data, and produces an output. For example, a model in a house pricing scenario might take inputs such as the size and the number of rooms, and infer the price of a house.

### Training

The act of using the data that describes a certain problem, to teach the model how to solve such a problem. Training a model is a repetitive task that typically passes training data to the model to infer the outputs, and compares such outputs with the expected ones. The training algorithm repeats this process, adjusting the coefficients of the model's internal equations, until the outputs of the model are similar to the outputs in the training data. This process is also commonly called *fitting*.

After training, you can use the model on cases that the model has never seen before. This ability, called *generalization*, is a key concept in machine learning, and refers to the capacity of models to adapt to unseen data. For example, assume that you want to build an application that can recognize user emotions based on their facial expressions. To this end, you can train a model by using images of user faces, each one tagged with a certain emotion. After training, you can use the model to infer emotions in users that the model has not seen before.



#### Note

You can also consider a model as a condensed representation of a set of observations, which describe a real-world problem. Under this perspective, training is the process of compressing the knowledge of these observations into a smaller artifact: the model.

### Features

The *model features* are the input values that you pass to the model to produce an output. For example, consider a model that detects digits in RGB images of 100x100 pixels. Each pixel contains three color channels, and each image contains 10000 pixels. Therefore, the model uses 30000 features, which you must pass to the model to predict the digit.

From a software perspective, the features define the parameters that you must pass to the model function. Data scientists typically put most of their effort into engineering and selecting the best features for training. If you train a model with a specific set of features, then you must use the same exact features for inference, when the model is in production.

## Target Variable(s)

This is the value, or values, that the model tries to infer. Depending on your problem, the model might produce one or more target variables. These variables can be continuous values or categorical values. In a weather forecasting scenario, for example, the temperature is a continuous value, and a severe weather warning is a categorical value (yellow, amber, or red).

## Hyperparameters

Parameters specific to the model that configure the structure and the behavior of the model itself. For example, for a decision tree, the depth of the tree and the number of leaf nodes are hyperparameters. The configuration values of the learning algorithm that trains the model are also considered hyperparameters.

During the training phase, data scientists often run multiple training experiments or rounds, each one with different combinations of hyperparameters and features. Then, they compare the experiment outcomes and select the set of hyperparameters that produce the best results. This technique is called *hyperparameter tuning*.

## Evaluation

After a training round, data scientists evaluate the performance of a model on unseen cases. They evaluate the model by using a specific subset of the data that includes samples that the model has not seen during training. Then, they calculate the deviation between the expected values, which are present in the evaluation data set, and the actual values that the model produces. To calculate the deviation, or the performance, data scientists use metrics such as Root-mean-square error, precision, and recall.

# Machine Learning Types

From a learning process perspective, machine learning is divided into three high-level categories.

## Supervised learning

Learning from labeled data, meaning that the training data includes both the features and the target variables. In the preceding digits recognition example, supervised learning requires a training data set in which each case contains the 30000 features and the target digit. For each case during training, the learning algorithm computes the error between the inferred digit and the digit in the training data set, adjusts the model to minimize that error, and repeats the process. Examples of algorithms fitting into this category are linear regressions, logistic regressions, support vector machines (SVM), neural networks, decision trees, and random forests.

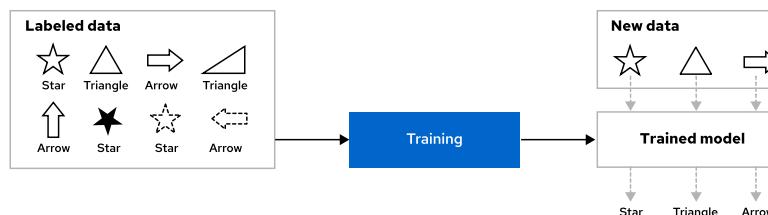


Figure 7.1: Example of supervised learning with shapes and their labels



### Note

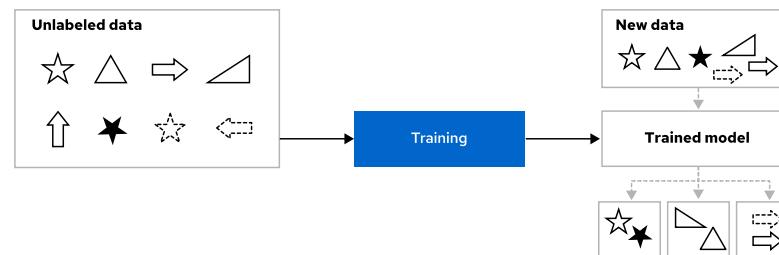
In this context, labeled data refers to target variables. These variables do not necessarily have to be categorical values. Labeled data can contain continuous numeric values.

## Unsupervised learning

Learning from unlabeled data to discover hidden structures or insights within the data, without explicit guidance about the target. In this case, the data set does not include target variables. Unsupervised learning can identify groupings such as segments of customers based on their spending habits, items that have a high possibility of being purchased together in a supermarket, or anomalous behavior.

In the preceding digits example, an unsupervised learning approach would use only the 30000 features for training, and as a result, detect groups of similar cases. The model would not be aware of any particular digit, but still would be able to segment the data in groups of images with the same number.

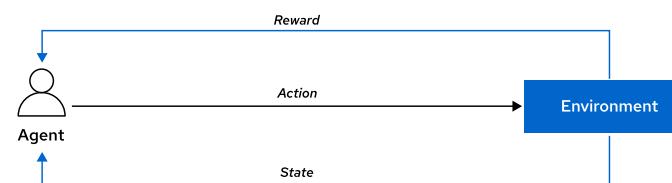
Examples of algorithms fitting into this category are K-means clustering, K-nearest neighbors (KNN), Principal Component Analysis (PCA), and autoencoder neural networks.



**Figure 7.2: Example of unsupervised learning to find groups of shapes**

## Reinforcement learning

Finding the optimal behavior of a model, called *agent*, in a certain environment. In contrast to supervised and unsupervised learning, *Reinforcement Learning (RL)* typically uses no training data. The agent learns to make decisions through trial-and-error interactions with an environment, and it receives feedback in the form of rewards or penalties for its actions. This process guides the model towards achieving a goal by maximizing the rewards. Noteworthy examples of RL are autonomous driving, learning to play video games, or classic table games such as Go or Chess. Typical RL algorithms are Monte Carlo and Deep-Q networks.



**Figure 7.3: Typical learning workflow in RL**



### Note

Other ramifications of these categories exist, such as semi-supervised learning and self-learning.

## Problem Types

Most of the tasks solved by the use of Machine Learning fall into these categories:

## Regression

The model output is a single continuous numeric value, such as the price of a house, sea level, or temperature.

## Classification

The model output is a category or a label. There are multiple types of classification problems:

- Binary: a yes/no problem. The model outputs a single Boolean value. Spam detection is a classic example of binary classification.
- Multiclass: the model outputs a single value that can take more than two discrete values. For example, in digit recognition, the output value corresponds to 10 classes (0-9).
- Multilabel: the model outputs multiple discrete values. Identifying the music genres of a song is an example of this type.

## Clustering

The model outcome is a set of *clusters*, or groups of cases. Each case is located within a cluster. The output also includes information such as the center of each cluster and the number of elements. Clustering problems are typically solved with unsupervised learning.

## Generation

The model outcome is a vector, or a matrix of continuous values. Text or image generation are examples of generative ML. Generation problems often require powerful and large models, such as autoencoder or transformer neural networks.

## Environment Interaction

The model outcomes are actions that an agent or a system must execute within a given environment, taking into account the state of the environment and the rewards or penalties received as a result of previous actions. Reinforcement learning techniques have been prolific in this area.

# Data Concepts

During training, models adjust their internal coefficients to gradually fit the training data. However, if a model is rigidly fitted to the training data, then it might not generalize well on new, unseen cases. This problem is called *overfitting*.

To verify that a model can perform well on real-life scenarios, you should evaluate the model with data that has not been seen during training. For this reason, data scientists typically split the available data into subsets for training and evaluation:

### Training subset

The data used for training. Machine learning models require moderate to large amounts of data for training, so the training set should be the biggest subset. A generally accepted approach is using 70-80% of the available data for training.

### Validation subset

The data used for hyperparameter and data tuning. Use this subset to evaluate the model after each training experiment. Compute the evaluation metrics of the model and compare which hyperparameters and features perform best across multiple training experiments. A generally accepted approach is using 10-15% of the available data for validation.

### Test subset

Use this subset to perform the final test after you have selected the best features and hyperparameters. This final test ensures that your model is not overfitted to the validation set. A generally accepted approach is using 10-15% of the available data for testing.

**Note**

When data is scarce, you can apply a technique called *cross validation*, which allows you to use all the available data for training.

The available data also defines the essence of the problem that you try to solve. For example, predicting system failures based on CSV logs is a completely different problem than detecting human voice in an audio signal. Several methods support multiple problems, such as classification and regression. However, depending on the structure of the data, certain models might be more promising.

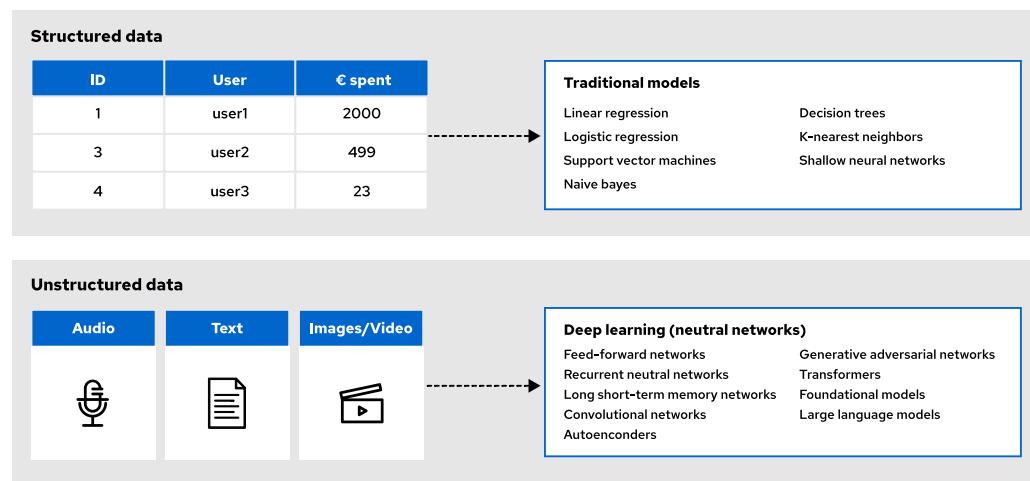
**Structured Data**

Information that is available under a certain data structure, such as the values in a data base or a CSV file. More complex structures such as time series, graphs, or trees are also considered structured data.

**Unstructured Data**

Information that is not available under a clear data structure. Text, images, audio, and video are examples of unstructured data. Unstructured data typically requires more preprocessing because ML models require data structures as inputs, such as vectors, or similar algebraic objects called tensors.

## Machine Learning Methods



## Traditional ML

Traditional ML methods use algorithms that are common in probability, statistics, and algebra, such as linear regression, support vector machines (SVM), Naive Bayes classifiers, or decision trees. In general, these algorithms work well for regression and classification problems with moderately sized, structured data, and typically do not take long to train. Some of these models, such as decision trees, are moderately interpretable, so you can inspect why the model has inferred a certain value.

## Deep Learning

Deep Learning (*DL*) techniques have been the most prolific in terms of performance in the last decade. Deep learning techniques use large neural networks as their main building block and have proven to be effective in the following scenarios:

- Large amount of unstructured, high-dimensional data such as images, text, and audio.
- Complex, non-linear relationships between the features and the target variables.
- Entity detection, such as image or speech recognition.
- Generation of images, music, and text.

Although more powerful for unstructured and high-dimensional data than traditional approaches, deep learning typically presents interpretability problems. These methods are based on neural networks, which are black-box models.

Deep learning also requires significant hardware resources. Training DL models needs large amounts of memory, and performs heavy matrix-based algebraic operations that take a long time. Some use cases might require hours, or days for training. Data scientists typically speed up this process by using graphical processing units (GPU).



## References

### What is machine learning?

<https://www.redhat.com/en/topics/ai/what-is-machine-learning>

### An introduction to machine learning today

<https://opensource.com/article/17/9/introduction-machine-learning>

### Getting started with data science and machine learning: what architects need to know

<https://www.redhat.com/architect/intro-data-science-machine-learning>

*Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, 2nd Edition.* Aurélien Géron.  
O'Reilly Media, Inc. ISBN: 9781492032649

## ► Quiz

# Machine Learning Concepts

Choose the correct answers to the following questions:

### ► 1. What is the main goal of machine learning?

- a. To make generalized predictions by using large language models.
- b. To learn how to maximize rewards from trial-and-error experience and minimize GPU usage.
- c. To learn how to solve a problem by learning complex patterns from data or from trial-and-error experience.
- d. To generate new multimedia content that is similar to the training data.

### ► 2. What is a major difference between supervised and unsupervised learning?

- a. Supervised learning is better for classification problems, and unsupervised learning is used for regression problems.
- b. Supervised learning is used for image recognition, and unsupervised learning is better for text classification.
- c. Supervised learning uses labeled data. Unsupervised learning uses unlabeled data.
- d. Supervised learning uses unlabeled data. Unsupervised learning uses labeled data.

### ► 3. What is the main advantage of deep learning over traditional ML methods?

- a. Deep learning is more interpretable than traditional ML methods.
- b. Deep learning is more memory efficient than traditional ML methods.
- c. Deep learning is more powerful than traditional ML methods for unstructured, high-dimensional data.
- d. Deep learning does not need GPUs to speed up training or inference.

### ► 4. Which two statements are disadvantages of deep learning compared to traditional ML methods? (Choose two.)

- a. Deep learning is generally less accurate than traditional ML methods.
- b. Deep learning models are less interpretable than traditional ML methods.
- c. Deep learning is more efficient than traditional ML methods.
- d. Deep learning requires more compute and memory resources than traditional ML methods.

- 5. Suppose that you must create a model that recognizes flowers in pictures. Your data set contains 5 million high-resolution flower images and, for each image, a label indicating the flower type. Which of the following options is a promising solution for this problem?
- a. Use traditional programming methods because ML is not required in this case.
  - b. Use a supervised learning approach that trains a deep learning model. Deep learning models work well for high dimensional data.
  - c. Use reinforcement learning to train the model by trial and error. The model can learn the labels by receiving rewards.
  - d. Use an unsupervised learning approach that trains a regression model to predict the labels.
  - e. Use a large language model or a foundation model. These models are efficient and can solve classification problems.

## ► Solution

# Machine Learning Concepts

Choose the correct answers to the following questions:

### ► 1. What is the main goal of machine learning?

- a. To make generalized predictions by using large language models.
- b. To learn how to maximize rewards from trial-and-error experience and minimize GPU usage.
- c. To learn how to solve a problem by learning complex patterns from data or from trial-and-error experience.
- d. To generate new multimedia content that is similar to the training data.

### ► 2. What is a major difference between supervised and unsupervised learning?

- a. Supervised learning is better for classification problems, and unsupervised learning is used for regression problems.
- b. Supervised learning is used for image recognition, and unsupervised learning is better for text classification.
- c. Supervised learning uses labeled data. Unsupervised learning uses unlabeled data.
- d. Supervised learning uses unlabeled data. Unsupervised learning uses labeled data.

### ► 3. What is the main advantage of deep learning over traditional ML methods?

- a. Deep learning is more interpretable than traditional ML methods.
- b. Deep learning is more memory efficient than traditional ML methods.
- c. Deep learning is more powerful than traditional ML methods for unstructured, high-dimensional data.
- d. Deep learning does not need GPUs to speed up training or inference.

### ► 4. Which two statements are disadvantages of deep learning compared to traditional ML methods? (Choose two.)

- a. Deep learning is generally less accurate than traditional ML methods.
- b. Deep learning models are less interpretable than traditional ML methods.
- c. Deep learning is more efficient than traditional ML methods.
- d. Deep learning requires more compute and memory resources than traditional ML methods.

► 5. Suppose that you must create a model that recognizes flowers in pictures. Your data set contains 5 million high-resolution flower images and, for each image, a label indicating the flower type. Which of the following options is a promising solution for this problem?

- a. Use traditional programming methods because ML is not required in this case.
- b. Use a supervised learning approach that trains a deep learning model. Deep learning models work well for high dimensional data.
- c. Use reinforcement learning to train the model by trial and error. The model can learn the labels by receiving rewards.
- d. Use an unsupervised learning approach that trains a regression model to predict the labels.
- e. Use a large language model or a foundation model. These models are efficient and can solve classification problems.

# Machine Learning Workflow

---

## Objectives

- Describe the machine learning workflow and MLOps.

## The ML Workflow

The machine learning workflow is a cyclic process that covers the steps required to create, deliver, and maintain machine learning models. This workflow covers the whole lifecycle of machine learning projects.

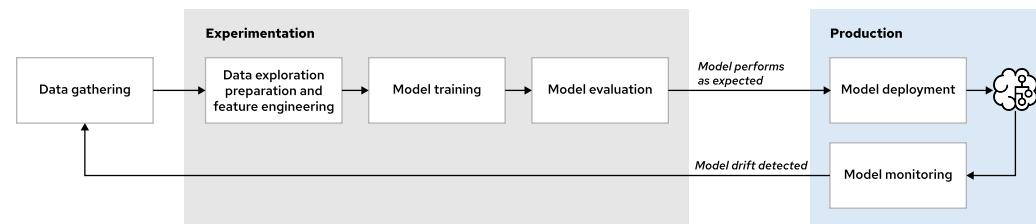


Figure 7.5: Common machine learning workflow

### Data gathering

The acquisition of the data required for training a model. Data gathering is often a continuous process, such as logging the requests of a web application into the file system or a database. In other cases, you can use precompiled data sets that represent the problem that you want to solve. Another common scenario is the integration of several data sources or data lakes into a rich data set. Complex organizations and scenarios might require Big Data techniques to collect, store, and query the data.

### Data exploration and preparation

Data scientists start their experimentation by loading the collected data, cleaning, and preprocessing it. They typically inspect statistical descriptors of the data, clean null values, and remove outliers, among other tasks. To this end, they use common data science and visualization techniques and libraries such as Pandas and Matplotlib. Data scientists also carefully select the most relevant features by scaling, normalizing, combining, and transforming the data when necessary. This process is called *feature engineering*.

Exploring the data and understanding the task at hand is key to select the most promising models. Data scientists select the models that they want to experiment with, and then adapt the input data to the input format of each model.

This phase also involves the separation of the data set into training and evaluation subsets.

### Model training

After the data is ready, data scientists pick one of the models that they have selected as promising. They instantiate and configure the model with a set of hyperparameters and use the training subset to train the model. Training is the longest and most complex process in the ML workflow. It can take significant time and hardware resources, depending on the size of the data and the model. With certain techniques, such as deep learning, GPUs are required to speed up a process that otherwise could take days or weeks, in many cases.

**Note**

Usually, data scientists do not program models from scratch. They write only the code to instantiate and train the model.

Popular libraries such as Scikit-learn, PyTorch, and TensorFlow offer different types of models that you can instantiate, such as decision trees and neural networks.

These implementations define the internal mechanisms, structures, and equations of each model.

**Model evaluation**

After the model finishes training, the data scientist evaluates its performance and quality by using standard metrics, such as accuracy, precision, and recall. A model generally requires several experiments to reach the desired quality. If the model does not yet perform as expected, then the data scientist prepares another experiment by using different input features, alternative data preprocessing approaches, adjusting the model hyperparameters, or trying a different model.

The ability to reproduce experiments and log their results is particularly important for data scientists to compare results.

**Model deployment and monitoring**

When the experiments produce a model that reaches the desired performance, the model is ready for deployment to production. Deploying a model requires access to a production environment, experience with operations, version management, scalability considerations, and ability to develop APIs. Additionally, for cloud-native environments, experience with containers, Kubernetes, and OpenShift is essential.

**Model monitoring**

ML models typically act on changing scenarios. Input data might change or evolve over time. New types of events might occur, which the algorithm has not observed during training. Because of these changing conditions, the effectiveness of a model might drift over time. When data scientists detect a model drift, they can retrain the model with newly collected data and verify whether the model adapts to the new nature of the problem.

## MLOps

Organizations that start out with machine learning often use a manual workflow. In those scenarios, data scientists execute most workflow steps by hand, and sometimes lack the collaboration or resources to gather data, train models, or deploy them to production.

*Machine learning operations (MLOps)* is a set of practices and principles aimed at optimizing the lifecycle of machine learning projects, allowing teams to iterate quickly and efficiently to deliver ML models to production. MLOps streamlines the ML workflow by applying the same principles as DevOps: continuous improvement, maximized automation, and a culture of collaboration between data scientists, engineers, and administrators.

Red Hat OpenShift AI (RHOAI) offers a platform to implement MLOps techniques, helping teams improve the steps of the ML workflow in the following ways:

## Data Gathering

Data scientists collaborate with developers and engineers to agree about which data should be collected. Engineers develop automated solutions to collect and store data in ways that comply with data privacy regulations and ease the access of data scientists to such data. In more

advanced scenarios, engineers can design mechanisms that trigger model training pipelines when new data arrives, enabling techniques such as *continuous training* and *online learning*.

## Data Preparation, Model Training, and Evaluation

Data scientists usually carry out many experiments to find the right solution. With MLOps, they can track their experiments to quickly compare and reproduce results. Data scientists can use continuous integration and deployment (CI/CD) techniques to reproduce experiments in a automated and consistent way.

Additionally, data scientists get on-demand access to the right infrastructure for running their experiments. They collaborate actively with administrators to define the memory, GPU, CPU, and storage resources required for training. The requirements vary depending on factors such as the size of the data or the model type. Deep learning experiments, for example, typically use GPUs and require large amounts of memory.

## Model Deployment and Monitoring

When experiments are reproducible, teams can use CI/CD pipelines to test the model evaluation metrics. In this way, a pipeline can automatically deploy a model if the model passes quality thresholds. You can combine pipelines with GitOps strategies, which automate model version management, deployments, rollbacks, and monitoring.

Data scientists, engineers and administrators cooperate to dimension production environments. Inference typically requires less resources than training, and aspects such as the user load or the latency limits become more relevant. For example, many models that require a GPU for training might not require a GPU for inference in production.

Teams use monitoring systems to automatically detect model drifts in production. These systems send alerts when they detect a drift in the model accuracy. The data scientists react by starting a new training round to capture the new observations and improve the model. In highly automated environments, a model drift might trigger a training pipeline with new data.



### References

#### OpenShift and AI

<https://ai-on-openshift.io/getting-started/openshift/>

#### What is MLOps?

<https://www.redhat.com/en/topics/ai/what-is-mlops>

#### Red Hat AI/ML and MLOps Customer Success Stories

<https://www.redhat.com/en/blog/red-hat-ai/ml-and-mlops-customer-success-stories>

# Summary

---

- Machine learning creates algorithms that learn from observations or trial-and-error experience.
- You can use machine learning to tackle problems that traditional programming methods cannot solve optimally.
- You must have access to data to train most machine learning models.
- The machine learning workflow requires the collaboration of data scientists, engineers, and administrators.
- MLOps optimizes ML workflows with automation, collaboration, and continuous improvement.
- Red Hat OpenShift AI provides the tools to implement MLOps practices.

## Chapter 8

# Training Models

### Goal

Train models by using default and custom workbenches.

### Sections

- Training ML Models (and Guided Exercise)
- Training Models with Custom Workbenches (and Guided Exercise)



# Training ML Models

## Objectives

- Train ML models with Scikit-learn, PyTorch and TensorFlow.

## Training ML Models

The data science and ML Python ecosystem is comprehensive and diverse, and includes a number of different libraries and methods to train ML models. Currently, some of the most popular libraries for machine learning in Python are NumPy, Scikit-learn, PyTorch, and TensorFlow.



### Note

This lecture is a hands-on introduction to basic ways of training models in Python. As such, the lecture provides only a limited view of the model training process. Understanding the complexities and foundations of this process requires comprehensive study and experience in the AI/ML fields.

## NumPy Arrays

NumPy is arguably the most popular library for scientific computation in Python. Implemented mostly in C, NumPy provides a Python interface with essential and advanced mathematical functions to operate on arrays and matrices in a high-performance way.



### Note

To clearly display instructions and output, some code blocks in this section are Python shell examples. Each >>> prompt in the Python shell precedes a Python instruction.

NumPy is designed around the core concept of *NumPy arrays*. A NumPy array is a multidimensional data structure, optimized for vectorized operations, rapid access, and memory-efficient management. You can think of a NumPy array as a vector, or as a matrix if the array includes more than one dimension.

A common way to create a NumPy array is by passing a Python list to the `numpy.array` function, as follows:

```
>>> import numpy as np
>>> data = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> data
array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])
```

The dimensions in a NumPy array must be homogeneous. For example, if, in the preceding example, one row includes two columns instead of three, then NumPy would raise an error. From a

## Chapter 8 | Training Models

typing perspective, NumPy arrays generally contain elements of the same type, such as `float` or `int`. However, an array can contain elements of different types, in which case the type of the array uses a more abstract `object` type.

To get the length of each array dimension, use the `shape` attribute:

```
>>> data.shape
(3, 3)
```

NumPy applies operations at the array level. This is one of the key features of the library. The following example illustrates how you can apply basic arithmetic and logical operators on NumPy arrays:

```
>>> data > 5
array([[False, False, False],
 [False, False, True],
 [True, True, True]])
>>> data * 10
array([[10, 20, 30],
 [40, 50, 60],
 [70, 80, 90]])
>>> data - [[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9]]
array([[0.9, 1.8, 2.7],
 [3.6, 4.5, 5.4],
 [6.3, 7.2, 8.1]])
```

You can also slice an array to select specific portions of the array. To do so, use the `start:stop:step` syntax as the index. For example, in a two-dimensional array, with rows and columns, you can select the first two rows as follows:

```
>>> data[0:2]
array([[1, 2, 3],
 [4, 5, 6]])
```

To slice across multiple dimensions, you can use a tuple of `start:stop:step` indexes. For example, in a two-dimensional array, you can select all the rows and all the columns except for the last one, as follows:

```
>>> data[:, :-1]
array([[1, 2],
 [4, 5],
 [7, 8]])
```

The first element of the index tuple `:` selects all the rows. The second element `:-1` selects all the columns except for the last one.



### Note

You can omit the `stop` and `step` elements in the `start:stop:step` syntax.

NumPy is an essential tool for data science and machine learning in Python. Most of the more ML-specialized libraries depend on NumPy, use NumPy arrays as main building blocks, or integrate with them.

## Data Exploration and Preprocessing

Before training an ML model, you typically explore the data set to understand the characteristics of the data, and prepare the data for training. In Python, the use of Pandas, Scikit-learn, and Matplotlib libraries is common for such tasks.

### Pandas

Pandas is a library that loads, analyzes, manipulates, and saves data. Pandas is based on NumPy, and uses NumPy arrays at its core, but offers a more human-friendly interface to interact with data.

Pandas provides the `DataFrame` data structure, which is similar to a spreadsheet or database table. Pandas typically stores the values of a DataFrame in NumPy arrays.

Pandas offers the `read_*` functions to load data from several formats or sources, such as CSV, XML, JSON, Parquet, HDF5, or SQL. For example, you can load a CSV file into a data frame as follows:

```
import pandas as pd

dataframe = pd.read_csv("path/to/file.csv")
```

If the data source contains column names, then Pandas identifies and parses them. You can later use these column names for indexing and selection.

Then, you can use the `pandas.DataFrame` methods to explore and manipulate the data. For example, you can get statistical and structural information about the data by using methods such as `info` and `describe`. You can also preview portions of the data.

In the following example, the `head` method displays the first five rows of the *US Tornado* data set 1950–2021. Refer to the references section for more details about this data set.

```
>>> dataframe.head()
 yr mo dy date st mag inj fat slat slon elat elon len
0 1950 1 3 1950-01-03 IL 3 3 0 39.10 -89.30 39.12 -89.23 3.6
1 1950 1 3 1950-01-03 MO 3 3 0 38.77 -90.22 38.83 -90.03 9.5
2 1950 1 3 1950-01-03 OH 1 1 0 40.88 -84.58 0.00 0.00 0.1
3 1950 1 13 1950-01-13 AR 3 1 1 34.40 -94.37 0.00 0.00 0.6
4 1950 1 25 1950-01-25 IL 2 0 0 41.17 -87.33 0.00 0.00 0.1
```

Note that the first column, which ranges from 0 to 4, is the index.

To select columns by name, you can use a dictionary or a class syntax:

```
>>> dataframe.mag
0 3
1 3
2 1
3 3
```

**Chapter 8 |** Training Models

```
4 2
...
67553 1
67554 1
67555 1
67556 1
67557 1
Name: mag, Length: 67558, dtype: int64
>>> dataframe["mag"]
0 3
1 3
2 1
3 3
4 2
...
67553 1
67554 1
67555 1
67556 1
67557 1
```

In Pandas, each individual row and column is represented as a `Series` object.

```
>>> type(dataframe.mag)
<class 'pandas.core.series.Series'>
```

Pandas supports the NumPy slicing syntax with the `iloc` attribute. For example, you can select all rows and all columns except for the last one. Pandas returns the result in a separate `DataFrame` object.

```
another_dataframe = dataframe.iloc[:, :-1]
```

You can also access the underlying NumPy array in a data frame by using the `values` attribute of the data frame.

```
>>> dataframe.values
array([[1950, 1, 3, ..., -89.23, 3.6, 130],
 [1950, 1, 3, ..., -90.03, 9.5, 150],
 [1950, 1, 3, ..., 0.0, 0.1, 10],
 ...,
 [2021, 12, 31, ..., -85.7805, 0.95, 50],
 [2021, 12, 31, ..., -84.9633, 2.75, 150],
 [2021, 12, 31, ..., -83.9498, 2.5, 75]], dtype=object)
```

In machine learning, the input values that you pass to a model are typically called features, and the output values are called target variables. Most ML models require features and target variables as separated parameters for training. By convention, data scientists often store the features in a variable called `X`, and the target variables in a variable called `y`. With Pandas, you can extract these two elements by using the `iloc` method and positional indexes, or you can use the column names of the data frame. For example, assume that you want to predict the magnitude of a tornado, so the target variable is the `mag` column. You can separate the `mag` column from the rest as follows:

## Chapter 8 | Training Models

```
X = dataframe.drop('mag', axis=1)
y = dataframe.mag
```

You can also apply conditions on DataFrame and Series objects. For example, to identify cases with a magnitude of 1, you can use the == operator on the y series.

```
>>> y == 1
0 False
1 False
2 True
3 False
4 False
...
67553 True
67554 True
67555 True
67556 True
67557 True
Name: mag, Length: 67558, dtype: bool
```



### Note

Boolean operations do not filter data. Instead, these operations return a new data frame with the same shape, in which each cell contains the result of the logical operation, `True` or `False`.

If you need to filter the data frame, then you can use conditional expressions as the index inside square brackets, as follows:

```
>>> y[y == 1]
2 1
13 1
17 1
19 1
20 1
...
67553 1
67554 1
67555 1
67556 1
67557 1
Name: mag, Length: 22885, dtype: int64
```

Pandas also supports grouping and aggregating data with methods such as `groupby`, `count`, `min`, or `max`. For example, to count the number of tornadoes with a magnitude of 1, you can filter by a condition, and then count all the cases with the `count` method.

```
>>> y[y == 1].count()
22885
```

## Chapter 8 | Training Models

Alternatively, you can use the `sum` method without filtering, which interprets every `True` value as 1.

```
>>> (y == 1).sum()
22885
```

Pandas supports more complex statistical methods, such as the Pearson correlation coefficient. You can calculate the correlation between each column by using the `corr` method, as follows

```
>>> dataframe[["inj", "fat", "slat", "slon", "elat", "elon", "len"]].corr()
 inj fat slat slon elat elon len
inj 1.000000 0.761776 -0.011010 0.031863 0.042888 -0.039848 0.257219
fat 0.761776 1.000000 -0.011070 0.022091 0.037168 -0.035873 0.237817
slat -0.011010 -0.011070 1.000000 -0.173159 0.181996 -0.043552 -0.002392
slon 0.031863 0.022091 -0.173159 1.000000 0.040213 0.045419 0.052290
elat 0.042888 0.037168 0.181996 0.040213 1.000000 -0.972665 0.277226
elon -0.039848 -0.035873 -0.043552 0.045419 -0.972665 1.000000 -0.271882
len 0.257219 0.237817 -0.002392 0.052290 0.277226 -0.271882 1.000000
wid 0.187878 0.176205 -0.032145 0.060446 0.219981 -0.223435 0.376088
```



### Note

Pandas can calculate correlation only on numerical columns. The preceding example selects columns containing numbers, before calculating the correlation.

## Matplotlib

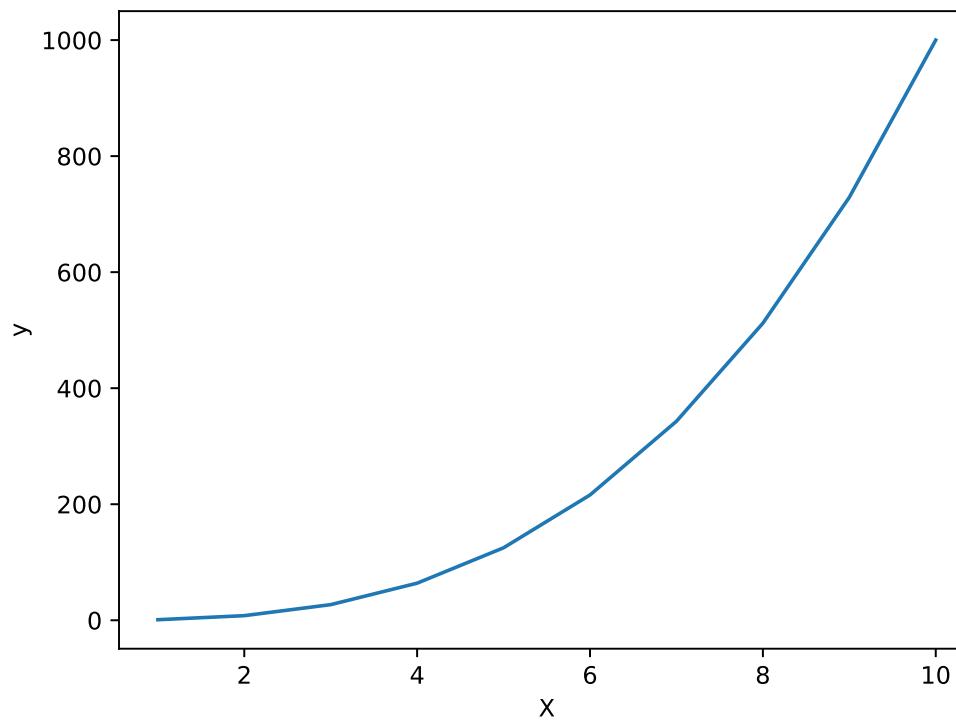
Matplotlib is a popular and powerful visualization library in the Python data science ecosystem. With Matplotlib you can plot your data to visualize and explore it. One of the most common ways to generate a plot is by using the Matplotlib `pyplot` module. For example:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
y = X ** 3

plt.plot(X, y)
plt.xlabel('X')
plt.ylabel('y')
plt.show()
```

The preceding code generates the following plot:



Pandas offers shortcut functions that integrate with Matplotlib, to quickly visualize data frames. For example, you can plot the distribution of data for each column by using a histogram.

```
data.hist(bins=20)
```

The `bins` parameter controls how many divisions you use to visualize the data.



### Note

JupyterLab can display inline plots in notebooks, but if you use Python scripts or the CLI, then you might need additional configuration. For more information, refer to the Matplotlib back ends documentation.

## Scikit-learn

*Scikit-learn* is a library that provides a suite of tools specialized in machine learning and data science. Scikit-learn is typically the first option when you work with classic machine learning algorithms. In addition to that, data scientists commonly use Scikit-learn to preprocess data for other libraries, regardless of the library they use for creating and training the model. Among many features, Scikit-learn offers modules for:

- Feature engineering and data preprocessing
- Model training
- Metrics evaluation

Before training a model, you must split your data into training and test subsets. With Scikit-learn, you can split the data with the `train_test_split` method. Most of the methods in Scikit-

## Chapter 8 | Training Models

learn integrate with Pandas and NumPy, so you can pass data frames or NumPy arrays as data parameters. For example:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(①
 X, ②
 y, ③
 test_size=0.2, ④
 random_state=0 ⑤
)
```

- ① The function returns a tuple of four subsets: training features, testing features, training targets, and testing targets.
- ② The original features.
- ③ The original targets.
- ④ The percentage of data for the testing set. In this case, 20%.
- ⑤ A seed to ensure reproducibility.

If you need to split data further, for example, to create an additional validation set, then you can call `train_test_split` again on one of the subsets.

## Model Training with Scikit-learn

In Scikit-learn, a model is called an *estimator*. Scikit-learn provides many estimators as classes that inherit from `sklearn.base.BaseEstimator`. Each estimator provides the `fit` method for training, and the `predict` method for inference.

For example, assume that you want to use a `LogisticRegression` estimator to solve a classification problem. Logistic regression is a statistical technique that predicts the probability of a categorical target variable, based on the input features. You can train this estimator as follows:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

model = LogisticRegression() ①

model.fit(X_train, y_train) ②

y_predicted = model.predict(X_test) ③
report = classification_report(y_test, y_predicted) ④
print(report)
```

- ① Create an instance of a model. You can tune the hyperparameters of a model by passing parameters to its constructor.
- ② Train the model with the training data. The first parameter is the features array or data frame. The second is the target variable. Scikit-learn accepts Python lists, NumPy arrays, and Pandas data frames.
- ③ Compute the model predictions for the test subset.

## Chapter 8 | Training Models

- ④ Compare the predicted targets with the actual targets in the test set and create a classification report.



### Important

ML models require features and targets to be numeric. Before training a model, you must preprocess the data to drop non-numerical variables or convert them to numbers. For more information, refer to the *Preprocessing data* guide in the Scikit-learn documentation.

The classification report presents different classification metrics in a single view to evaluate the trained model. A printed report for a binary classification problem displays as a table view:

| Classification report: |           |        |          |         |   |
|------------------------|-----------|--------|----------|---------|---|
|                        | precision | recall | f1-score | support | ① |
| 0                      | 0.84      | 0.92   | 0.88     | 107     | ② |
| 1                      | 0.76      | 0.62   | 0.68     | 47      | ③ |
| accuracy               |           | 0.82   |          | 154     | ④ |
| macro avg              |           | 0.80   | 0.77     | 154     | ⑤ |
| weighted avg           |           | 0.82   | 0.82     | 154     | ⑥ |

- ① The report includes precision, recall, and f1-score as classification metrics. The support value refers to the number of cases.
- ② Metrics for the false case (0).
- ③ Metrics for the true case (1).
- ④ The accuracy metric of the model. In this case, the accuracy is 0.82, which means that the model classified 82% of the test cases correctly.
- ⑤ Averages precision, recall, and f1-score across all classes (0 and 1).
- ⑥ Averages precision, recall, and f1-score across all classes, taking into account the number of cases for each class.

The use of the `sklearn.metrics` module is not restricted to Scikit-learn model training. In TensorFlow or PyTorch cases, data scientists commonly combine the use of Scikit-learn for data preprocessing and model evaluation with TensorFlow or PyTorch code.



### Note

Red Hat OpenShift AI (RHOAI) includes NumPy, Pandas, Scikit-learn, and Matplotlib in Standard Data Science workbenches, as well as in other derived workbenches, such as Pytorch and TensorFlow.

## Model Training with PyTorch

PyTorch is a deep learning library based on the core concept of *tensors*. A tensor is a multidimensional data structure, similar to a NumPy array, but optimized for compute accelerators, such as GPUs. In PyTorch, you must generally convert your data to tensors to train a model. PyTorch integrates with NumPy, so you can convert NumPy arrays and data frames to tensors.

## Chapter 8 | Training Models

The following example illustrates some ways of creating tensors:

```
import torch

tensor1 = torch.tensor(my_array) ①
tensor2 = torch.tensor(my_dataframe.values) ②
tensor3 = torch.tensor(X_train.values, dtype=torch.float) ③
tensor4 = torch.FloatTensor(X_train.values) ④
```

- ① Create a tensor from a NumPy array.
- ② Create a tensor from a Pandas data frame.
- ③ You can convert the tensor values to a specific type by using `dtype`.
- ④ PyTorch offers shortcut methods to create tensors of a specific type.

A printed tensor looks as follows:

```
>>> torch.tensor(my_array)
tensor([[7, 150, 78, ..., 35, 0, 54],
 [4, 97, 60, ..., 28, 0, 22],
 ...,
 [5, 136, 82, ..., 0, 0, 69]], dtype=torch.int32)
```

Use PyTorch to create neural network models. To define a neural network in PyTorch, create a class that inherits from `torch.nn.Module` and implements the `forward` method. The implementation of a neural network typically defines the layers of the network and how data passes forward through such layers.

A simple neural network in PyTorch might look as follows:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MyModel(nn.Module):

 def __init__(self):
 super(MyModel, self).__init__()
 # The layers of the neural network model
 self.layer1 = nn.Linear(num_features, num_neurons_layer1) ①
 self.layer2 = nn.Linear(num_neurons_layer1, num_outputs)

 def forward(self, x): ②
 x = F.relu(self.layer1(x))
 output = F.relu(self.layer2(x)))
 return output

model = MyModel() ③
```

## Chapter 8 | Training Models

- ① Define the layers of the neural network. In this example, the network includes two linear layers. A linear layer is a tensor of weights, also called layer parameters, that the layer uses to multiply the inputs and produce the outputs. During training, the network learns these weights for each layer. Each layer defines the size of its inputs and outputs. The output size of one layer must match the input size of the next one.
- ② The `forward` function defines how the input features (`X`) pass forward through the layers to produce an output. Each layer is a function that accepts the inputs as a parameter. In addition to passing data to layers, neural networks typically use *activation functions*, such as `F.relu`. Activation functions are mathematical functions that enable neural networks to learn non-linear relationships and complex patterns.
- ③ Create an instance of the model.

After creating the model, you must configure the *back propagation* training process. Back propagation is a method used for training neural networks that repeatedly adjusts the internal parameters of the network layers to minimize the error in the predictions. To configure back propagation in PyTorch, define the following elements:

```
loss_function = nn.CrossEntropyLoss() ❶
optimizer = torch.optim.Adam(model.parameters(), lr=0.01) ❷
epochs = 500 ❸
```

- ❶ The *loss function* computes the prediction error, which is the difference between the predicted and the actual values. This example uses `CrossEntropyLoss`, a typical loss function in classification scenarios.
- ❷ The *optimizer* changes the internal parameters of each layer to minimize the error of the loss function. This example uses the `Adam` optimizer, a popular *gradient descent* method to adjust the model parameters. Gradient descent is a family of calculus techniques that minimize the loss function based on the model parameters.
- ❸ An *epoch* is a full pass over the entire data set. This example configures the training process to loop over the entire data set 500 times.

Train the model by passing the data set to the model `epoch` times. The following example shows a simple training loop:

```
for i in range(epochs): ❶
 y_predicted = model.forward(X_train) ❷
 loss = loss_function(y_predicted, y_train) ❸
 optimizer.zero_grad() ❹
 loss.backward() ❺
 optimizer.step() ❻
```

- ❶ The training phase runs for `epochs` iterations.
- ❷ Pass the `X_train` input tensor through the model layers and save the outputs in the `y_predicted` tensor.
- ❸ The loss function calculates the difference between the predicted values, `y_predicted`, and the actual ones, `y_train`.

## Chapter 8 | Training Models

- ④ Reset gradients to zero before calculating them in this iteration. The *gradients* are numerical values that measure how much the parameters of the network layers have to change to minimize the loss function.
- ⑤ Apply the back propagation algorithm to recalculate gradients across all layers.
- ⑥ Update the layer parameters by applying the gradients.

After the model is trained, you can run predictions by calling `model` as a function with an input tensor:

```
model(inputs)
```

To evaluate a PyTorch model, you can use the TorchEval library, or you can use Scikit-learn evaluation metrics.



### Note

To work with PyTorch in RHOAI, use PyTorch workbenches.

## Model Training with TensorFlow

TensorFlow is the other most popular deep learning library, along with PyTorch. Similarly to PyTorch, TensorFlow is also built on top of its own implementation of tensors (`tf.Tensor`), which are multidimensional arrays of elements optimized for compute accelerators.

You can convert NumPy arrays to tensors with the `convert_to_tensor` function. However, if you use the TensorFlow Keras API, then this conversion is not required.

TensorFlow provides the Keras API, which provides routines at a higher level of abstraction than tensor operations. To train a model with TensorFlow Keras, define the model first. The following example is equivalent to the PyTorch model definition explained before.

```
import tensorflow as tf

model = tf.keras.Sequential([
 tf.keras.layers.Dense(
 num_neurons_layer1, activation='relu', input_shape=(8,)),
 tf.keras.layers.Dense(
 units=2, activation='softmax')
])
```

- ➊ Define the model as a sequence of layers.
- ➋ The Keras Dense layer is a regular neural network layer, similar to the `nn.Linear` layer in PyTorch.
- ➌ The `input_shape` parameter must be a tuple of integers. In this case, each input case is a one-dimensional vector that contains eight features.
- ➍ The `units` parameter of the last layer corresponds to the number of outputs for each case. In a binary classification problem, the model outputs probability values for two possible

## Chapter 8 | Training Models

classes: yes (1) or no (0). The `softmax` function normalizes the output for each class to a probability value between 0 and 1. As a result, if you pass a tensor of three cases to the model, then the model outputs a tensor of 3x2 shape: three cases and two classes for each case. The contents of such a tensor might look like [ [0.1, 0.9], [0.4, 0.6], [0.75, 0.25] ].

Similar to PyTorch, you must also configure the training process. The `compile` method of a model in the Keras API configures the model for training.

```
model.compile(
 optimizer='adam',
 loss='sparse_categorical_crossentropy',
 metrics=['accuracy'])
)
```

Finally, you can run the training loop by calling the `fit` method of the model. Note that you can pass the features as NumPy arrays.

```
epochs = 500
model.fit(X_train, y_train, epochs=epochs, verbose=0.5)
```

After the model is trained, you can run predictions by calling the `predict` method of the model. You can pass a NumPy array as the inputs.

```
model.predict(inputs)
```

To evaluate a TensorFlow model, you can use the `tf.keras.metrics` module, or you can use Scikit-learn evaluation metrics. For example:

```
y_predicted_probabilities = model.predict(X_test) ❶
y_predicted = tf.argmax(y_predicted_probabilities, axis=1) ❷

print(classification_report(y_test, y_predicted)) ❸
```

- ❶ Predict the targets for the test data.
- ❷ Select the predicted class: 0 or 1. In this example, for each case, the model outputs a tensor that contains two values, such as [0.85, 0.15]. The predicted class is the index that contains the maximum value, which you can find with the `argmax` function.
- ❸ Compare the predictions with the actual targets of the test set and build a report.



### Note

To work with TensorFlow in RHOAI, use TensorFlow workbenches.



## References

### **NumPy: the absolute basics for beginners**

[https://numpy.org/doc/stable/user/absolute\\_beginners.html](https://numpy.org/doc/stable/user/absolute_beginners.html)

### **NumPy: Basic indexing**

<https://numpy.org/doc/stable/user/basics.indexing.html>

### **Pandas: Intro to data structures**

[https://pandas.pydata.org/docs/user\\_guide/dsintro.html#intro-to-data-structures](https://pandas.pydata.org/docs/user_guide/dsintro.html#intro-to-data-structures)

### **Scikit-learn: User guide**

[https://scikit-learn.org/stable/user\\_guide.html](https://scikit-learn.org/stable/user_guide.html)

### **Scikit-learn: Preprocessing data**

<https://scikit-learn.org/stable/modules/preprocessing.html>

### **Scikit-learn: Model evaluation - Classification report**

[https://scikit-learn.org/stable/modules/model\\_evaluation.html#classification-report](https://scikit-learn.org/stable/modules/model_evaluation.html#classification-report)

### **Matplotlib: Back ends**

<https://matplotlib.org/stable/users/explain/figure/backends.html>

### **PyTorch: Tensors tutorial**

[https://pytorch.org/tutorials/beginner/basics/tensorqs\\_tutorial.html#](https://pytorch.org/tutorials/beginner/basics/tensorqs_tutorial.html#)

### **PyTorch: Neural networks**

[https://pytorch.org/tutorials/beginner/blitz/neural\\_networks\\_tutorial.html](https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html)

### **PyTorch: torch.nn.Linear**

<https://pytorch.org/docs/stable/generated/torch.nn.Linear.html>

### **Keras: The high-level API for TensorFlow**

<https://www.tensorflow.org/guide/keras>

### **US Tornado Dataset 1950–2021 by Random Draw**

<https://www.kaggle.com/datasets/danbraswell/us-tornado-dataset-1950-2021>

. Derived from data produced by NOAA's Storm Prediction Center.

## ► Guided Exercise

# Training ML Models

Train ML models with Scikit-learn, PyTorch and TensorFlow.

## Outcomes

- Explore and analyze data.
- Train a model by using Scikit-learn, TensorFlow, and PyTorch.

## Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI264 && lab start training-models
```

## Instructions

In this exercise, you train an ML model for medical diagnosis. The exercise provides the necessary data to predict whether a patient might have diabetes.

- 1. Within the `training-models` data science project, create three separate workbenches, one for working with Scikit-learn, another for PyTorch, and a third one for TensorFlow.
  - 1.1. In a web browser, navigate to the Red Hat OpenShift AI (RHOAI) dashboard at <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com> and authenticate as the developer user by using the developer password.
  - 1.2. Navigate to the project details by clicking **Data Science Projects** and then the `training-models` project.
  - 1.3. Create a small workbench called `sklearn-wb` that uses the `Standard Data Science` image.
  - 1.4. Create a small workbench called `pytorch-wb` that uses the `PyTorch` image.
  - 1.5. Create a small workbench called `tensorflow-wb` that uses the `TensorFlow` image.
- 2. Explore the data set in the `sklearn-wb` workbench.
  - 2.1. Wait for the `sklearn-wb` workbench to start.
  - 2.2. Access the workbench by clicking the **Open** button next to it. Use the username `developer` and password `developer` to authenticate to the workbench.
  - 2.3. Accept the default selected permissions by clicking **Allow selected permissions**. The JupyterLab interface displays.

**Chapter 8 |** Training Models

- 2.4. In the JupyterLab interface, click the **Git** icon from the left tools pane.
  - 2.5. Click **Clone a Repository**.
  - 2.6. Enter <https://github.com/RedHatTraining/AI26X-apps> as the repository, and click **Clone**.
  - 2.7. In the JupyterLab file browser, navigate to the `AI26X-apps/models/training-models/diabetes-prediction` directory and double click the `01-data-exploration.ipynb` notebook to open it.
  - 2.8. Follow the instructions within the `01-data-exploration.ipynb` notebook.
- **3.** Train a model by using Scikit-learn.
- 3.1. Within the same `sklearn-wb` workbench, open the `02-model-training-sklearn.ipynb` notebook.
  - 3.2. Follow the instructions within the `02-model-training-sklearn.ipynb` notebook.
- **4.** Train the same model by using PyTorch.
- 4.1. Return to the RHOAI **training-models** project dashboard.
  - 4.2. Open the `pytorch-wb` workbench and authenticate as you did in the previous workbench.
  - 4.3. Clone the <https://github.com/RedHatTraining/AI26X-apps> repository and navigate to the `AI26X-apps/models/training-models/diabetes-prediction` directory.
  - 4.4. Open the `02-model-training-pytorch.ipynb` notebook and follow the instructions.
- **5.** Train the same model by using TensorFlow.
- 5.1. Return to the RHOAI **training-models** project dashboard.
  - 5.2. Open the `tensorflow-wb` workbench and authenticate as you did in the previous workbench.
  - 5.3. Clone the <https://github.com/RedHatTraining/AI26X-apps> repository and navigate to the `AI26X-apps/models/training-models/diabetes-prediction` directory.
  - 5.4. Open the `02-model-training-tensorflow.ipynb` notebook and follow the instructions.

## Finish

On the **workstation** machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish training-models
```

# Training Models with Custom Workbenches

## Objectives

- Train classic models with libraries not included in the default workbenches.

## Additional Python Packages for Machine Learning

Beyond the popular foundational libraries covered in the previous lecture, many other libraries exist for working with data and training models. The following libraries are noteworthy examples found in the tool sets of many data scientists:

### SciPy

*SciPy* is a library for scientific computing, based on NumPy. The library extends NumPy by defining high-level functions and adding modules for specific use cases, such as signal processing. Often, SciPy also acts as a fundamental building block for other ML Python libraries.

Red Hat OpenShift AI (RHOAI) includes SciPy in most of its default workbenches, such as Standard Data Science, PyTorch and TensorFlow.

### Seaborn

*Seaborn* is a visualization library built on top of Matplotlib. It is not as customizable as Matplotlib, but offers simpler routines to create rich statistical graphics with less code. Additionally, Seaborn includes a number of themes and styles to generate elegant and pleasing visualizations.

Standard RHOAI workbenches do not include Seaborn.

### Plotly

Similar to Seaborn, *Plotly* is a high-level visualization library. In contrast to Seaborn, Plotly is not based on Matplotlib. Plotly offers capabilities to create web-based, interactive visualizations, including more advanced graphics like 3D charts and maps.

RHOAI includes Plotly in most of its default workbenches.

### XGBoost

XGBoost is relatively popular library that implements gradient boosting methods for machine learning. Gradient boosting techniques train multiple decision tree models in combination to optimize the performance of the final result. XGBoost implements the Scikit-learn estimator interface, so you can switch between Scikit-learn and XGBoost models quickly.

Standard RHOAI workbenches do not include XGBoost.

### Transformers

The `transformers` library is based on a specific deep learning architecture, called *transformer model*. Transformer models are deep learning models specifically designed for sequence-to-sequence problems. In these problems, a model outputs a set or a sequence of target variables. This library is popular for the creation of large language models (LLMs). Examples of use cases are text generation, machine translation, or image generation.

Standard RHOAI workbenches do not include the `transformers` library.

## Customizing Workbench Packages

If you need to use a package that is not included in any of the RHOAI standard workbenches, then you must add it to a workbench. Generally, you should start from a workbench with a set of installed libraries that is closer to your needs. For example, if you need to use the `transformers` library, then you might want to start from a deep learning workbench, such as PyTorch. To add the package to the workbench, you have two options:

- Open a standard workbench and use `pip` via JupyterLab to install the required packages.
- Create a custom workbench image that includes the package.

The first option is generally preferred for initial experimentation. When a package is missing in your workbench, you can run `pip` through JupyterLab in the following ways:

- Open a terminal and run `pip install package`.
- In a notebook, run `!pip install package`. IPython runs the command prefixed with `!` directly in the system shell, so this option is equivalent to running the command in the terminal.
- In a notebook, run `%pip install package`. IPython treats commands prefixed with `%` as *magic commands*, which are IPython functions to control the kernel environment. If you run a `pip` command in a notebook cell without a prefix, then IPython treats it as the `%pip` magic command. The `%pip` magic command runs `pip` in the current kernel process. As a result, this command installs the package in the virtual environment of the current kernel.



### Note

JupyterLab uses IPython within Python notebooks. It is the component that handles the interactive Python code execution.

After you verify that a particular package version suits your needs and does not conflict with other packages in the workbench, then you can build a custom workbench image. Building a container image is recommended because it can provide a stable and homogeneous working environment for your team. It can also save time for data scientists, because installing some dependencies can be a long process. For more details about building a custom container image, refer to the *Red Hat OpenShift AI Administration (AI263)* course.



### Note

You can install a specific package version by using the `==` version operator, as follows: `pip install package==version`.

## Machine Learning with Other Languages

The standard RHOAI workbenches are Python-based. The ML ecosystem, however, is not restricted to Python. In fact, many other languages offer tools and libraries that you can use for machine learning and data science. The following non-comprehensive list includes some of these alternatives:

### R

R is a language historically used in scientific research for statistical analysis, including machine learning. Popular R packages for ML include *Caret*, *TensorFlow for R*, *XGBoost*, and *Glmnet*, among others.

### **Julia**

Julia is a relatively recent language. Created in 2012, it quickly gained traction in academia and research environments. In these environments, Julia ML in data science is common and utilizes libraries such as *MLJ.jl*, *Knet.jl* and *TensorFlow.js*.

### **Java**

Traditionally, Java has been a common option for machine learning, especially before the popularization of Python. Noteworthy examples of Java ML applications and libraries include Weka or OpenNLP.

### **JavaScript**

Although JavaScript has been historically used for web development, the popularity of Node.js has made JavaScript ubiquitous. The myriad of JavaScript applications also includes machine learning. Libraries such as *BrainJS* and *TensorFlowJS* can train and run models directly in a web browser.

### **MATLAB**

MATLAB is a classic tool for numerical computation in engineering and science. Rather than just a programming language, MATLAB is a full-fledged platform that includes its own syntax and IDE. MATLAB is a proprietary solution, but you can find open source alternatives such as *GNU Octave*.

If you wish to create machine learning models in RHOAI by using languages other than Python, then you must create a custom workbench. Such a workbench should include a Kernel that can execute code in your language of choice. For example, to create a workbench for the Julia language, you could create a workbench image that includes the **IJulia** kernel. With this image, you could create notebooks that can run Julia code.

For a comprehensive list of available kernels and how to integrate them in JupyterLab, refer to the Jupyter Kernels list in the references.



## References

### SciPy user guide

<https://docs.scipy.org/doc/scipy/tutorial/index.html#user-guide>

### Seaborn tutorial

<https://seaborn.pydata.org/tutorial.html>

### Plotly for Python

<https://plotly.com/python/>

### XGBoost Documentation

[https://xgboost.readthedocs.io/en/release\\_2.0.0/](https://xgboost.readthedocs.io/en/release_2.0.0/)

### Transformers documentation

<https://huggingface.co/docs/transformers/index>

### Python packaging user guide: Version specifiers

<https://packaging.python.org/en/latest/specifications/version-specifiers/#version-specifiers>

### Julia: Machine learning packages

<https://juliapackages.com/c/machine-learning>

### Weka 3: Machine learning software in Java

<https://www.cs.waikato.ac.nz/ml/weka/>

### Apache OpenNLP

<https://opennlp.apache.org/>

### MATLAB help

<https://www.mathworks.com/help/matlab/>

### GNU Octave

<https://octave.org/>

### CRAN Task View: Machine Learning & Statistical Learning

<https://cran.r-project.org/web/views/MachineLearning.html>

### IPython: System shell access

<https://ipython.readthedocs.io/en/stable/interactive/reference.html#system-shell-access>

### Jupyter kernels list

<https://github.com/jupyter/jupyter/wiki/Jupyter-kernels>

## ► Guided Exercise

# Training Models with Custom Workbenches

Train classic models with libraries not included in the default workbenches.

### Outcomes

- Add missing libraries with pip.
- Train an ML model by using a custom workbench.

### Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI264 && lab start training-custom
```

### Instructions

Train a model by using XGBoost. This library is not included in the standard workbenches of Red Hat OpenShift AI (RHOAI).

- ▶ 1. Create and access a new workbench called `training-custom-wb` within the `training-custom` data science project.
  - 1.1. In a web browser, navigate to the RHOAI dashboard at <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com> and authenticate as the `developer` user by using the `developer` password.
  - 1.2. Navigate to the project details by clicking **Data Science Projects** and then `training-custom` project.
  - 1.3. Create a small workbench called `training-custom-wb` that uses the `Standard Data Science` image. Wait for the workbench to start.
  - 1.4. Access the workbench by clicking the **Open** button next to it. Use the username `developer` and password `developer` to authenticate to the workbench.
  - 1.5. Accept the default selected permissions by clicking **Allow selected permissions**.  
The JupyterLab interface displays.
- ▶ 2. Clone the repository and open the `model-training-xgboost.ipynb` notebook.
  - 2.1. In the JupyterLab interface, click the **Git** icon from the left tools pane.
  - 2.2. Click **Clone a Repository**.

**Chapter 8 |** Training Models

- 2.3. Enter <https://github.com/RedHatTraining/AI26X-apps> as the repository, and click **Clone**.
  - 2.4. In the JupyterLab file browser, navigate to the `AI26X-apps/models/training-custom` directory and double click the `model-training-xgboost.ipynb` notebook to open it.
- 3. Try to run the notebook cells and inspect the error.
- 3.1. Click **Run > Run All Cells**.
  - 3.2. Verify that the first Python cell raises the following error:

```
ModuleNotFoundError: No module named 'xgboost'
```

This error indicates that the `xgboost` package is not installed in the workbench.

- 4. Install the `xgboost` package in the current kernel.
- 4.1. Add a cell at the top of the notebook with the following contents:
- ```
%pip install xgboost==2.0.3
```
- 4.2. Click **Kernel > Restart Kernel and Run All Cells....**
 - 4.3. Verify that all cells execute without errors.
- 5. Return to the RHOAI dashboard and add the XGBoost image.
- 5.1. Return to the RHOAI dashboard and log out by clicking **developer > Log out**.
 - 5.2. Log in again as the **admin** user by using the `redhatocp` password.
 - 5.3. Click **Settings > Notebook images**.
 - 5.4. Click **Import new image**.
 - 5.5. Enter `quay.io/redhattraining/ai264-xgboost-notebook:v2-20240319` as the image location, **XGBoost** as the name, and click **Import**.

**Note**

In the interest of time, this exercise provides the prebuilt `ai264-xgboost-notebook` workbench image, which includes XGBoost. For more details about building custom workbench images, refer to the *Red Hat OpenShift AI Administration (AI263)* course. You can also find the Containerfile of this image at <https://github.com/RedHatTraining/AI26X-apps/tree/main/models/training-custom/xgboost-workbench/>

- 6. Create an XGBoost workbench.
- 6.1. Navigate to the `training-custom` project page by clicking **Data Science Projects** and then the `training-custom` project.
 - 6.2. Create a small workbench called `xgboost-wb` that uses the newly added XGBoost image. Wait for the workbench to start.

Chapter 8 | Training Models

- 6.3. Access the workbench by clicking the **Open** button next to it. Use the username **developer** and password **developer** to authenticate to the workbench.
 - 6.4. Accept the default selected permissions by clicking **Allow selected permissions**.
- 7. Test the notebook on the XGBoost workbench.
- 7.1. In the JupyterLab interface, click the **Git** icon from the left tools pane.
 - 7.2. Click **Clone a Repository**.
 - 7.3. Enter <https://github.com/RedHatTraining/AI26X-apps> as the repository, and click **Clone**.
 - 7.4. In the JupyterLab file browser, navigate to the `AI26X-apps/models/training-custom` directory and double click the `model-training-xgboost.ipynb` notebook to open it.
 - 7.5. Run all the notebook cells.
 - 7.6. Verify that the cells run without issues.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish training-custom
```

Summary

- Tensors and NumPy arrays are key data structures in machine learning.
- You can use Scikit-learn to train classic ML models.
- You can use TensorFlow or PyTorch to train deep learning models.
- You can preprocess data with Pandas and Scikit-Learn.
- Red Hat OpenShift AI (RHOAI) provides workbenches to work with all those libraries.
- If you need to use a library that is not in the default RHOAI workbenches, then create a custom workbench.
- You can create workbenches with kernels that support a language other than Python.

Chapter 9

Enhancing Model Training with RHOAI

Goal

Use RHOAI to apply best practices in machine learning and data science.

Sections

- Inspecting Workbench Resources (and Guided Exercise)
- Scaling Data Loading (and Guided Exercise)
- Monitoring the Training Process (and Guided Exercise)
- Software Engineering Principles for Data Science (and Guided Exercise)

Inspecting Workbench Resources

Objectives

- Verify installed dependencies, the CPU memory and the GPU memory.

Inspecting Workbench Resources

AI workbenches can use many computing resources during model development and execution. Tracking the resources used by a project is important for many reasons, including the following:

- Determine necessary workbench size based on actual resource usage
- Diagnose performance issues due to lack of sufficient resources
- Create resource estimates for production deployments

Resources are allocated at the following three levels:

- The *OpenShift node* hosting the workbench container has a fixed amount of CPU and memory.
- The *container* running the workbench is allocated CPU and memory based on the size selected when the workbench was created.
- The *Jupyter Notebook Kernel* uses a subset of resources available to the workbench container.



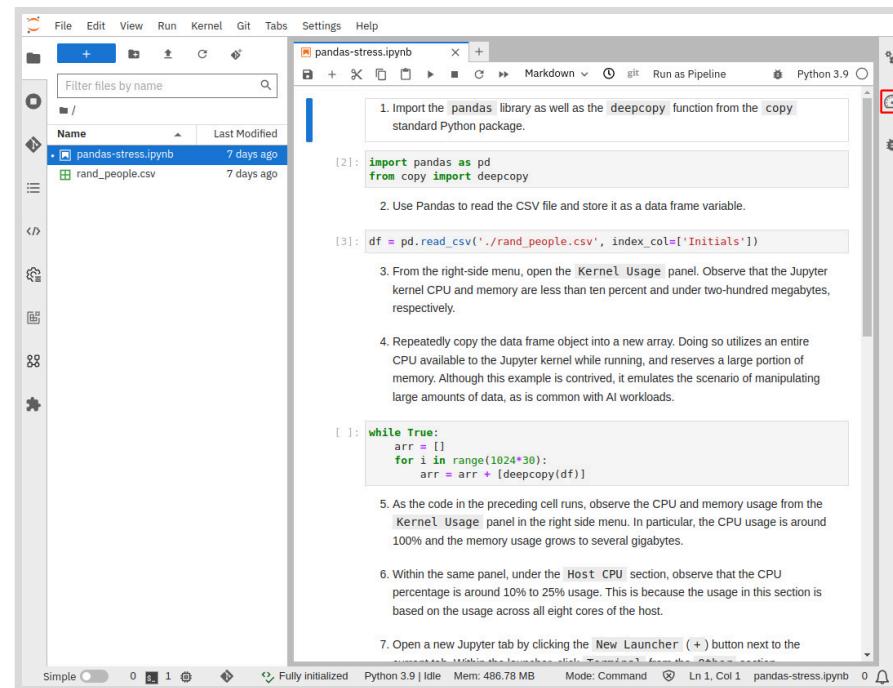
Note

The Jupyter Kernel is a process responsible for executing the code in your notebook. Kernels are *programming-language-specific* processes that run independently from each other and interact with Jupyter applications and interfaces.

Workbenches include mechanisms for viewing resource availability and usage at all three levels, such as the Kernel Usage panel, the `top` command, and container system files.

The Kernel Usage Panel

Provided by the *jupyter-resource-usage* extension, the panel is accessed by clicking the tachometer icon on the far right sidebar.

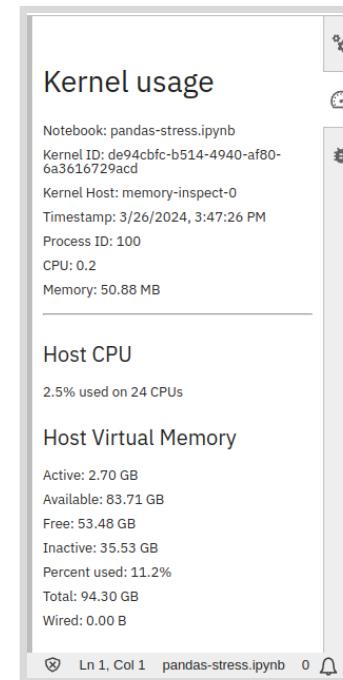


The screenshot shows a Jupyter Notebook interface with a file tree on the left containing 'pandas-stress.ipynb' and 'rand_people.csv'. The main area displays a code cell with the following content:

```
[1]: Import the pandas library as well as the deepcopy function from the copy standard Python package.  
[2]: import pandas as pd  
      from copy import deepcopy  
  
[3]: df = pd.read_csv('./rand_people.csv', index_col=['Initials'])  
  
1. Import the pandas library as well as the deepcopy function from the copy standard Python package.  
2. Use Pandas to read the CSV file and store it as a data frame variable.  
3. From the right-side menu, open the Kernel Usage panel. Observe that the Jupyter kernel CPU and memory are less than ten percent and under two-hundred megabytes, respectively.  
4. Repeatedly copy the data frame object into a new array. Doing so utilizes an entire CPU available to the Jupyter kernel while running, and reserves a large portion of memory. Although this example is contrived, it emulates the scenario of manipulating large amounts of data, as is common with AI workloads.  
5. As the code in the preceding cell runs, observe the CPU and memory usage from the Kernel Usage panel in the right side menu. In particular, the CPU usage is around 100% and the memory usage grows to several gigabytes.  
6. Within the same panel, under the Host CPU section, observe that the CPU percentage is around 10% to 25% usage. This is because the usage in this section is based on the usage across all eight cores of the host.  
7. Open a new Jupyter tab by clicking the New Launcher (+) button next to the
```

The status bar at the bottom indicates: Fully initialized, Python 3.9 | Idle, Mem: 486.78 MB, Mode: Command, Ln 1, Col 1, pandas-stress.ipynb, 0.

The `Kernel usage` section displays the CPU and memory that the kernel is currently using. These values update every five seconds by default. The `Host CPU` section displays the total amount of CPU and memory available on the OpenShift node. It also displays the current amount of CPU and memory that the node is using.



The status bar at the bottom of the page displays the total amount of memory being used by the container.

The screenshot shows a Jupyter Notebook interface with a sidebar containing file management and search tools. The main area displays a notebook titled 'pandas-stress.ipynb' with several code cells and associated numbered steps:

- Step 1: Import the pandas library as well as the deepcopy function from the copy standard Python package.
- Step 2: Use Pandas to read the CSV file and store it as a data frame variable.
- Step 3: From the right-side menu, open the Kernel Usage panel. Observe that the Jupyter kernel CPU and memory are less than ten percent and under two-hundred megabytes, respectively.
- Step 4: Repeatedly copy the data frame object into a new array. Doing so utilizes an entire CPU available to the Jupyter kernel while running, and reserves a large portion of memory. Although this example is contrived, it emulates the scenario of manipulating large amounts of data, as is common with AI workloads.
- Step 5: As the code in the preceding cell runs, observe the CPU and memory usage from the Kernel Usage panel in the right side menu. In particular, the CPU usage is around 100% and the memory usage grows to several gigabytes.
- Step 6: Within the same panel, under the Host CPU section, observe that the CPU percentage is around 10% to 25% usage. This is because the usage in this section is based on the usage across all eight cores of the host.
- Step 7: Open a new Jupyter tab by clicking the New Launcher (+) button next to the tabs.

The status bar at the bottom indicates the notebook is 'Fully Initialized' with 'Python 3.9 | Idle' and 'Mem: 486.78 MB'.

The Top Command

A terminal can be opened within the Jupyter notebook by creating a new tab and selecting the Terminal icon. The terminal runs within the workbench container.

Type the command `top` and then press Enter.

The output displays information on the processes in the container that use the most resources.

Press `Ctrl+c` to exit the `top` command.

Container System Files

The following container system files provide insight into the container's resource usage:

`/sys/fs/cgroup/cpu.max`

The allowed CPU quota in microseconds for a period, followed by the period length. Divide these values to get the amount of CPU available to the container. This matches the amount of CPU selected when the workbench was created.

`/sys/fs/cgroup/memory.max`

The maximum amount of memory that can be used by the container, in bytes. This matches the amount of memory selected when the workbench was created.

`/sys/fs/cgroup/memory.current`

The amount of memory currently being used by the container. This value matches the amount of memory displayed in the status bar.

View the contents of each file by using the `cat` command, such as in the following example:

```
[user@host ~]$ cat /sys/fs/cgroup/cpu.max
```



References

Kernels (Programming Languages)

<https://docs.jupyter.org/en/latest/projects/kernels.html>

Using cgroups-v2 to control distribution of CPU time for applications

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_monitoring_and_updating_the_kernel/using-cgroups-v2-to-control-distribution-of-cpu-time-for-applications_managing-monitoring-and-updating-the-kernel

► Guided Exercise

Inspecting Workbench Resources

Verify installed dependencies, the CPU memory and the GPU memory.

Outcomes

- Monitor the CPU and memory usage within a Jupyter notebook.
- Use the `top` command-line tool to also inspect resource usage.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI264 && lab start practices-inspect
```

Instructions

- ▶ 1. Create and access a new workbench called `memory-inspect` within the `practices-inspect` data science project.
 - 1.1. In a web browser, navigate to the RHOAI dashboard at <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com> and authenticate as the `developer` user by using the `developer` password.
 - 1.2. Navigate to the project details by clicking **Data Science Projects** and then the `practices-inspect` project.
 - 1.3. Create a small workbench called `memory-inspect` that uses the `Standard Data Science` image. Wait for the workbench to start.
 - 1.4. Access the workbench by clicking the **Open** button next to it. Use the username `developer` and password `developer` to authenticate to the workbench.
 - 1.5. Accept the default selected permissions by clicking **Allow selected permissions**. The JupyterLab interface displays.
- ▶ 2. Clone the repository and open the `pandas-stress.ipynb` notebook.
 - 2.1. In the JupyterLab interface, click the **Git** icon from the left tools pane.
 - 2.2. Click **Clone a Repository**.
 - 2.3. Enter <https://github.com/RedHatTraining/AI26X-apps> as the repository, and click **Clone**.
 - 2.4. In the JupyterLab file browser, navigate to the `AI26X-apps/models/practices-inspect` directory and double click the `pandas-stress.ipynb` notebook to open it.

- 3. For the remainder of this exercise, follow the instructions within the `pandas-stress.ipynb` notebook.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish practices-inspect
```

Scaling Data Loading

Objectives

- Load data in a scalable way by reading pieces of data that fit in memory.

Many machine learning models are trained by using large sets of data. These data sets can often exceed the available memory of the training system.

One solution is to acquire more memory, but that increases infrastructure costs. Additionally, building systems with increasing amounts of memory can also increase overall system complexity and maintenance effort. Some data sets are large enough that they cannot fit into the memory of any single system. Data sets used in many deep learning use cases, such as in large language models, are examples of such large data sets.

To avoid these issues, you can train the models incrementally. In other words, you can train models via smaller consecutive training rounds, with each call including only a subset of the data.

This method of incremental training is called *out-of-core* or *online* training. Conversely, training on all data at the same time is called *batch* or *offline* training.



Note

To incrementally train, you must select a training model that can handle it. For example, in Scikit-learn, the naive Bayes classifiers and passive-aggressive regressor all support out-of-core training.

Additionally, online learning provides you with flexibility to incorporate new or preprocessed data into the model. For example, on many occasions, you might need to apply data engineering techniques that are memory intensive, even if the raw data set fits in memory. In these cases, applying an incremental approach also helps to preprocess and prepare your data without running out of memory.

Offline or Batch Learning

With offline learning, you train the model by using all data, minus the data set aside for validation, at the same time. Thus, all training data must fit into system memory. This creates a hard constraint on the amount of data you can provide your model.

Additionally, if you integrate new data into the model, then you must rebuild the entire model from all of the data. Depending on how often your organization needs to incorporate new data, rebuilding the entire model can require a lot of retraining time and resource usage. In particular, main memory can become a constraint as the data set grows.

However, the main advantage to offline learning is that it is simpler to build and maintain a model. You train and deploy each new version of the model independently, and you incorporate new data with all the existing data simultaneously. These factors, and others, can result in simpler training and deployment needs.

Online or Out-of-core Learning

With online learning, you train the model in multiple passes. You use only a subset of the data, called a *chunk*, or a *batch*, during each pass. Some data should still be set aside for validation, with the split occurring on each new chunk.

Over time, models trained on constantly changing data incur more *model rot* or *data drift*, which is how out of date the model is from the real-world environment. Because you can feed an online model with new data continuously, the model can adapt to changing situations and environments without needing to completely retrain the entire model.

However, this means that the model is also more susceptible to bad data. With an online learning strategy, you must exercise increased caution around the training data. Additionally, developing an online learning model can be more challenging for data scientists.

Generally, it would be best to favor an offline learning strategy unless you hit its constraints, such as not having enough memory or needing to adapt to new data faster.

Incrementally Training with Scikit-learn

As suggested, online learning requires training the model in an incremental fashion, as opposed to all at the same time.

To incrementally train by using Scikit-learn, in place of calling the `fit` method on the model, you call the `partial_fit` method within a loop. Within each iteration, before calling the function, you perform the usual data split: one part for training and another for model validation. Then, the call to the partial fit method adapts the model based on the portion of data provided.

To update the model as new data arrives, you continue to call the partial fit method.

For example, assume you have a model object called `model` and a CSV file containing your training data. The following Python code snippet highlights the main pieces for incrementally training the model.

```
for chunk in pd.read_csv("path/to/data.csv", chunksize=chunk_size): ①
    X = chunk.drop('target', axis=1) ②
    y = chunk['target']

    ...

    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=test_size) ③

    ...

    model.partial_fit(X_train_partial, y_train_partial, classes=y.unique()) ④
```

- ① Read the CSV in a loop, storing each partial set of data in a variable called `chunk`.
- ② Create variables `X` and `y` that store the feature and target columns of the dataframe, respectively.
- ③ Split the chunk of data into separate training and validation sets.
- ④ Instead of calling the usual `fit` method one time, repeatedly call the `partial_fit` method on the `model` object.

**Note**

When using PyTorch or TensorFlow, the overall strategy is similar, but the specifics of doing so are outside of the scope of this course.

Batches in Deep Learning

If you use deep learning techniques, then you probably need to split your data into chunks. The size and large data requirements of deep learning models typically implies the use of incremental learning to make the models and the data fit in memory.

Data scientists usually design deep learning training loops to iterate over data chunks, which are typically called *batches* in deep learning scenarios. In fact, libraries such as TensorFlow and PyTorch offer routines and properties to quickly configure batches and batch sizes.

For example, in TensorFlow, you can split the data in batches with the `tf.data.Dataset` object. In PyTorch, you can use the `DataLoader` object to achieve a similar result.



References

Difference between Online & Batch Learning

<https://vitalflux.com/difference-between-online-batch-learning/>

Scikit-learn: Strategies to scale computationally: bigger data

https://scikit-learn.org/1.4/computing/scaling_strategies.html

Machine Learning Glossary: Out-of-core

<https://machinelearning.wtf/terms/out-of-core/>

Scikit-learn

https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB.partial_fit

TensorFlow: Writing a training loop from scratch

https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch

PyTorch: Loading Batched and Non-Batched Data

<https://pytorch.org/docs/stable/data.html#loading-batched-and-non-batched-data>

► Guided Exercise

Scaling Data Loading

Load data in a scalable way by reading pieces of data that fit in memory.

Outcomes

- Use incremental training to train a model from a data set that cannot fit into system memory.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI264 && lab start practices-data
```

Instructions

- ▶ 1. Create a workbench called `practices-data-wb` within the `practices-data` Red Hat OpenShift AI (RHOAI) data science project.
 - 1.1. In a web browser, open the RHOAI dashboard by navigating to <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com>.
 - 1.2. Authenticate by using the `developer` username and `developer` as the password.
 - 1.3. View the list of data science projects by clicking **Data Science Projects** from the left navigation menu. Select the `practices-data` project by clicking its name.
 - 1.4. Create a RHOAI workbench by using the name `practices-data-wb` and selecting the `Standard Data Science` image.
- ▶ 2. Clone the repository containing the course exercises and open the `incremental-sklearn.ipynb` notebook.
 - 2.1. After the workbench is running, open it by clicking **Open**, authenticating as the `developer` user with password `developer`, and accepting the default permissions.
 - 2.2. Clone the <https://github.com/RedHatTraining/AI26X-apps> git repository into the workbench.
 - 2.3. Navigate to the `AI26X-apps/models/practices-data` directory and open the `full-training.ipynb` and the `incremental-sklearn.ipynb` files.
- ▶ 3. Continue the exercise by following the instructions in the `full-training.ipynb` notebook. After running the `full-training.ipynb` notebook, continue with the `incremental-training.ipynb` notebook.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish practices-data
```

Monitoring the Training Process

Objectives

- Use TensorBoard to visualize the evaluation metrics of trained models.

Monitoring the Training Process

During training, some machine learning models require more attention than others. Many classic ML techniques, such as Naive Bayes and Decision trees, use deterministic and explainable algorithms that do not require much introspection during training. When you train a Naive Bayes classifier, for example, the model uses the Bayes' theorem to learn probabilities.

Gradient Descent and Loss

Many other techniques, such as logistic regression and neural networks, rely on an optimization algorithm for training, typically a form of *gradient descent*. Gradient descent is a family of algorithms that attempt to minimize a *loss function*. The *loss* is the difference between the expected target values and the values that the model infers. Gradient descent adjusts the internal weights and parameters of a model to minimize the loss.

Gradient descent is not deterministic, because it typically starts from a random set of model internal weights, and relies on a extensive set of hyperparameters, especially in deep learning scenarios. Examples of these hyperparameters are the learning rate, loss function, the batch size, and the number of training epochs.



Note

In deep learning, an *epoch* is a full training pass over the entire data set.

The loss is considered an internal, non-interpretable value that drives the training process. It depends on the randomly initialized model weights and on the data itself. This means that the scale and the values of the loss are arbitrary. They do not fall inside a given scale and might vary across training rounds. In general, rather than observing the specific loss values, you should focus on how loss decreases as training progresses.

Evaluation Metrics

To measure the performance of the model and compare performance across models, use evaluation metrics. In contrast to the loss, evaluation metrics are well-known, interpretable standards for assessing the quality of models. Each type of ML problem requires a particular set of metrics, for example:

- In classification problems, where you predict a label or a class, *accuracy*, *precision*, *recall*, or *F-score* are popular metrics. All these metrics range between zero and one, and higher values are better. For example, an accuracy value of `0.999` means that the model is almost perfectly accurate.

- In regression problems, where you predict a continuous numeric value, some typical metrics are *mean squared error (MSE)*, *mean absolute error (MAE)*, or the *coefficient of determination (R^2)*. In MSE and MAE, lower values are better. In R^2 , higher values are better.

During training, you can compute these metrics in combination with the loss, both in the training and validation data sets, to monitor the training process.

Inspecting Loss and Metrics During Training

Depending on the library that you use, you might have a different set of built-in training logging mechanisms.



Note

This section focuses on TensorFlow. For other libraries, refer to the corresponding library documentation.

In the TensorFlow Keras API, you can start by setting the evaluation metrics that you want to use during training. The following example uses accuracy.

```
model.compile(  
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),  
    loss='sparse_categorical_crossentropy',  
    metrics=['accuracy'])  
)
```

Next, call the `fit` method. You can use the `verbose` parameter of the `fit` function to control the verbosity of the training process.

```
model.fit(X_train, y_train, epochs=100, verbose=1)
```

During training, the `fit` method outputs the loss and the configured metric, which in this case is accuracy, after each epoch:

```
Epoch 1/100  
20/20 [=====] - 0s 1ms/step - loss: 0.5091 - accuracy:  
0.7573  
Epoch 2/100  
20/20 [=====] - 0s 1ms/step - loss: 0.4959 - accuracy:  
0.7573  
Epoch 3/100  
20/20 [=====] - 0s 1ms/step - loss: 0.4987 - accuracy:  
0.7866  
Epoch 4/100  
20/20 [=====] - 0s 1ms/step - loss: 0.4747 - accuracy:  
0.7891  
...output omitted...
```

Note that as training progresses, the loss keeps decreasing, and the accuracy keeps increasing. In other words, gradient descent reduces the drift between the predicted and the expected values, therefore tuning the model to produce more accurate results.

**Note**

The `1ms/step` value refers to how long a training step takes. In deep learning, a *training step* is a training pass over a batch of data.

For example, if you train a model with a data set of 1000 cases, loaded in batches of 10, then the training process takes 100 steps. If you train during 30 epochs, then the training process takes 3000 steps.

By default, the TensorFlow Keras API computes both the loss and the other evaluation metrics only on the training set. To compute these values also on a validation set, pass the set as tuple of features (X) and target variables (y) to the `fit` method, as follows:

```
model.fit(  
    X_train,  
    y_train,  
    epochs=100,  
    validation_data=(X_validation, y_validation)  
)
```

When you add a validation set to the training, Keras also computes the loss and the evaluation metrics on this data set. Keras displays these values by prefixing them with `val_`.

```
Epoch 1/100  
17/17 [=====] - 1s 12ms/step - loss: 13.1611 - accuracy:  
0.5307 - val_loss: 3.6104 - val_accuracy: 0.5286  
Epoch 2/100  
17/17 [=====] - 0s 3ms/step - loss: 2.5767 - accuracy:  
0.4749 - val_loss: 2.0732 - val_accuracy: 0.5143  
Epoch 3/100  
17/17 [=====] - 0s 3ms/step - loss: 1.7958 - accuracy:  
0.5028 - val_loss: 1.7802 - val_accuracy: 0.4000  
Epoch 4/100
```

During training, reporting the loss and evaluation metrics for both data sets is crucial to verify that your model is learning correctly.

Understanding Metrics During Training

Ideally, you expect a model to improve its metrics as training progresses. As the model learns new patterns, the loss function should decrease and the evaluation metric should improve, for both the training and the validation sets. For example, if you use accuracy as a metric, then you might expect a high accuracy for the training set. You can also expect a reasonably high accuracy on the validation set, which probes that the model can generalize well.

Underfitting

Assume that you start training a simple model, but, during training, you observe that the accuracy of your model is low for both the training and validation data, as the following plot shows:

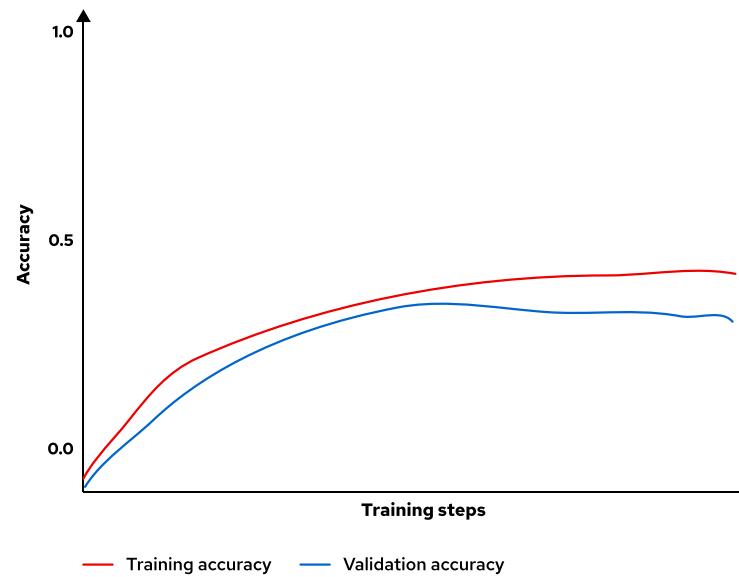


Figure 9.4: Example of underfitting: a model with low accuracy during training

This problem is called *underfitting*. It means that the model cannot learn the complex patterns present in the data. Consequently, the accuracy of your model is low for both the training and validation data, regardless of how many training epochs or steps you run.

To solve underfitting, you might want to add more features to your data set or use more complex models. For example, a logistic regression model for artificial vision can underfit, so you might want to switch to a deep neural network.

Overfitting

Assume that you now use a complex model, such as deep neural network. During training, you observe that although the accuracy of the model keeps improving for the training set, the accuracy for the validation set starts decreasing after a certain number of learning steps or epochs, as follows:

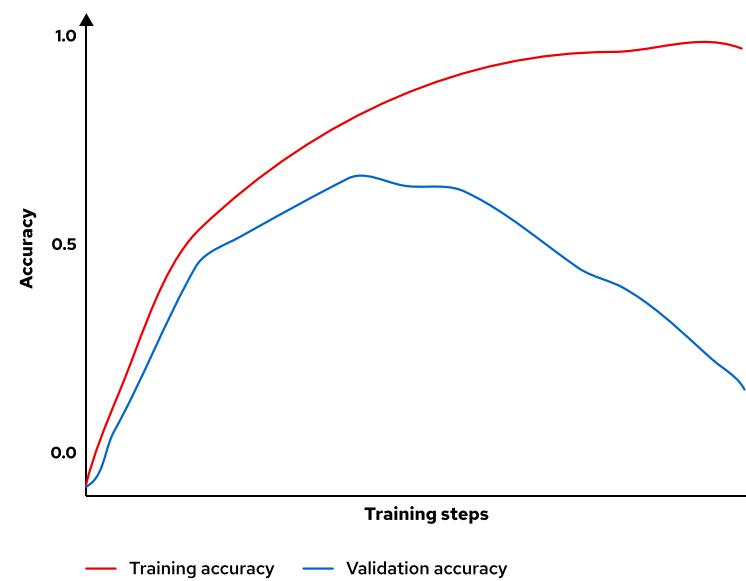


Figure 9.5: Example of overfitting: a model with low accuracy on the validation set

This problem is called *overfitting*. When a model overfits, the model is tightly adjusted to the particularities and potential noise of the training set. As a result, although the model presents good metrics for the training set, its performance is poor on the validation set, meaning that the model cannot generalize well with unseen examples. The more you train the model, the more the performance degrades on the validation set.

To solve overfitting, you might want to add more training cases so that the model can learn a broader perspective of the problem. If you do not have access to additional samples, then you can try a simpler model or reduce the number of features.



Note

Deep learning models are prone to overfitting due to their complexity, so they typically need vast amounts of cases to avoid this problem.

The balance between overfitting and underfitting is directly influenced by a key concept in machine learning: the *bias-variance trade off*. This trade off explains the subtle balance between the bias (training error) and the variance (validation error). You must achieve this balance by carefully tuning your data, your model, and the training hyperparameters. After solving these problems, the metrics for a training process might look like the following chart:

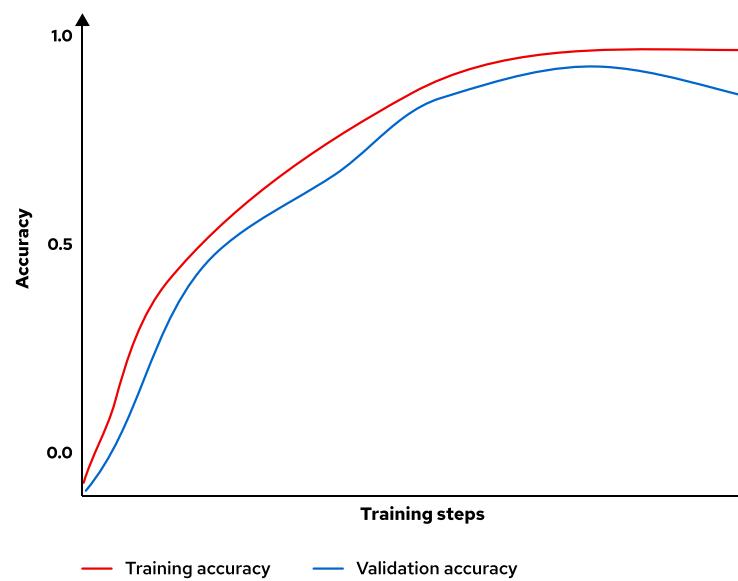


Figure 9.6: Example of a model with a good training accuracy

Similarly, the loss evolution curve might look like the following visualization. Both the training loss (loss) and the validation loss (val_loss) decrease. As training progresses, the loss values keep decreasing. This decrease causes them to get closer to zero.



Figure 9.7: Example of ideal loss curves during training

**Note**

Even if your model can generalize well, deep learning models tend to overfit when you train the model during too many epochs. Techniques such as *EarlyStopping* can detect when a metric degrades, stop the training, and select the state of the model that achieved the best metric.

The right number of training epochs depends on the problem to solve, the model, and the data.

TensorBoard

The TensorFlow project includes an application called TensorBoard. TensorBoard is a web application for metrics visualization in deep learning scenarios. Using TensorBoard can help you to inspect charts such as the preceding ones, to compare the loss and the metrics between the training and the validation set.

By default, TensorBoard runs on `localhost`, on port 6007, and requires a directory of logs. You can run TensorBoard as follows:

```
[user@host ~]$ tensorboard --logdir path/to/logdir
...output omitted...
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --
bind_all
TensorBoard 2.16.2 at http://localhost:6007/ (Press CTRL+C to quit)
```

The TensorBoard dashboard looks like the following:

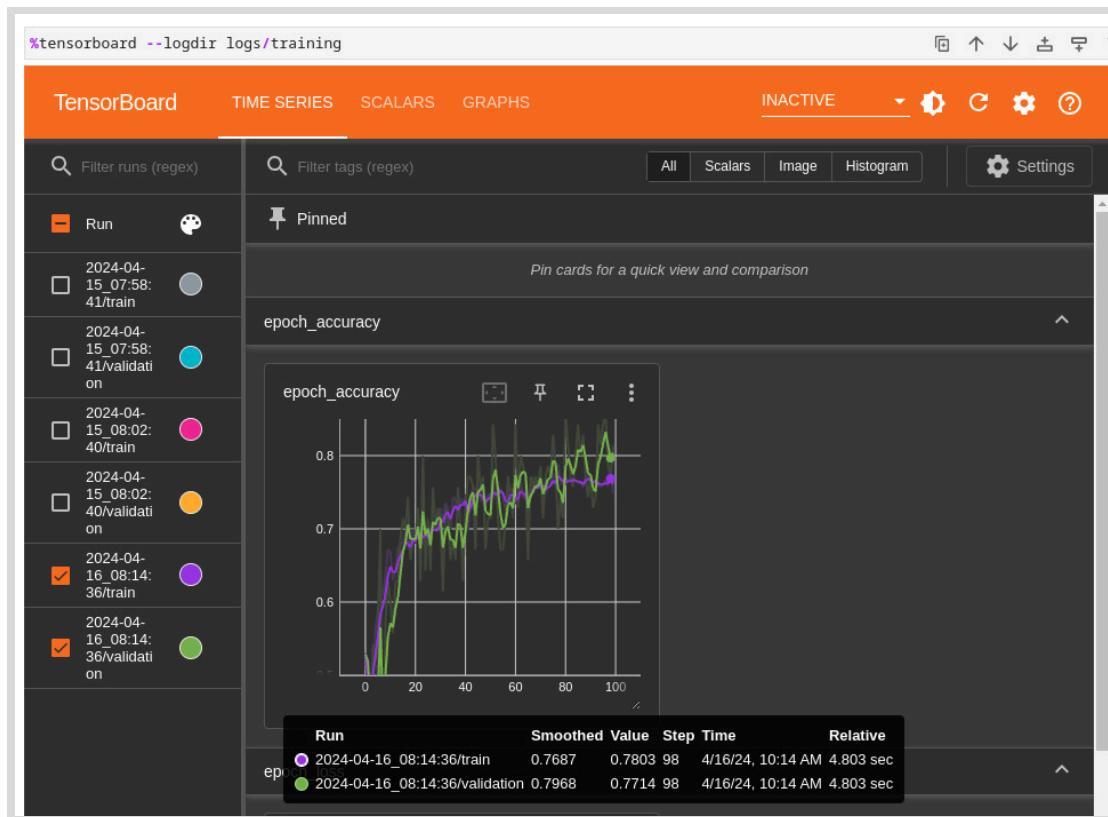


Figure 9.8: TensorBoard dashboard

For more details about using the UI, refer to the TensorBoard documentation.

Running TensorBoard on Workbenches

In Red Hat OpenShift AI (RHOAI), you can run TensorBoard in PyTorch and TensorFlow workbenches. These workbenches include TensorBoard and the `tensorboard` notebook extension. With this extension, you can run and open TensorBoard directly from a notebook.

To do so, run this code in a cell:

```
import os  
  
os.environ["TENSORBOARD_PROXY_URL"] = os.getenv("NB_PREFIX") + "/proxy/6006/" ①  
  
%tensorboard --logdir path/to/logdir ②
```

- ① The extension renders TensorBoard as an HTML IFrame inside the notebook. The `TENSORBOARD_PROXY_URL` controls the URL of this IFrame. Because TensorBoard runs inside the workbench container, you must set this URL to be `os.getenv("NB_PREFIX") + "/proxy/6006/"`. The `NB_PREFIX` is a Kubeflow variable that contains the URL path of the notebook. The `/proxy/6006/` path is where the workbench exposes TensorBoard.
- ② Start TensorBoard, if it is not already running, and render the dashboard in the notebook.

**Note**

At the time of creating this course for RHOAI 2.8, setting the proxy URL is required, according to the RHODS-4799 [<https://issues.redhat.com/browse/RHODS-4799>] issue.

Writing Logs for TensorBoard

To display training metrics, TensorBoard requires the logs directory to contain log files in a specific format. You must configure TensorFlow to write these log files as follows:

```
logging_callback = tf.keras.callbacks.TensorBoard(log_dir="path/to/logdir") ❶

model.fit(
    X_train,
    y_train,
    epochs=100,
    validation_data=(X_validation, y_validation),
    callbacks=[logging_callback] ❷
)
```

- ❶ Create a TensorBoard callback. This callback writes metrics to log files in the `log_dir` directory.
- ❷ Use the `callbacks` parameter of the `fit` function to provide the callback to the training loop. You can provide more than one callback.

**Note**

To export metrics from PyTorch in TensorBoard format, use the `torch.utils.tensorboard` module. Refer to the PyTorch documentation for more details.



References

What is gradient descent?

<https://www.ibm.com/topics/gradient-descent>

Metrics and scoring: quantifying the quality of predictions

https://scikit-learn.org/stable/modules/model_evaluation.html

TensorFlow: Training & evaluation with the built-in methods

https://www.tensorflow.org/guide/keras/training_with_builtin_methods

Training with PyTorch: The Training Loop

<https://pytorch.org/tutorials/beginner/introyt/trainingyt.html#the-training-loop>

TensorBoard

<https://www.tensorflow.org/tensorboard>

Keras: Callbacks API

<https://keras.io/api/callbacks/>

How to use TensorBoard with PyTorch

https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html

► Guided Exercise

Monitoring the Training Process

Use TensorBoard to visualize the evaluation metrics of trained models.

Outcomes

- Use TensorBoard in workbenches
- Inspect training and validation metrics

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI264 && lab start practices-monitor
```

Instructions

Monitor the training process of a model by using TensorBoard.

- ▶ 1. Create and access a new workbench called `practices-monitor-wb` within the `practices-monitor` data science project.
 - 1.1. In a web browser, navigate to <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com> and authenticate as the developer user by using the developer password.
 - 1.2. Click **Data Science Projects** and then click the `practices-monitor` project.
 - 1.3. Create a small workbench called `practices-monitor-wb` that uses the `TensorFlow` image. The workbench might take a few minutes to start.
 - 1.4. Access the workbench by clicking the **Open** button next to it. Use the developer username with developer password to authenticate to the workbench.
 - 1.5. Accept the default selected permissions by clicking **Allow selected permissions**. The JupyterLab interface displays.
- ▶ 2. Clone the repository and open the `model-training.ipynb` notebook.
 - 2.1. In the JupyterLab interface, click the **Git** icon from the left tools pane.
 - 2.2. Click **Clone a Repository**.
 - 2.3. Enter <https://github.com/RedHatTraining/AI26X-apps> as the repository, and click **Clone**.
 - 2.4. In the JupyterLab file browser, navigate to the `AI26X-apps/models/practices-monitor` directory and double click the `model-training.ipynb` notebook to open it.

- 3. For the remainder of this exercise, follow the instructions within the `model-training.ipynb` notebook.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish practices-monitor
```

Software Engineering Principles for Data Science

Objectives

- Apply software engineering principles when training models in Jupyter notebooks.

Software Engineering

Software engineering is an engineering field that studies processes and principles that help deliver quality software quickly.

These processes are a set of software development methods and techniques, such as Agile or test-driven development (TDD), that provide a framework for developers working in a project.

Good quality software is easy to maintain and adapt to changing requirements. Software is easy to maintain if the code is easy to understand, and has enough tests for validating that the software works as expected. This lets developers become familiar with the code base quickly and reduces the chance of introducing bugs. To build software that can adapt to changing requirements, you must apply software architecture and design principles.

Writing good quality software requires practice and study. This section focuses on only a small set of principles to help you improve your notebook's code quality.

Writing readable code

Write concise functions and use meaningful names for functions, parameters, and variables.

Don't Repeat Yourself (DRY)

Avoid code duplication.

Single-Responsibility Principle (SRP)

Functions should only have one reason to change.

You Aren't Gonna Need It (YAGNI)

Avoid overengineering.

Like other engineering and science fields, machine learning requires writing and maintaining software. Applying software engineering principles when you create ML models can help you to build and deliver better models.

Writing Readable Code

Writing functions that contain a few lines of code, and using meaningful variable names and function signatures, helps your code to be readable and easy to understand.

A short function that does one thing is easier to understand, maintain, and test, than a long function that does many tasks. Also, using meaningful names that communicate the function's purpose and meaningful parameter names and variables, can reduce the amount of documentation you need to write.

For example, compare using the letter x to store the temperature average value of a data set, to using the variable name `average_temp` for the same purpose. When you see x variable used in the code you must remember what it means, but the `average_temp` name is self-explanatory.

DRY

Code duplication is a potential source of bugs, it increases the number of lines that you must read to understand the code base. If you have a repeated code block that contains bugs, then you might forget to apply the bug fix in all of the affected blocks.

SRP

Your functions should have only one reason to change. If you use the same function in two places for different requirements, then a change to the function to satisfy a change in one of the requirements might break the logic for the other requirement.

YAGNI

Writing code for something that is not a current requirement but might happen in the future could increase your code complexity in a nonproductive way.

Modularity in Notebooks

One of the key features that make notebooks great for experimentation is the fact that their cells behave like a shell. You can quickly try your ideas and experiment by adding new Python statements. Unfortunately, because of quick experimentation, you might end up with notebooks that contain a lot of messy, unorganized, and duplicated code sequences.

One way to add structure, and avoid code duplication among notebooks, is to create a module within your project that contains common functionality. These modules can expose common utility functions and functions with a high level of abstraction.

A high-level abstraction function expresses what your code does without forcing the person reading it to understand the implementation. For example, you might want to create a method that reads a data set from a path, applies a list of transformations to add features, and returns the training and test data for training a model.

You could have that code block with the implementation of every step repeated on your project notebooks, but it would make your notebooks hard to read and hard to maintain.

Instead, you could create a `training` module with a function that has the following signature:

```
def prepare_training_data(dataset: Path, features: List[callable]) -> list
```

Note that naming matters because it helps other developers to find existing functionality. In the example, the `prepare_training_data` function exists in the `training` module.

Utility functions are functions that replace a common pattern in your code reducing the number of lines needed to perform a task. For example, you might want to fill missing values in some columns with the average value for the column.

```
import pandas as pd

patients = pd.read_csv("patients.csv")

patients["days_to_recover"] =
    patients["days_to_recover"].fillna(patients.days_to_recover.mean())
```

If you find this pattern several times in your code, then replacing the pattern with a `fill_na_with_mean` utility function improves readability.

```
def fill_na_with_mean(data_frame: DataFrame, column_names: list)
```

The previous method can replace several lines of code in your notebook with just one method call. The implementation could iterate the `column_names` list applying the mean to the missing values on each column.



Note

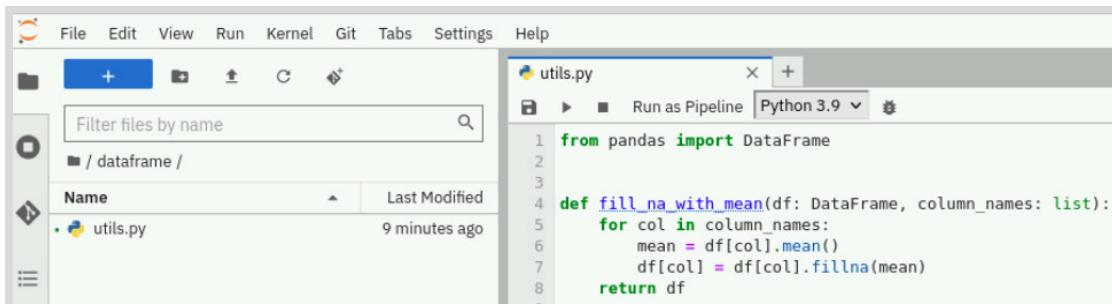
Extracting some parts of your ML workflow into reusable modules can also be helpful after model training. For example, automated data science pipelines might benefit from a function that automates data gathering.

Also, a reusable data cleaning module might be helpful for inference, when you need to clean the new data that you pass to the model.

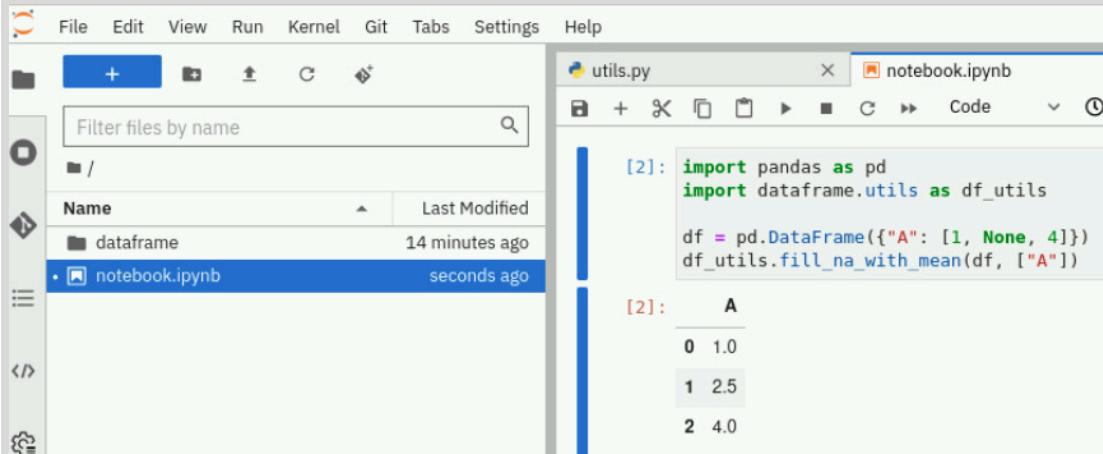
Working with Modules in JupyterLab

To create a module in JupyterLab, you must create a regular Python module. You can do this by creating a Python file that contains the source code. The name of the file is the name of the module. You can also create module hierarchies by creating directory trees. For more information, refer to the Python documentation on modules.

The following image shows a `dataframe.utils` module that contains the `fill_na_with_mean` function.



The following image shows how you can import and use the `fill_na_with_mean` function in your notebook.



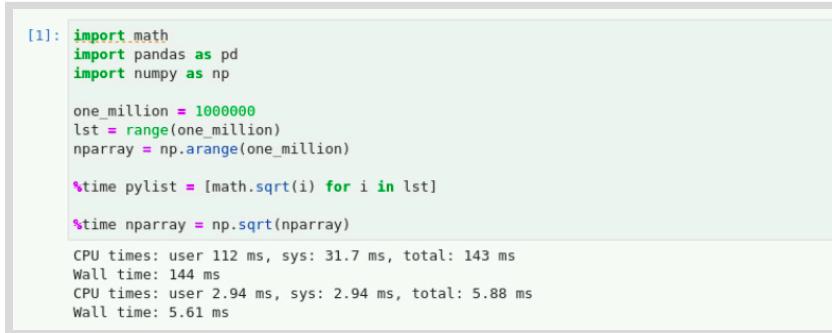
The screenshot shows a Jupyter Notebook interface. On the left, there's a file browser with a search bar and a list of files. The 'notebook.ipynb' file is selected. On the right, there are two code cells. Cell [1] contains imports for pandas, pd, and df_utils. Cell [2] contains code to create a DataFrame 'df' with one column 'A' containing values [1, None, 4] and then fills the 'None' value with the mean of the other values. The resulting DataFrame 'df' is displayed as:

	A
0	1.0
1	2.5
2	4.0

Performance with NumPy Arrays

NumPy arrays are more efficient to work with than Python lists. NumPy arrays use vectorization, which applies an operation on every element of an array instead of applying the operation sequentially.

For example, you can compare the timings for calculating the square root for a one dimensional array with one million elements.



The screenshot shows a Jupyter Notebook cell [1] with the following code and timing results:

```
[1]: import math
import pandas as pd
import numpy as np

one_million = 1000000
lst = range(one_million)
nparray = np.arange(one_million)

%time pylist = [math.sqrt(i) for i in lst]
%time nparray = np.sqrt(nparray)
```

CPU times: user 112 ms, sys: 31.7 ms, total: 143 ms
Wall time: 144 ms
CPU times: user 2.94 ms, sys: 2.94 ms, total: 5.88 ms
Wall time: 5.61 ms

Figure 9.11: Timing comparison

Using the NumPy method, which uses vectorization, is around 20 times faster in this particular case.

NumPy offers a set of mathematical and common operations. You can also transform a custom function to a vectorized one by using NumPy's `vectorize`. The following images show the timing comparison of adding an even or odd column to a dataframe by either using iterative or vectorized operations.

```
[2]: from pandas import Series
ten_million = 10000000
df = pd.DataFrame({"A": np.random.randint(low=0, high=1000000, size=ten_million)})

def calc_even_odd(series: Series):
    even_odd_list = []
    for num in series:
        even_odd_list.append(num % 2)
    return even_odd_list

%time df['even_odd_iterations'] = calc_even_odd(df['A'])

CPU times: user 3.45 s, sys: 257 ms, total: 3.71 s
Wall time: 3.74 s
```

Figure 9.12: Iterative operations

```
[5]: def even_odd(x):
    return x % 2

calc_even_odd_vectorized = np.vectorize(even_odd)

%time df['even_odd_vectorized'] = calc_even_odd_vectorized(df['A'])

CPU times: user 1.62 s, sys: 484 ms, total: 2.11 s
Wall time: 2.14 s
```

Figure 9.13: Vectorized operations

**Note**

Whenever possible, try to use NumPy routines rather than iterating over an array.

**References**

The Pragmatic Programmer: From Journeyman to Master. Andrew Hunt, David Thomas. Addison-Wesley Professional, Inc. ISBN: 9780201616224

YAGNI principle

<https://martinfowler.com/bliki/Yagni.html>

SRP principle

<https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleResponsibilityPrinciple.html>

Why is NumPy fast?

<https://numpy.org/devdocs/user/whatisnumpy.html#why-is-numpy-fast>

► Guided Exercise

Software Engineering Principles for Data Science

Apply software engineering principles when training models in Jupyter notebooks.

Outcomes

- Refactor a notebook to extract common code to a Python module.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI264 && lab start practices-engineering
```

Instructions

In this exercise, you will analyze two notebooks and extract common functionality to a Python module to avoid code duplication.

In the interest of time you will perform the refactoring only in the `training-sklearn.ipynb` notebook.

- ▶ 1. Create and access a new workbench called `practices-engineering-wb` within the `practices-engineering` data science project.
 - 1.1. In a web browser, navigate to the RHOAI dashboard at <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com> and authenticate as the `developer` user by using the `developer` password.
 - 1.2. Navigate to the project details by clicking **Data Science Projects** and then `practices-engineering` project.
 - 1.3. Create a small workbench called `practices-engineering-wb` that uses the `Standard Data Science` image. Wait for the workbench to start.
 - 1.4. Access the workbench by clicking the **Open** button next to it. Use the username `developer` and password `developer` to authenticate to the workbench.
 - 1.5. Accept the default selected permissions by clicking **Allow selected permissions**. The JupyterLab interface displays.
- ▶ 2. Clone the repository and open the `training-sklearn.ipynb` notebook.
 - 2.1. In the JupyterLab interface, click the **Git** icon from the left tools pane.
 - 2.2. Click **Clone a Repository**.

- 2.3. Enter <https://github.com/RedHatTraining/AI26X-apps> as the repository, and click **Clone**.
- 2.4. In the JupyterLab file browser, navigate to the `AI26X-apps/models/practices-engineering` directory and double click the `training-sklearn.ipynb` notebook to open it.
- 2.5. Analyze steps two and three and compare them to the same steps in the `training-pytorch.ipynb` notebook.

You can verify that the code in these steps is doing the same on both notebooks:

- Reading the CSV file and splitting the data into features and the target variable.
- Using the available data to create a training and a testing data set.

Creating an abstraction that takes a CSV path and returns the training and testing data sets would simplify the notebooks and avoid code duplication.

- 3. Analyze the `common/data_preparation.py` module. Verify that the Python file declares a `prepare_training_data` function that can be reused in the exercise notebooks.

```
...output omitted...

def prepare_training_data(
    csv_path: Path, test_size: float = 0.2, random_state: int = 0
) -> List:
    path = Path(csv_path)
    data = pd.read_csv(path.as_posix())

    x = data.drop("Outcome", axis=1)
    y = data["Outcome"]

    return train_test_split(x, y, test_size=test_size, random_state=random_state)
```

- 4. Run the `training-sklearn.ipynb` notebook to verify it works before refactoring the notebook.

- 4.1. Click **Run > Run All Cells** in the menubar.
- 4.2. Wait for the notebook to finish and scroll to the notebook end to verify the following output.

```
['Diabetes', 'No Diabetes']
```

- 5. Refactor the `training-sklearn.ipynb` notebook to use the `prepare_training_data` function. A complete refactor requires that you use the function on the other notebook, but for this exercise you only update the `training-sklearn.ipynb` notebook.

- 5.1. Add the correct imports to the first notebook step to make the Python code cell look like the following:

```
from typing import List, Dict

import pandas as pd

from common import data_preparation as prep
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
```

- 5.2. Replace steps two and three with the following code cell. You might want to remove all the previous cells except the code cell with the imports.

```
split_result = prep.prepare_training_data('./data/diabetes.csv')
X_train, X_test, y_train, y_test = split_result

print(f"Number of samples in training set: {X_train.shape[0]}")
print(f"Number of samples in test set: {X_test.shape[0]}")
```

► 6. Verify that the `training-sklearn.ipynb` works after the refactor.

- 6.1. On the menubar, click **Kernel > Restart Kernel** and **Run All cells**.
- 6.2. Wait for the notebook to finish and scroll to the notebook end to verify the following output.

```
['Diabetes', 'No Diabetes']
```

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish practices-engineering
```

Summary

- Tracking the resources used by a project helps you determine the necessary workbench size, diagnose performance issues, and size production deployments.
- To scale data loading, preprocessing, and training you can apply techniques such as online learning and batched data loading. These techniques split the training process into multiple training rounds that fit in memory.
- Use evaluation metrics to monitor the training process.
- You can visually inspect metrics with TensorBoard. TensorBoard is available in TensorFlow and PyTorch workbenches.
- Applying software engineering principles to ML model creation can help you build and deliver better models.

Chapter 10

Introduction to Model Serving

Goal

Describe the concepts and components required to export, share, and serve trained machine learning models.

Sections

- Concepts and Key Aspects of Serving AI Models (and Quiz)
- Saving Trained Machine Learning Models (and Guided Exercise)
- Serving Models as Stand-Alone Applications (and Guided Exercise)

Concepts and Key Aspects of Serving AI Models

Objectives

- Describe the concepts and terminology used when deploying models.

Model Serving Concepts

Machine learning (ML) model serving refers to the process of deploying, managing, and exposing ML models for inference, enabling them to be used as inference services on unseen data. After training and fine-tuning a ML model on a specific task, serving involves making the model available for real-time or batch predictions. In particular, serving a model from a production environment is a crucial step in the ML lifecycle. Doing so delivers real-world value from the models by enabling integration of models into applications, systems, and services.

Key aspects of ML model serving include:

Deployment

After training, a ML model needs to be deployed to an environment where it is accessible for inference. This can be on-premises, in the cloud, or on edge devices, depending on the application's requirements.

Scalability

Model serving systems must be scalable to handle varying levels of demand. This involves ensuring that the infrastructure can handle increased load and that the serving architecture is designed for efficiency and resource optimization.

Real-time inference

Many applications require real-time predictions, where the model responds to input data quickly. This is common in scenarios such as fraud detection, image recognition, and natural language processing.

Batch Inference

Inference requests might increase latency in a real-time system to a degree that is not practical for a production environment. Bundling many requests in a batch reduces the communication overhead. Also, in some scenarios, a single inference request can be so computationally expensive, that precalculating and caching the results might be the only viable option. This is common in scenarios such as data preprocessing, recommender systems, analytics, and large-scale data transformations.

Monitoring and Logging

The success of a model's predictions might decrease over time because of changes in the environment. You should monitor these predictions to detect drifts and decide whether you need to retrain the model with new input data. Also, just like non-ML applications, it is important to monitor response times and hardware use.

Model serving systems often include monitoring and logging functionalities to track the performance of deployed models.

Versioning

It is important to manage different versions of a model. This eases rollback in case of issues and facilitates comparing versions side-by-side when deploying new models.

Security

Ensuring the security of the deployed models is crucial. This involves controlling access to the models, encrypting communications, and protecting against potential attacks or adversarial inputs.

A secure model also depends on the data that you use for training. Make sure that your data meets privacy and compliance regulations, and carefully inspect potential biases.

Integration

Model serving systems need to integrate seamlessly with other components of the application or system architecture. This often involves using APIs to facilitate communication between the model and the rest of the application. Popular tools and frameworks for model serving include TensorFlow Serving, Flask, FastAPI, Amazon SageMaker, and TensorFlow Extended (TFX), among others. These tools provide solutions for deploying, managing, and scaling ML models in production environments.

Model Serving Runtime

A *model serving runtime* or model server is the execution environment that runs trained ML models to infer predictions. A model server defines standard HTTP or gRPC endpoints, loads the model into memory, handles client requests, performs the inference, and returns the results.

A model serving runtime can be part of a larger deployment platform, such as KServe, which includes features such as scalability, versioning, monitoring, and security. Examples of model serving runtimes include TensorFlow Serving, TorchServe, OpenVINO and ONNX Runtime. These runtimes support the deployment of model formats trained by using popular ML frameworks. Some model servers only work with models from a single training framework, other model servers support multiple frameworks.

For use cases that require a high model-per-pod density, *multi-model servers*, such as ModelMesh, can serve multiple models in a single pod.

By default, Red Hat OpenShift AI includes a preconfigured model serving runtime, OpenVINO, which can load, execute, and expose models trained with TensorFlow and PyTorch. OpenVINO supports various model formats, such as the following examples:

ONNX

An open standard for ML interoperability.

OpenVINO IR

The proprietary model format of OpenVINO, the model serving runtime used in OpenShift AI.



References

TensorFlow Serving

<https://www.tensorflow.org/tfx/guide/serving>

Pytorch Serve

<https://pytorch.org/serve/index.html>

ONNX

<https://onnx.ai/>

ONNX Runtime

<https://onnxruntime.ai/docs>

OpenVINO IR

https://docs.openvino.ai/latest/openvino_ir.html

OpenVINO

<https://docs.openvino.ai>

KServe Project

<https://kserve.github.io/website/latest/>

For more information, refer to the *Configuring Model Servers* section in the *Serving Models* chapter in the Red Hat Red Hat OpenShift AI Self-managed 2.8 *Serving Models* documentation at

https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/serving_models/index#about-model-serving_about-model-serving

► Quiz

Concepts and Key Aspects of Serving AI Models

Choose the correct answers to the following questions:

► 1. What is machine learning (ML) model serving?

- a. The process of training, deploying, managing, and exposing ML models for inference, enabling them to make predictions on unseen data.
- b. The process of deploying, managing, and exposing ML models for inference, enabling them to make predictions on unseen data.
- c. The process of deploying, managing, and exposing ML models for inference, enabling them to make predictions on test data.
- d. The process of serving ML models from a centralized store.

► 2. Which statement is true regarding real-time inference in model serving?

- a. The model prediction times are not relevant.
- b. It requires a message queue to store pending inference requests.
- c. It requires quick response times to ease synchronous communications.

► 3. Which two statements are true regarding batch inference in model serving? (Choose two.)

- a. If your inference requests take one hour each, then you should consider batch inference.
- b. It makes sense if your inference requests take a few milliseconds each.
- c. It is adequate for business use cases where caching inference results is appropriate.
- d. It is great for chat-bot use cases because you get higher inference throughput.

► 4. Which of the following metrics are common to ML and non-ML applications? (Choose two.)

- a. Accuracy
- b. CPU usage
- c. Response latency
- d. Precision

- 5. You are designing a book recommendation system that uses thousands of models that will run on Red Hat OpenShift AI. Each model is specialized on a type of user, segmented by many categories such as country, profession, or age. Which one of the following model serving architectures would you choose?
- a. A single model serving runtime to expose the model for real-time inferencing.
 - b. A multi-model server to expose the model for real-time inferencing.
 - c. A single model serving runtime to expose the model for asynchronous batch inferencing.
 - d. A multi-model server with a weekly job to pre-process and cache recommendations for each user type.
 - e. Many custom lightweight APIs for the different AI frameworks used to create the models.

► Solution

Concepts and Key Aspects of Serving AI Models

Choose the correct answers to the following questions:

► 1. **What is machine learning (ML) model serving?**

- a. The process of training, deploying, managing, and exposing ML models for inference, enabling them to make predictions on unseen data.
- b. The process of deploying, managing, and exposing ML models for inference, enabling them to make predictions on unseen data.
- c. The process of deploying, managing, and exposing ML models for inference, enabling them to make predictions on test data.
- d. The process of serving ML models from a centralized store.

► 2. **Which statement is true regarding real-time inference in model serving?**

- a. The model prediction times are not relevant.
- b. It requires a message queue to store pending inference requests.
- c. It requires quick response times to ease synchronous communications.

► 3. **Which two statements are true regarding batch inference in model serving? (Choose two.)**

- a. If your inference requests take one hour each, then you should consider batch inference.
- b. It makes sense if your inference requests take a few milliseconds each.
- c. It is adequate for business use cases where caching inference results is appropriate.
- d. It is great for chat-bot use cases because you get higher inference throughput.

► 4. **Which of the following metrics are common to ML and non-ML applications? (Choose two.)**

- a. Accuracy
- b. CPU usage
- c. Response latency
- d. Precision

- 5. You are designing a book recommendation system that uses thousands of models that will run on Red Hat OpenShift AI. Each model is specialized on a type of user, segmented by many categories such as country, profession, or age. Which one of the following model serving architectures would you choose?
- a. A single model serving runtime to expose the model for real-time inferencing.
 - b. A multi-model server to expose the model for real-time inferencing.
 - c. A single model serving runtime to expose the model for asynchronous batch inferencing.
 - d. A multi-model server with a weekly job to pre-process and cache recommendations for each user type.
 - e. Many custom lightweight APIs for the different AI frameworks used to create the models.

Saving Trained Machine Learning Models

Objectives

- Save, export, and share machine learning models.

Saving Models

After training a model, you probably want to share and use it in real-word cases. You might want to share the model with your teammates, coworkers, or customers. You might also plan to deploy it as a web API for public use. In any of those cases, the first step is to save or serialize the model to disk.

The way you persist an ML model depends on the ML framework that you use. Each machine learning framework provides its own routines and formats to save and load models.

Saving Models in Scikit-learn

To save models in Scikit-learn, use the `joblib` module. This module is implemented in the Python package of the same name, which is installed as a dependency of Scikit-learn. The `joblib` module offers serialization capabilities that you can use to save Scikit-learn models to the file system. To save a model, call the `joblib.dump` function by passing the model object and the file path where you want to save the model, as follows:

```
import joblib
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()

# ...training omitted...

joblib.dump(model, "my_model.joblib")
```

To load a saved model, use the `joblib.load` function:

```
model = joblib.load("my_model.joblib")
```

Scikit-learn also supports the `pickle` module for model persistence, although the recommendation is to use `joblib`.



Note

The `pickle` and `joblib` Python packages are not ML-specific. You can use them to serialize and deserialize any Python object.

Saving Models in TensorFlow

TensorFlow can save models in different formats:

Keras v3

The newest, simplest, and most efficient format for saving models in TensorFlow. This format saves the model as a file with the `.keras` extension. TensorFlow recommends this format if you use the Keras API. However, this format is not yet supported by the Model Serving feature of Red Hat OpenShift AI (RHOAI).

SavedModel

This format saves the model as a directory that contains the model as *Protobuf* binary, with the `.pb` extension. You can use this format to deploy a model with Model Serving in RHOAI.

HDF5

This format saves the model as a file with the `.h5` extension. This is a legacy format that uses a family of formats called Hierarchical Data Format (HDF). You can use this format for compatibility with earlier libraries, such as TensorFlow 1.

To save a model in the TensorFlow Keras API, use the `save` method of the model instance. Pass the output path as a parameter. The format of this path dictates the format. If the path ends with `.keras` or `.h5` extension, then TensorFlow saves the model in Keras or HDF5 format, respectively. Otherwise, it saves the model in SavedModel format. The following example saves the model in SavedModel format.

```
model.save("my_model/v1/")
```

To load a model saved in these formats, use the `tf.keras.models.load_model` function, as follows:

```
model = tf.keras.models.load_model("my_model/v1/")
```

Checkpoints

In TensorFlow, you can also save partially trained models as artifacts called *checkpoints*. This is a usual practice in deep learning, and helps to prevent the loss of progress during long training sessions, which can span several hours or days. To save checkpoints during training, pass a `tf.keras.callbacks.ModelCheckpoint` callback to the `fit` method of a Keras API model. This callback regularly saves checkpoints in the TensorFlow *Checkpoint* format.

Saving Models in PyTorch

In PyTorch, you can save a model with the `torch.save` method. Pass the model state dictionary as the first parameter. You can get this dictionary with the `state_dict` method of the model instance. As a second parameter, pass the output path:

```
torch.save(model.state_dict(), "path/to/my_model")
```

You can also use this approach for saving checkpoints during training or saving the final model after training.

To load a model from disk, instantiate the model and then use the `torch.load` function and the `load_state_dict` method from the model, as follows:

```
model = MyModel()
model.load_state_dict(torch.load("path/to/my_model"))
```

ONNX

One of the downsides of saving models in different formats is the difficulty to standardize the delivery and deployment of these models across different platforms. Depending on the framework and platform that you use for training, you need a specific implementation for running and loading the model. For example, assume that your team has trained a model with PyTorch. To make a Python application run that model, you need PyTorch installed. Additionally, you must load the model from disk in the PyTorch way. If your team decides to retrain the same model in TensorFlow, then you must change your application so that it can load and run TensorFlow models.

Red Hat OpenShift AI (RHOAI) attempts to ease this problem by supporting a single, standard format to serve models: the *Open Neural Network Exchange (ONNX)* format. ONNX is an open format for interoperability across machine learning systems. As well as the format itself, the ONNX project provides a set of libraries for creating models in ONNX format, and a runtime for running those models. ONNX represents models as graphs of operations, which are called *operators*. Examples of operators are Add, Sub, Transpose, and Relu, among many others.



Note

To run and serve ONNX models, regardless of the library used for training, RHOAI provides the OpenVINO model server, which is covered later in this course.

Typically, instead of building an ONNX model directly, you first train the model by using a common ML library, such as PyTorch, and then you convert the model to ONNX. You can convert models from specific libraries to ONNX with converter libraries, such as the following ones:

- `sklearn-onnx`
- `tensorflow-onnx`
- `torch.onnx`

From Scikit-learn to ONNX

To export a model in Scikit-learn, you need the `sklearn-onnx` library. This library provides the `skl2onnx` module, which you can use as follows:

```
from skl2onnx import to_onnx

X, y = load_and_preprocess_data()

model = KMeans()
...training omitted...

onnx_model = to_onnx(❶
    model, ❷
    X[:1] ❸
)

with open("my_model.onnx", "wb") as f:
    f.write(onnx_model.SerializeToString()) ❹
```

❶ The `skl2onnx.to_onnx` function converts the Scikit-learn model to ONNX.

❷ The Scikit-learn model instance.

- ③ A NumPy array that contains the input features. The converter uses this array to infer the input types. This example passes only the first row, because a single row is enough to determine the input types.
- ④ Serialize the model and write the results to a file.

**Note**

The Standard Data Science workbench includes the `sklearn-onnx` library. Note that the package name of this library is `skl2onnx`.

From TensorFlow to ONNX

To export a TensorFlow model to ONNX, you need the `tensorflow-onnx` library. This library provides the `tf2onnx` module, which you can use to convert a model ONNX, as follows:

```
import onnx
from tf2onnx import convert
from tensorflow.keras.applications.resnet import ResNet50

model = ResNet50()
# ...training omitted...

onnx_model, _ = convert.from_keras(model)
onnx.save(onnx_model, "my_model.onnx")
```

**Note**

The TensorFlow workbench includes the `tensorflow-onnx` library. Note that the package name of this library is `tf2onnx`.

From PyTorch to ONNX

PyTorch includes an ONNX exporter in the `torch.onnx` module. The `onnx` dependency is also required for this module work, although the PyTorch workbench includes it by default. You can export a PyTorch model as follows:

```
import torch

my_model = MyModel()

# ...training omitted...

input_example = torch.tensor([[1, 2, 3], [4, 5, 6]]) ❶
torch.onnx.export(model, input_example, "my_model.onnx") ❷
```

- ❶ Generate a tensor that represents input data. Usually, you can just use a single case extracted from your data set.
- ❷ Export the model. PyTorch infers the type and shape of the input tensor by using the data provided.

**Important**

At the time of writing this section, RHOAI offers a PyTorch 2.0 workbench. In more recent versions of PyTorch, such as 2.3, the recommended export mechanism uses the TorchDynamo JIT compiler, which is as follows:

```
onnx_model = torch.onnx.dynamo_export(my_model, inputs)
onnx_model.save("my_model.onnx")
```

Running Inference with ONNX Models

An initial, simple approach to running inference in ONNX is using `onnxruntime` Python package. The ONNX runtime includes implementations of all the ONNX operators. This runtime is a recommended option when, after exporting your model to ONNX, you want to test that the ONNX model works as expected. You can use the runtime as follows:

```
import onnxruntime

session = onnxruntime.InferenceSession("my_model.onnx") ❶

X = read_input_data() ❷
inputs = { "X": X.astype(np.float32) } ❸
output_names = ["y"] ❹

results = session.run(output_names, inputs) ❺
```

- ❶ Create an inference session by passing the path of the ONNX model file.
- ❷ Read and preprocess the input data that you want to pass to the model.
- ❸ Create a dictionary containing the input data. You might need to cast the data to the right type.
- ❹ Define the output names.
- ❺ Run the inference session by passing the input and output names.

**Note**

RHOAI workbenches do not include the `onnxruntime` by default.

The ONNX runtime is not the only runtime capable of running ONNX models. Model serving platforms, such as OpenVINO, offer support for ONNX models. In the case of RHOAI, OpenVINO is the default option to serve ONNX models.



Important

RHOAI ships with OpenVINO 2023.3, which can read the ONNX format with limitations. Specifically, OpenVINO 2023.3 is compatible with versions up to 18 of the ONNX operators set. However, this compatibility is limited, because OpenVINO does not implement certain ONNX operators that are not used in deep learning. This limitation prevents OpenVINO from serving some models trained with Scikit-learn.

The list of supported operations is generally enough to support ONNX models that have been trained with TensorFlow or PyTorch. For more information, refer to the list of supported ONNX operations in OpenVINO.

Additional Formats

In RHOAI, Model Serving also supports these formats:

OpenVINO IR

This format is an intermediate representation (IR) that OpenVINO uses internally to manage models.

Caikit

Caikit is an open source toolkit for developers to manage and run ML models. RHOAI combines Caikit with a *Text Generation Inference Server (TGIS)* to support serving text generation and conversational models. If you plan to use Caikit, then you must convert your model to Caikit format first.



Important

By default, Model Serving does not support all the formats covered in this lecture. For example, OpenVINO does not support the Scikit-learn Pickle and Joblib formats. Most ONNX models converted from Scikit-learn are not supported either, because they use operations that OpenVINO does not implement.

To use any other unsupported format you need to define a *custom model server*, as explained later in this course. For more information about supported formats, refer to the *Using Model Servers to Deploy Models* section.

Sharing the Model

After saving the model, you might want to distribute the model file or files in several ways. You can do this in different ways. For example:

- By uploading the model to S3
- By pushing the model to a Git repository
- By embedding the model in a container image

Including the model in a Git repository is usually discouraged, except for tiny models. Models can be large and their size can degrade the performance of the repository.

Copying the model to a different location or embedding it in a container image requires you to also take care of deploying that model. You must also create an interface to access the model, such as a CLI, or an HTTP API.

Chapter 10 | Introduction to Model Serving

The recommended option is to use a S3 bucket as a model registry to store the model. In RHOAI, the Model Serving feature can download models from any S3-compatible API. Model Serving uses a data connection to get the S3 connection parameters. Next, Model Serving automatically downloads, executes, and exposes the model via HTTP and gRPC interfaces.

To upload a model to S3, use the boto3 library, as the following example demonstrates:

```
import os
import boto3

# Read the data connection variables
key_id = os.getenv("AWS_ACCESS_KEY_ID")
secret_key = os.getenv("AWS_SECRET_ACCESS_KEY")
endpoint = os.getenv("AWS_S3_ENDPOINT")
bucket_name = os.getenv("AWS_S3_BUCKET")

# Upload the model
model_path = "path/to/model.onnx"
s3_client = boto3.client(
    "s3",
    aws_access_key_id=key_id,
    aws_secret_access_key=secret_key,
    endpoint_url=endpoint,
    use_ssl=True
)
s3_client.upload_file(model_path, bucket_name, Key=onnx_filename)
```

**Note**

The boto3 library is preinstalled in the Standard Data Science, PyTorch and TensorFlow workbenches.



References

Scikit-learn: Model persistence

https://scikit-learn.org/1.4/model_persistence.html

TensorFlow: Save and load models

https://www.tensorflow.org/tutorials/keras/save_and_load

TensorFlow: Training checkpoints

<https://www.tensorflow.org/guide/checkpoint>

PyTorch: Saving and Loading Models

https://pytorch.org/tutorials/beginner/saving_loading_models.html

Introduction to ONNX

<https://onnx.ai/onnx/intro/index.html>

OpenVINO: ONNX Support Coverage

<https://onnxruntime.ai/docs/execution-providers/OpenVINO-ExecutionProvider.html#support-coverage>

ONNX Operators

<https://onnx.ai/onnx/operators/index.html#l-onnx-operators>

OpenVINO: Supported Operations

https://docs.openvino.ai/2023.3/openvino_resources_supported_operations_frontend.html

Getting Started Converting TensorFlow to ONNX

<https://onnxruntime.ai/docs/tutorials/tf-get-started.html>

PyTorch 2.0: torch.onnx module

<https://pytorch.org/docs/2.0/onnx.html?highlight=onnx#module-torch.onnx>

Sklearn-onnx

<https://github.com/onnx/sklearn-onnx>

Tensorflow-onnx

<https://github.com/onnx/tensorflow-onnx>

Caikit: Put the power of AI in your apps without being an AI expert

<https://developer.ibm.com/articles/awb-power-of-ai-caikit-huggingface/>

Converting Hugging Face Hub models to Caikit Format

<https://github.com/opendatahub-io/caikit-tgis-serving/blob/main/demo/kserve/built-tip.md#converting-hugging-face-hub-models-to-caikit-format>

OpenVINO IR format

https://docs.openvino.ai/2023.3/openvino_ir.html

OpenVINO supported model formats

https://docs.openvino.ai/2023.3/openvino_docs_model_processing_introduction.html

Boto3: Uploading files

<https://boto3.amazonaws.com/v1/documentation/api/latest/guide/s3-uploading-files.html>

► Guided Exercise

Saving Trained Machine Learning Models

Save, export, and share machine learning models.

Outcomes

- Save a Scikit-learn model in Open Neural Network Exchange (ONNX) format.
- Upload the model to S3.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI265 && lab start serving-saving
```

Instructions

This exercise provides you with an S3 bucket and a data connection.

The bucket is called `saved-models` and is available at `http://minio-minio.apps.ocp4.example.com`. You can use the `minio` access key and the `minio123` secret key to log in.

► 1. Verify the data connection.

1. In a new browser tab, navigate to `https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com`.
2. Click **Log in with OpenShift**, and then click `htpasswd_provider`.
3. Authenticate as the `developer` user by using the `developer` password.
4. Click **Data Science Projects** and then click the `serving-saving` project.
5. Scroll down to the **Data connections** section and verify that the `internal-minio` data connection exists.

► 2. Create and open a workbench associated with the data connection.

- 2.1. In the `serving-saving` project page, click **Create workbench**.
- 2.2. Create the workbench with the following values:

Field	Value
Name	serving-saving-wb
Image selection	Standard Data Science
Container size	Small
Data connection	Click Use a data connection, then Use existing data connection, and select internal-minio.

- 2.3. Click **Create workbench**. The workbench might take a few minutes to start.
- 2.4. Access the workbench by clicking the **Open** button next to it. Click **httpasswd_provider**, and then authenticate by using **developer** as the username and the password.
- 2.5. Accept the default selected permissions by clicking **Allow selected permissions**. The JupyterLab interface displays.

▶ 3. Clone the repository and open the `model-upload.ipynb` notebook.

 - 3.1. In the JupyterLab interface, click the **Git** icon from the left tools pane.
 - 3.2. Click **Clone a Repository**.
 - 3.3. Enter <https://github.com/RedHatTraining/AI26X-apps> as the repository, and click **Clone**.
 - 3.4. In the JupyterLab file browser, navigate to the `AI26X-apps/deploying/serving-saving` directory and double click the `model-upload.ipynb` notebook to open it.

▶ 4. Follow the instructions within the `model-upload.ipynb` notebook.

Finish

On the **workstation** machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish serving-saving
```

Serving Models as Stand-Alone Applications

Objectives

- Create and deploy a Python application to serve a machine learning model.

Exposing Models with API Frameworks

To make your model publicly accessible, you can use a popular framework to create an API that serves inference requests. This is a traditional and popular approach, but it requires development and operational skills and effort.

First, you must choose the programming language for the API. This is typically dictated by the ML model framework. Most popular ML frameworks use Python. This reduces the framework choice to popular Python API frameworks such as Flask or FastAPI.

You can also choose the technology you use for serving inference requests. For example, you can use JSON over HTTP, which is widely adopted, or a technology such as gRPC, which performs better.

After you write your API, you can create a container to package and distribute your model API. Deploying a model is a workflow similar to deploying applications, and has some similarities with microservices architectures. If you use a container platform such as Kubernetes or OpenShift, then you can scale this approach and use the container platform to address cross-cutting concerns such as:

- Security: you can implement authentication and authorization.
- Monitoring: you can use your cluster monitoring capabilities.
- Hardware resource management.

Writing a Simple Inference Endpoint with FastAPI

FastAPI integrates Python type hints and the Pydantic Python library in a way that simplifies writing APIs.

The following `main.py` Python script creates an API that defines a GET endpoint, and returns a fixed string:

```
import uvicorn
from fastapi import FastAPI

# Create the application object
app = FastAPI()

# Define the `/hi` GET endpoint and return a fixed string
@app.get("/hi")
def greet():
    return "Hello Red Hat!"
```

```
# Run the script in iterative development mode
if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8000, reload=True)
```

To start the previous script, you can install the dependencies and run the following command:

```
[user@host ~]$ python main.py
...output omitted...
```

FastAPI uses JSON as the default format, and automatically converts JSON between the Python request and response objects.

The following code sample creates an /is-cat inference endpoint to predict whether there is a cat in the image:

```
from pydantic import BaseModel

# Define the request object
class CatImage(BaseModel):
    file: str
    image: bytes

# Define the `/is-cat` POST endpoint and return the saved CatImage
@app.post("/is-cat")
def predict(cat_image: CatImage) -> bool:
    prediction = predict(cat_image)
    return is_cat(prediction)
```

To test the /is-cat endpoint you can use a curl command like the following:

```
[user@host ~]$ curl -X POST \
-H "Content-Type: application/json" \
http://localhost:8000/isCatImage \
-d '{"file": "my_cat.png", "image": "@./images/my_cat.png"}'
{"cat": true, "breed": "persian"}
```

Advanced Model Serving with Custom APIs

Implementing an inference endpoint might be more involved for more complex use cases.

The following code is a high-level implementation of the process, which divides the serving process into pre-processing, prediction, and post-processing:

```
@app.post("/predict")
def predict(req: PredictRequest):

    data = pre_process(req)

    prediction = predict(data)

    return post_process(prediction, req)
```

The **pre-process** part should prepare the data in the format that the model expects. The inference input data that the **pre-process** phase generates should be the same as the input data generated in the training process. These transformations perform the following tasks:

- Transforming the request object, such as an image or text, to a vectorized representation for inference.
- Cleaning data, for example by removing empty or invalid cases.
- Generating or retrieving pre-calculated features for the inference request.

The **prediction** phase uses a model to make predictions.

The **post-process** phase translates the output inference vector to a usable format for the client requesting the prediction. In the case of an LLM, the **post-process** phase might translate an output vector to human-readable text. In a classification model, the **post-process** phase maps the output vector to the category with the highest probability.

Custom API Limitations

Using custom APIs has disadvantages such as the following:

- You must manually provide and maintain the implementation around a specific model format and inference engine. If you want to experiment with different models and inference engines, then you must provide these implementations.
- Serving inference requests that use several models requires extra custom code to implement model interaction.
- The preprocessing, prediction, and postprocessing phases are coupled in the same software package. A change in the `pre_process` method forces you to redeploy a model that has not changed. Also, you cannot independently scale the `predict` phase which might underutilize your ML-specific hardware resources.
- Your model is always loaded even if it is not being used.

Model servers address the previous issues in a standard and documented way. For example, KServe lets you use different pods for the prediction phase and for preprocessing and postprocessing. Model servers are addressed in the following chapters.

► Guided Exercise

Serving Models as Stand-alone Applications

Serve a model by implementing a FastAPI Python API.

Outcomes

- Deserialize a model from a file.
- Create an API to serve inference requests.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI265 && lab start serving-apps
```

Instructions

In this exercise you will implement the python code to deserialize the `sklearn_diabetes_model.joblib` binary model file and expose it via a FastAPI Python application.

► 1. Examine the `~/AI265/serving-apps` project and install the requirements.

- 1.1. Change to the exercise directory.

```
[student@workstation ~]$ cd ~/AI265/serving-apps
```

- 1.2. Observe that the `serving-apps` directory contains the `sklearn_diabetes_model.joblib` file along with other files that support the Python project.

```
[student@workstation serving-apps]$ tree .
tree .

.
├── requirements.txt
├── sklearn_diabetes_model.joblib
└── solution
    └── diabetes_api.py
└── src
    ├── diabetes_api.py
    └── dto
        ├── __init__.py
        └── patient_dto.py
└── test
```

```
└── has_diabetes.json  
└── no_diabetes.json  
  
4 directories, 8 files
```

- 1.3. Create and activate a Python virtual environment.

```
[student@workstation serving-apps]$ python -m venv .venv \  
&& source .venv/bin/activate  
(.venv) [student@workstation serving-apps]$
```

- 1.4. Install the project requirements.

```
(.venv) [student@workstation serving-apps]$ pip install -r requirements.txt  
...output omitted...
```

► 2. Deserialize the `sklearn_diabetes_model.joblib` file.

- 2.1. Open the `src/diabetes_api.py` file with your editor of choice, and add the following code to implement the model deserialization.

```
...output omitted...  
app = FastAPI()  
  
# Load the model from a file  
model = load('sklearn_diabetes_model.joblib')  
  
...output omitted...
```

- 2.2. Verify that the API `diabetes_api.py` starts with no errors after defining the `model` variable.

```
(.venv) [student@workstation serving-apps]$ python src/diabetes_api.py  
...output omitted...  
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)  
...output omitted...  
INFO:     Started server process [5183]  
INFO:     Waiting for application startup.  
INFO:     Application startup complete.
```

The script starts a server application that expects requests on port 8000. Leave the terminal open for the rest of the exercise.

The `diabetes_api.py` script is configured for fast iteration development by reloading the application on file updates.

► 3. Verify that the `/patient/diagnose` inference endpoint responds but returns a fixed string.

- 3.1. Open another terminal and change to the exercise directory.

```
[student@workstation ~]$ cd ~/AI265/serving-apps
```

- 3.2. Use the following curl command to test the inference endpoint.

```
[student@workstation serving-apps]$ curl -X POST \
-H "Content-Type: application/json" \
http://localhost:8000/patient/diagnose \
-d @test/no_diabetes.json; echo
"TODO: implement prediction"
```

The endpoint works, but the implementation returns a fixed string.

- 4. Edit the `src/diabetes_api.py` script to add the prediction to the inference endpoint.

- 4.1. Implement the `predict` function.

```
...output omitted...
classes = ('No diabetes', 'Diabetes')

def predict(patients: Patient):
    inputs = pd.DataFrame([patients.dict()])
    return classes[model.predict(inputs)[0]]

...output omitted...
```

- 4.2. Return the `predict` function output as the response for the inference endpoint.

```
...output omitted...

@app.post("/patient/diagnose")
def diagnose_diabetes(patient: Patient):
    try:
        # Return the model prediction
        return predict(patient)
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

...output omitted...
```

- 4.3. Optionally, inspect the contents of the test files.

```
[student@workstation serving-apps]$ tail -n+1 ./test/*
==> ./test/has_diabetes.json <==
{
    "Pregnancies": 6.0,
    "Glucose": 110.0,
    "BloodPressure": 65.0,
    "SkinThickness": 15.0,
    "Insulin": 1.0,
    "BMI": 45.7,
    "DiabetesPedigreeFunction": 0.627,
    "Age": 50.0
}
```

```
==> ./test/no_diabetes.json <==  
{  
    "Pregnancies": 0.0,  
    "Glucose": 88.0,  
    "BloodPressure": 60.0,  
    "SkinThickness": 35.0,  
    "Insulin": 1.0,  
    "BMI": 45.7,  
    "DiabetesPedigreeFunction": 0.27,  
    "Age": 20.0  
}
```

4.4. Verify that the inference endpoint works by running the `curl` command.

Verify the response by using the `no_diabetes.json` file.

```
[student@workstation serving-apps]$ curl -X POST \  
-H "Content-Type: application/json" \  
http://localhost:8000/patient/diagnose \  
-d @test/no_diabetes.json; echo  
"No diabetes"
```

Verify the response by using the `has_diabetes.json` file.

```
[student@workstation serving-apps]$ curl -X POST \  
-H "Content-Type: application/json" \  
http://localhost:8000/patient/diagnose \  
-d @test/has_diabetes.json; echo  
"Diabetes"
```

► 5. Exit the application by pressing `Ctrl+C` in the terminal window.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish serving-apps
```

Summary

- Machine learning (ML) model serving refers to the process of deploying, managing, and exposing ML models for inference.
- A model server defines standard HTTP or gRPC endpoints, loads the model into memory, handles client requests, performs the inference, and returns the results.
- By default, Red Hat OpenShift AI includes a preconfigured model serving runtime, OpenVINO, which can load, execute, and expose models trained with TensorFlow and PyTorch.
- The way you persist an ML model depends on the ML framework that you use. Each machine learning framework provides its own routines and formats to save and load models.
- You can implement custom APIs to serve your ML models, but using a model server provides standard out-of-the-box solutions for common model serving tasks.

Chapter 11

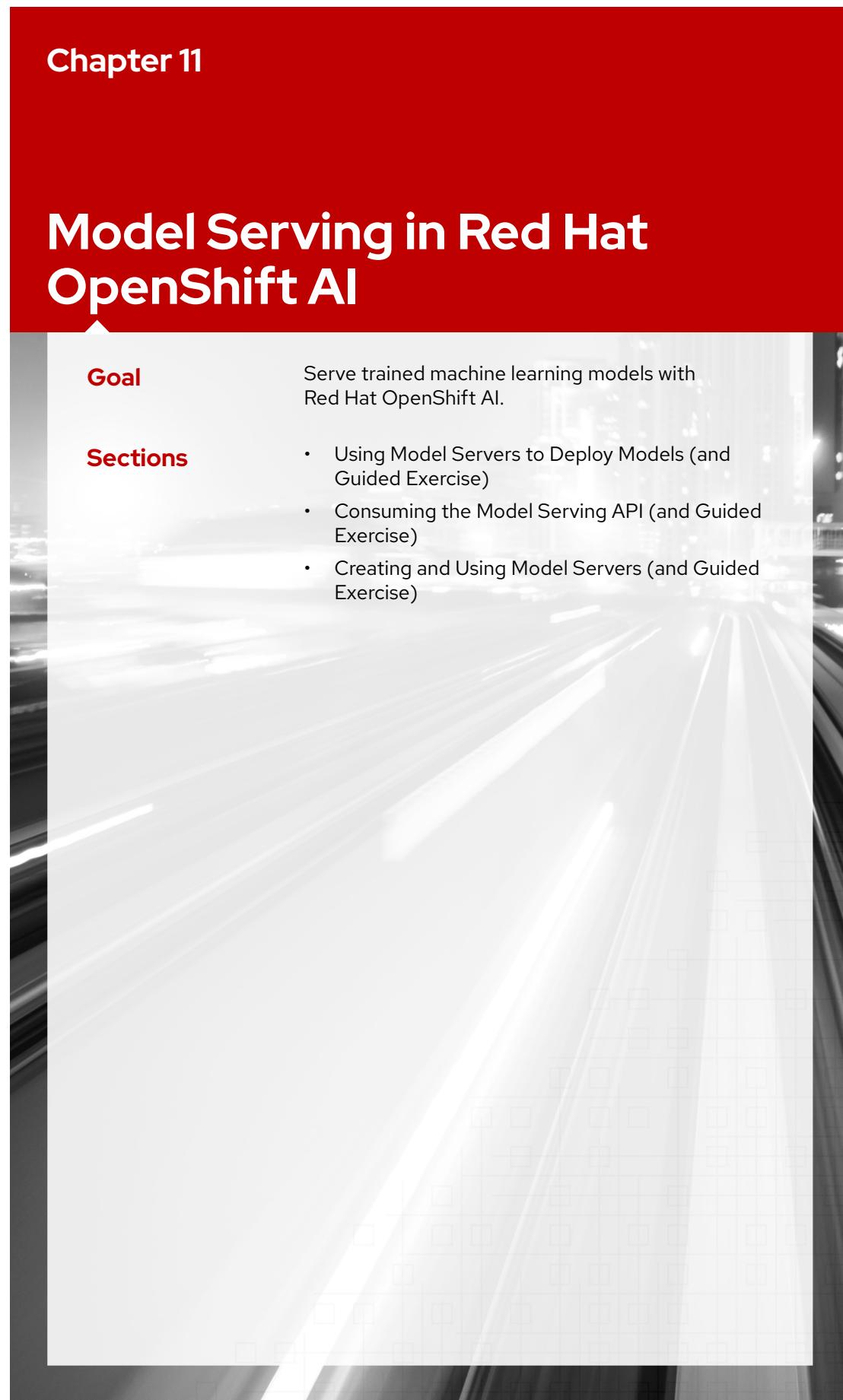
Model Serving in Red Hat OpenShift AI

Goal

Serve trained machine learning models with Red Hat OpenShift AI.

Sections

- Using Model Servers to Deploy Models (and Guided Exercise)
- Consuming the Model Serving API (and Guided Exercise)
- Creating and Using Model Servers (and Guided Exercise)



Using Model Servers to Deploy Models

Objectives

- Deploy machine learning models by using OpenShift AI Model Servers.

Serving Models with Red Hat OpenShift AI

Red Hat OpenShift AI (RHOAI) simplifies the tasks that expose an AI/ML model for internal or external consumption by using the infrastructure on Red Hat OpenShift. You can expose the models trained in a data science project by using one of the two serving platforms that RHOAI provides. After serializing and exporting the model to a supported format, you must choose the model serving runtime for inference.

RHOAI uses KServe and ModelMesh as the model serving or model inferencing platforms, which support multiple model serving runtimes. Also, both model serving platforms support custom serving runtimes, which you can add for models created with incompatible model frameworks.

RHOAI Custom Resources for Model Serving

RHOAI defines two OpenShift custom resource definitions (CRD) to implement the serving runtime.

`serving.kserve.io.ServingRuntime`

The RHOAI dashboard calls `ServingRuntime` instances *Model Servers*. A Model Server is the execution environment or platform where a model runs to make predictions. It is responsible for loading the model into memory, handling client requests, inferencing, and returning the results.

The `ServingRuntime` object defines the needed templates for pods that serve one or more model formats.

`serving.kserve.io.InferenceService`

The `InferenceService` object specifies the model location and format, and binds the model with the model server. The `InferenceServices` define the different HTTP or gRPC endpoints for the exposed model.

RHOAI uses these objects to create an inference API common to different AI/ML frameworks. For more information about the two versions of the RHOAI inference API, see the KServe upstream documentation in the references section.

Furthermore, the RHOAI model serving platforms provide the `InferenceGraph` custom resource. An `InferenceGraph` enables you to create more complex inference services by using more than one model to generate a result. For more information about inference graphs, see the KServe upstream documentation in the references section.

Single-model versus Multi-model Serving Platforms

RHOAI uses the `InferenceService` and `ServingRuntime` OpenShift objects to implement two different serving platforms: the single-model and the multi-model platform.

Chapter 11 | Model Serving in Red Hat OpenShift AI

The open source community project KServe, and its Model Serving component, implements the *single-model serving platform* on RHOAI. This serving platform uses objects from the Red Hat OpenShift Serverless and Red Hat OpenShift Service Mesh operators. Therefore, the single model serving platform provides serverless advantages, such as scale pods to, and from zero.

The ModelMesh component of KServe implements the *multi-model serving platform* on RHOAI. This serving platform uses only objects from OpenShift, or the RHOAI operator. Therefore, it does not depend on any other OpenShift operator.

Both serving platforms pull models from S3 buckets, where the models are stored. When you deploy a model to a model server, RHOAI creates the inference endpoints on the data science projects to allow applications to send inference requests. Both serving platforms inject sidecar containers in each model server pod. However, the OpenShift resources and objects are different in the two serving platforms.

In the single-model platform, RHOAI serves each model by using a pod with the following containers:

- `kserve-container`: container image that runs the serving runtime.
- `queue-proxy`: container image that runs the serverless features.
- `istio-proxy`: container image that runs the HTTP services under Red Hat OpenShift Service Mesh control.

If you deploy another model by using the single-model platform, then RHOAI creates another pod with another three containers.

In the multi-model platform, RHOAI serves all models by using a pod with the following containers:

- `rest-proxy`: runs an HTTP proxy to redirect inference requests to the gRPC endpoint.
- `oauth-proxy`: runs an OAuth proxy to integrate the model access with the OpenShift authentication system.
- `ovms`: runs an OpenVINO model server.
- `ovms-adapter`: intermediates between KServe and the OpenVINO model server, and loads models from the S3 buckets.
- `mm`: orchestrates model placement and routes the gRPC inference requests.

When you deploy another model by using the multi-model platform, the same pod can serve the new model.

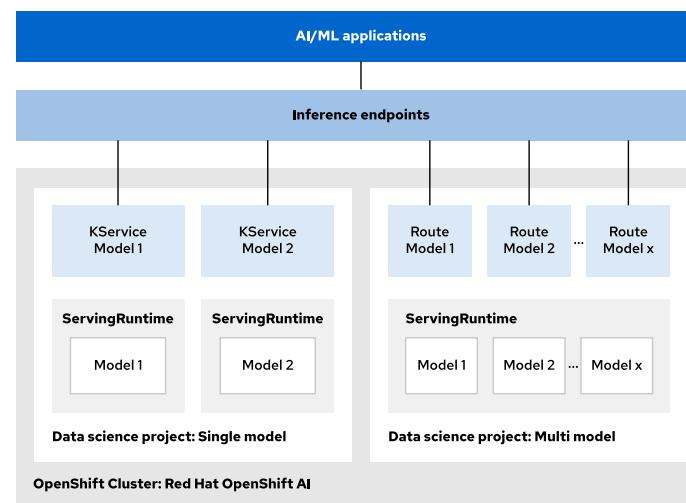


Figure 11.1: Serving platform single and multi-model modes

Chapter 11 | Model Serving in Red Hat OpenShift AI

As shown in the preceding diagram, the main difference between these serving platforms is that the single model platform uses one model server for each AI/ML model. The multi-model, or model-mesh platform can serve more than one model from the same model server.

In the single-model serving platform, the `InferenceService` object injects the `istio-proxy` and `queue-proxy` sidecar containers within the model server pod. These sidecar containers add about 0.5 CPU and 0.5G memory resources per replica. This implies that the multi-model mode can make better use of the cluster resources when you have an increasing number of deployed models.

The following list shows other differences between the single model and multi-model serving runtimes:

- In the single model platform, RHOAI uses one Kubernetes `Deployment` object, and one Red Hat OpenShift Serverless `KService` object for each model server. The `KService` object provides the routing functionality to the inference service, and the `Deployment` object contains the serving runtime for one model. In the multi-model platform there is only one `Deployment` object that contains many models, and many OpenShift `Route` objects that route to the inference service of each model.
- Authentication is only supported in multi-model serving mode. The inference endpoints are not authenticated in RHOAI 2.8.



Note

Future versions of RHOAI will include authentication for single serving mode. See the references section for more information about authentication in the single serving platform.

The choice of serving platform runtime depends on factors such as the machine learning framework used, deployment requirements, and the specific optimizations needed for the target hardware.

Choosing a Model Server Runtime

RHOAI provides four preinstalled serving runtimes that can act as model servers, as the following screen capture shows. If these servers do not fit your needs, then you can add your own custom serving runtimes.

The screenshot shows the 'Serving runtimes' section of the RHOAI administrator interface. On the left, a sidebar menu includes 'Applications', 'Data Science Projects', 'Data Science Pipelines', 'Model Serving' (which is selected), 'Resources', 'Settings' (with 'Notebook images', 'Cluster settings', 'Accelerator profiles', 'Serving runtimes', and 'User management' listed), and 'User management'. The main content area has a header with 'Add serving runtime' and columns for 'Name', 'Enabled', 'Serving platforms supported', and 'API protocol'. There are four entries:

Name	Enabled	Serving platforms supported	API protocol
Calikit TGIS ServingRuntime for KServe Pre-installed	Enabled	Single-model	REST
OpenVINO Model Server Pre-installed	Enabled	Single-model	REST
OpenVINO Model Server Pre-installed	Enabled	Multi-model	REST
TGIS Standalone ServingRuntime for KServe Pre-installed	Enabled	Single-model	gRPC

Figure 11.2: Pre-installed RHOAI serving runtimes in the administrator view of RHOAI dashboard

The definition of the pre-installed runtimes is in four OpenShift templates:

NAME	DESCRIPTION	PARAMETERS
OBJECTS		
caikit-tgis-serving-template	Caikit is an AI toolkit... OpenVino Model Serving Definition	0 (all set) 1 0 (all set) 1
kserve-ovms	OpenVino Model Serving Definition	0 (all set) 1
ovms	Text Generation Inference Server...	0 (all set)
tgis-grpc-serving-template		
1		

**Note**

If you need to modify the default parameters in a template, then you must instantiate the model server template as an OpenShift template. For more details, refer to https://access.redhat.com/documentation/en-us/openshift_container_platform/4.14/html-single/images/index#using-templates

OpenVINO Model Server

RHOAI 2.8 includes OpenVINO 2023.3 for serving regular neural networks and deep learning models. In a typical use case, you train a model with PyTorch and TensorFlow for predictive AI, export the model to ONNX, and then serve the model with OpenVINO. You can use the OpenVINO model server both in single, and multi-model serving platforms.

**Note**

OpenVINO does not implement the full set of ONNX operations required to support Scikit-learn models. However, OpenVINO still can serve some basic ONNX models trained with Scikit-learn, such as KMeans.

In RHOAI 2.8 multi-model serving, OpenVINO can serve the following saved model formats:

- OpenVINO IR
- ONNX
- TensorFlow SavedModel format. The .pb file is needed.
- TensorFlow Lite (.tflite)

In RHOAI 2.8 single-model serving, OpenVINO can serve the following saved model formats:

- OpenVINO IR
- ONNX
- TensorFlow 1 and 2
- Paddle
- PyTorch

TGIS

This option is intended for generative AI, specifically text generation based on Large Language Models (LLMs). The Text Generation Inference (TGI) is a toolkit to deploy and serve LLMs. RHOAI 2.8 provides two single runtime servers that act as TGI servers:

Caikit TGIS serving runtime for KServe

TGI server specialized in serving the Caikit format.

TGIS Standalone serving runtime for KServe

Fork from the HuggingFace TGI server repository. It can serve models in PyTorch format.

See the references section for more information about model formats in KServe.

Deploying a Model by Using the RHOAI Dashboard

If you have a model saved in any supported format in an S3 bucket, then you can create a model server to serve it. To deploy a model, navigate to a data science project in the **Data Science Projects** page, and then scroll down to the **Models and model servers** section.

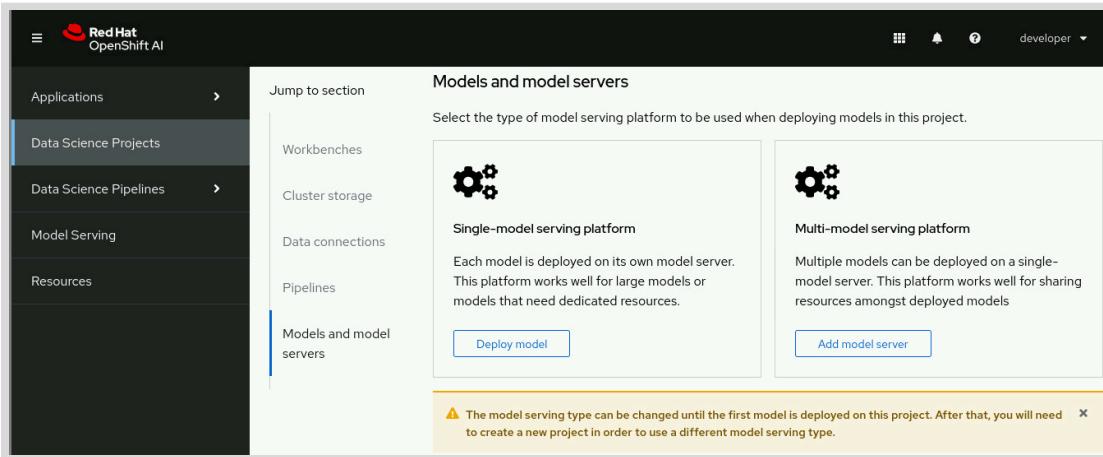


Figure 11.3: Model server section in the data science project page



Note

When you create a model server in a data science project for the first time, you choose single mode or multi-model platform. You cannot change the model server platform for that project afterwards.

Deploying a Model on a Single Server

Click **Deploy model**. In the modal window that opens, you can enter the following fields:

- The model name
- One of the three preinstalled single model serving runtimes
- The model framework or format to serve
- The model server replicas, or number of pods to serve the model
- The model server size with enough resources to serve the model
- The model location in the S3 bucket, with a path to a saved model through a data connection

Fill all the mandatory fields, and click **Deploy**.

Model name	Serving runtime	Inference endpoint	API protocol	Status
flan-t5-small	TGIS Standalone ServingRuntime for KServe	https://flan-t5-small-single-model-runtime.svc.cluster.local:8080	gRPC	✓
	Framework	pytorch		
	Model server replicas	1		
	Model server size	Small 1 CPUs, 4Gi Memory requested 2 CPUs, 8Gi Memory limit		
	Accelerator	No accelerator enabled		

Figure 11.4: Single model server running a TGI model

Deploying a Model on a Multi-model Server

Click Add model server. In the modal window that opens, you can enter the following fields:

- The model server name
- The only preinstalled multi-model serving runtime, OpenVINO
- The model server replicas, or number of pods to serve the model
- The model server size with resources enough to serve the model
- Whether RHOAI should create an OpenShift external route for each deployed model
- Whether the model server access needs to use a token for authorization, and the service account that owns the generated tokens

Fill all the mandatory fields, and click Add.

Model Server Name	Serving Runtime	Deployed models	Tokens
mm-server	OpenVINO Mod...	0	Tokens disabled

Figure 11.5: Multi-model server without deployed models

With the multi-model server created, you can deploy a model by clicking Deploy model. In the modal window that opens, you can enter the following fields:

- The model name
- One of the five preinstalled model formats for OpenVINO
- The model location in the S3 bucket, with a path to a saved model through a data connection

Fill all the fields, and click Deploy

**Note**

Both in single and multi-model servers, the **Path** field accepts either a file or a directory path. If you specify a directory, then make sure that the directory contains only one ONNX file. For example, if your only model in the S3 bucket is stored at `model/iris.onnx`, then you can enter `model` in the **Path** field.

The best practice, however, is to be specific and use the file path when possible. In this way, you ensure that you are selecting the correct model.

The screenshot shows the 'Models and model servers' interface. At the top, there is a button labeled 'Add model server' and a status indicator 'Multi-model serving enabled'. Below this, a table lists a single model server entry:

Model Server Name	Serving Runtime	Deployed models	Tokens	Actions
many-models	OpenVINO Mod...	2	1	<button>Deploy model</button> ...

Below the server entry, another table displays the deployed models:

Model name	Inference endpoint	API protocol	Status	Actions
diabetes	https://diabetes-multimodel-runtime...	REST	✓	...
flan-t5-small	https://flan-t5-small-multimodel-runt...	REST	✓	...

Figure 11.6: Multi-model server running two models



References

KServe Inference API for Predicting and Inferencing

https://kserve.github.io/website/latest/modelserving/data_plane/data_plane/#data-plane-v1-v2

KServe Documentation: Inference Graph

https://kserve.github.io/website/latest/modelserving/inference_graph/

KServe Documentation: The Model Deployment Scalability Problem

<https://kserve.github.io/website/latest/modelserving/mms/multi-model-serving>

KServe Issues Track: Authorino Integration in Model Serving

<https://issues.redhat.com/browse/RHOAIENG-1085>

KServe Documentation: Supported Model Formats in Multi-model Serving

<https://github.com/kserve/modelmesh-serving/tree/main/docs/model-formats>

KServe Documentation: Model Serving (Single) Runtimes Documentation

https://kserve.github.io/website/latest/modelserving/v1beta1/serving_runtime/#model-serving-runtimes

OpenVINO Documentation: OpenVINO IR Format

https://docs.openvino.ai/2023.3/openvino_ir.html

ONNX Format

<https://onnx.ai/onnx/intro/concepts.html>

TensorFlow Formats

https://www.tensorflow.org/tutorials/keras/save_and_load#save_the_entire_model

Upstream Documentation: Text Generation Inference Server in OpenDataHub

<https://github.com/opendatahub-io/text-generation-inference>

► Guided Exercise

Using Model Servers to Deploy Models

Deploy machine learning models by using OpenShift AI Model Servers.

Outcomes

- Deploy AI/ML models to Red Hat OpenShift AI (RHOAI) by using single model servers.
- Deploy AI/ML models to RHOAI by using a multi-model server.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI265 && lab start rhoaiserving-using
```

Instructions

In this exercise you expose inference endpoints for different model formats by deploying models in single-model and multi-model serving modes. First, you deploy the Flan-T5 large language model (LLM) to a single-model server. Then, you deploy the `diabetes-from-tensorflow-keras.onnx` model in both single multi-model modes. Finally, you deploy the `iris` model to a multi-model server.

- 1. Deploy the Flan-T5 model in the `serving-singlemodel` project. The `flan-t5-small` model, in PyTorch H5 format, is present at the `models` directory in the `models-bucket` Minio S3 bucket. You can see the files by accessing the Minio console at <https://minio-minio.apps.ocp4.example.com>. Use `minio` as the access key, and `minio123` as the secret key.
- 1.1. In a web browser, navigate to the RHOAI dashboard at <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com>. Use `developer` as the username and the password.
 - 1.2. Go to the project details by clicking **Data Science Projects** and then **rhoai-singlemodel**.
 - 1.3. In the **Models and model servers** section, within the **Single-model serving platform** box, click **Deploy model**.
 - 1.4. Enter `flan-t5-small` as the model name, select **TGIS Standalone ServingRuntime for KServe** as the serving runtime, and `pytorch` as the model framework.
In the **Model location** section select `s3-minio` as the existing data connection name, and enter `models/flan-t5-small` as the path to the model.
Leave the other fields with the default values, and click **Deploy**.

**Note**

When you create a model server or deploy a model to a model server, RHOAI pulls the needed container images. This operation can take some minutes. You can continue with the exercise while the inference endpoint reaches the ready status.

- ▶ 2. Deploy the diabetes model to the `rhoai-singlemodel` project.
 - 2.1. To deploy the diabetes model, click **Deploy model** in the **Models and model servers** section.
 - 2.2. Enter **diabetes** as the model name, select **OpenVINO Model Server** as the serving runtime, and **onnx - 1** as the model framework.
In the **Model location** section select **s3-minio** as the existing data connection name, and enter **models/diabetes-from-tensorflow-keras.onnx** as the path to the model.
Leave the other fields with the default values, and click **Deploy**.

The screenshot shows the Red Hat OpenShift AI dashboard. In the center, there is a table titled "Models and model servers". It lists two entries:

Model name	Serving runtime	Inference endpoint	API protocol	Status
diabetes	OpenVINO Model Server	https://diabetes-rhoai-singlemodel.apps.ocp...	REST	READY
flan-t5-small	TGIS Standalone ServingRuntime for KServe	https://flan-t5-small-rhoai-singlemodel.apps...	gRPC	READY

At the top of the dashboard, there is a navigation bar with the Red Hat logo and the text "Red Hat OpenShift AI". Below the navigation bar, there are sections for "Workbenches", "Cluster storage", "Data connections", and "Pipelines". A "Deploy model" button is located in the "Models and model servers" section. A "Configure pipeline server" button is also visible.

Figure 11.7: Two models deployed on two single-model serving runtimes

- ▶ 3. Deploy the diabetes model in the `rhoai-multimodel` project. Use the same ONNX file that you used to deploy this model in the previous step.
 - 3.1. In the RHOAI dashboard, navigate to the project details by clicking **Data Science Projects** and then **rhoai-multimodel**.
 - 3.2. In the **Models and model servers** section, within the **Multi-model serving platform** box, click **Add model server**.
 - 3.3. Enter **multimodel-server** as the model server name, and select **OpenVINO Model Server** as the serving runtime.
Select **Model route**, and **Token authorization**, and leave the other fields with the default values. Then click **Add**.
 - 3.4. Deploy the diabetes model to the server called **multimodel-server**.
Click **Deploy model**. Enter **diabetes** as the model name, and select **onnx - 1** as the model framework.
In the **Model location** section, select **s3-minio** as the existing data connection name, and enter **models/diabetes-from-tensorflow-keras.onnx** as the path to the model.

Click Deploy.

Model name	Inference endpoint	API protocol	Status
diabetes	https://diabetes-rhoai-multimodel.ap...	REST	Green

Figure 11.8: Diabetes model deployed on a multi-model serving runtime

► 4. Deploy the iris model to the server called `multimodel-server`.

Click **Deploy model**. Enter `iris` as the model name, and select `onnx - 1` as the model framework.

In the **Model location** section, select `s3-minio` as the existing data connection name, and enter `models/rf_iris.onnx` as the path model.

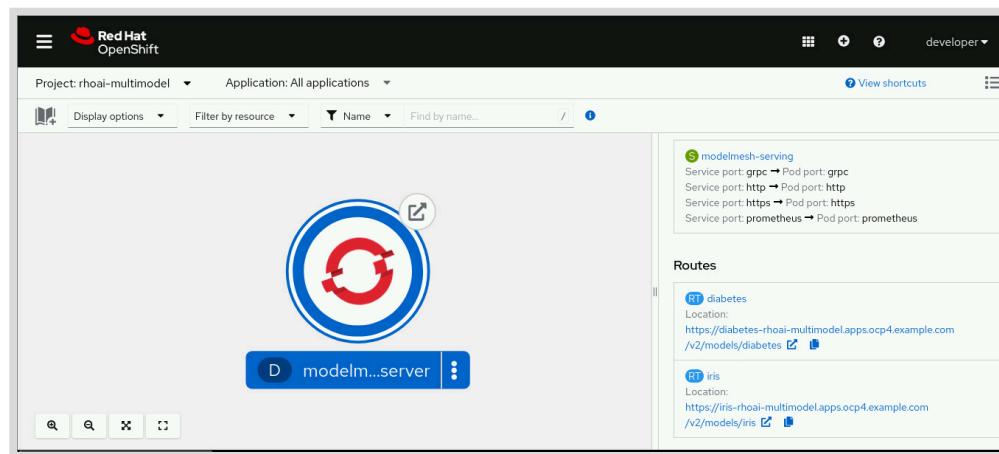
Click **Deploy**.

► 5. Inspect the RHOAI resources, and OpenShift resources in the OpenShift console.

- 5.1. Access the OpenShift console at <https://console-openshift-console.apps.ocp4.example.com>. Use `developer` as the username and the password.
- 5.2. In the developer perspective, navigate to **Topology**, and select the `rhoai-singlemodel` project.

Take some time inspecting the single-model platform objects and containers for the two deployed models. There are two Deployment Kubernetes objects, and two KService Serverless Operator objects. Each Deployment object starts one pod for each model server. Click each of the Deployment objects in the topology diagram to inspect their details. Drill down through the Pod object to identify the containers that form the pod.

- 5.3. In the developer perspective, navigate to **Topology**, and select the **rhoai-multimodel** project.



The screenshot shows the Red Hat OpenShift developer perspective with the 'Topology' view selected. The project is set to 'rhoai-multimodel'. In the center, there's a large circular icon with a red and blue design, labeled 'modelmesh-serving'. To the right, under 'Routes', there are two entries: 'diabetes' and 'iris'. Each entry includes a small icon, the route name, and its corresponding URL: 'https://diabetes-rhoai-multimodel.apps.ocp4.example.com/v2/models/diabetes' and 'https://iris-rhoai-multimodel.apps.ocp4.example.com/v2/models/iris' respectively.

Take some time inspecting the **Deployment** object that implements the serving runtime and the deployed model. Click the only **Deployment** Kubernetes object to inspect its details. Drill down through the **Pod** object to identify the containers that form the pod.

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish rhoaiserving-using
```

Consuming the Model Serving API

Objectives

- Make inference requests to models deployed with Model Serving.

There are various types of AI models. For example, some models are classic machine learning algorithms, such as logistic regression. Others are well-known neural networks, such as ResNet, and others are large language models (LLM), such as Mistral 7B. Because the inputs and outputs of each type can differ dramatically, different model serving interfaces might cater to only one or a few types of models. Some interfaces and tools, such as KServe, try to abstract away these differences.

Which model serving interface you should use depends on the type of model and your particular use cases.

KServe Predict API

In RHOAI, both single and multi-model serving platforms use the KServe inference API. KServe defines two APIs that are independent of any specific AI/ML framework or model server. This lecture focuses on version two of the KServe inference API.

KServe includes an API schema that systems can implement so that different underlying servers behave similarly. This schema includes endpoints to retrieve metadata about the model server and hosted models. It also defines an endpoint for making an inference request to a model. KServe provides schemas for both HTTP REST and gRPC.



Note

The gRPC protocol is a Remote Procedure Call (RPC) framework. It is built on top of Protocol Buffers (protobuf) and HTTP/2 to define schemas and transmit data, respectively.

For more information on gRPC, see the official website at <https://grpc.io>.

Requesting Metadata

The API specification includes endpoints for retrieving data about available models.

For example, you can use the v2/models/MODEL_NAME endpoint to verify that the model is ready for inference requests.

```
[user@host ~]$ curl -s \ ①
-H "Authorization: Bearer $TOKEN" \ ②
$HOST/v2/models/example-model | jq ③
{
  "name": "example-model__isvc-4e49574209", ④
  "versions": [ ⑤
    "1"
  ],
}
```

```
...output omitted...
"inputs": [ ⑥
  {
    "name": "X",
    ...output omitted...
  }
],
"outputs": [ ⑦
  {
    "name": "label",
    ...output omitted...
  },
  {
    "name": "scores",
    ...output omitted...
  }
]
```

- ① The `-s` option hides the transfer output.
- ② The API calls require token header authentication.
- ③ The `jq` command formats the JSON output.
- ④ The `name` field of the response provides the full name of the model.
- ⑤ The `versions` field of the response provides all available versions of the model.
- ⑥ The `inputs` field of the response lists the input tensors that the model requires.
- ⑦ The `outputs` field of the response lists the outputs that the model produces.



Note

By default, the OpenShift Route object that RHOAI exposes for a model only includes the path for the model. It does not expose the underlying KServe service entirely, so server metadata endpoints are only available within the cluster.

Inference Input

To make an inference request to the model remotely, use the `v2/models/MODEL_NAME/infer` endpoint, replacing `MODEL_NAME` with the selected model. For example, the following is the body of a POST request made to the inference endpoint.

```
{
  "id": "abc123", ①
  "parameters": [], ②
  "outputs": [{ ③
    "name": "output",
    "parameters": [] ④
  }],
  "inputs": [{ ⑤
    "name": "input", ⑥
    "datatype": "INT64", ⑦
  }]
```

```

    "shape" : [2, 3], ❸
    "data" : [[34, 54, 65], [4, 12, 21]] ❹
  }]
}

```

- ❶ The `id` field is an optional identifier for the request. If provided, it is returned in the response.
- ❷❸ The `parameters` field is an optional list of parameter key-value pairs. This is reserved as a future enhancement as no parameters are available at time of writing.
- ❹ The `outputs` field is an optional list of requested tensors. The response includes all outputs from the model if this field is not specified.
- ❺ The `inputs` field is a required list of input tensors.
- ❻ The `name` field of each input must match the name of the input tensor within the model.
- ❼ The `datatype` field of an input specifies the type of values in the `data` field.
- ❽ The `shape` field of an input defines the structure of the values in the `data` field.
- ❾ The `data` field of an input provides the actual input data.



Note

To verify the required input name, data type, and shape, you can make a request to the metadata endpoint.

The `data` field can be formatted as either a single list or as nested lists matching the `shape` field. The latter format is called *natural format*.

For example, the preceding example request's data is in natural format. You could instead format it in the following way:

```

...output omitted...
"shape" : [2, 3],
"data" : [34, 54, 65, 4, 12, 21]
...output omitted...

```

In either format, the number of elements must match the defined `shape`. In the example request, because the `shape` is defined as a two-by-three matrix, the `data` field must include exactly six values.

Inference Responses

After making an inference request, the server replies with either the results of the inference or an error response. On success, the JSON response matches the structure of the following example output:

```

{
  "model_name": "MODEL_NAME",
  "model_version": "1",
  "id": "abc123", ❶
  "outputs": [{ ❷

```

```
        "name": "scores", ③
        "shape": [1,3],
        "datatype": "FP32",
        "data": [3.122946, 4.825508, 3.4817147] ④
    }]
}
```

- ①** The `id` field in the response matches the `id` field from the request.
- ②** The `outputs` list includes the results of the inference.
- ③** Each output is organized by name and specifies the shape and type.
- ④** The `data` field of each output includes the resulting data for each output of the inference.

Making a Request

To make an HTTP inference request from the command line, you can use the `curl` command. For example, the following command triggers an inference to a model called `example-model` and provides required input data.

```
[student@workstation ~]$ curl -s \ ①
-H "Authorization: Bearer $TOKEN" \ ②
$HOST/v2/models/example-model/infer \
-X POST \ ③
--data '{"inputs" : [{"name" : "X","shape" : [ 1, 4 ],"datatype" : "FP32","data" :
[ 3, 4, 3, 2 ]}]}' \ ④
| jq ⑤
{
  ...
  "outputs": [ ⑥
    {
      "name": "scores",
      "datatype": "FP32",
      "shape": [
        1,
        2
      ],
      "data": [
        3.122946,
        4.825508,
        ...
      ]
    }
  ]
}
```

- ①** The `-s` option hides the transfer output.
- ②** The API calls require token header authentication.
- ③** The `-X` option specifies the HTTP request method. In this case, the request uses the `POST` method.
- ④** The `--data` option sets the request body.

- 5 The jq command formats the JSON output.
- 6 The outputs field of the response lists the results of the inference.

**Note**

Alternative to using HTTP, you can use gRPC to make inference requests. To do this from the command line, you can use the gRPCurl tool.

You can also achieve a similar result in Python. For example, you can use the `requests` library, as follows:

```
import requests

INFERENCE_ENDPOINT = "https://your-project.apps.cluster.example.com/v2/models/
example-model/infer"

payload = {
    "inputs": [
        {
            "name": "X",
            "shape": [1, 4],
            "datatype": "FP32",
            "data": [3, 4, 3, 2]
        }
    ]
}

response = requests.post(INFERENCE_ENDPOINT, json=payload)
result = response.json()
```

Datatypes

Both the request and response objects use `datatype` fields to specify the types of data represented. If an element does not match the specified type, then a given server implementation can decide whether to attempt coercing the value or returning an error.

The following are some of the available types and their meanings:

- **BOOL**: A Boolean true-false value
- **UINT**: An unsigned integer
- **INT**: A signed integer
- **FP**: A 32-bit floating-point number

The numeric values, such as `UINT` and `FP`, use a postfix value to specify the number of bits. For example, a datatype of `FP32` is a 32-bit floating-point number. The number of bits must be in full byte increments and floating-point types must use sixteen bits or more.

Refer to the KServe Predict API specification for a full list of available types.

Alternative Interfaces

Text Generation Inference Server (TGIS)

To invoke an LLM to generate text, you must use a model serving runtime that supports LLMs specifically, such as TGIS. This runtime enables you to pass plain text in the body of the inference request, instead of passing lower-level array structures.

TGIS is an IBM fork of Hugging Face's Text Generation Inference (TGI). The fork diverged from TGI version 1.0 and maintains similar features to TGI, but with differing implementations for some features.

Caikit

Many of the model-serving APIs and abstractions, such as KServe, use raw tensor values as their inputs and outputs. This can be challenging for developers and users who intend to treat the AI model as a single obscure entity. Using tensor values also requires additional code to handle translating those values to and from useful application objects.

The Caikit framework strives to consolidate interaction with AI models by providing an abstraction layer that is agnostic to many of the models' implementation details. By doing so, the project also enables mixing many models with differing interfaces.

Setting up Caikit is more involved than exposing and consuming an API. The framework mainly intends for both the client and the server to use the provided Python structures. However, the server runtime uses gRPC or HTTP as the foundation, so theoretically any client could consume the API.



References

KServe Predict API v2

https://github.com/kserve/kserve/blob/master/docs/predict-api/v2/required_api.md

KServe Datatypes

https://github.com/kserve/kserve/blob/master/docs/predict-api/v2/required_api.md#tensor-data-types

Caikit GitHub project page

<https://github.com/caikit/caikit>

IBM Text Generation Inference Server

<https://github.com/IBM/text-generation-inference>

Hugging Face - Consuming Text Generation Inference

https://huggingface.co/docs/text-generation-inference/basic_tutorials/consuming_tgi

For more information, refer to the *Serving large language models* chapter in the *Red Hat OpenShift AI Self-managed* documentation at

https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.6/html/serving_models/serving-large-language-models_serving-large-language-models#accessing-api-endpoints-for-models-deployed-on-single-model-serving-platform_serving-large-language-models

► Guided Exercise

Consuming the Model Serving API

Make inference requests to models deployed with Model Serving.

Outcomes

- Send inference requests to a model that runs in a multi-model server.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI265 \
&& lab start rhoaiserving-consuming
```

Instructions

- 1. Create a model server called `infer-model-server`.
- 1.1. Navigate to the Red Hat OpenShift AI (RHOAI) dashboard at <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com>. Use `developer` as the username, and the password.
Go to the project details by clicking **Data Science Projects** and then select `rhoaiserving-consuming`.
 - 1.2. In the **Models and model servers** section, click **Add model server** within the **Multi-model serving platform** card.



Important

Be sure to create a multi-model server.

- 1.3. Fill out the form with the following values:
 - Model server name: `infer-model-server`
 - Serving runtime: `OpenVINO Model Server`
 - Enable the `Make deployed models available through an external route` option
 - Enable the `Require token authentication` option
 - Service account name: `infer-serviceaccount`
- 1.4. Leave the rest of the fields as their default values and click **Add** to create the model server.

- 2. Deploy a model called `iris-model` to the model server.

- 2.1. Next to the `infer-model-server` model server, click **Deploy model**.

2.2. Fill out the form with the following values:

- Model name: **iris-model**
- Model framework: **onnx - 1**
- Select **Existing data connection**
- Select the **iris-data-connection** data connection
- Path: **iris**

2.3. Click **Deploy** to deploy the model.



Note

When you create a model server or deploy a model to a model server, RHOAI pulls the needed container images. This operation can take some minutes.

- 3. Use the `curl` command-line tool to execute a prediction from the model by using the `/infer` endpoint.
- 3.1. In the `Deployed models` column, click the cell with numeral one to show information about the deployed model.
 - 3.2. Copy the `Inference endpoint` URL for the model and, in a new command-line terminal window, store the model portion of it in a `MODEL_PATH` environment variable. Do not include the trailing `/infer` portion of the URL.

```
[student@workstation ~]$ export MODEL_PATH=https://iris-model.../v2/models/iris-model
```

- 3.3. In the `Tokens` column, click the cell with numeral one to show information about the token.
- 3.4. Copy the `Token secret` value and store it in a `TOKEN` environment variable.

```
[student@workstation ~]$ export TOKEN=eyJH...Rk12
```

- 3.5. Use the URL and authorization token to retrieve metadata about the deployed model.

```
[student@workstation ~]$ curl -s \
-H "Authorization: Bearer $TOKEN" \
$MODEL_PATH | jq
{
  "name": "iris-model...",
  "versions": [
    "1"
  ],
  "platform": "OpenVINO",
  "inputs": [
    {
      "name": "X",
      "datatype": "FP32",
      "shape": [
        "-1",
        "4"
      ]
    }
  ]
}
```

```
        },
      ],
    "outputs": [
      {
        "name": "label",
        "datatype": "INT64",
        "shape": [
          "-1"
        ]
      },
      {
        "name": "scores",
        "datatype": "FP32",
        "shape": [
          "-1",
          "3"
        ]
      }
    ]
  }
```

- 3.6. Call the inference endpoint by providing input tensors to the model. Note that the data body of the POST method is in JSON format.

```
[student@workstation ~]$ curl -s \
-H "Authorization: Bearer $TOKEN" \
$MODEL_PATH/infer \
-X POST \
--data '{"inputs": [{"name": "X", "shape": [1, 4], "datatype": "FP32", "data": [3, 4, 3, 2]}]}' \
| jq
{
  ...
  "outputs": [
    {
      "name": "label",
      ...
    },
    {
      "name": "scores",
      ...
      "data": [
        3.122946,
        4.825508,
        3.4817147
      ]
    }
  ]
}
```

The values in the scores output represent the results of the model's prediction.

- 4. Use the Python requests library to programmatically execute a prediction from the model.

- 4.1. Within the RHOAI dashboard, create a new workbench called `infer-wb` by using the `Standard Data Science` image. After the workbench finishes starting, open the workbench by clicking `Open`. Use `developer` as the username, and the password.
- 4.2. Within the workbench, clone the `https://github.com/RedHatTraining/AI26X-apps` repository and open the `deploying/rhoaiserving-consuming/inference-request.ipynb` file.
- 4.3. Continue the exercise by following the instructions in the notebook.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish rhoaiserving-consuming
```

Creating and Using Model Servers

Objectives

- Create and use a model server in OpenShift AI.

Custom Model Serving Runtimes

In Red Hat OpenShift AI (RHOAI), you can add and implement additional model runtimes for single-model and multi-model servers.

To extend the serving runtime catalog, you can provide RHOAI with `ServingRuntime` YAML definitions that use a serving runtime container. The KServe project provides containers and `ServingRuntime` definitions for model serving runtimes such as ONNX Runtime, Triton, or Pytorch.

Using a multi-model server other than the default OpenVINO is out of the scope for this course. For more information on adding a multi-model server, refer to *Adding a Model Server for the Multi-model Serving Platform* in the references section.



Note

RHOAI enables you to add your own custom runtimes, but does not support the runtimes themselves. You are responsible for correctly configuring and maintaining custom runtimes.

Creating a Model Server Container

To create a custom model serving container, you can create a container with your custom Python code. You must create a subclass of `kserve.Model` and implement the `load` handler to load your model into memory and the `predict` handler to execute the inference for your model.

The following code shows the basic structure for creating a `kserve.Model` Python subclass:

```
from typing import Dict
from kserve import Model, ModelServer

class AlexNetModel(Model):
    def __init__(self, name: str):
        super().__init__(name)
        self.name = name
        self.load()

    def load(self):
        ...code omitted...

    def predict(self, payload: Dict, headers: Dict[str, str] = None) -> Dict:
        ...code omitted...
```

```
if __name__ == "__main__":
    model = AlexNetModel("custom-model")
    ModelServer().start([model])
```

Then, you must create a container image to run the previous code, push the image to a container registry, and make the image accessible to your RHOAI cluster. Finally, to make the serving runtime available, you must create a `ServingRuntime` resource from the RHOAI console. This resource must point to the container image that you just created.

Configuring a Single-Model Serving Runtime

You can add serving runtimes from the RHOAI console. To add a serving runtime, log in to RHOAI as an administrator, then go to `Settings > Serving Runtimes`, and click `Add serving runtime`. Select the type of serving platform and protocol, and provide a `ServingRuntime` manifest, which defines the container used for model serving.

The following is an example of a Paddle `ServingRuntime` definition:

```
apiVersion: serving.kserve.io/v1alpha1
kind: ServingRuntime
metadata:
  annotations:
    openshift.io/display-name: Example Model Server ①
    name: kserve-paddleserver
spec:
  annotations:
    prometheus.kserve.io/path: /metrics
    prometheus.kserve.io/port: "8080"
  containers:
    - args:
        - --model_name={{.Name}}
        - --model_dir=/mnt/models ②
        - --http_port=8080
      image: paddle_container_image ③
      name: kserve-container
    multiModel: false ④
  protocolVersions:
    - v2
  supportedModelFormats: ⑤
    - autoSelect: true
      name: paddle
      priority: 1
      version: "2"
```

- ① The serving runtime name that shows on RHOAI's UI.
- ② The default path where all the single-model servers search for the models.
- ③ The container that runs the serving runtime.
- ④ This runtime is intended for a single-model server.
- ⑤ Formats that the serving runtime supports.

Click `Create` at the bottom of the form to finish the addition of the new serving runtime.

After adding a custom runtime, you can navigate to your data science project and create a model server based on the new runtime.

**Note**

The form to add a serving runtime performs only a syntactic validation of the `ServingRuntime` resource.

Other errors, such as a typo in the image name, will show when you instantiate a model server. In such cases, you might want to use the OpenShift console or the `oc` CLI to troubleshoot the pods inside your data science project.

**References****KServe Model Serving Runtimes**

https://kserve.github.io/website/latest/model-serving/v1beta1/serving_runtime/

For more information, refer to the *Adding a Model Server for the Multi-model Serving Platform* section in the *Serving Small and Medium-sized Models* chapter in the Red Hat Red Hat OpenShift AI Self-managed 2.8 *Serving Models* documentation at

https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/serving_models/index#adding-a-model-server-for-the-multi-model-serving-platform_model-serving

► Guided Exercise

Creating and Using Model Servers

Create and deploy a custom model server with OpenShift AI.

Outcomes

- Create a custom model server by creating a Python container image and importing the ServingRuntime YAML into RHOAI.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI265 && lab start rhoaiserving-custom
```

Instructions

In this exercise you create a custom Scikit-learn model server by creating a Python container and a RHOAI serving runtime. Then, you deploy a version of the diabetes model trained with Scikit-learn by using the RHOAI console.

- 1. Implement the `predict` method of the `SKLearnModel` class, a `kserve.Model` subclass.

- 1.1. Change to the exercise directory.

```
[student@workstation ~]$ cd ~/AI265/rhoaiserving-custom
```

- 1.2. Inspect the `SKLearnModel` class, which implements the `load` and `predict` methods.

```
[student@workstation rhoaiserving-custom]$ cat model_server/model.py
...output omitted...
def predict(
    self, payload: Union[Dict, InferRequest], headers: Dict[str, str] = None
) -> Union[Dict, InferResponse]:
    try:
        instances = get_predict_input(payload)
        # result = self.model.predict(instances)
        # return get_predict_response(payload, result, self.name)
        raise InferenceError("Use the model to make predictions to finish the
exercise")
    except Exception as e:
        raise InferenceError(str(e))
...output omitted...
```

13. Use your preferred editor to update the `predict` method. Uncomment the code and remove the first `InferenceError` line to make the implementation look like the following:

```
...code omitted...
def predict(
    self, payload: Union[Dict, InferRequest], headers: Dict[str, str] = None
) -> Union[Dict, InferResponse]:
    try:
        instances = get_predict_input(payload)
        result = self._model.predict(instances)
        return get_predict_response(payload, result, self.name)
    except Exception as e:
        raise InferenceError(str(e))
...code omitted...
```

- ▶ 2. Use the Container file that the exercise provides to create a container and push it to the classroom internal registry.

2.1. Analyze the `Containerfile`.

```
[student@workstation rhoaiserving-custom]$ cat Containerfile
...
ENTRYPOINT ["python", "-m", "model_server.model"]
```

2.2. Analyze the `build_and_push.sh` script.

```
[student@workstation rhoaiserving-custom]$ cat build_and_push.sh
#!/bin/bash
QUAY_HOST="registry.ocp4.example.com:8443"
podman login -u=developer -p=developer ${QUAY_HOST}
podman build . -t ${QUAY_HOST}/developer/sklearn-model-server
podman push ${QUAY_HOST}/developer/sklearn-model-server
```

2.3. Run the script to push the container to the classroom registry.

```
[student@workstation rhoaiserving-custom]$ ./build_and_push.sh
...
Writing manifest to image destination
Storing signatures
```

Building the image takes some minutes.

- ▶ 3. Edit the `serving_runtime.yaml` file in the exercise directory to add the new `sklearn-model-server` container.

```
apiVersion: serving.kserve.io/v1alpha1
kind: ServingRuntime
labels:
  opendatahub.io/dashboard: "true"
```

```
metadata:  
  name: sklearn-server  
spec:  
...output omitted...  
  containers:  
    - name: kserve-container  
      image: registry.ocp4.example.com:8443/developer/sklearn-model-server  
      args:  
        - --model_name={{.Name}}  
        - --model_dir=/mnt/models  
        - --http_port=8080  
        - --enable_docs=True  
...output omitted...
```

- ▶ **4.** Import the serving runtime in the `serving_runtime.yaml` file to the `rhoaiserving-custom` data science project.
- 4.1. In a web browser, navigate to the RHOAI dashboard at `https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com`. Use `admin` as the username and `redhatocp` as the password.
 - 4.2. In the left menu panel, go to **Settings > Serving runtimes**, and then click **Add serving runtime**.
 - 4.3. Select **Single-model serving platform** and **REST**, and click **Upload files** to upload the `serving_runtime.yaml` file at `/home/student/AI265/rhoaiserving-custom`.
 - 4.4. Click **Create** and verify that the `sklearn-server` shows in the **Serving runtimes** list.
- ▶ **5.** Deploy the diabetes model to the new **Scikit-learn** server.
- 5.1. Log out RHOAI, and log back in as the developer user with the developer password.
 - 5.2. Open the `rhoaiserving-custom` data science project.
 - 5.3. Click **Deploy model** in the **Single-model serving platform** section.
 - 5.4. Complete the form with the following values and click **Deploy**:
 - Model name: `diabetes`
 - Serving runtime: `sklearn-server`
 - Model framework (name- version): `sklearn - 1`
 - Existing data connection:
 - Name: `s3-minio`
 - Path: `models/diabetes.joblib`
 - 5.5. Wait for your model to reach the Ready status.
- ▶ **6.** Send an inference request to verify the model deployment.
- 6.1. Open a terminal and use the `oc` command to log in to OpenShift as the developer user.

Chapter 11 | Model Serving in Red Hat OpenShift AI

```
[student@workstation rhoaiserving-custom]$ oc login api.ocp4.example.com:6443 \
-u developer -p developer
Login successful.
...output omitted...
```

- 6.2. Ensure that you use the rhoaiserving-custom project.

```
[student@workstation rhoaiserving-custom]$ oc project rhoaiserving-custom
Already on project "rhoaiserving-custom" on server "https://
api.ocp4.example.com:6443".
```

- 6.3. Get the inference service URL from the diabetes InferenceService resource.

```
[student@workstation rhoaiserving-custom]$ oc get inferenceservice
NAME      URL                                     READY ...
diabetes   https://diabetes-rhoaiserving-custom.apps.ocp4.example.com  True ...
```

Create an environment variable with the URL.

```
[student@workstation rhoaiserving-custom]$ SERVER_HOST=\
"https://diabetes-rhoaiserving-custom.apps.ocp4.example.com"
```

- 6.4. Verify that the KSeverV2 API is accessible by sending an HTTP POST request to the inference endpoint. Send an inference request with a diabetes and a non-diabetes case and verify the output predicts 1 and 0 respectively.

```
[student@workstation rhoaiserving-custom]$ curl -sk -X 'POST' \
$SERVER_HOST/v2/models/diabetes/infer \
-H 'accept: application/json' -H 'Content-Type: application/json' \
-d '{
  "inputs": [
    {
      "data": [[6,148,72,35,0,33.6,0.627,50],[1,85,66,29,0,26.6,0.351,31]],
      "datatype": "FP32", "name": "input0", "shape": [2,8]
    }
  ],
  "outputs": [{"name": "output0"}]
}' | jq
{
  "model_name": "diabetes",
  "model_version": null,
  "id": "5827fc44-62fd-4e42-a0d4-3bf54c8750b5",
  "parameters": null,
  "outputs": [
    {
      "name": "output-0",
      "shape": [
        2
      ],
      "datatype": "INT64",
      "parameters": null,
      "data": [
        1
      ]
    }
  ]
}
```

```
    1,
    0
]
}
]
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish rhoaiserving-custom
```

Summary

- Model serving can deploy and expose models that are available in an S3 bucket.
- By default, model serving offers OpenVINO, Caikit TGIS, and TGIS standalone as serving runtimes.
- Use the KServe Predict API to send inference requests to models served with OpenVINO.
- In single-model serving, each model server can serve one model.
- In multi-model serving, each model server can serve multiple model.
- You can add a custom model server runtime by importing a `ServingRuntime` resource.

Chapter 12

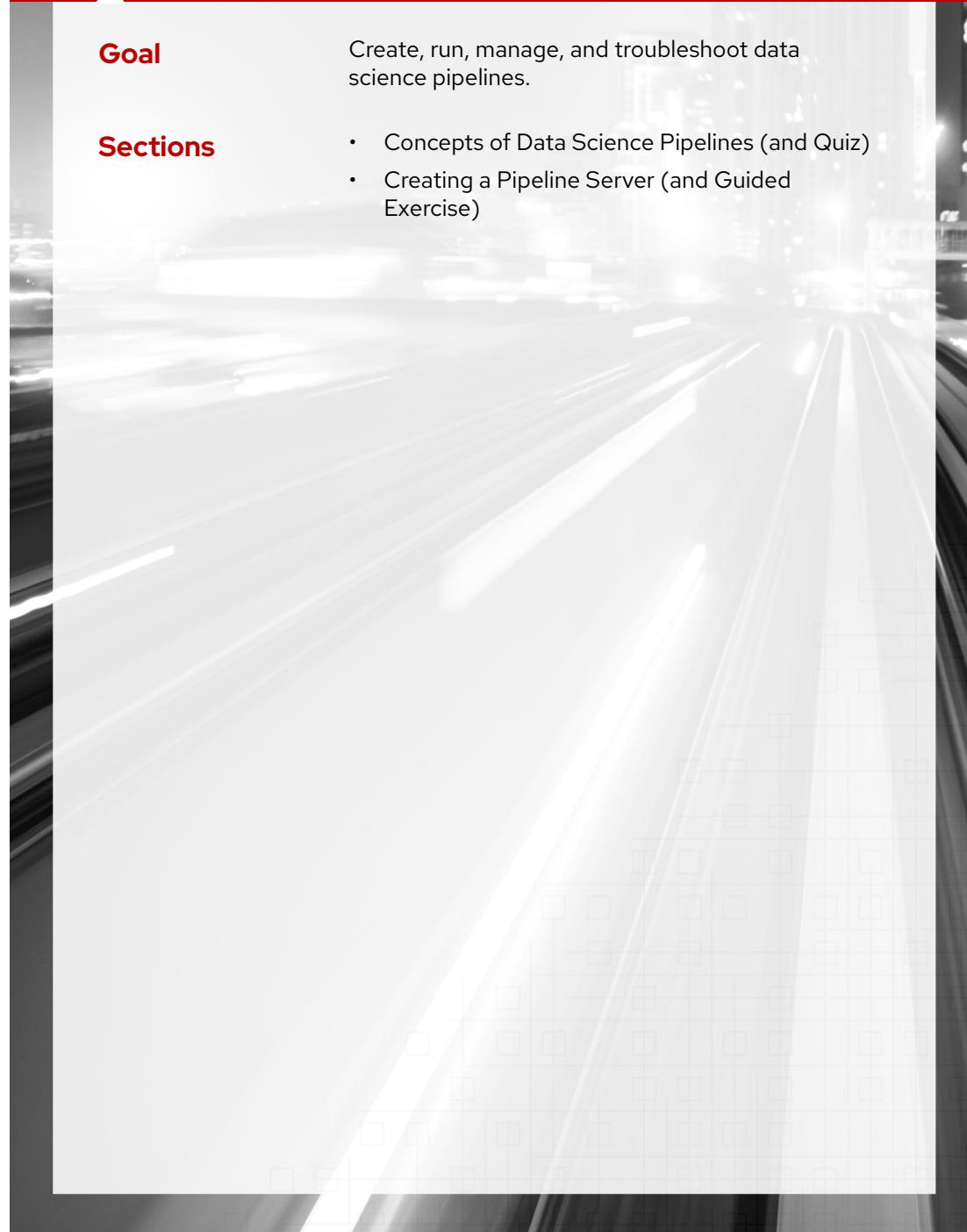
Introduction to Data Science Pipelines

Goal

Create, run, manage, and troubleshoot data science pipelines.

Sections

- Concepts of Data Science Pipelines (and Quiz)
- Creating a Pipeline Server (and Guided Exercise)



Concepts of Data Science Pipelines

Objectives

- Describe data science pipelines concepts.

Data science pipelines is the component of Red Hat OpenShift AI (RHOAI) that enables the automation of AI/ML workflows.

Data science pipelines is an implementation of Kubeflow Pipelines. RHOAI uses *OpenShift Pipelines*, which is based on Tekton, as the execution engine for data science pipelines. This is different from the upstream Kubeflow Pipelines, which uses Argo Workflows as the engine. Additionally, unlike Kubeflow Pipelines, data science pipelines are designed to support multitenancy.



Important

From version 2.9, RHOAI switches the engine from Tekton to Argo Workflows, for better alignment with the Kubeflow Pipelines upstream project.

Users can deploy an instance of the data science pipelines server in a namespace by using a `DataSciencePipelineApplication` OpenShift resource. Users can then create pipelines and submit them to the `DataSciencePipelineApplication` instance for execution.

Data Science Pipeline Concepts

- A *pipeline* is a workflow definition containing steps, inputs, and output artifacts.
- A *run*, or *pipeline run*, is a *single* execution of a pipeline that can be a one-off execution of a pipeline or scheduled as a *recurring run*.
- A *step* is a self-contained pipeline component that represents an execution stage in the pipeline.
- An *artifact* is an object that can be persisted after the execution of a step. Steps can use artifacts as inputs and some artifacts might be useful as post-completion references. Data science pipelines automatically store artifacts in S3-compatible storage.
- An *experiment* is a logical grouping of runs for the purpose of organization.



Note

A pipeline is an execution graph of tasks, commonly known as a Directed Acyclic Graph (DAG). A DAG is a directed graph without any cycles, also known as direct loops.

Pipelines improve the repeatability of data science experiments. Although the larger experimentation process includes steps such as data exploration, pipelines focus on making experiments repeatable and iterable.

A data science pipeline also fits within the context of a larger pipeline that manages the complete lifecycle of an application. The data science pipeline is responsible for data science parts of the process, such as training models.

Chapter 12 | Introduction to Data Science Pipelines

To train a model, data science pipelines consist of several key activities that are performed in a structured sequence. Such activities include the following:

- *Data collection* is gathering data from various sources, such as databases, APIs, spreadsheets, or external data sets. In complex scenarios with large amounts of data and several data sources, this activity might require the application of Big Data techniques. Examples of such activities are integrating data in data warehouses or data lakes, and *extract, load, transform (ETL)* processes.
- *Data preprocessing* is identifying and handling missing or inconsistent data, removing duplicates, and addressing data quality issues. This often involves normalizing, scaling, one-hot encoding, creating new variables, or reducing dimensionality. This process might also involve creating and transforming features, or variables, to improve performance. This step is crucial for machine learning (ML) algorithms that are sensitive to the scale of features. Another important task is splitting the data into subsets for testing and training to enable validating the model.
- *Model training* is when the ML algorithm iterates through the training data, making adjustments to the model until it reaches a satisfactory version of the model.
- *Model evaluation* is where model performance is executed with the previously unseen test data set. Then, the model is assessed by using various metrics, such as accuracy, precision, recall, F1 score, or mean squared error. You can use cross-validation techniques to ensure the model's robustness when data is scarce.

To improve the reproducibility of experiments, data scientists can use pipelines to quickly iterate on models, adjust data transformation, test different algorithms, and more. Although the preceding steps describe a common pattern, use cases and projects differ in requirements. Thus, they can vary in tool and framework selection, as well as techniques.

Viewing Pipelines Within RHOAI and OpenShift

Data science pipelines and their details are available from various places within RHOAI and the OpenShift cluster.

The screenshot shows the Red Hat OpenShift Data Science interface. The left sidebar has a navigation menu with 'Applications', 'Data Science Projects', 'Data Science Pipelines' (selected), 'Pipelines' (sub-selected), 'Runs', 'Model Serving', and 'Resources'. The main content area is titled 'Pipelines' with the sub-instruction 'Manage your pipelines.' Below this is a dropdown 'Project' set to 'user1'. A search bar at the top right contains 'Pipeline name' and 'Name' fields, and a blue 'Import pipeline' button. To the right of the search bar is a 'Pipeline server actions' dropdown with 'View pipeline server configuration' and 'Delete pipeline server' options. The main table lists three pipelines: 'model-training' (status: Completed, last run 28:28 ago, created 2 days ago), 'offline-scoring' (status: Completed, last run 4:21 ago, created 2 days ago), and 'offline-scoring' (status: Completed, last run 4:21 ago, created 2 days ago). Each pipeline row has a 'Runs' link and a three-dot menu icon.

Figure 12.1: The data science pipelines main page shows pipelines to which the user has access

Name	Experiment	Pipeline	Started	Duration	Status
offline-scoring-1116082509	offline-scoring	offline-scoring-1116082509	22 hours ago	1:07	✓
model-training-1115061139	model-training	model-training	2 days ago	28:28	✓
offline-scoring-1115060602	offline-scoring	offline-scoring	2 days ago	4:21	✓

Figure 12.2: The data science pipelines Runs page shows the completed runs of data science pipelines

Pipeline name	Last run	Last run status	Last run time	Created
model-training	model-training-1115061139	✓ Completed	28:28	2 days ago
offline-scoring	offline-scoring-1115060602	✓ Completed	4:21	2 days ago

Figure 12.3: The data science pipelines view within a data science Project visualizes the runs and other details of pipelines that ran within that project

Because data science pipelines use OpenShift Pipelines as the underlying execution engine, additional execution information and logs are available from the OpenShift console.

Elyra Pipelines

Elyra is a JupyterLab extension that provides a visual editor to create pipelines based on notebook files (.ipynb), Python files (.py), and R scripts (.r). Elyra enables dragging and dropping files, as well as visually creating pipelines.

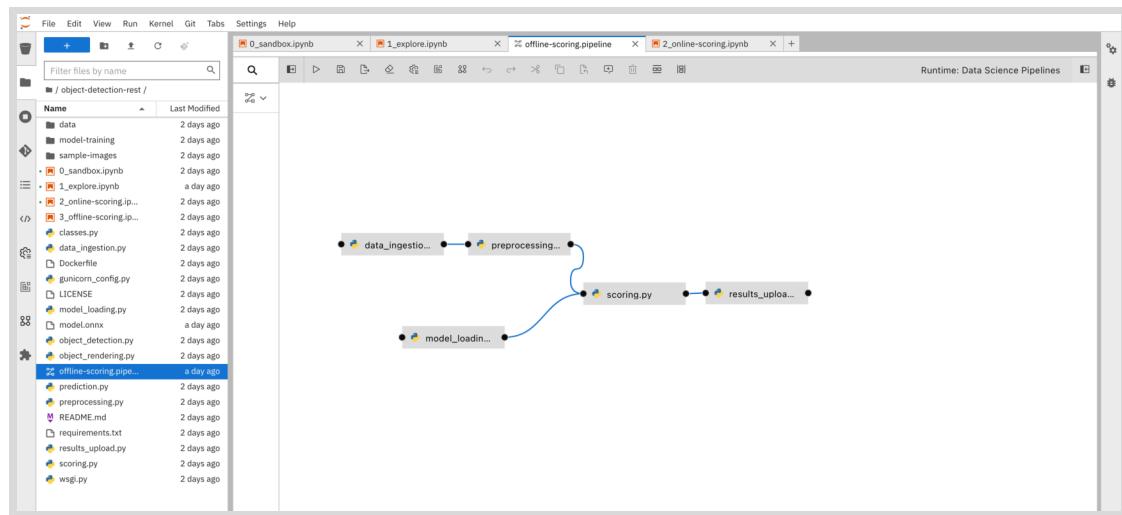


Figure 12.4: An offline scoring pipeline in Elyra

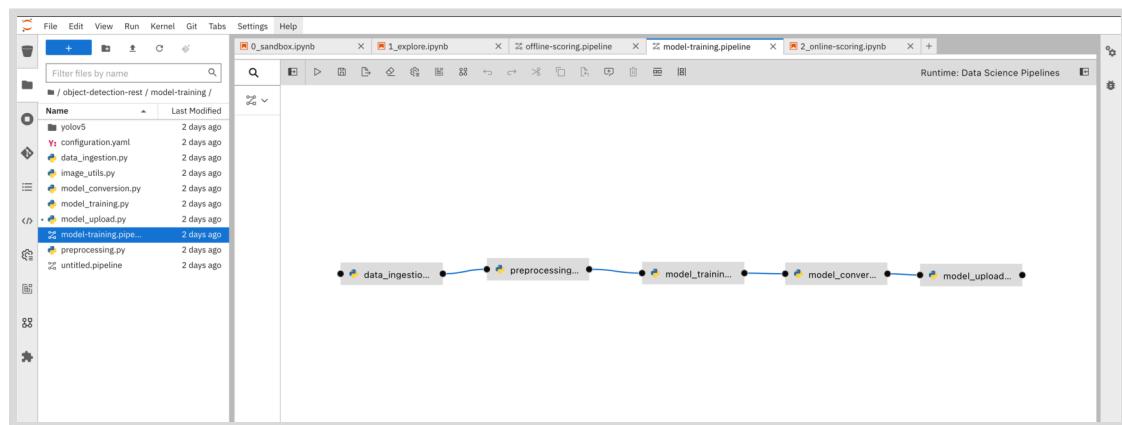


Figure 12.5: A model training pipeline in Elyra

By using Elyra, you can visually assign resources, such as CPUs and GPUs, to each individual step in the pipeline.

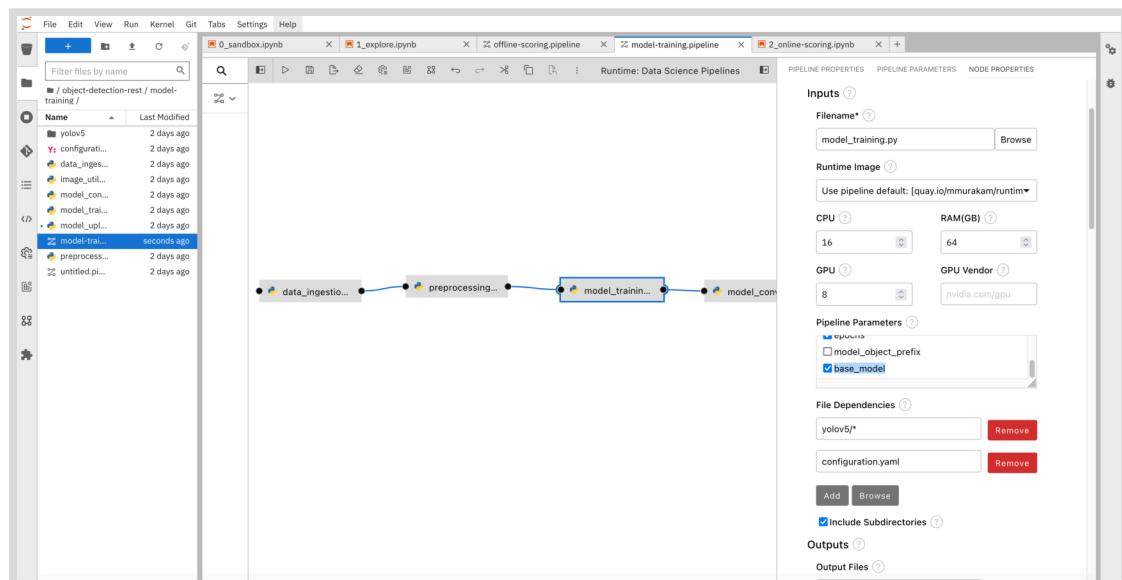


Figure 12.6: Configuring GPUs and other resources for an Elyra pipeline step

After constructing a pipeline with Elyra, users can submit a job directly to Data science pipelines for execution.

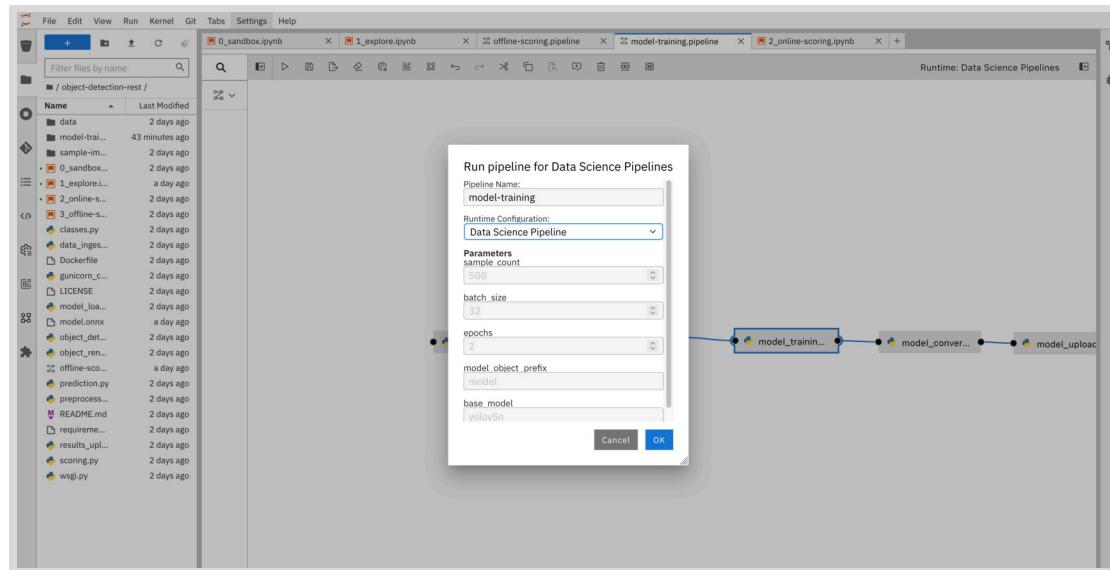


Figure 12.7: Starting an Elyra pipeline

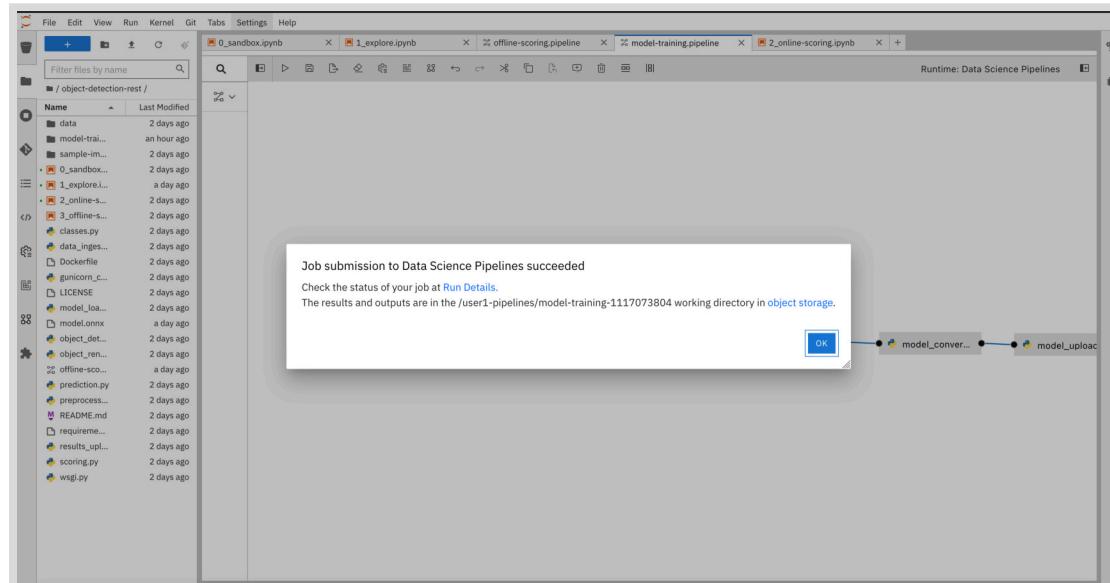


Figure 12.8: Details of a started Elyra pipeline

Kubeflow Pipelines SDK

The Kubeflow Pipelines SDK is a programmatic alternative to Elyra. It involves the use of the `kfp` and `kfp-tekton` Python packages, which eases writing Kubeflow Pipelines with Python. After installing these packages, you can then import them into Python scripts to build the pipeline. You also use these packages to compile pipelines to a YAML object, which is the format required to import a pipeline into the RHOAI dashboard.

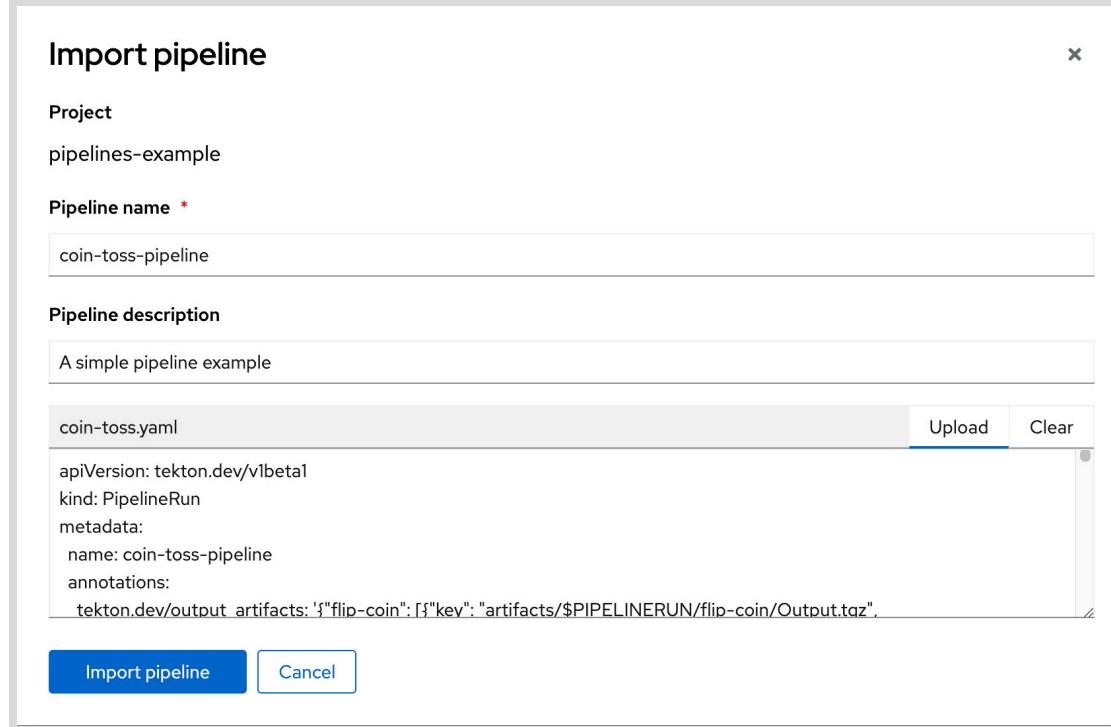


Figure 12.9: Importing a pipeline into the RHOAI dashboard

After uploading, you can execute the pipeline or schedule a recurring run for the pipeline.

Additionally, the `kfp` and `kfp-tekton` packages enable you to connect directly to the `DataSciencePipelineApplication` instance from a Python environment to execute a run without manually compiling and uploading the pipeline.



References

For more information, refer to the *Working with data science pipelines* chapter in the *Red Hat OpenShift AI Self-Managed* documentation at https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/working_on_data_science_projects/index#working-with-data-science-pipelines_ds-pipelines

► Quiz

Concepts of Data Science Pipelines

Choose the correct answers to the following questions:

- ▶ 1. Which of the following statements are advantages of using Data Science Pipelines as part of a machine learning project? (Choose two.)
 - a. Repeatable steps via automation
 - b. Tight integration with machine learning frameworks, such as PyTorch and Tensorflow
 - c. Iterable solutions via reduction of manual steps
 - d. Reduction of bugs via static analysis
- ▶ 2. Which of the following are typical steps in a Data Science Pipeline? (Choose three.)
 - a. Data storage
 - b. Data cleaning
 - c. Data analysis
 - d. Model training
 - e. Model reduction
 - f. Model evaluation
- ▶ 3. Elyra provides a visual editor.
 - a. True
 - b. False
- ▶ 4. On which pipeline framework are Data Science Pipelines based? (Choose one.)
 - a. Elyra
 - b. Argo Workflows
 - c. OpenShift Pipeline Grid
 - d. Kubeflow Pipelines
- ▶ 5. Which Python packages facilitate managing Data Science Pipelines within Python code? (Choose one.)
 - a. pipeline and oc
 - b. kfp and mlops
 - c. kfp and kfp-tekton
 - d. requests and pipeline

► 6. What is the OpenShift resource to which users can submit pipelines? (Choose one.)

- a. DataSciencePipelineApplication
- b. DataSciencePipelines
- c. PipelineManager
- d. PipelineDeployment

► Solution

Concepts of Data Science Pipelines

Choose the correct answers to the following questions:

- ▶ 1. Which of the following statements are advantages of using Data Science Pipelines as part of a machine learning project? (Choose two.)
 - a. Repeatable steps via automation
 - b. Tight integration with machine learning frameworks, such as PyTorch and Tensorflow
 - c. Iterable solutions via reduction of manual steps
 - d. Reduction of bugs via static analysis
- ▶ 2. Which of the following are typical steps in a Data Science Pipeline? (Choose three.)
 - a. Data storage
 - b. Data cleaning
 - c. Data analysis
 - d. Model training
 - e. Model reduction
 - f. Model evaluation
- ▶ 3. Elyra provides a visual editor.
 - a. True
 - b. False
- ▶ 4. On which pipeline framework are Data Science Pipelines based? (Choose one.)
 - a. Elyra
 - b. Argo Workflows
 - c. OpenShift Pipeline Grid
 - d. Kubeflow Pipelines
- ▶ 5. Which Python packages facilitate managing Data Science Pipelines within Python code? (Choose one.)
 - a. pipeline and oc
 - b. kfp and mlops
 - c. kfp and kfp-tekton
 - d. requests and pipeline

► 6. What is the OpenShift resource to which users can submit pipelines? (Choose one.)

- a. DataSciencePipelineApplication
- b. DataSciencePipelines
- c. PipelineManager
- d. PipelineDeployment

Creating a Pipeline Server

Objectives

- Create a pipeline server to manage and execute data science pipelines.

To run and manage data science pipelines, you need a *pipeline server*. A pipeline server is a namespace-scoped instance of the `DataSciencePipelineApplication` custom resource, which creates the necessary pods. This includes the creation of an API endpoint and a database for storing metadata. The Red Hat OpenShift AI (RHOAI) dashboard, as well as tools such as Elyra and the `kfp-tekton` package, use this endpoint to manage and execute pipelines.

In RHOAI, the data science pipeline runtime consists of the following components:

- A data science pipeline server container
- A MariaDB database for storing pipeline definitions and results
- A pipeline scheduler for scheduling pipeline runs
- A persistent agent to record the set of containers that executed as well as their inputs and outputs

Additionally, the `DataSciencePipelineApplication` requires an S3-compatible storage solution to store artifacts that are generated in the pipeline.



Note

You can use any S3-compatible storage solution for data science pipelines, including AWS S3, OpenShift Data Foundation, or MinIO. This course uses MinIO as its S3 storage solution. Red Hat recommends OpenShift Data Foundation in scenarios where security, data resilience, and disaster recovery are important concerns.

Creating a Server

Before you can create pipelines, you must first create a pipeline server. To create the server, use the RHOAI dashboard to navigate to the data science project where you want to create the server.

Scroll down to the **Pipelines** section and then click **Configure pipeline server**.

The screenshot shows the RHOAI Pipeline configuration interface. At the top left is the 'Pipelines' tab. Next to it is an 'Import pipeline' dropdown menu with a downward arrow. In the center is a large wrench icon with the text 'Enable pipelines' below it. At the bottom is a blue rectangular button with the text 'Configure pipeline server' in white. The overall background is light gray.

Complete the form by providing a data connection. This data connection enables the pipeline server to access an S3 bucket.

The creation of the server might take a few minutes until the `DataSciencePipelineApplication` instance and its dependent objects are ready.

When the server is ready, the dashboard activates the **Import pipeline** button. This indicates that you are ready to start using data science pipelines, either with Elyra or Kubeflow Pipelines.

Multitenancy with Data Science Pipeline Applications

As previously mentioned, data science pipelines are designed to support multitenancy. This means that multiple users and teams can use their own instances of data science pipelines without fear of leaking data from pipelines to other users or groups.

Multitenancy requires each user or group to have their own `DataSciencePipelineApplication` instance. Additionally, Red Hat recommends that each `DataSciencePipelineApplication` instance have its own S3 instance that disallows access to other groups.

Although a `DataSciencePipelineApplication` is a namespace-scoped object, workbenches and pods running in other namespaces can interact with the pipeline instance, provided they have appropriate permissions.

Data Science Pipelines Application Permissions

The `DataSciencePipelineApplication` API endpoint route is protected by an OpenShift OAuth Proxy sidecar. OAuth Proxy attempts to access the endpoint to be authenticated by using OpenShift authentication credentials. Thus, OpenShift admits or rejects requests to the endpoint based on the Role-Based Access and Control configuration of the resources in the namespace.

In particular, the `DataSciencePipelineApplication` requires users and service accounts to have get access to the `DataSciencePipelineApplication` route object. Any user with Admin or Edit access to the namespace in which the `DataSciencePipelineApplication` is installed has access to the namespace routes, and therefore, to running and managing pipelines.



Note

Depending on cluster configuration, you might need to grant access to other resources, such as a service account in the cluster, to be able to interact with the API endpoint.

To grant access to the `DataSciencePipelineApplication` API, you must first create a role that grants get permissions in the `DataSciencePipelineApplication` namespace. For example, the following Role can grant access to a `DataSciencePipelineApplication` in a namespace called `my-project`.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: dspa-access
  namespace: my-project
rules:
  - verbs:
    - get
  apiGroups:
```

```
- route.openshift.io
resources:
- routes
```

Given an existing role, a role binding grants the appropriate permissions to the user or service account. For example, the following RoleBinding grants access to a service account called `my-service-account`:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: dspa-access-my-service-account
  namespace: my-project
subjects:
- kind: ServiceAccount
  name: my-service-account
  namespace: my-project
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: dspa-access
```

When programmatically accessing the API endpoint, a user can authenticate to the endpoint by passing a `BearerToken` header value in the HTTP request. Users can obtain tokens from the `Copy Login Command` menu option in the OpenShift Web Console or via the following `oc` command:

```
[user@host ~]$ oc whoami --show-token
fbac...12c7
```



References

For more information, refer to the *Configuring a pipeline server* section in the *Working with data science pipelines* chapter in the Red Hat OpenShift AI Self-managed 2.8 *Working on Data Science Projects* documentation at https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/working_on_data_science_projects/index#managing_data_science_pipelines

► Guided Exercise

Creating a Pipeline Server

Create a pipeline server to manage and execute data science pipelines.

Outcomes

- Create a `DataSciencePipelineApplication` pipeline server in Red Hat OpenShift AI (RHOAI).

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI266 && lab start workflows-pipeline
```

Instructions

- ▶ 1. Observe that the `data-science-pipelines` S3 bucket does not exist.
 - 1.1. In a web browser, navigate to `https://minio-minio.apps.ocp4.example.com`.
 - 1.2. Log in to the MinIO UI by using `minio` as the access key and `minio123` as the secret key.
 - 1.3. Observe that the `data-science-pipelines` bucket does not display in the left-hand navigation panel.
- ▶ 2. Log in to the RHOAI dashboard.
 - 2.1. In a new web browser tab, navigate to `https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com`.
 - 2.2. Click **Log in with OpenShift** and then click `htpasswd_provider`. Log in by using `developer` as both the username and the password.
 - 2.3. In the left navigation pane, click **Data Science Projects** and click `workflows-pipeline` in the list of projects.
- ▶ 3. Create a data science pipeline application.
 - 3.1. In the **Pipelines** section, click **Configure pipeline server**.
 - 3.2. In the right side of the **Access Key** field, click the key icon and select the `data-science-pipelines` data connection.
The other fields in the form are automatically populated.
 - 3.3. Click **Configure pipeline server** to create the pipeline server.

Chapter 12 | Introduction to Data Science Pipelines

After about a minute, the **Pipelines** section shows **No pipelines**, which indicates that the pipeline server is ready.

- 3.4. Verify that the pipeline server automatically creates the **data-science-pipelines** bucket configured in the data connection.

Refresh the MinIO UI browser tab and observe that the **data-science-pipelines** bucket is available in the left-hand navigation panel.

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish workflows-pipeline
```

Summary

- Data science pipelines use OpenShift Pipelines, which are based on Tekton, as the execution engine for pipelines.
- An execution of a pipeline is called a *run* and produces zero or more *artifacts*.
- Data science pipelines are accessible from several places in Red Hat OpenShift AI (RHOAI), such as the details page of a data science project.
- To execute data science pipelines, you need a pipeline server, which is represented by a custom OpenShift resource: `DataSciencePipelineApplication`.

Chapter 13

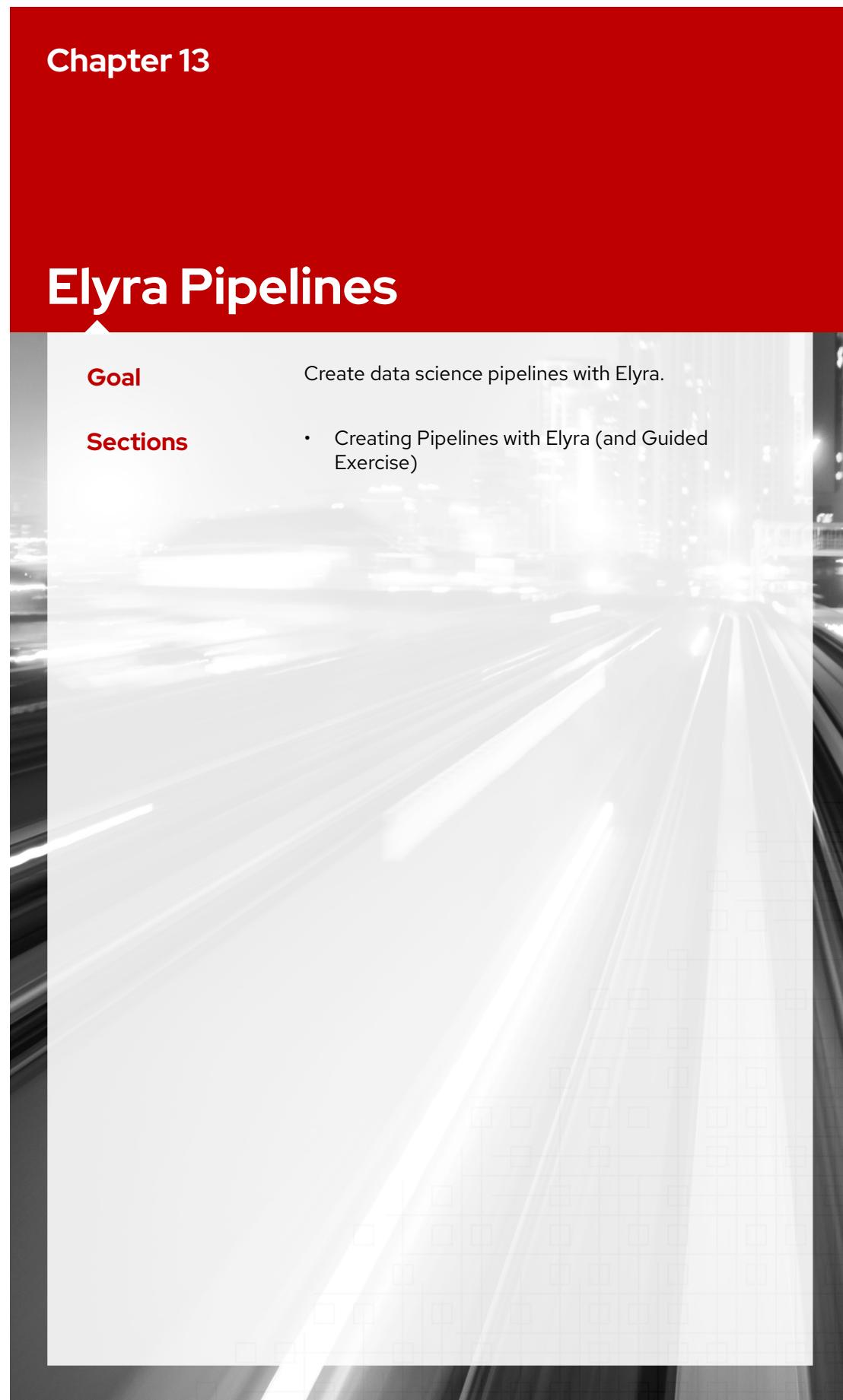
Elyra Pipelines

Goal

Create data science pipelines with Elyra.

Sections

- Creating Pipelines with Elyra (and Guided Exercise)



Creating Pipelines with Elyra

Objectives

- Create a data science pipeline in JupyterLab by using the Elyra extension.

Elyra Pipelines

Elyra is an AI-centric toolkit that extends the JupyterLab notebooks functionalities. Elyra provides a visual pipeline editor for building pipelines from Python and R scripts as well as Jupyter notebooks. Elyra simplifies pipeline creation by converting multiple files into connected nodes, defining their dependencies, and the execution order.

The Elyra visual pipeline editor enables you to assemble pipelines by dragging supported files onto a canvas. To run the pipeline, Elyra converts the visual representation of the pipeline into a Tekton YAML definition and submits this definition to Red Hat OpenShift Pipelines.

Red Hat OpenShift AI (RHOAI) includes Elyra in four of its predefined workbench images:

- Standard Data Science
- PyTorch
- TensorFlow
- TrustyAI

Creating a Data Science Pipeline with Elyra

To create a pipeline with Elyra, follow these steps:

- Launch a JupyterLab workbench with the Elyra extension installed.
- Create a pipeline by clicking on the **Elyra Pipeline Editor** icon in the launcher view of a workbench.
- Drag notebooks or scripts from the file browser onto the pipeline editor canvas.
- Connect the nodes to define the execution flow.
- Configure each node by right-clicking on it, clicking **Open Properties**, and setting the appropriate runtime image and file dependencies.
- You can also inject environment variables, secrets, and define output files.
- Submit your Elyra pipeline to the data science pipelines engine.

Chapter 13 | Elyra Pipelines

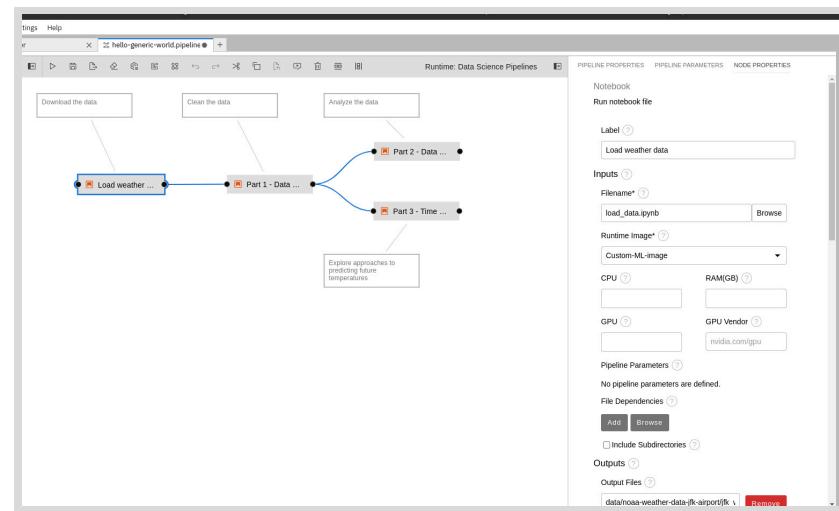


Figure 13.1: Pipeline visual edition from a notebook



Note

To use the Elyra pipelines, the RHOAI cluster depends on the Red Hat OpenShift Pipelines operator. For more information about RHOAI component dependencies, see the AI263: Red Hat OpenShift AI Administration course.

Elyra Runtime Configuration

A runtime configuration provides Elyra access to the data science pipelines back end for pipeline execution. You can manage runtime configurations by using the JupyterLab UI or the Elyra CLI. The runtime configuration is preconfigured for submitting pipelines to data science pipelines. Refer to the Elyra documentation for more information about Elyra and the available runtime configuration options.



Important

To submit pipelines, you must first create a pipeline server, which is managed via a `DataSciencePipelineApplication` resource object.

Runtime Images

RHOAI uses three types of container images:

Notebook images

Images for workbenches

Runtime images

Images for running each pipeline node

Model serving images

Images for model servers

You can create custom runtime images by following the same process as with notebook images. See the references section for more information about runtime images.

Using Custom Runtime Images

After you create your own specialized, custom runtime images, you can use them in RHOAI. Import a custom runtime image to provide the scripts in your nodes with the language runtime and required dependencies. To add a custom runtime image, open the **Runtime Images** menu from the left toolbar, and click the + plus sign icon at the panel's top.

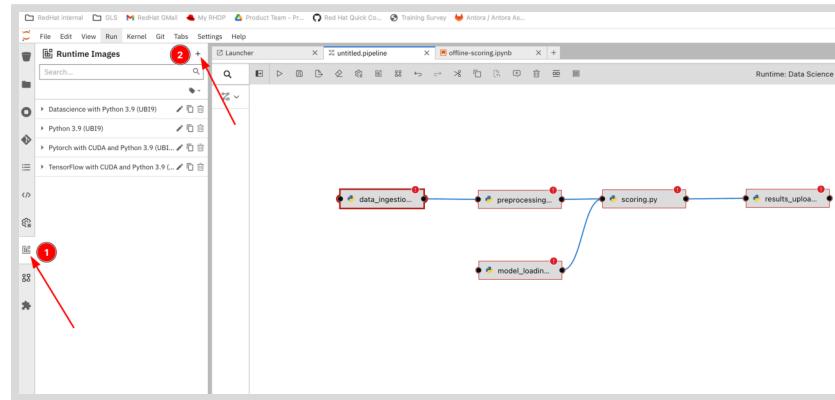


Figure 13.2: Adding a runtime image

1. Complete the form with the required values:

Display Name

The name of the image as shown in the Elyra UI.

Image Name

The fully qualified name that RHOAI uses to pull the image.

This screenshot shows the 'Add new Runtime Image runtime image' dialog box. It has fields for 'Display Name' (set to 'fraud detection runtime'), 'Description', 'Tags', and 'Source'. Under 'Source', there is a 'Image Name*' field containing 'quay.io/murukam/runtimes:fraud-detection-v0.2.0', an 'Image Pull Policy' dropdown, and an 'Image Pull Secret' field. At the bottom are 'Save & Close' buttons.

Figure 13.3: Form to add a runtime image

After you define the runtime image, you can configure all nodes in your pipeline to use it. To do this, open the pipeline settings in the Elyra pipeline editor via the **Open Panel** icon in the top right corner of the editor. You can also define the runtime image for each of the pipeline nodes.

- Select the **PIPELINE PROPERTIES** tab of the settings menu. Configurations in this section apply defaults to all nodes in the pipeline.
- Scroll down to the **Generic Node Defaults** section, and click the drop-down menu of **Runtime Image**. Select the image that you want to use.

Configuring the Pipeline to Use a Data Connection

You can inject S3 configuration from a data connection into a pipeline. To do so, define the following four entries on the PIPELINE PROPERTIES tab of the settings menu:

- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY
- AWS_S3_ENDPOINT
- AWS_S3_BUCKET

Each of these entries must include the following three options:

- Environment Variable: the parameter name
- Secret Name: the name of the OpenShift secret that contains the parameters of the RHOAI data connection
- Secret Key: the parameter name



Note

If you create the RHOAI data connection from the RHOAI dashboard, then the secret name is always preceded by aws-connection-.

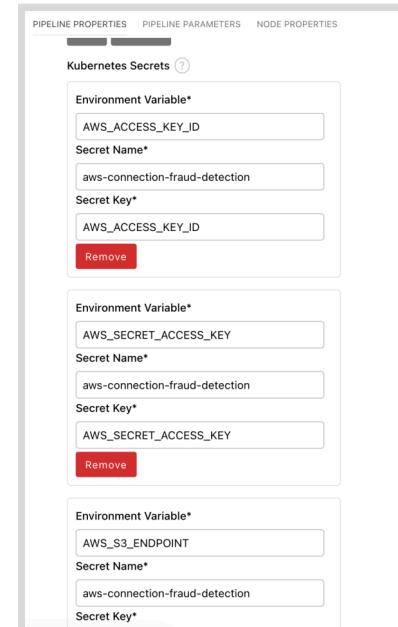


Figure 13.4: S3 configuration through node secrets

Connecting the Pipeline Nodes

To create pipeline nodes, drag notebooks, Python files, or R files to the Elyra canvas. To connect nodes, click the Output Port black dot of the first node, and drag the line towards the Input Port black dot of the next node.

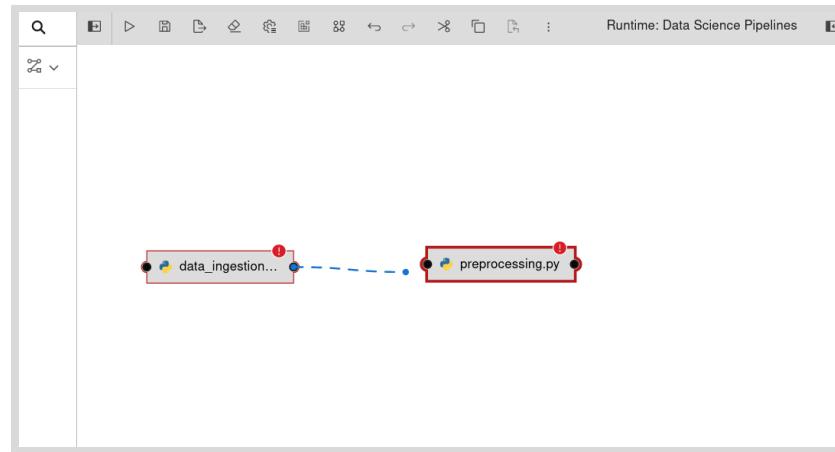


Figure 13.5: Connecting the pipeline nodes

Configuring Node Outputs

To configure the data to be passed between the nodes, click the node that you want to configure. If you are in the configuration menu, then you see the NODE PROPERTIES tab. If not, right-click the node, and select Open Properties.

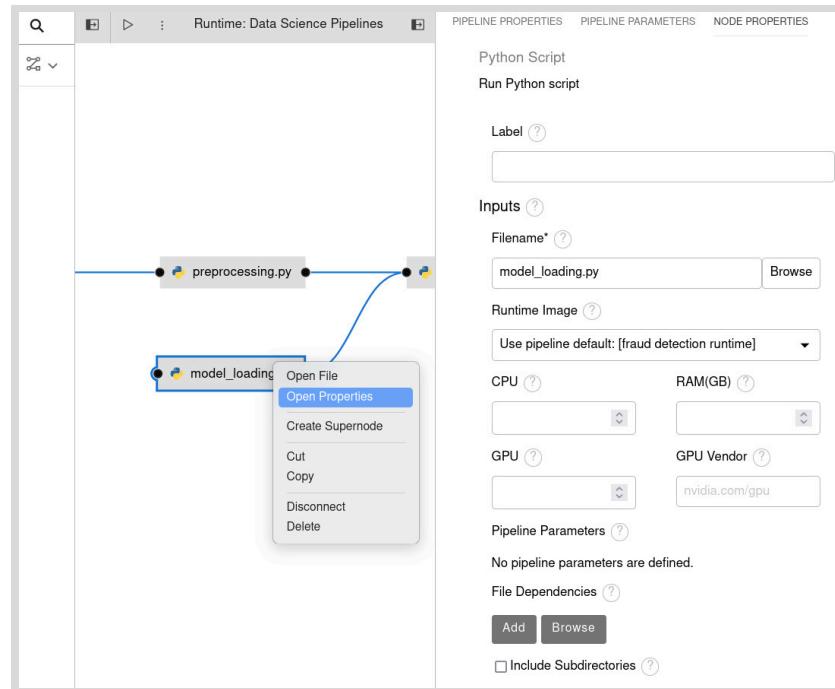


Figure 13.6: Menu to open the node properties configuration

- Under Runtime Image and Kubernetes Secrets, you can see that the global pipeline settings are used by default.
- In the Outputs section, you can declare one or more output files. These output files are created by this pipeline task and are made available to all subsequent tasks.

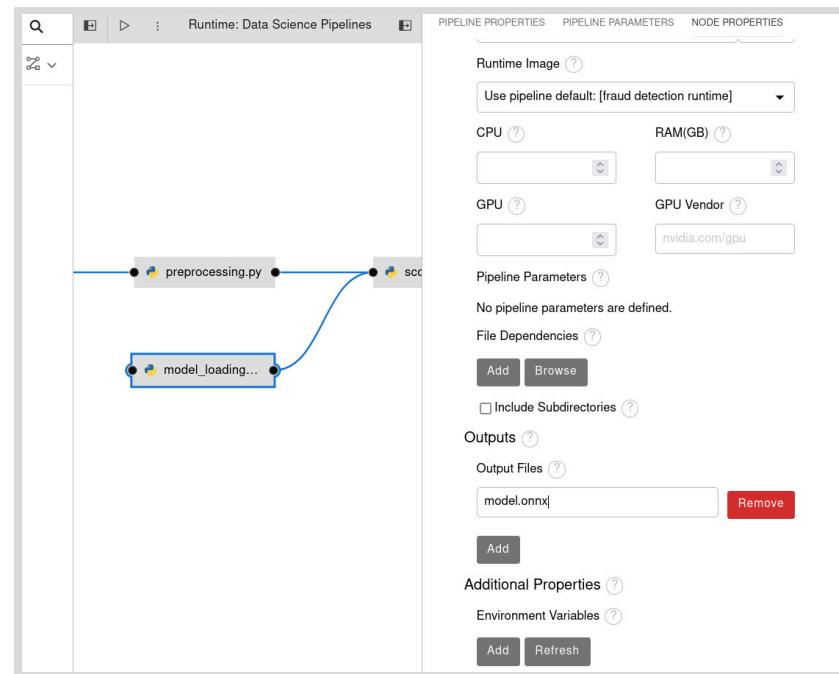


Figure 13.7: Configuring output files for the node

By default, all files within a containerized task are removed after its execution, so declaring files explicitly as output files is one way to ensure that they can be reused in downstream tasks.

RHOAI manages the output files of pipelines automatically. RHOAI stores these files in the S3 bucket of the data connection that you have used to create the `DataSciencePipelineApplication` server.

In addition to using S3 for passing data between nodes, you can also mount volumes.

1. Configure a data volume to allow the nodes to store additional data as a mounted volume.

In the `NODE_PROPERTIES` section of the node, scroll to the bottom of the `NODE_PROPERTIES` panel, and click `Add` in the `Data Volumes` section. You must define the following mandatory configuration options:

- **Mount Path:** The path to the mounted volume inside the container.
- **Persistent Volume Claim Name:** The name of a persistent storage defined on the data science project.

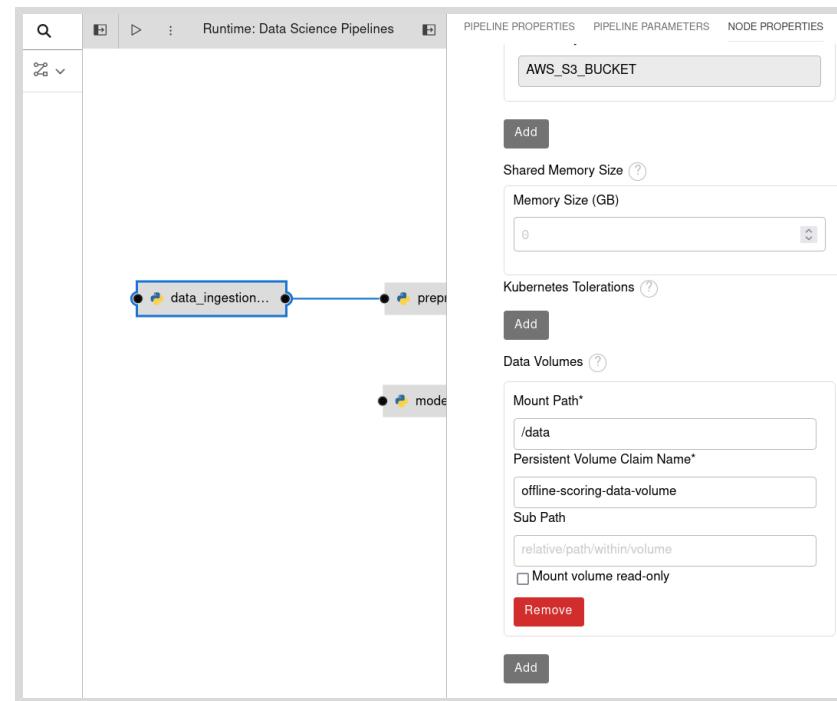


Figure 13.8: Node data volumes configuration

2. Repeat the same Data Volumes configuration for all the tasks in the pipeline.



Note

Mount Volumes and Output Files both provide the ability for files to persist between tasks, and each has different strengths and weaknesses.

Output Files are generally easy to configure and do not require the creation of any additional kubernetes resources. One disadvantage is that output files can generate a large amount of additional read and writes to S3, which might slow down pipeline execution.

Mount Volumes can be helpful when a large amount of files, or a large dataset is required to be stored. Mount Volumes also have the ability to persist data between runs of a pipeline, which can allow a volume to act as a cache for files between executions.



Note

You can declare a data volume as a global pipeline property, for simplicity. However, this prevents parallel execution of model loading and data ingestion/preprocessing because data volumes can only be used by a single task by default.

Renaming and Saving the Pipeline

You can rename the pipeline by right clicking the pipeline tab name, and then **Rename pipeline**. To save the changes, click the disk icon on the pipeline editor upper toolbar.

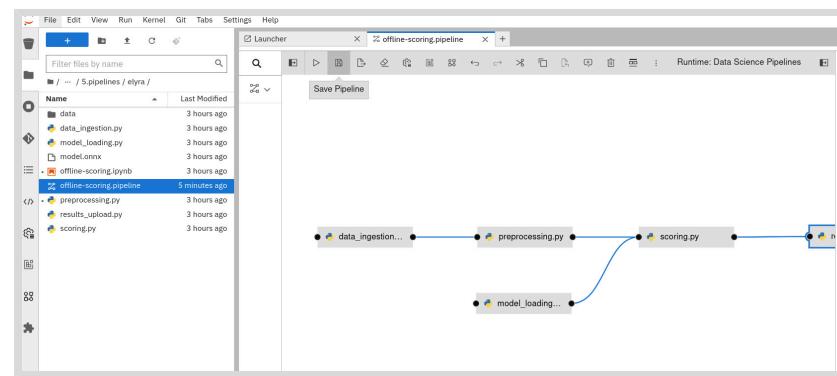


Figure 13.9: Save pipeline button

Running the Pipeline

To run a pipeline, click Run Pipeline on the pipeline editor upper toolbar. Elyra converts the pipeline definition to a YAML manifest, and sends it to the pipeline server. Then, Elyra shows a submission confirmation message.

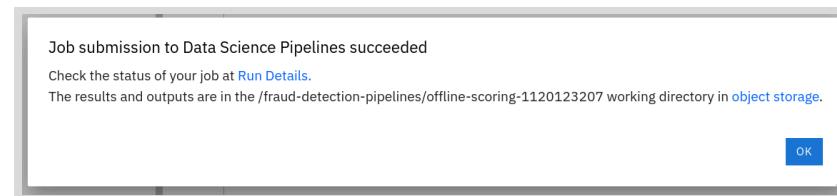


Figure 13.10: Confirmation of the pipeline submission

You can monitor the pipeline execution by clicking the Run details link on the confirmation message.

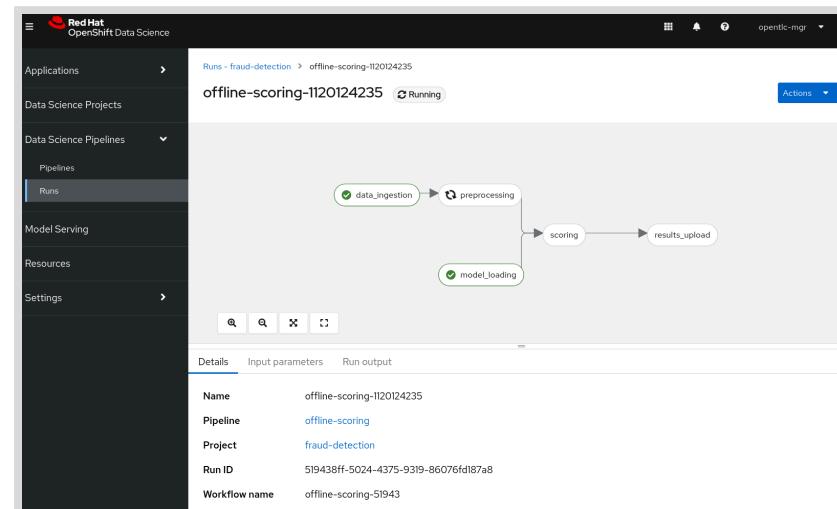


Figure 13.11: Pipeline running

**Important**

If you forget to create the pipeline server before creating the workbench, then the server is not available in the workbench. Restarting the workbench might not solve the problem.

To solve the issue, you must recreate the workbench pod. You can achieve this effect by stopping the workbench, and then editing the workbench by adding a new environment variable.

Inspecting Pipeline Results

Elyra pipelines store artifacts on the S3 bucket of the data connection assigned to the pipelines server. The pipeline runs create the following directories on the S3 bucket:

pipelines

stores the pipeline definitions in YAML manifests.

artifacts

stores the metadata of the pipeline run.

pipeline-name-timestamp

stores dependencies, log files, and output files of each node from each pipeline run.



References

Elyra Community GitHub Repository

<https://github.com/elyra-ai/elyra>

Elyra Community Documentation: CLI

https://elyra.readthedocs.io/en/v3.15.0/user_guide/command-line-interface.html

Quay repository for RHOAI runtime images

<https://quay.io/repository/modh/runtime-images>

GitHub repository for OpenDataHub Runtime Images

<https://github.com/opendatahub-io/notebooks/tree/main/runtimes>

Quay repository for community-contributed runtime images

<https://quay.io/repository/opendatahub-contrib/runtime-images>

GitHub repository for community-contributed runtime images

<https://github.com/opendatahub-io-contrib/workbench-images?tab=readme-overview#runtime-images>

Elyra Community Documentation: Runtime Configuration

https://elyra.readthedocs.io/en/latest/user_guide/runtime-conf.html#kubeflow-pipelines-configuration-settings

For more information, see the *Working with Pipelines in JupyterLab* section in the *Working on Data Science Projects* guide in the Red Hat OpenShift AI Self-managed 2.8 documentation at

https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/working_on_data_science_projects/index#working_with_pipelines_in_jupyterlab

► Guided Exercise

Creating Pipelines with Elyra

Create a data science pipeline in JupyterLab by using the Elyra extension.

Outcomes

- Create an Elyra pipeline that ingests, preprocesses, and makes predictions about a time series data set.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI266 && lab start elyra-pipeline
```

Instructions

In this exercise, you create an Elyra pipeline to estimate the number of incoming support cases that can arrive in to an issues management system in the next four weeks. The pipeline has three nodes:

- `data_ingestion.py` reads all comma-separated values (CSV) support cases data files from the S3 bucket.
- `data_preprocessing.py` prepares the data.
- `data_training_and_forecasting.py` reads the cleaned data, and applies a time series regression analysis model to predict the number of future support cases.

The pipeline uses a persistent storage, and a custom runtime image, common to the three nodes.

The `elyra-pipeline-bucket` S3 resource contains the `/data` directory with the CSV files needed by the `data_ingestion.py` script node. You can access the S3 resource by navigating to <https://minio-minio.apps.ocp4.example.com>. Log in by using `minio` as the access key, and `minio123` as the secret key.

- 1. In the `elyra-pipeline` Data Science Project, create a new data connection called `s3-minio`.
 - 1.1. In a web browser, open a new tab and navigate to <https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com>. Log in as the `developer` user by using the `developer` password.
 - 1.2. From the left navigation pane, click **Data Science Projects** and click the project called `elyra-pipeline`.
 - 1.3. From the **Data connections** section, click **Add data connection**.
 - 1.4. Use the following values to complete and submit the form.

Field	Value
Name	s3-minio
Access key	minio
Secret key	minio123
Endpoint	http://minio-minio.apps.ocp4.example.com
Region	any
Bucket	elyra-pipeline-bucket

Click Add data connection.

- ▶ 2. In the elyra-pipeline Data Science Project, create a pipeline server.

In the **Access key** field, click the key icon to select s3-minio. Then, ensure that the **Bucket** field contains elyra-pipeline-bucket. Leave the other fields with their default values, and click **Configure pipeline server**.

Wait until OpenShift configures a new pipeline server with the selected configuration.

- ▶ 3. Create a small workbench called elyra-pipeline-wb with a data connection pointing to the S3 bucket.

3.1. From the workbenches section, click **Create workbench**

3.2. Enter elyra-pipeline-wb as the name and select Standard Data Science as the image. Do not submit the form yet.

3.3. Add an environment variable by clicking **Add variable**.

Select the **Config Map** type, and then select **Key / value**. Set the Key to PIPELINES_SSL_SA_CERTS and the value to /var/run/secrets/kubernetes.io/serviceaccount/ca.crt.



Note

The PIPELINES_SSL_SA_CERTS variable for the workbench points to the internal OpenShift certificate authority. This enables Elyra to access HTTPS services identified by OpenShift certificates.

3.4. Scroll down to the **Cluster storage** section, and leave **Create new persistent storage** selected. Reduce the persistent storage capacity to 10 Gi.

3.5. At the bottom of the form, select **Use a data connection**.

Click **Use existing data connection**, and select s3-minio. Then, click **Create workbench**.

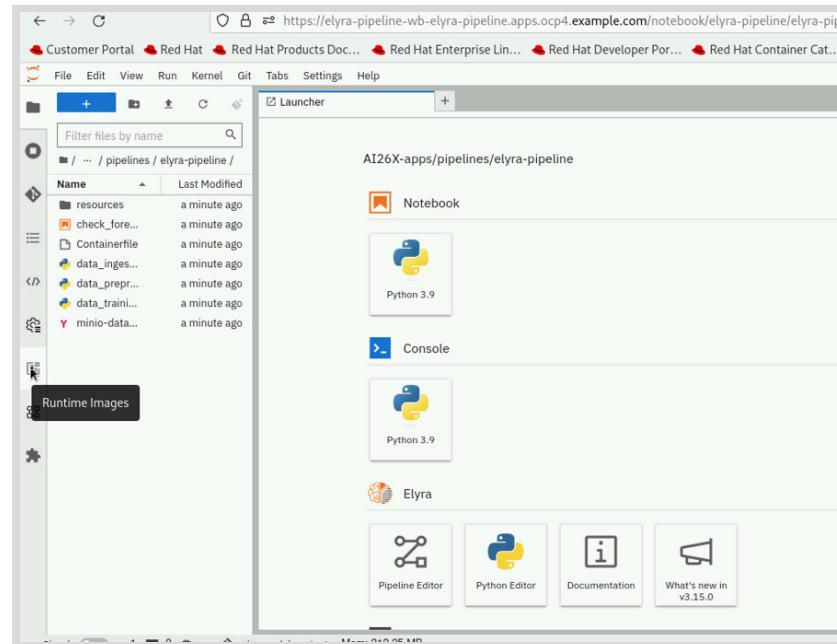
- ▶ 4. Open the elyra-pipeline-wb workbench as the developer user, and clone the git repo with the code needed for the pipeline.

4.1. In the Workbenches section, click **Open**.

Authenticate with the **htpasswd_provider** option, as the developer user by using password developer. Accept permissions by clicking **Allow selected permissions**.

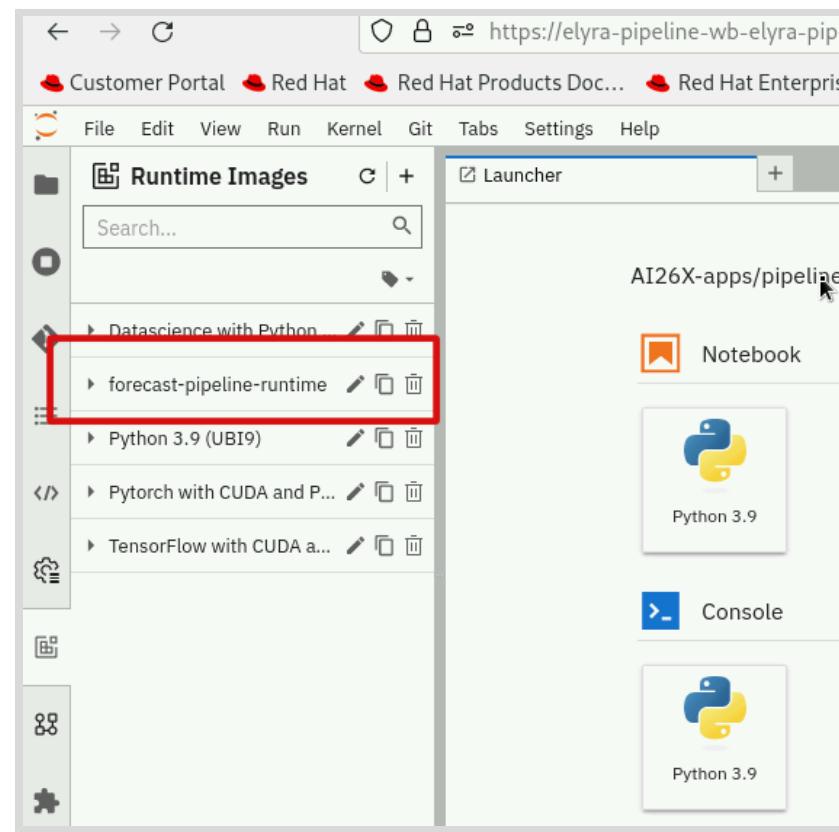
- 4.2. In the JupyterLab interface, click the **Git** icon from the left tools pane.
 - 4.3. Click **Clone a repository**.
 - 4.4. Enter <https://github.com/RedHatTraining/AI26X-apps> as the repository, and click **Clone**.
 - 4.5. In the JupyterLab file browser, verify the new content by navigating to the AI26X-apps/pipelines/elyra-pipeline directory.
- 5. Create a pipeline runtime image.

- 5.1. In the left pane, click **Runtime Images**.



- 5.2. Add a new **Runtime Image** by clicking the **+** icon.
- 5.3. Introduce **forecast-pipeline-runtime** as the **Display Name** field, and **quay.io/redhattraining/forecast-pipeline-runtime:1.0** as the **image name**.

Then, click **Save & Close**.

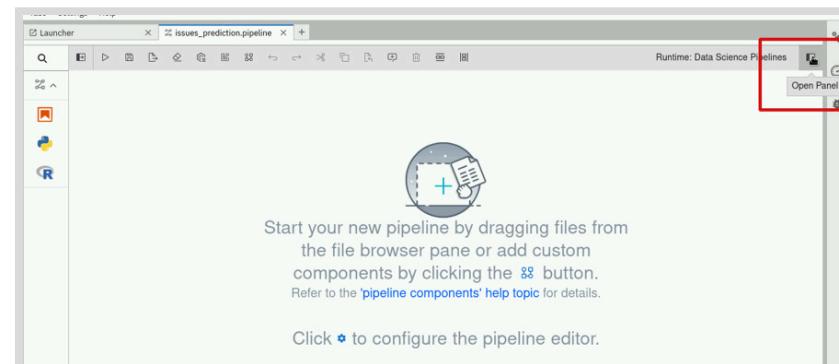


- ▶ 6. Create the pipeline, configure the access to the data connection, and set the common image runtime for all nodes.

- From the launcher view, in the Elyra section, click **Pipeline Editor**.

The Elyra canvas for pipelines opens. Right click the pipeline tab, and rename the pipeline to `issues_prediction.pipeline`.

- Open the pipeline configuration panel by clicking **Open Panel**, next to the Runtime: Data Science Pipelines label.



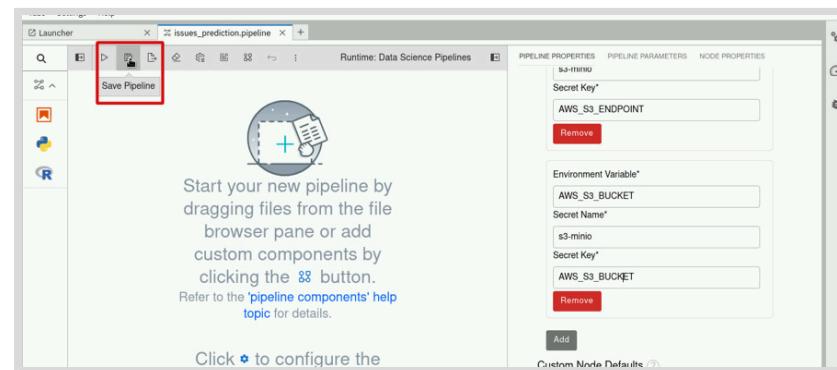
- Scroll down to the **Generic Node Defaults** section, and select `forecast-pipeline-runtime` as the **Runtime Image**.
- Click **Add** in the **Kubernetes Secrets** section, and add the following entries:

Environment Variable	Secret Name	Secret Key
AWS_ACCESS_KEY_ID	aws-connection-s3-minio	AWS_ACCESS_KEY_ID
AWS_SECRET_ACCESS_KEY	aws-connection-s3-minio	AWS_SECRET_ACCESS_KEY
AWS_S3_ENDPOINT	aws-connection-s3-minio	AWS_S3_ENDPOINT
AWS_S3_BUCKET	aws-connection-s3-minio	AWS_S3_BUCKET

**Note**

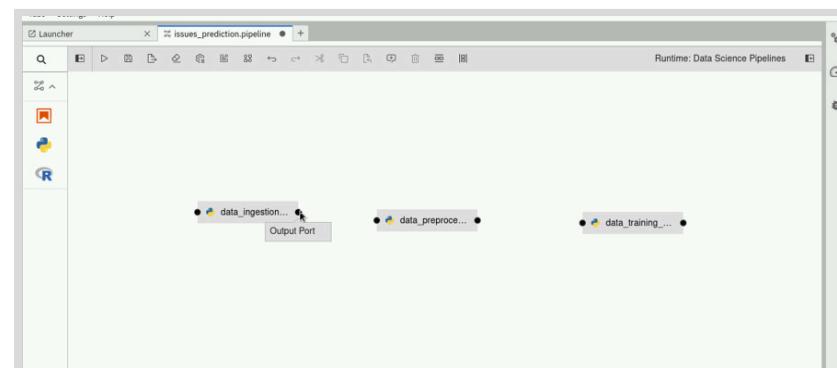
Note that you use `aws-connection-` as a prefix for the secret's name. Although created with `s3-minio` as the display name, the OpenShift internal name includes the prefix.

6.5. Save the pipeline by clicking **Save pipeline**.



► 7. Define and connect the pipeline nodes.

- 7.1. In the file browser, ensure that you are in the `AI26x-apps/pipelines/elyra-pipeline` directory. Next, drag the `data_ingestion.py`, `data_preprocessing.py`, and `data_training_and_forecasting.py` scripts to the visual pipeline editor.
- 7.2. Connect the nodes by clicking on the **Output Port** black dot of the origin node, and dragging to the **Input Port** of the target node.

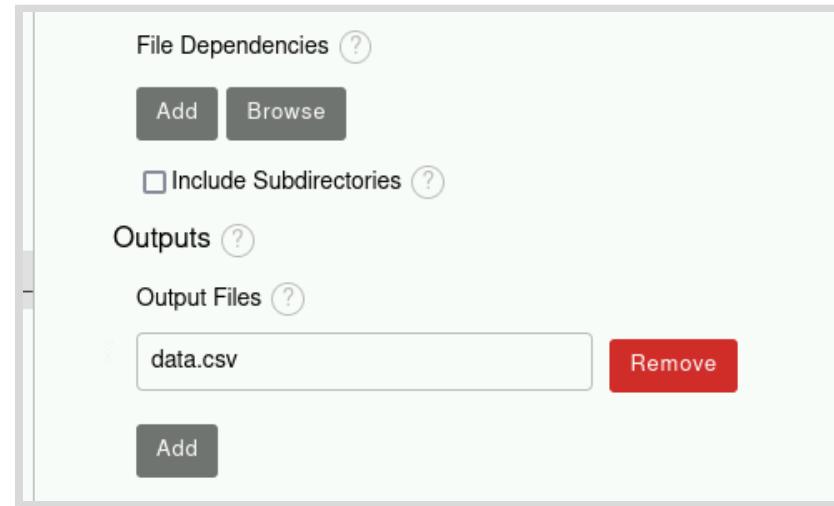


Connect from `data_ingestion.py` to `data_preprocessing.py`, and then connect from `data_preprocessing.py` to

`data_training_and_forecasting.py`. By default, Elyra uses the output artifacts of one node as the input artifacts of the next node.

- 7.3. Configure the output path for the `data_ingestion.py` node.

Right click the `data_ingestion.py` node, and select **Open Properties**. In the **Outputs** section add `data.csv`



- 7.4. Configure the output path for the `data_preprocessing.py` node.

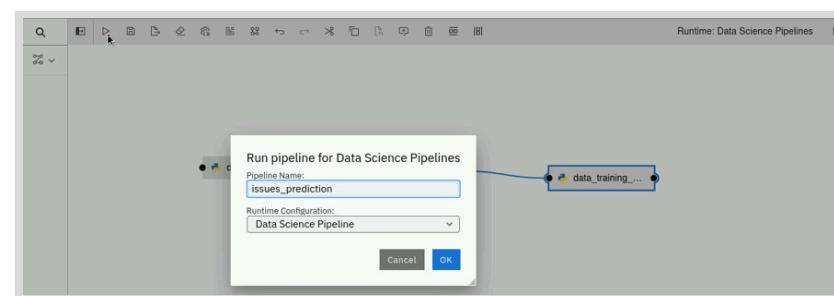
Right click the `data_preprocessing.py` node, and select **Open Properties**. In the **Outputs** section add `clean-data.csv`.

- 7.5. Configure the output path for the `data_training_and_forecasting.py` node.

Right click the `data_training_and_forecasting.py` node, and select **Open Properties**. In the **Outputs** section add `forecast-data.csv`

► 8. Run the pipeline.

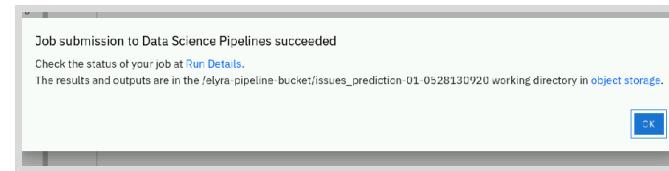
- 8.1. Save the pipeline, and click the play button icon with the **Run Pipeline** text.



Modify the **Pipeline Name** to be `issues_prediction-01`, and click **OK**.

Elyra sends the pipeline definition to the pipeline server runtime. The pipeline server creates the pipeline run.

- 8.2. Click **Run Details** to access the pipeline **Runs** panel in the Red Hat OpenShift AI (RHOAI) dashboard.



- 8.3. Wait for the pipeline to finish.

All three nodes should pass.

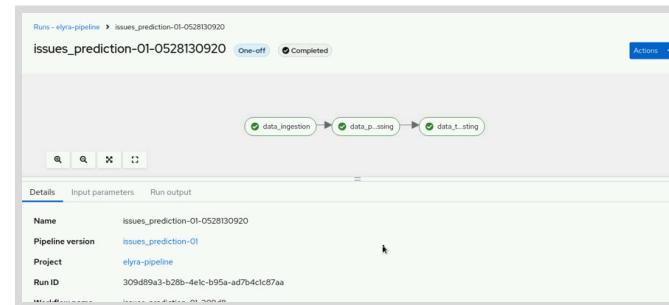


Figure 13.20: The Elyra pipeline after a successful run

- 9. Inspect the results in the S3 bucket.

- 9.1. Access the S3 resource by navigating to <https://minio-minio.apps.ocp4.example.com>. Log in using minio as the access key, and minio123 as the secret key.
- 9.2. Inspect the files in the `issues_prediction-01-timestamp` directory of the S3 bucket.
Within this directory, you can inspect the log file from each node, and the three output files. Inspect the three output files from each node:

- `data.csv`
- `clean-data.csv`
- `forecast-data.csv`

- 10. (Optional) Visualize the forecasted values.

- 10.1. In the `elyra-pipeline-wb` workbench, open the `check_forecast.ipynb` notebook by double clicking it.
- 10.2. In the second cell of the notebook, replace `EDIT THIS` with the directory name generated by the pipeline run.
- 10.3. Execute all the notebook cells. The notebook generates a graph to visualize the past and the forecasted time-series data.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish elyra-pipeline
```

Summary

- Elyra is a JupyterLab extension that provides a graphical UI for creating data science pipelines.
- You can use Elyra to create pipeline nodes by dragging and dropping Jupyter notebooks, Python, and R scripts to the pipeline editor.
- You can define custom runtime images for node execution with Elyra.
- With Elyra, you can submit a pipeline to the pipeline server for execution.

Chapter 14

Kubeflow Pipelines

Goal

Create data science pipelines with Kubeflow Pipelines.

Sections

- Creating Pipelines with Kubeflow Pipelines (and Guided Exercise)

Creating Pipelines with Kubeflow Pipelines

Objectives

- Create a data science pipeline with the Kubeflow Pipelines SDK.

Kubeflow Pipelines SDK

Red Hat OpenShift AI (RHOAI) offers two mechanisms to build and run data science pipelines:

- Elyra, a JupyterLab extension that provides a visual editor for creating pipelines based on Jupyter notebooks, Python, and R scripts.
- The Kubeflow Pipelines (KFP) SDK, which is discussed in this section.

With the KFP SDK, you can define pipelines in Python, and then submit them to the data science pipeline runtime for execution. Programmatic pipelines offer a flexible and powerful way to automate complex processes and workflows. This approach also has the added benefit of offering all the tooling, frameworks, and developer experience that comes with Python. Another benefit of using this approach is that you can share and version control the source code of your pipelines. This is especially useful for GitOps.



Note

GitOps is a set of practices that combine the DevOps principles with Git. This approach uses Git as the repository to manage infrastructure as code (IaC), which results in better team collaboration and automation.

Kubeflow Pipelines SDK for Tekton

In RHOAI 2.8, the data science pipelines runtime is based on *Red Hat OpenShift Pipelines*.

OpenShift Pipelines is an operator that provides OpenShift clusters with continuous integration and continuous delivery (CI/CD) capabilities. This operator is based on the *Tekton* upstream, a community project for building cloud-native CI/CD systems.

To integrate the KFP SDK with OpenShift Pipelines, RHOAI relies on the *Kubeflow Pipelines SDK for Tekton* project, which is available as the `kfp-tekton` Python package. This package provides the standard KFP SDK as well as additional tools to integrate the SDK with Tekton and, consequently, with OpenShift Pipelines and RHOAI.

The Kubeflow Pipelines Process

To build data science pipelines with KFP, you must follow these steps:

1. Develop the pipeline in Python by using the KFP SDK.
2. Convert the Python code of the pipeline into a Tekton YAML manifest.
3. Import the YAML into a data science project in RHOAI.

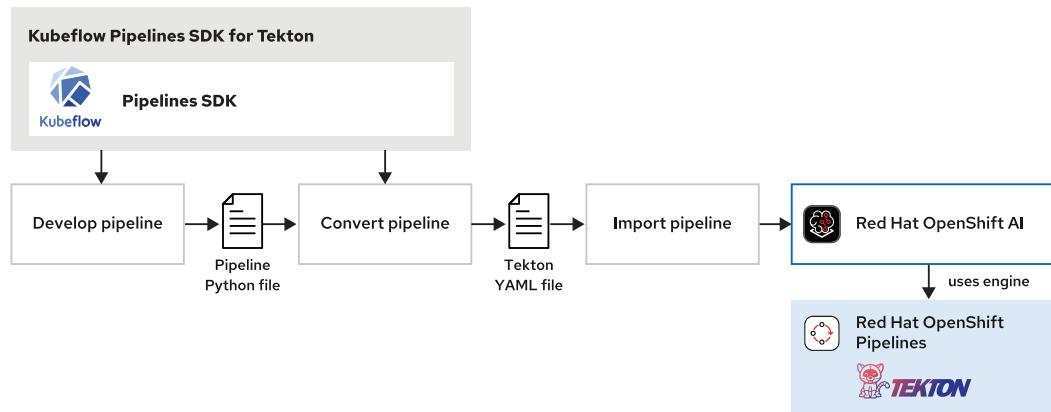


Figure 14.1: Components and steps of the Kubeflow Pipelines process

**Note**

From version 2.9, RHOAI switches the engine from OpenShift Pipelines/Tekton to Argo Workflows, for better alignment with the Kubeflow Pipelines upstream project.

Developing Pipelines

First, make sure that the `kfp-tekton` package is installed. To ensure that you create pipelines compatible with RHOAI 2.8, install the latest of the `1.5.x` versions of the package, as follows:

```
[user@host ~]$ pip install kfp-tekton~=1.5.9
```

**Warning**

Using the correct versions of `kfp-tekton` is critical to avoid conflicts between the KFP SDK and RHOAI versions.

- The recommendation is to install the latest of the `1.5.x` versions of the package.
- Installing versions from `1.8.x` results in failures during pipeline runs.
- Other versions between `1.5` and `1.8` might work, such as `1.7.x`, but have not been extensively tested.

The `kfp-tekton` package provides the `kfp` module. Import this module in your Python scripts to use the KFP SDK.

Components

Start by defining the phases of your data science pipeline. Examples of pipeline phases are data cleaning, preprocessing, model training, and model upload.

The KFP SDK calls these phases *components*. Each of these pipeline components runs as a container. Thus, each component, or phase, can use a different container image.

To define a component, use the `kfp.components.create_component_from_func` function in a Python script. For example, the following code defines a component that preprocesses data:

```
from kfp import components

def preprocess_data(path: str): ❶
    ...code omitted...
    return "path/to/clean_data.csv" ❷

preprocess_data_op = components.create_component_from_func(
    preprocess_data, ❸
    base_image='registry.access.redhat.com/ubi9/python-39' ❹
)
```

- ❶ Define the function that implements the component logic. To add input parameters to the component, define them as regular function parameters.
- ❷ The return value of the function is the return value of the component.
- ❸ Call `create_component_from_func` by passing the function that contains the logic of the component. The return value is a function with the same signature as the original function, which in this case is `preprocess_data`.
- ❹ The second parameter is the container image that RHOAI must use to run this pipeline component.

Pipelines

Next, define a pipeline by composing the components that you have created. To do so, first create a Python function that represents the pipeline. You must apply the `dsl.pipeline` decorator to the function, as follows:

```
from kfp import dsl

@dsl.pipeline(name="my-pipeline", description="A simple pipeline")
def my_pipeline():
    ...code omitted...
```

You can now implement the body of the pipeline function by calling the components that you have created before.

```
@dsl.pipeline(...)
def my_pipeline():
    task1 = preprocess_data_op("/path/to/dataset.csv") ❶
    task2 = train_model_op(training_set_path)

    model_path = task2.output ❷
    task3 = evaluate_model_op(model_path, test_set_path)
```

- ❶ Call a component by using the same signature as the function that you used to define the component.
- ❷ Access the return value of a component by using the `output` property of the value returned by the component. The preceding example assumes that the `train_model_op` returns the path of the trained model.

Control Flow, Inputs, Outputs, and Advanced Features

Optionally, you can manually control the flow of your pipeline. For example, the following excerpt uses a condition to push the model to S3 only if the accuracy metric is greater than 0.9:

```
def my_pipeline():
    ...code omitted...
    task3 = evaluate_model_op(model_path, test_set_path)
    accuracy, model_path = task3.output
    with dsl.Condition(accuracy > 0.9):
        push_model_op(model_path)
```

The KFP SDK supports additional instructions to perform the following operations:

- Instructions for control flow, such as loops and conditions
- Data passing methods include persistent volume claims and Tekton workspaces
- Node selectors, taints, and tolerations
- Input parameter passing and output values returned as files

For more information about these features, refer to the Kubeflow Pipelines v1 documentation.

Exporting a Pipeline to Tekton Format

After you create a pipeline in Python, you must convert it to a YAML manifest. This manifest defines the Tekton resources required for the pipeline to run on RHOAI. To compile the pipeline, use the `dsl-compile-tekton` command provided by `kfp-tekton`, as follows:

```
[user@host ~]$ dsl-compile-tekton \
--py my_pipeline.py \ ❶
--output my_pipeline.yaml ❷
```

- ❶ The path of the Python script that defines the pipeline function.
- ❷ The path of the generated YAML file.

Alternatively, you can embed the compilation step in the same Python script where you define the pipeline. You can do this by using the `kfp_tekton.compiler.TektonCompiler` class.

```
from kfp import dsl
from kfp_tekton.compiler import TektonCompiler

@dsl.pipeline(...)
def my_pipeline():
    ...code omitted...

if __name__ == "__main__":
    compiler = TektonCompiler() ❶
    compiler.compile(my_pipeline, "output.yaml") ❷
```

- ❶ Create an instance of a KFP Tekton compiler.
- ❷ Compile the pipeline function to a Tekton YAML manifest.

If you use the `TektonCompiler` class, then you do not need the `dsl-compile-tekton` command. To compile the pipeline, you can just run the Python script.

```
[user@host ~]$ python3 my_pipeline.py
```

Importing a Pipeline into a Data Science Project

To add the pipeline YAML to your data science project, you can use the RHOAI dashboard. Navigate to your data science project and then, under the **Pipelines** section, click **Import Pipeline**.



Note

Before you can import pipelines, you must first create a pipeline server, which is managed via a `DataSciencePipelineApplication` resource object.

In the form that shows, enter a pipeline name and optionally a description. Next, click **Upload** to upload the YAML file that you compiled in the previous step.

A screenshot of a modal dialog box titled 'Import pipeline'. Inside the dialog, there are several input fields and a code editor area. The 'Project' field contains 'my-project'. The 'Pipeline name *' field contains 'my-pipeline'. The 'Pipeline description' field contains 'My first pipeline'. Below these fields is a code editor containing a YAML file named 'my_pipeline.yaml'. The code in the editor is as follows:

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: coin-toss-pipeline
  annotations:
    tekton.dev/output_artifacts: '{"flip-coin": {"key": "artifacts/$PIPELINERUN/flip-coin/Output.tgz",
```

At the bottom of the dialog are two buttons: a blue 'Import pipeline' button and a white 'Cancel' button.

Finally, click **Import pipeline**.

The imported pipeline should now be visible under the pipelines section. To run the pipeline, click the : pipeline menu and then click **Create run**.

The screenshot shows the Kubeflow Pipelines interface. At the top, there's a header with 'Pipelines', an 'Import pipeline' button, and a dropdown. Below the header is a table with columns: Pipeline name, Versions, Created, and Updated. There is one entry: 'my-pipeline' (My first pipeline), Version 1, created and updated 3 minutes ago. To the right of the table is a context menu with options: 'Upload new version', 'Create run' (highlighted with a blue border), and 'Delete pipeline'. Below the table is a section titled 'Models and model servers' with a 'Add model server' button. At the bottom of the interface is a table with columns: Model Server Name, Serving Runtime, Deployed models, and Tokens.

In the form, enter a name for the pipeline run. You can also select whether to run the pipeline immediately or schedule it. If the pipeline has input parameters, then you can specify them in this form.

To run or schedule the pipeline, click **Create**. RHOAI displays a page with the details of the pipeline run.



References

Kubeflow Pipelines v1 Documentation

<https://www.kubeflow.org/docs/components/pipelines/v1/>

Kubeflow Pipelines on Tekton

<https://github.com/kubeflow/kfp-tekton>

Kubeflow Pipelines SDK for Tekton

<https://github.com/kubeflow/kfp-tekton/tree/master/sdk>

For more information, refer to the *Defining a Pipeline* section in the *Working on Data Science Projects* chapter in the Red Hat OpenShift AI Self-managed 2.8 *Working on Data Science Projects* documentation at

https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/working_on_data_science_projects/index#define-a-pipeline_nb-server

For more information, refer to the *Running a Data Science Pipeline Generated from Python Code* section in the *Implementing Pipelines* chapter in the Red Hat OpenShift AI Self-managed 2.8 *OpenShift AI Tutorial - Fraud Detection Example* documentation at

https://access.redhat.com/documentation/en-us/red_hat_openshift_ai_self-managed/2.8/html-single/openshift_ai_tutorial_-_fraud_detection_example/index#running-a-pipeline-generated-from-python-code

► Guided Exercise

Creating Pipelines with Kubeflow Pipelines

Create a data science pipeline with the Kubeflow Pipelines SDK.

Outcomes

- Install the Kubeflow Pipelines (KFP) SDK CLI.
- Create an S3 storage to persist pipeline artifacts.
- Create and execute a simple Kubeflow pipeline.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your environment for this exercise, and to ensure that all required resources are available. This command also ensures that you use the lab scripts specific to this course.

```
[student@workstation ~]$ lab install -u AI266 && lab start kubeflow-pipeline
```

Instructions

► 1. Install the KPF SDK CLI in a Python virtual environment.

- 1.1. Change to the exercise directory.

```
[student@workstation ~]$ cd ~/AI266/kubeflow-pipeline
```

- 1.2. Create and activate a Python virtual environment.

```
[student@workstation kubeflow-pipeline]$ python3 -m venv .venv  
[student@workstation kubeflow-pipeline]$ source .venv/bin/activate
```

- 1.3. Use the `pip` command to install the KFP Tekton CLI version 1.7.2.

```
(.venv) [student@workstation kubeflow-pipeline]$ pip install kfp-tekton==1.7.2  
...output omitted...
```



Note

kfp-tekton version 1.7.2 implements the KFP DSL version 1.8.22.

► 2. Define the pipeline components by using Python code.

- 2.1. Open the `kubeflow-pipeline.py` file with your favourite editor, and examine the content. The file contains methods to print messages, generate random numbers, and flip a coin.
- 2.2. Create a component from the `flip_coin` function, and store it in a variable called `flip_coin_op`. The component must use `registry.access.redhat.com/ubi9/python-39` as the base image.

```
...code omitted...

# TODO: Create a component from the flip_coin function
flip_coin_op = components.create_component_from_func(
    flip_coin, base_image='registry.access.redhat.com/ubi9/python-39')

...code omitted...
```

- 2.3. Create a component from the `print_msg` function, and store it in a variable called `print_op`. The component must use `registry.access.redhat.com/ubi9/python-39` as the base image.

```
...code omitted...

# TODO: Create a component from the print_msg function
print_op = components.create_component_from_func(
    print_msg, base_image='registry.access.redhat.com/ubi9/python-39')

... code omitted...
```

- 2.4. Create a component from the `random_num` function, and store it in a variable called `random_num_op`. The component must use `registry.access.redhat.com/ubi9/python-39` as the base image.

```
...code omitted...

# TODO: Create a component from the random_num function
random_num_op = components.create_component_from_func(
    random_num, base_image='registry.access.redhat.com/ubi9/python-39')

...code omitted...
```

► 3. Define a pipeline in Python.

- 3.1. Decorate the `flipcoin_pipeline` function with the `@dsl.pipeline` decorator to define a pipeline. Set `coin-toss-pipeline` as the pipeline name, and `A simple pipeline` as the description.

```
...code omitted...

# TODO: Decorate function to define a pipeline
@dsl.pipeline(
    name='coin-toss-pipeline',
    description='A simple pipeline'
)
def flipcoin_pipeline():
    ...code omitted...
```

- 3.2. Add a condition named `heads-result` that is executed when the `flip.output` value is `heads`. Inside the condition, print the message `Heads!`, and then, define a variable named `random_number` that stores a random number between 0 and 9. You must use the `random_num_op` component to generate the random number.

```
...code omitted...
def flipcoin_pipeline():
    flip = flip_coin_op()
    # TODO: Add pipeline logic
    with dsl.Condition(flip.output == 'heads', 'heads-result'):
        print_op('Heads!')
        random_number = random_num_op(0, 9)
```

- 3.3. Inside the `heads` condition, add a condition that prints `A high value!` when the `random_number.output` is greater than 7, and print `A low value!` otherwise.

```
...code omitted...
def flipcoin_pipeline():
    flip = flip_coin_op()
    # TODO: Add pipeline logic
    with dsl.Condition(flip.output == 'heads', 'heads-result'):
        print_op('Heads!')
        random_number = random_num_op(0, 9)
        with dsl.Condition(random_number.output > 7):
            print_op('A high value!')
        with dsl.Condition(random_number.output <= 7):
            print_op('A low value!')
```

- 3.4. Add a condition named `tails-result` that is executed when the `flip.output` value is `tails`. Inside the condition, print the message `Tails result`.

```
...code omitted...
def flipcoin_pipeline():
    flip = flip_coin_op()
    # TODO: Add pipeline logic
    with dsl.Condition(flip.output == 'heads', 'heads-result'):
        print_op('Heads!')
        random_number = random_num_op(0, 9)
        with dsl.Condition(random_number.output > 7):
            print_op('A high value!')
        with dsl.Condition(random_number.output <= 7):
```

```
    print_op('A low value!')  
  
    with dsl.Condition(flip.output == 'tails', 'tails-result'):   
        print_op('Tails result')
```

- ▶ 4. Compile the Python file into a Tekton resource definition.
- 4.1. Return to the terminal window, and run the following command from within the virtual environment.

```
(.venv) [student@workstation kubeflow-pipeline]$ dsl-compile-tekton \  
--py kubeflow-pipeline.py \  
--output pipeline.yaml
```

The preceding command generates a YAML file named `pipeline.yaml` that contains a Tekton PipelineRun resource.

- 4.2. Inspect the `pipeline.yaml` file to understand how the kfp-tekton SDK transformed the Python pipeline definition into a runnable Tekton PipelineRun resource.
- 4.3. Exit the Python virtual environment.

```
(.venv) [student@workstation kubeflow-pipeline]$ deactivate
```

- ▶ 5. Deploy and configure a MinIO server to provide storage for the pipeline artifacts.

- 5.1. In the same terminal, log in to the cluster with the `oc` CLI.

```
[student@workstation kubeflow-pipeline]$ oc login -u developer -p developer \  
https://api.ocp4.example.com:6443  
...output omitted...  
Login successful.  
...output omitted...
```

- 5.2. Change to the `kubeflow-pipeline` project.

```
[student@workstation kubeflow-pipeline]$ oc project kubeflow-pipeline  
...output omitted...
```

- 5.3. Apply the provided `minio.yaml` manifest file. This creates several Kubernetes resources, including the MinIO pod, service, and routes.

```
[student@workstation ~]$ oc apply -f minio.yaml  
...output omitted...
```

- 5.4. In a web browser, navigate to the MinIO UI by using the address from the route.

```
[student@workstation ~]$ oc get route minio-ui -o jsonpath='{.spec.host}'; echo  
minio-ui-kubeflow-pipeline.apps.ocp4.example.com
```

- 5.5. Log in to the UI by using the `minio` username with password `minio123`.

- 5.6. Because no bucket exists, the UI prompts you to create one. Click **Create a Bucket**, provide the name `kubeflow-pipeline-bucket`, and click **Create Bucket**. Leave all other form values as their default.
- 6. In the `kubeflow-pipeline` Data Science Project, create a new data connection called `kubeflow-pipeline-connection`.
- 6.1. In a web browser, open a new tab and navigate to `https://rhods-dashboard-redhat-ods-applications.apps.ocp4.example.com`. Log in as the `developer` user by using the `developer` password.
 - 6.2. From the left navigation pane, click **Data Science Projects** and click the project called `kubeflow-pipeline`.
 - 6.3. From the **Data connections** section, click **Add data connection**.
 - 6.4. Use the following values to fill out and submit the form.

Field	Value
Name	<code>kubeflow-pipeline-connection</code>
Access key	<code>minio</code>
Secret key	<code>minio123</code>
Endpoint	<code>http://minio-api-kubeflow-pipeline.apps.ocp4.example.com</code>
Region	<code>us-east-1</code>
Bucket	<code>kubeflow-pipeline-bucket</code>

- 7. Import the pipeline.
- 7.1. From the **Pipelines** section, click **Configure pipeline server**.
 - 7.2. In the **Access key** list select `kubeflow-pipeline-connection`. Then, ensure that the **Bucket** field, contains `kubeflow-pipeline-bucket`. Finally, click **Configure pipeline Server**.
 - 7.3. Wait until OpenShift configures a new pipeline server with the selected configuration.
 - 7.4. Click **Import pipeline**, in the **Pipeline name** field, enter `toss-a-coin-v1`. Click **Upload**, and select the `/home/student/AI266/kubeflow-pipeline/pipeline.yaml` file. Finally, click **Import pipeline**
- 8. Execute the pipeline.
- 8.1. Expand the `toss-a-coin-v1` pipeline, and then click the `toss-a-coin-v1` pipeline version.
 - 8.2. Click **Actions** and then click **Create run**.
 - 8.3. In the **Name** field, enter `toss-a-coin-run-1`, click **Run once immediately after creation** on the **Run type** section, and finally click **Create**.

- 8.4. Wait until the pipeline run finishes.
- 8.5. Click each pipeline task, and examine the input and output data, the details of each task, and the logs.
- 8.6. Optionally, run the pipeline multiple times to generate different execution outcomes.

Finish

On the workstation machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish kubeflow-pipeline
```

Summary

- You can use Kubeflow Pipelines to define complex pipelines with Python.
- Red Hat OpenShift AI (RHOAI) 2.8 uses Tekton as the data science pipelines runtime.
- To import a pipeline in RHOAI, you must first compile the Python pipeline script into a Tekton YAML manifest.
- Use the `kfp-tekton` Python package to define pipelines with the Kubeflow Pipelines SDK and compile them to Tekton.