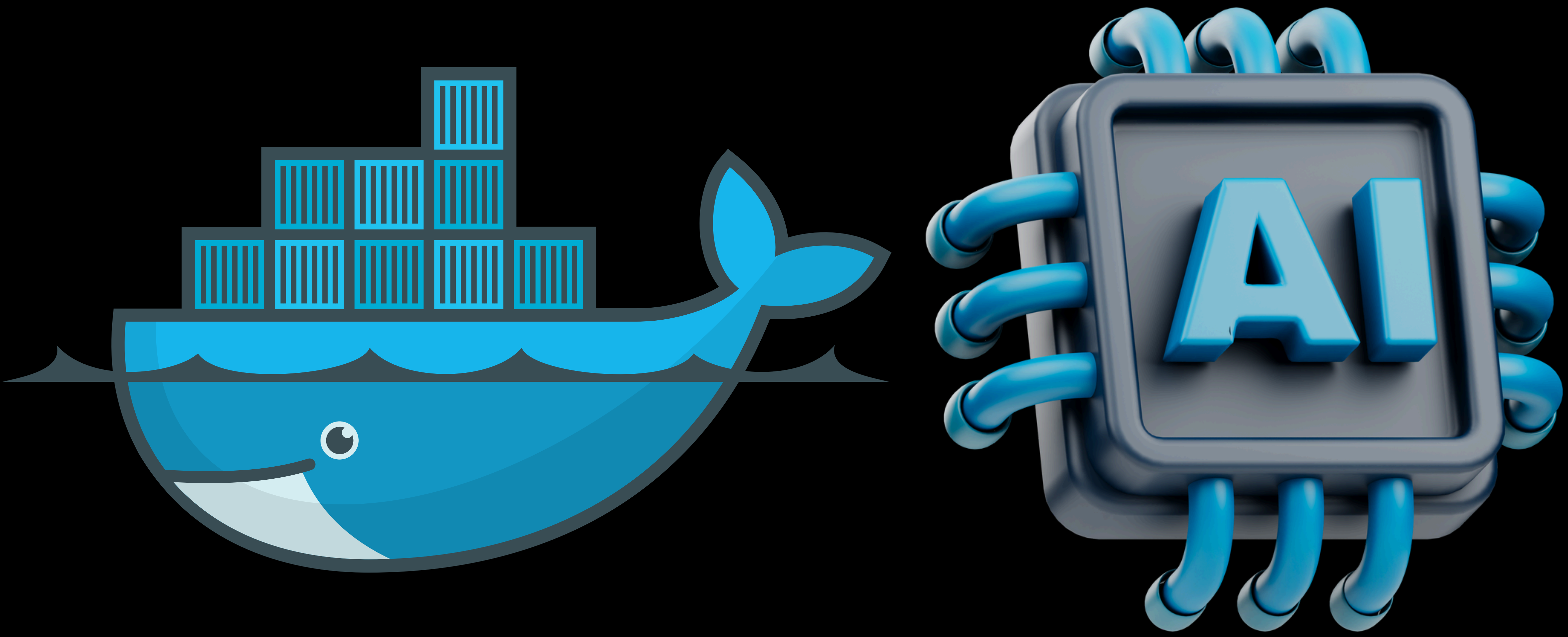
Hands-on Docker for Data Science & AI Projects



To Data & Beyond

Youssef Hosni

Hands-on Docker for Data Science & AI Projects

Hands-on Docker for Data Science & AI Projects

By: Youssef Hosni

To Data & Beyond

Brief of Contents

Table of Contents

About this book

Are you a data scientist, AI practitioner, or machine learning engineer grappling with environment inconsistencies, reproducibility nightmares, or the infamous "it works on my machine" syndrome? Do you want to streamline your development workflows, collaborate more effectively, and deploy your AI models with greater ease and confidence? If so, Hands-on Docker for Data Science & AI Projects is your definitive guide to mastering containerization for your specific needs.

In the rapidly evolving landscape of data science and artificial intelligence, managing complex dependencies, ensuring consistent environments across different machines, and simplifying deployment are paramount. Docker has emerged as an indispensable tool to address these challenges, and this book is designed to equip you with the practical skills to leverage its full potential for your projects.

We begin with a Beginner-Friendly Introduction to Docker for Data Science Projects, demystifying core concepts like containers versus images and highlighting why Docker is crucial for data scientists. You'll quickly get Docker installed and dive straight into Dockerizing your first Machine Learning Application, learning how to define an environment, write a Dockerfile, and build your image.

Next, we explore a curated list of 11 Essential Docker Container Images for Generative AI & ML Projects. This chapter serves as a practical toolkit, showcasing pre-built environments for

Python, Jupyter, Hugging Face Transformers, NVIDIA CUDA, TensorFlow, PyTorch, Ollama, Qdrant, Airflow, MLflow, and Kubeflow Notebooks, saving you valuable setup time.

Building on this foundation, Chapter 3 guides you through Building your first comprehensive Docker container for Machine Learning Applications, reinforcing your understanding with practical steps. To ensure your Docker practices are efficient and professional, Chapter 4 delves into Best Practices, covering crucial topics like minimizing layers, using official images, optimizing with multi-stage builds, persisting data with volumes, and effectively organizing and versioning your images.

Finally, to empower your daily workflow, Chapter 5 presents 10 Docker Commands for 90% of Containerization Tasks. This practical reference will make you proficient in commands like docker run, docker build, docker ps, and docker push, enabling you to manage your containers with confidence.

Whether you're new to Docker or looking to specifically apply it to the unique demands of data science and AI, this book provides a hands-on, step-by-step approach. By the end, you'll be adept at building, managing, and deploying containerized data science applications, paving the way for more reproducible, scalable, and collaborative projects.

About the author

Youssef Hosni is a data scientist and machine learning researcher who has been working in machine learning and AI for more than half a decade. In addition to being a researcher and data science practitioner, Youssef has a strong passion for education. He is known for his leading data science and AI blog, newsletter, and eBooks on data science and machine learning.

Youssef is a senior data scientist at Ment focusing on building Generative AI features for Ment Products. He is also an AI applied researcher at Aalto University working on mutlimodat agents and their applications for next generation smart cities . Before that, he worked as a researcher in which he applied deep learning and computer vision techniques to medical images.

1. Docker for Data Science Projects: A Beginner-Friendly Introduction

When shipping your machine learning code to the engineering team, encountering compatibility issues with different operating systems and library versions can be frustrating.

Docker can solve compatibility issues between operating systems and library versions when shipping machine learning code to engineering teams, making code execution seamless regardless of its underlying setup.

In this comprehensive tutorial, we will introduce Docker's essential concepts, guide you through installation, demonstrate its practical use with examples, uncover industry best practices, and answer any related queries along the way — so say goodbye to compatibility woes and streamline machine learning workflow with Docker!

Chapter Content:

1.1. Introduction to Docker

1.1.1. Docker vs Containers vs Docker Images

Docker is a commercial containerization platform and runtime that helps developers build, deploy, and run containers. It uses a client-server architecture with simple commands and automation through a single API.

With Docker, developers can create containerized applications by writing a Dockerfile, which is essentially a recipe for building a container image. Docker then provides a set of tools to build and manage these container images, making it easier for developers to package and deploy their applications in a consistent and reproducible way.

A container is a lightweight and portable executable software package that includes everything an application needs to run, including code, libraries, system tools, and settings.
Containers are created from images that define the contents and configuration of the container, and they are isolated from the host operating system and other containers on the same system.

This isolation is made possible by the use of virtualization and process isolation technologies, which enable containers to share the resources of a single instance of the host operating system while providing a secure and predictable environment for running applications.
A Docker Image is a read-only file that contains all the necessary instructions for creating a container. They are used to create and start new containers at runtime.

1.1.2. Importance of Docker for Data Scientists

Docker lets developers access these native containerization capabilities using simple commands and automate them through a work-saving application programming interface (API).

Docker offers:

- **Improved and seamless container portability**: Docker containers run without modification across any desktop, data center, or cloud environment.
- **Even lighter weight and more granular updates**: Multiple processes can be combined within a single container. This makes it possible to build an application that can continue running while one of its parts is taken down for an update or repair.
- **Automated container creation**: Docker can automatically build a container based on application source code.
- **Container versioning**: Docker can track versions of a container image, roll back to previous versions, and trace who built a version and how. It can even upload only the deltas between an existing version and a new one.
- **Container reuse**: Existing containers can be used as base images — essentially like templates for building new containers.
- **Shared container libraries**: Developers can access an open-source registry containing thousands of user-contributed containers.

1.2. Getting Started with Docker

Now, after introducing Docker, let's see how we can use it for our data science projects. Let's first start with installing Docker on your local machine, and after that, we will introduce basic Docker commands.

1.2.1. Installing Docker on Your Machine

Installing Docker on your machine is fairly easy. You can follow the instructions available on the official documentation:

- Instructions to install Docker for **Linux**.
- Instructions to install Docker for **Windows**.
- Instructions to install Docker for **Mac**.

It is important to note that if you like to create your own images and push them to Docker Hub, you must create an account on Docker Hub. Think of Docker Hub as a central place where developers can store and share their Docker images.

1.2. Dockerizing a Machine Learning Application

To dockerize a machine learning application, there are three main steps:

- Create a requirements.txt file
- Write a Dockerfile
- Build the Docker image

Let's build a simple machine learning application and see a step-by-step guide on how to dockerize it. The application below trains a simple classification model (logistic regression) on the iris dataset.

```
# Load the libraries

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score


# Load the iris dataset

iris = load_iris()

X = iris.data

y = iris.target


# Split the data
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Train a logistic regression model
clf = LogisticRegression()
clf.fit(X_train, y_train)

# Make predictions
y_pred = clf.predict(X_test)

# Print the accuracy of the model
accuracy_score = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy_score}')
```

1.2.1. Defining the environment

The first step is to define the current environment to be able to replicate it in another location. The most effective and the most straightforward way is to create a requirements.txt file that outlines all the libraries your project is using, including their versions. To create this file, you can simply run the following command in the command line:

```
pip3 freeze > requirements.txt  # Python3
```

1.2.2. Write a Dockerfile

The next step is to create a file named Dockerfile that can create the environment and execute our application in it.

```
FROM python:3.9

WORKDIR /src
```

```
COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["python","iris_classification.py"]
```

This Dockerfile uses the official Python image as the base image, sets the working directory, copies the requirements.txt file, installs the dependencies, copies the application code, and runs the python iris_classification.py command to start the application.

1.2.3. Build the Image

The final step to create a reproducible environment is to create an image (also known as a template) that can be run to create any number of containers with the same configurations.

You can build the image by running the command **docker build -t <image-name>** . in the same directory where the Dockerfile is located.

2. 11 Docker Container Images for Generative AI & ML Projects

Docker containers offer significant advantages for machine learning by ensuring consistent, portable, and reproducible environments across different systems.

By encapsulating all dependencies, libraries, and configurations in a container, Docker eliminates compatibility issues and the “it works on my machine” problem.

This makes it easier to move ML projects between development, cloud, or production environments without worrying about differences in setup. Additionally, Docker enables scalability and isolation, allowing machine learning workflows to be easily scaled using tools like Kubernetes, and ensuring that dependencies do not conflict between different projects.

In this chapter, we will explore 11 Docker container images for Generative AI and machine learning projects. These include tools for development environments, deep learning frameworks, machine learning lifecycle management, workflow orchestration, and large language models.

Chapter Content:

2.1. Python

This is as simple and practical as it gets for starting a machine learning or data processing project.

```
FROM python:3.8
```

```
RUN pip install --no-cache-dir numpy pandas
```

Here’s what each line does:

- **FROM python:3.8:** This line sets the base image to Python 3.8. That means your container will come pre-installed with Python 3.8 and a minimal Linux OS underneath (usually Debian or Alpine, depending on the tag). It’s a good choice when you want to control exactly what gets installed next.
- **RUN pip install — no-cache-dir numpy pandas**: This line installs two essential Python libraries:
    1. **numpy**: for numerical computations (arrays, matrices, etc.)
    2. **pandas**: for handling tabular data and time series
- **The — no-cache-dir flag is a small optimization**: it prevents pip from saving the downloaded .whl files in the container, which helps keep the image size smaller.

Use it when you’re writing lightweight scripts, preprocessing data, or building simple ML pipelines. It’s fast to build, easy to extend, and perfect for tasks that don’t need heavy frameworks like TensorFlow or PyTorch.

2.2. Jupyter Notebook data science stack

This command spins up a ready-to-use JupyterLab environment loaded with popular data science tools — all inside a Docker container.

```
docker run -it --rm -p 8888:8888 jupyter/datascience-notebook
```

Here’s a breakdown of what’s happening:

- docker run: Starts a new container from the image you specify.
- -it: Stands for interactive terminal. This keeps the container attached to your terminal session, so you can interact with it if needed.
- – rm: Automatically deletes the container once it’s stopped. Super handy for one-off sessions where you don’t need to persist anything after you’re done.
- -p 8888:8888: Maps port 8888 on your local machine to port 8888 inside the container—this is how you’ll access the Jupyter interface from your browser.
- jupyter/datascience-notebook: This is the official image from the Jupyter project. It includes:
  1. JupyterLab interface
  2. Python 3
  3. Libraries like numpy, pandas, scikit-learn, matplotlib, seaborn, and even statsmodels and bokeh.

This will provide you with a full-featured data science workbench in your browser with zero setup. You don’t need to worry about installing dependencies or conflicting Python versions — it all just works inside the container.

When to use this image:

- When you’re exploring datasets
- Teaching or learning data science
- Prototyping ML models
- Collaborating on notebooks without polluting your local Python environment.

2.3. Hugging Face Transformers

Hugging Face Transformers is a popular library used for everything from working with large language models to creating image generation systems. It’s built on top of major deep learning frameworks like PyTorch and TensorFlow, so you can easily load models, fine-tune them, monitor performance, and save your progress directly to Hugging Face.

```
FROM huggingface/transformers-pytorch-gpu
RUN python main.py
```

Here’s what this simple Dockerfile does:

- **FROM huggingface/transformers-pytorch-gpu:** This base image is maintained by Hugging Face and comes preinstalled with:

  1. The transformers library
  2. datasets, tokenizers, and other related tools
  3. PyTorch with GPU (CUDA) support

     ○ It’s specifically designed to support training and inference of transformer models like BERT, GPT-2, T5, and many more.

- RUN python main.py: This line will run your Python script (main.py) as soon as the container starts. That script can load a pretrained model, fine-tune it on your data, or serve it via an API — totally up to you.

This image saves you a ton of setup time. No need to manually install PyTorch, transformers, or manage CUDA compatibility — everything is pre-configured and GPU-optimized.
When to use this image:

- Fine-tuning LLMs with PyTorch
- Running inference pipelines (e.g., summarization, Q&A, classification)
- Building model-serving apps using Hugging Face pipelines
- Prototyping Gen AI apps using Transformers

You can also use other variants if your project is based on TensorFlow instead.

```
FROM huggingface/transformers-tensorflow-gpu
```

2.4. NVIDIA CUDA deep learning runtime

The NVIDIA CUDA deep learning runtime is key to speeding up deep learning tasks on GPUs. By adding it directly to your Dockerfile, you can skip the hassle of setting up CUDA manually and get GPU-accelerated machine learning workflows up and running more easily.

```
FROM nvidia/cuda
RUN python main.py
```

This is a very basic starting point for building GPU-accelerated deep learning containers. Here's what's happening:

- FROM nvidia/cuda: This sets the base image to NVIDIA's official CUDA runtime. It gives you the drivers and libraries needed to interface with NVIDIA GPUs from within a container. By default, this base image:
- Doesn't include Python
- Doesn't include deep learning frameworks (like TensorFlow or PyTorch)
- Comes in many tags like:
    - nvidia/cuda:12.2.0-runtime-ubuntu22.04
    - nvidia/cuda:11.8.0-cudnn8-runtime

So, for production use, you'd usually choose a more specific CUDA version + runtime + OS.

- RUN python main.py: This line assumes you've already installed Python and your project's dependencies in a previous step, which this current version doesn't do yet.

This image is more like a blank slate with GPU support. You'd typically build on top of this image if:

- You want full control over the Python environment
- You're installing custom ML/DL libraries from source
- You're matching CUDA versions to specific model/training code requirements

2.5. TensorFlow

TensorFlow is another major deep learning framework that's widely used in both research and industry. It integrates smoothly with the Google ecosystem and supports a range of tools for experiment tracking and model deployment.

```
FROM tensorflow/tensorflow:latest
RUN python main.py
```

This Dockerfile gives you a quick and clean way to run TensorFlow-based projects. Let's break it down:

- **FROM tensorflow/tensorflow:latest**: This line pulls the latest official TensorFlow Docker image. By default, it includes:
    - TensorFlow (CPU version)
    - Python (usually 3.x)
    - Commonly used Python libraries (like numpy, six, protobuf)

If you want GPU support, you'd use:

```
FROM tensorflow/tensorflow:latest-gpu
```

or even more specific versions like:

```
FROM tensorflow/tensorflow:2.14.0-gpu
```

- **RUN python main.py:** This line runs your TensorFlow script — training a model, doing inference, or whatever task you've defined in main.py.

This image saves you from setting up TensorFlow manually on different systems (which can be a headache, especially with GPU drivers and CUDA). Plus, the image is actively maintained by the TensorFlow team, so it stays up to date and compatible.

When to use this image:

- Building and training TensorFlow models
- Serving TensorFlow models in production (via tensorflow/serving)
- Experimenting with deep learning workflows in a reproducible environment

If you want to have a JupyterLab pre-installed. You can use the following command to give you a ready-to-go Jupyter environment for TensorFlow development.

```
FROM tensorflow/tensorflow:latest-jupyter
```

2.6. PyTorch

PyTorch is a top deep learning framework, recognized for its modular approach to building neural networks. You can easily create a Dockerfile and run model inference without the need to manually install the PyTorch package using the pip command.

```
FROM pytorch/pytorch:latest
RUN python main.py
```

This image is the fastest way to get up and running with PyTorch in a consistent, repeatable way — no setup hassles, no version mismatches.

When to use it:

- Training or fine-tuning deep learning models (CV, NLP, etc.)
- Running inference on PyTorch models
- Prototyping transformer-based Gen AI workflows
- Collaborating on projects where you want to guarantee version consistency

If you’re working on notebooks, there’s an official Jupyter version too:

```
FROM pytorch/pytorch:latest
RUN pip install notebook && jupyter notebook --ip=0.0.0.0 --port=8888
--no-browser
```

2.7. Ollama

Ollama is built for deploying and running large language models locally. With the Ollama Docker, you can easily download and serve LLMs, making it simple to integrate them into your applications. It’s the most straightforward way to run both quantized and full large language models in production.

```
docker run -it --rm ollama/ollama
```

This container is specifically designed for Ollama’s LLMs, making it easy to run LLMs out of the box. It’s great for experimenting with large models and using them in production, all without worrying about managing complex setups.

When to use this image:

- When you want to quickly experiment with and run Ollama's pre-trained models.
- For deploying LLMs in a production setting where Ollama's framework is being used.
- For projects focused on generative text-based AI.

2.8. Qdrant

Qdrant is a vector search engine designed for deploying similarity search applications. It includes a dashboard and server, allowing you to quickly send and retrieve vector data.

```
docker run -it --rm -p 6333:6333 qdrant/qdrant
```

Qdrant provides an optimized way to handle vector data and perform similarity searches efficiently. It works well with embeddings from models like OpenAI's, Hugging Face's, or any custom models you've fine-tuned for your specific domain.

When to use this image:

- When you're working with large-scale semantic search applications.
- For building and querying a vector database for tasks like nearest neighbor search or recommendation engines.
- When you want to store and search through embeddings generated by AI models, such as text embeddings, image embeddings, etc.

Qdrant's API is flexible and can integrate seamlessly with other tools in your pipeline (like LangChain, Haystack, or direct LLMs). It's highly optimized for real-time searches over vectorized data.

2.9. Airflow

Airflow is a popular tool for orchestrating complex machine learning pipelines. It allows you to author, schedule, and monitor workflows as directed acyclic graphs (DAGs), making it essential for automating and managing data workflows in machine learning engineering.

```
docker run -it --rm -p 8080:8080 apache/airflow
```

Airflow is ideal for orchestrating complex workflows, including tasks related to data engineering, machine learning, and deployment pipelines. This Docker image provides a quick way to set up and run Airflow without the need for manual installations or configurations.

When to use this image:

- When you need to automate ETL (Extract, Transform, Load) processes.
- Orchestrating machine learning workflows (e.g., data preprocessing, model training, model serving).
- When you need a centralized dashboard to monitor and manage your workflows.

Airflow comes with a rich web UI that allows you to visualize your workflow's execution, retry failed tasks, check logs, and more. It's especially helpful when managing complex dependencies in your pipelines.

2.10. MLflow

MLflow is an open-source platform that helps manage the entire machine learning lifecycle, from experimentation and reproducibility to deployment. By running the MLflow server with the following command, you can store and version experiments, models, and artifacts. It also offers an interactive dashboard for managers to easily track experiments and models.

```
docker run -it --rm -p 9000:9000 ghcr.io/mlflow/mlflow
```

MLflow simplifies managing machine learning experiments. With its built-in UI, you can easily track models, parameters, and outputs across multiple experiments. It also allows you to deploy models and manage their versions.

When to use this image:

- To track machine learning experiments and log parameters, metrics, and artifacts.
- When working with collaborative teams, you can maintain reproducibility and version control.
- For managing models, creating model registries, and promoting models through stages (e.g., from development to production).

2. 11. Kubeflow Notebooks

Kubeflow Notebooks is part of the larger Kubeflow ecosystem, which is designed for running machine learning pipelines on Kubernetes. This specific image is tailored for working with PyTorch in a Jupyter environment, giving you a streamlined workflow for ML experimentation and model development. You can scale it across distributed systems if needed, especially with Kubernetes.

```
docker run -it --rm -p 8888:8888 kubeflownotebookswg/jupyter-pytorch
```

When to use this image:

- When working in a Kubernetes environment and you want an easy-to-deploy notebook interface for deep learning models (especially with PyTorch).
- For prototyping and running PyTorch models in Jupyter, while benefiting from Kubernetes' orchestration capabilities.
- When you need to run Jupyter notebooks in a cloud-native, scalable environment for ML experiments.

You can easily extend this image to include other frameworks or tools like TensorFlow, scikit-learn, or Hugging Face by installing them directly in the notebook.

3. Building your first Docker container for Machine Learning Applications

One of the major operational challenges I faced when starting my career as a Data scientist was the dependency-conflict issue. Back in college, a simple pip install was usually enough to get your code running without a hitch.

Aside from a few group projects, it was a one-man show. I was the developer, the only server (my trusty laptop), and the sole user. It was a perfectly balanced ecosystem that vanished the moment I stepped out into the real world.

The simple pip install would now turn into a regular nightmare called dependency conflict- one project needs this, while another needs that. Every update or fresh deployment was another gamble that threatened to break the existing setup and required me to invest a lot of time manually reconfiguring the environments.

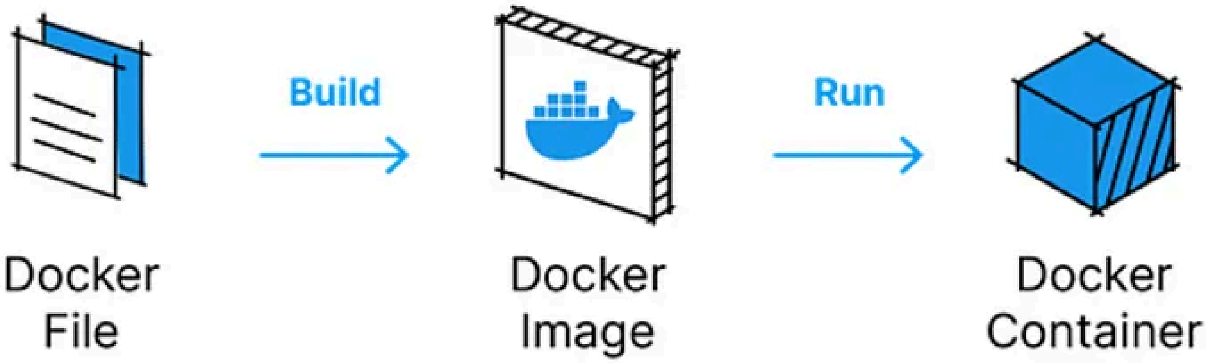
A trivial solution was to build a separate environment—using conda or venv—for each project. But every time we migrated to a new server, I had to redo the tedious work of recreating each environment from scratch.

Remember the quote “Work Smart, not hard”? This was the reason I began learning Docker and CI/CD for machine learning.

Docker is simply an open-source tool that enables developers to automate the tedious deployment process mentioned before by introducing the concept of containerization.In the Docker world, we have three concepts:

- Image
- Container
- DockerFile

A good analogy for containers and images is to think of them as your PC/Laptop and the installed OS, where docker image is a mini OS that contains everything your application needs to run, the container is the dedicated pc -or environment- that you create to run your application, and the DockerFile is the text file containing the instructions to build the container.



You can install Docker for the different operating systems:

- For Windows
- For Linux
- For MacOS

We will follow along using the Docker Desktop version. We will create a train.py where we generate dummy training data and save the fitted model to be used for inference in a Flask API. Here is the code for the train.py:

```
import numpy as np
import joblib
from sklearn.linear_model import LinearRegression
# Generate dummy data
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2, 4, 6, 8, 10])
# Train model
model = LinearRegression()
model.fit(X, y)
# Save model
joblib.dump(model, 'model.pkl')
print("Model saved!")
```

And this will be our Flask application code

```
from flask import Flask, request, jsonify
import joblib
import numpy as np
import logging
# Initialize Flask App
app = Flask(__name__)
# Configure Logging
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s
- %(message)s")
# Load Model
MODEL_PATH = "model.pkl"
try:
    model = joblib.load(MODEL_PATH)
    logging.info(f"✅ Model loaded successfully from {MODEL_PATH}")
except Exception as e:
    logging.error(f"❌ Error loading model: {e}")
    model = None

@app.route("/", methods=["GET"])
def home():
    """Root endpoint - API Welcome Message."""
    return jsonify({
        "message": "🚀 Welcome to the ML Prediction API!",
        "endpoints": {
            "Predict": "/predict (POST)"
        }
    })

@app.route("/predict", methods=["POST"])
def predict():
    """Endpoint to make predictions using the trained model."""
    if model is None:
        logging.error("Prediction request failed: Model not loaded.")
        return jsonify({"error": "Model not loaded"}), 500
    try:
        # Get JSON data from request
        data = request.json.get("input")
        if not data:
            logging.warning("Prediction request failed: Missing 'input'
data.")
            return jsonify({"error": "Missing 'input' data"}), 400
        # Convert input to NumPy array and make prediction
        prediction = model.predict(np.array(data).reshape(-1, 1))
        logging.info(f"Prediction successful: {prediction.tolist()}")
        return jsonify({"prediction": prediction.tolist()})
    except Exception as e:
        logging.error(f"Prediction error: {e}")
        return jsonify({"error": "Invalid input format"}), 400

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

The app.py is to have two main routes

- /home : Just a simple welcome page.
- /predict : Endpoint that we will use for inference.
-

For this simple project, here are our requirements.txt

```
flask~=3.1.0
pandas
numpy~=2.2.2
joblib~=1.4.2
scikit-learn~=1.6.1
```

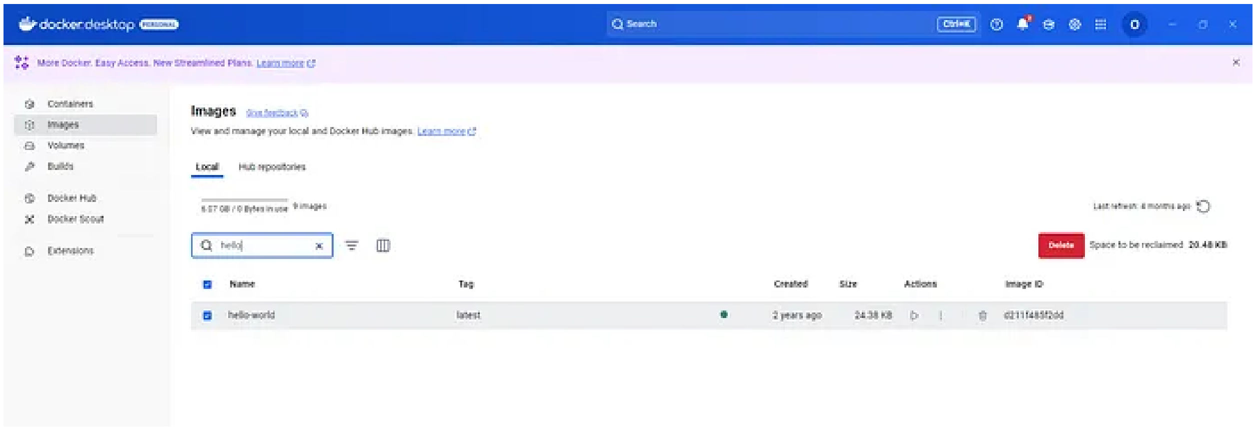
Here is the DockerFile for this demo project:

```
# Use the official Python 3.12 image as base
FROM python:3.12

# Set the working directory inside the container
WORKDIR /app

# Copy the current project directory into the container
COPY . /app

# Install dependencies
RUN pip install -r requirements.txt
```

```
# Expose the Flask port
EXPOSE 5000

# Run the application
CMD ["python", "app.py"]
```

In this file, we imported our desired base image with the selected Python version. Then, we copied all the required application files from our project directory to our container, installed all the required packages, exposed the port on which the API would run, and finally defined the main command to run the application.

Now, before running any Docker command, we have to make sure that we have started the Docker engine; in our case, we have to launch the Docker Desktop Application.

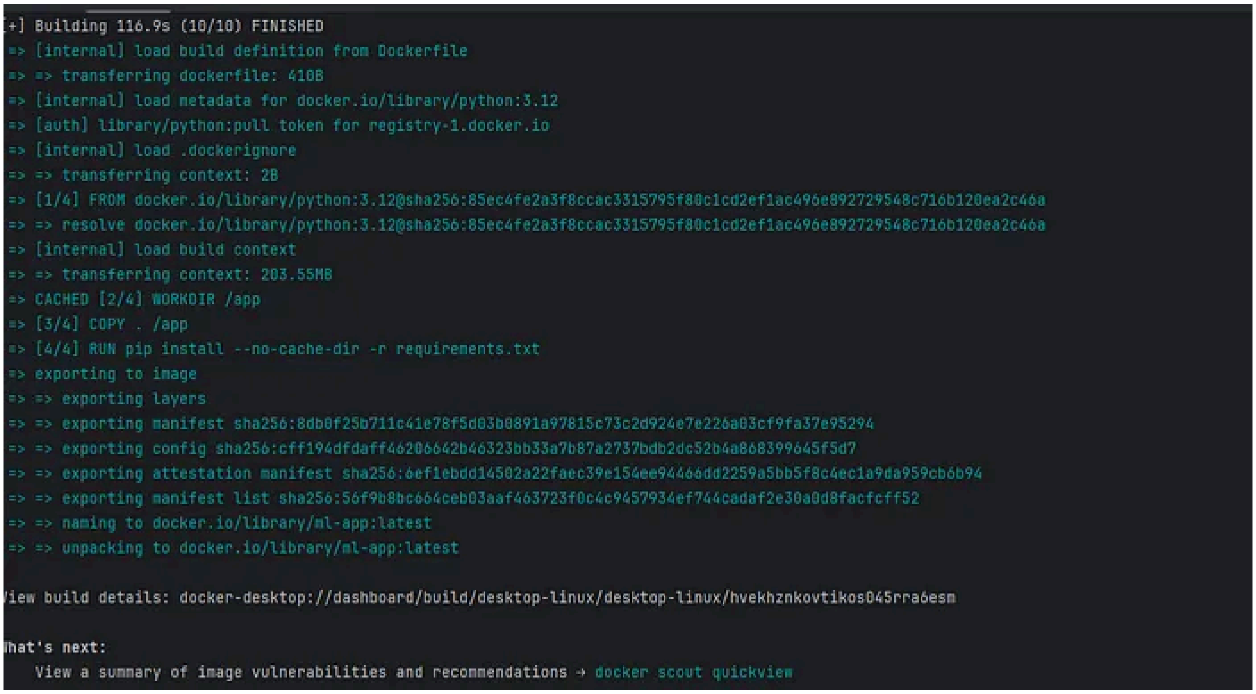


☑	Name	Tag		Created	Size	Actions	Image ID
☑	hello-world	latest	●	2 years ago	24.38 KB	▷ ⋮ 🗑	d2111f485f2dd

Having done that, we will now use our first command to build the image.

```
docker build -t ml-app .
```

By running the previous command, we see a step-by-step execution for each command we have written in our DockerFile.



```
[+] Building 116.9s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 410B
 => [internal] load metadata for docker.io/library/python:3.12
 => [auth] library/python:pull token for registry-1.docker.io
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [1/4] FROM docker.io/library/python:3.12@sha256:85ec4fe2a3f8ccac3315795f80c1cd2ef1ac496e892729548c716b120ea2c46a
 => => resolve docker.io/library/python:3.12@sha256:85ec4fe2a3f8ccac3315795f80c1cd2ef1ac496e892729548c716b120ea2c46a
 => [internal] load build context
 => => transferring context: 203.55MB
 => CACHED [2/4] WORKDIR /app
 => [3/4] COPY . /app
 => [4/4] RUN pip install --no-cache-dir -r requirements.txt
 => exporting to image
 => => exporting layers
 => => exporting manifest sha256:8db0f25b711c41e78f5d03b0891a97815c73c2d924e7e226a03cf9fa37e95294
 => => exporting config sha256:cff194dfdaff46206642b46323bb33a7b87a2737bdb2dc52b4a868399645f5d7
 => => exporting attestation manifest sha256:6ef1ebdd14502a22faec39e154ee94466dd2259a5bb5f8c4ec1a9da959cb6b94
 => => exporting manifest list sha256:56f9b8bc664ceb03aaf463723f0c4c9457934ef744cadaf2e30a0d8facfcff52
 => => naming to docker.io/library/ml-app:latest
 => => unpacking to docker.io/library/ml-app:latest

iew build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/hvekhznkovtikos045rra6esm

hat's next:
    View a summary of image vulnerabilities and recommendations → docker scout quickview
```

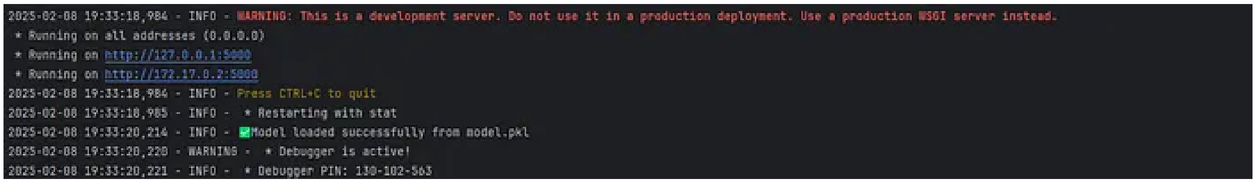
By now, we have finished building our first Docker container for our ML application to run the container, we will use this command

```
docker run -p 5000:5000 ml-app
```

This command could be generalized as

```
docker run -p <HOST_PORT>:<CONTAINER_PORT> <IMAGE_NAME>
```

Where HOST_PORT is the port that we will use from the host machine — your pc/laptop in this case-, CONTAINER_PORT is the port running from inside the Docker container, and the IMAGE_NAME is the name of the Docker image we have just created. When you run this command, the output will be something similar to the figure below.

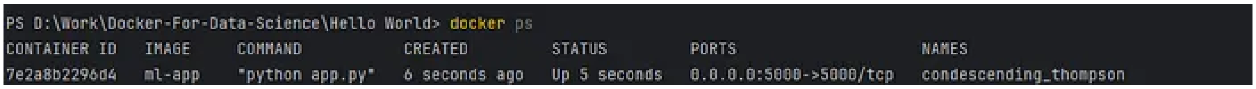


```
2025-02-08 19:33:18,984 - INFO - WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.17.0.2:5000
2025-02-08 19:33:18,984 - INFO - Press CTRL+C to quit
2025-02-08 19:33:18,985 - INFO -  * Restarting with stat
2025-02-08 19:33:20,214 - INFO - ✅Model loaded successfully from model.pkl
2025-02-08 19:33:20,220 - WARNING -  * Debugger is active!
2025-02-08 19:33:20,221 - INFO -  * Debugger PIN: 130-102-563
```

172.17.0.2:5000 is the internal IP address of the Docker container (Not accessible from your host machine). 127.0.0.1:5000 is localhost.

To get insight about the running containers, use this command

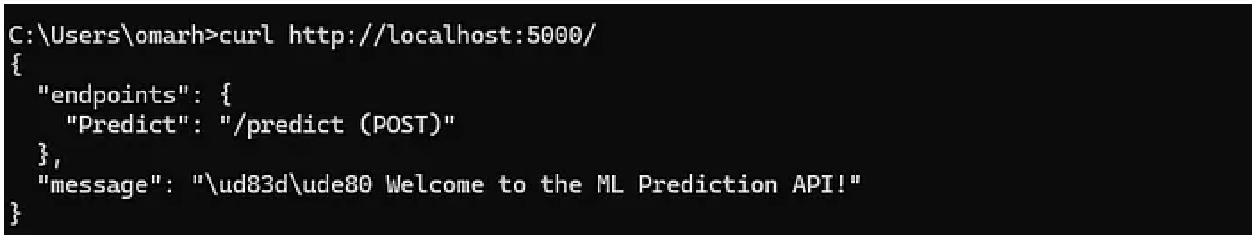
```
docker ps
```



```
PS D:\Work\Docker-For-Data-Science\Hello World> docker ps
CONTAINER ID   IMAGE     COMMAND           CREATED         STATUS         PORTS                    NAMES
7e2a8b2296d4   ml-app    "python app.py"   6 seconds ago   Up 5 seconds   0.0.0.0:5000->5000/tcp   condescending_thompson
```

The output of the previous command shows a detailed list of all the currently running containers. We can test the API on Postman, but for this example, we will use curl on the Windows command line like this

```
curl http://localhost:5000/
```

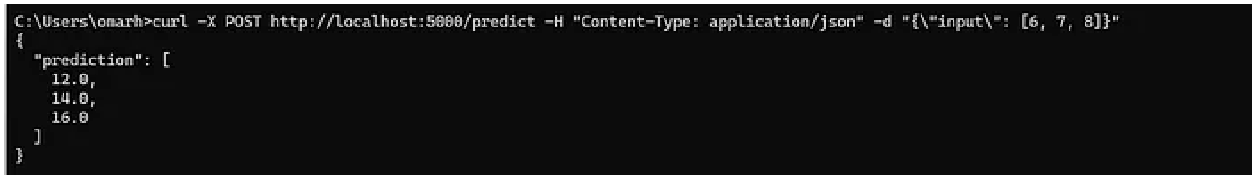


```
C:\Users\omarh>curl http://localhost:5000/
{
  "endpoints": {
    "Predict": "/predict (POST)"
  },
  "message": "\ud83d\ude80 Welcome to the ML Prediction API!"
}
```

We can use this command to test the model prediction

```
curl -X POST http://localhost:5000/predict -H "Content-Type:
application/json" -d "{\"input\": [6, 7, 8]}"
```

In this command, we passed the array [6,7,8] as input for the linear regression model.



```
C:\Users\omarh>curl -X POST http://localhost:5000/predict -H "Content-Type: application/json" -d "{\"input\": [6, 7, 8]}"
{
  "prediction": [
    12.0,
    14.0,
    16.0
  ]
}
```

You can stop the running container using this command

```
docker stop <container_id>
```

where <container_id> in this case is 7e2a8b2296d4

4. Best Practices For Using Docker for Data Science Projects

Docker has emerged as a powerful tool in the field of data science, revolutionizing how we develop, deploy, and manage data-driven applications. With its containerization capabilities, Docker offers unparalleled flexibility, reproducibility, and scalability, making it an invaluable asset for data scientists and developers alike. However, harnessing the full potential of Docker requires a solid understanding of industry-wide best practices to ensure efficient and effective utilization.

In this chapter, we will delve into a collection of industry-wide best practices specifically tailored for leveraging Docker in data science projects. From optimizing container performance to ensuring data persistence and maintaining a well-organized image repository, these practices address critical aspects of working with Docker in a data-driven environment.

Each section will explore a specific best practice, explain its importance, and provide practical insights on how to implement it effectively. By adhering to these best practices, data scientists and developers can streamline their Docker workflows, enhance reproducibility, minimize resource consumption, and improve collaboration within their teams.

Whether you are new to Docker or an experienced user looking to enhance your data science projects, this article will serve as a comprehensive guide to industry-proven best practices. By implementing these practices, you can unlock the true potential of Docker, enabling seamless and efficient management of your data-driven applications.

Let’s dive into the key best practices for using Docker in data science projects and elevate your containerization journey to new heights.

Chapter Content:

4.1. Keeping the Number of Layers Low

When working with Docker for data science projects, one crucial best practice is to keep the number of layers in your Docker image as low as possible. The concept of layers in Docker refers to the incremental changes made to an image during its construction. Each command in a Dockerfile creates a new layer, resulting in a stack of these layers that make up the final image.

Why is it important to keep the number of layers low? Here are a few key reasons:

1. **Efficient image builds**: Docker images are constructed by building layers on top of each other. When there are fewer layers, the build process becomes faster and more efficient. Each layer requires Docker to perform additional operations, such as file system operations, package installations, and dependency management. By reducing the number of layers, you can significantly speed up the image build process, saving valuable time and resources.
2. **Smaller image size**: Docker images with fewer layers tend to have smaller file sizes. This is because each layer adds its own set of files and dependencies to the image. When multiple layers contain similar or redundant files, it results in increased image size. By minimizing the number of layers, you can reduce the overall size of the Docker image. Smaller images not only save disk space but also improve network transfer times when sharing or deploying the image.
3. **Improved caching**: Docker utilizes caching mechanisms during the build process to optimize subsequent builds. Each layer is cached individually, allowing Docker to reuse previously built layers if the source code or dependencies haven't changed. When you keep the number of layers low, you increase the chances of hitting the cache and avoiding unnecessary rebuilds. This leads to faster iterations during development and reduces build times for CI/CD pipelines.
4. **Enhanced maintainability**: Managing and maintaining a Docker image with numerous layers can become complex and challenging. With a high number of layers, tracking changes, understanding dependencies, and troubleshooting issues can be more difficult. By minimizing the layers, you simplify the image structure, making it easier to understand, update, and maintain. It also improves the overall stability and reliability of your Dockerized data science projects.

To keep the number of layers low in your Docker image, consider the following best practices:

- **Combine related commands**: Identify commands in your Dockerfile that perform similar operations, such as package installations or file modifications. Consolidate these commands into a single step to reduce the number of layers created.
- **Leverage multi-stage builds**: Utilize multi-stage builds to separate the built environment from the runtime environment. This approach allows you to build dependencies and intermediate files in one stage and then copy only the necessary artifacts to the final stage, resulting in a smaller and more optimized image.
- **Use efficient base images**: Start your Docker image with a lightweight and minimal base image, such as Alpine Linux, rather than a larger and more bloated image. This reduces the number of initial layers and provides a lean foundation for your data science projects.

Let's take an example to make it clearer. Assume the following code:

```
# Use the official Python image as the build image

FROM python:3.9


# Install the dependencies

RUN pip install pandas

RUN pip install matplotlib

RUN pip install seaborn


# Copy necessary files

COPY my_script.py .

COPY data/ .


# Run the script

CMD ["python","my_script.py"]
```

The problem in the previous code is the use of several run and copy commands, which were unnecessary. Here’s how we could fix it:

```
# Use the official Python image as the build image

FROM python:3.9


# Install the dependencies using requirements.txt

COPY my_script.py requirements.txt data/ .

RUN pip install --no-cache-dir -r requirements.txt


# Run the script
```

```
CMD ["python","my_script.py"]
```

4.2. Using Official Images

Official Docker images are pre-built images provided by the Docker community or software vendors that are maintained and supported by the image publishers themselves. These images are created following industry best practices, undergo rigorous testing, and are regularly updated with security patches and bug fixes. Choosing official images as the base for your own Docker images offers several benefits, including increased stability and security.

Using the official images has the following advantages:

- **Stability**: Official images are created and maintained by experts who have in-depth knowledge of the software or framework being packaged. They follow strict guidelines and best practices to ensure the image is stable and reliable. This means you can have confidence in the quality and consistency of the official image, minimizing the chances of encountering unexpected issues or conflicts.
- **Security**: Security is a critical concern when working with Docker. Official images are carefully curated, and thorough security checks are performed to identify and address any vulnerabilities. Image publishers actively monitor and update the official images to ensure they incorporate the latest security patches. By using official images as your base, you leverage the expertise of the image publishers and reduce the risk of using outdated or compromised components.
- **Long-term maintenance**: Official images are designed with long-term support in mind. Image publishers are committed to maintaining and updating these images regularly, ensuring compatibility with new versions of dependencies, and addressing any reported issues. By starting with an official image, you align yourself with the ongoing support and maintenance efforts of the image publisher, which can save you time and effort in the long run.

While unofficial images may offer convenience or specific customizations, they come with inherent risks. Unofficial images are typically created and maintained by individual contributors or the community, without the same level of scrutiny and support as official images. They may lack proper documentation, security updates, or compatibility guarantees, making them less suitable for production environments.

When possible, it is recommended to prioritize the use of official images as the base for your Docker images. This practice ensures that you start with a solid foundation built by experts, which is actively maintained and updated. However, if you do choose to use unofficial images, it is essential to carefully evaluate their source, community reputation, documentation, and security practices before incorporating them into your workflow.

4.4 Multi-Stage Builds for Optimizing Performance

A multi-stage build in Docker allows you to use multiple FROM instructions in a single Dockerfile. We may use a larger image as a build image for building the application and then copy the necessary files to a smaller runtime image. By not including unnecessary files, we reduce the size of the final image, not only optimizing the performance but also making the application more secure.

Let’s look at an example to understand this better, as it gets repeatedly used in the industry:

```
# Use the official Python image as the build image

FROM python:3.9 AS build

# Set the working directory

WORKDIR /app

# Copy the requirements.txt file

COPY requirements.txt ./

# Install the dependencies

RUN pip install --no-cache-dir -r requirements.txt

# Copy the application files

COPY . .

# Train the model

RUN python train.py

# Use the official Alpine Linux image as the runtime image

FROM alpine:3

# Set the working directory

WORKDIR /app

# Copy the model files from the build image
```

```
COPY --from=build /app/models /app/models

# Copy the requirements.txt file

COPY --from=build /app/requirements.txt /app

# Install the dependencies

RUN pip install --no-cache-dir -r requirements.txt

# Run the application

CMD ["python","predict.py"]
```

After running the train.py file, we copy the generated model files and the requirements.txt file to a smaller Alpine Linux image for runtime, which we'll use to run the application. By utilizing a multi-stage build and not including build dependencies and the Python interpreter in the final image, we have effectively made the final image size much smaller.

4.4. Using Volumes to Persist Data

When working with Docker containers, data persistence is a crucial consideration, especially when you need to retain the results of experiments or share data across multiple containers. By utilizing the Docker volume command, you can ensure that data remains accessible and persistent even when containers are stopped or deleted.

When a container is stopped or deleted, the data stored within it is typically lost unless explicit measures are taken to preserve it. However, Docker volumes provide a solution to this challenge by creating a dedicated storage area outside the container. Volumes act as a bridge between the container and the host machine, allowing data to be stored separately and persistently.

Here are some key advantages of using Docker volumes for data persistence:

- **Data Retention**: By attaching a volume to a container, you can preserve critical data beyond the lifecycle of that container. This is especially useful when you need to reference or analyze experimental results at a later time. The data stored in the volume remains intact even if the container is stopped, deleted, or replaced, ensuring that valuable information is not lost.
- **Shared Data**: Docker volumes enable data sharing between multiple containers. This is particularly valuable when you have several interconnected services that need access to the same datasets or files. By creating a shared volume, you can ensure that all relevant

containers have consistent and up-to-date access to the required data, promoting collaboration and simplifying data management.

- **Separation of Concerns**: Volumes allow you to separate the container's execution environment from the persistent data it relies on. This clear separation simplifies container management and improves maintainability. You can easily update or replace containers without worrying about data loss or disruption, as the data resides independently within the volume.

Here's an example of how you could do it:

```
# Use the official Python image as the base image

FROM python:3.9

# Set the working directory

WORKDIR /app

# Copy the requirements.txt file

COPY requirements.txt ./

# Install the dependencies

RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application files

COPY . .

# Create a directory for storing data

RUN mkdir /app/data



# Define a volume for the data directory

VOLUME /app/data

# Run the application

CMD ["python","main.py"]
```

4.5. Organizing and Versioning Docker Images

As your Docker image collection grows, it's crucial to establish effective organizational practices to maintain clarity and ease of management. While specific approaches may vary between organizations, there are common practices that can be followed to organize Docker images effectively:

1. **Consistent Naming Convention**: Adopting a consistent naming convention for your Docker images is a fundamental practice. By using a standardized format like <registry>/<organization>/<image>:<version>, you can establish a clear structure that helps identify and categorize images. The registry represents the container registry where the image is stored (e.g., Docker Hub or a private registry), the organization indicates the entity or team responsible for the image, the image name describes the application or service encapsulated by the image, and the version denotes the specific version of the image. This naming convention facilitates easy navigation and management of images within your registry.
2. **Image Versioning**: Implementing versioning practices for your Docker images is essential for maintaining control and ensuring reproducibility. Versioning allows you to track changes, roll back to previous versions if necessary, and provide a clear understanding of the image's evolution over time. A common approach is to adopt the format <major version>.<minor version>.<patch version> (e.g., 1.0.2). Major versions usually indicate significant changes or new features, minor versions represent incremental updates, and patch versions denote bug fixes or minor adjustments. By incorporating versioning into your image naming convention, you can easily identify and manage different iterations of your images.
3. **Meaningful Image Tagging**: Tagging your Docker images with meaningful labels is another valuable practice for organization and management. Tags allow you to assign descriptive words or labels to images, indicating their purpose or stage in the deployment pipeline. For example, you can use tags like latest, staging, or production to differentiate images intended for different environments or deployment stages. This enables efficient identification and selection of the appropriate image for a particular use case. Additionally, you can use tags to signify specific configurations, branches, or customizations applied to the image. By leveraging meaningful tagging, you can streamline your image management and deployment processes.

By adhering to these organizational practices, you can establish a well-structured and manageable Docker image repository. Consistent naming conventions, versioning, and meaningful tagging enable easy identification, tracking, and selection of images for various environments and deployment scenarios. These practices contribute to a more efficient and organized Docker workflow, saving time and effort while ensuring clarity and control over your containerized applications.

5. 10 Docker Commands for 90% of Containerization Tasks

With Docker, developers can create isolated environments known as containers that can run consistently across different platforms, making it easier to build, deploy, and scale applications. However, to truly leverage the power of Docker, it is essential to master the fundamental commands that enable effective container management.

In this chapter, we will explore ten essential Docker commands that every developer and system administrator should know to streamline their container management process. From creating and starting containers to listing available images and gracefully stopping them, these commands will serve as building blocks for managing your Docker environment efficiently.

By mastering these commands, you will gain a solid foundation for managing Docker containers efficiently and effectively. Whether you are a beginner getting started with Docker or an experienced user looking to enhance your container management skills, this article will provide valuable insights and practical examples to help you navigate the world of Docker with confidence. Let’s dive in and unlock the full potential of Docker’s containerization capabilities.

Chapter Content:

5.1. docker run

The “docker run” command is used to create and start a new container based on a Docker image. Here’s the basic syntax for running a container:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

- **OPTIONS**: Additional options that can be used to customize the container’s behavior, such as specifying ports, volumes, environment variables, etc.
- **IMAGE**: The name of the Docker image to use for creating the container.
- **COMMAND**: (Optional) The command to be executed inside the container.
- **ARG**: (Optional) Arguments passed to the command inside the container.

For example, to run a container based on the “ubuntu” image and execute the “ls” command inside the container, you would use the following command:

```
docker run ubuntu ls
```

This will create a new container using the “ubuntu” image and run the “ls” command, which lists the files and directories inside the container’s file system.

Note that if the specified image is not available locally, Docker will automatically pull it from a Docker registry before creating the container.

5.2. docker ps

The “docker ps” command is used to list the running containers on your Docker host. It provides information such as the container ID, the image used, the command being executed, status, and port mappings. Here’s the basic syntax:

```
docker ps [OPTIONS]
```

By default, “docker ps” only shows the running containers. If you want to see all containers, including those that are stopped or exited, you can use the “-a” option:

```
docker ps -a
```

This will display a list of all containers on your Docker host, along with their respective details.

The output of the “docker ps” command includes columns like CONTAINER ID, IMAGE, COMMAND, CREATED, STATUS, PORTS, and NAMES. Here’s a brief explanation of these columns:

- **CONTAINER ID**: The unique identifier for the container.
- **IMAGE**: The name or ID of the image used to create the container.
- **COMMAND**: The command executed inside the container.
- **CREATED**: The timestamp indicates when the container was created.
- **STATUS**: The current state of the container (e.g., running, stopped, exited).
- **PORTS**: The port mappings between the container and the host system.
- **NAMES**: The automatically generated or user-assigned name of the container.

By default, the “docker ps” command provides a summarized view of the containers. If you want more detailed information, you can add the “—format” option followed by a format template. For example:

```
docker ps --format "ID: {{.ID}}, Image: {{.Image}}, Status: {{.Status}}"
```

This will display only the container ID, image, and status information for each container. Please note that running the “docker ps” command requires Docker to be installed and running on your system.

5.3. docker stop

The “docker stop” command is used to stop one or more running containers. It sends a signal to the container’s main process, requesting it to stop gracefully. Here’s the basic syntax:

```
docker stop [OPTIONS] CONTAINER [CONTAINER...]
```

- **OPTIONS**: Additional options that can be used to customize the stop behavior. For example, you can specify a timeout period with the “—time” or “-t” option to allow the container more time to stop gracefully before forcefully terminating it.
- **CONTAINER**: The name or ID of the container(s) to stop. You can specify multiple containers separated by spaces.

For example, to stop a container with the name “my-container”, you would use the following command:

```
docker stop my-container
```

If you want to stop multiple containers, you can list their names or IDs separated by spaces:

```
docker stop container1 container2 container3
```

When the “docker stop” command is executed, Docker sends a SIGTERM signal to the container, allowing the process inside to perform any necessary cleanup tasks and shut down gracefully. If the process doesn’t stop within the given timeout period (default is 10 seconds), Docker can send a SIGKILL signal to force the termination of the container.

5.4. docker rm

The “docker rm” command is used to remove one or more stopped containers from your Docker host. It permanently deletes the specified container(s) and frees up the associated resources. Here’s the basic syntax:

```
docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

- **OPTIONS**: Additional options that can be used to customize the removal behavior. For example, you can use the “-f” or “—force” option to force the removal of a running container.
- **CONTAINER**: The name or ID of the container(s) to remove. You can specify multiple

containers separated by spaces.

For example, to remove a container with the name “my-container”, you would use the following command:

```
docker rm my-container
```

If you want to remove multiple containers, you can list their names or IDs separated by spaces:

```
docker rm container1 container2 container3
```

By default, the “docker rm” command only removes stopped containers. If you want to remove running containers as well, you can use the “-f” or “—force” option:

```
docker rm -f container1 container2
```

Please note that removing a container will permanently delete it, including any data or changes made within the container. If you want to remove a running container, it will be stopped first before being removed.

Additionally, you can use the “-v” or “—volumes” option with the “docker rm” command to remove the associated volumes along with the container, if any.

5.5. Docker images

The “docker images” command is used to list the Docker images that are available on your Docker host. It displays information about the images, such as the repository, tag, image ID, creation date, and size. Here’s the basic syntax:

```
docker images [OPTIONS] [REPOSITORY[:TAG]]
```

- **OPTIONS**: Additional options that can be used to customize the output or filter the images. For example, you can use the “—format” option to specify a format template for the output or the “-a” or “—all” option to show all images, including intermediate image layers.
- **REPOSITORY**: (Optional) The repository name of the image.
- **TAG**: (Optional) The tag of the image.

By default, the “docker images” command lists all images available on your Docker host. For example:

```
docker images
```

This will display a table of images, including columns like REPOSITORY, TAG, IMAGE ID, CREATED, and SIZE. The REPOSITORY and TAG together form the unique identifier for an image.

If you want to filter the images based on the repository or tag, you can provide the repository and/or tag name as arguments. For example, to list images from a specific repository:

```
docker images my-repo
```

To list images with a specific tag:

```
docker images my-repo:my-tag
```

You can also combine options to further customize the output. For example, to show all images, including intermediate layers, and use a custom format for the output:

```
docker images -a --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"
```

This will display the images in a table format, showing only the image ID, repository, and tag information.

5.6. docker rmi

The “docker rmi” command is used to remove one or more Docker images from your Docker host. It permanently deletes the specified image(s) from your local image cache. Here’s the basic syntax:

```
docker rmi [OPTIONS] IMAGE [IMAGE...]
```

- **OPTIONS**: Additional options that can be used to customize the removal behavior. For example, you can use the “-f” or “—force” option to force the removal of an image, even if it’s being used by running containers.
- **IMAGE**: The name or ID of the image(s) to remove. You can specify multiple images separated by spaces.

For example, to remove an image with the name “my-image:latest”, you would use the following command:

```
docker rmi my-image:latest
```

If you want to remove multiple images, you can list their names or IDs separated by spaces:

```
docker rmi image1 image2 image3
```

By default, the “docker rmi” command only removes images that are not being used by any containers. If you try to remove an image that is currently being used by one or more containers, Docker will throw an error. In such cases, you can use the “-f” or “—force” option to forcefully remove the image, even if it’s in use:

```
docker rmi -f my-image:latest
```

Please note that removing an image will permanently delete it from your local image cache. If you need to use the image again in the future, you will need to pull it from a Docker registry or rebuild it using a Dockerfile.

Additionally, you can use the “—no-prune” option with the “docker rmi” command to prevent the automatic deletion of untagged parent images if they exist.

5.7. docker build

The “docker build” command is used to build a Docker image from a Dockerfile. It allows you to define the instructions and dependencies required to create a customized image. Here’s the basic syntax:

```
docker build [OPTIONS] PATH | URL | -
```

- **OPTIONS**: Additional options that can be used to customize the build process. Some commonly used options include “-t” or “—tag” to specify the name and optional tag for the image, “-f” or “—file” to specify the Dockerfile’s location, and “—build-arg” to pass build-time variables to the Dockerfile.
- **PATH | URL | -:** The path to the directory containing the Dockerfile, a URL to a Git repository, or “-” to build from the standard input.

For example, to build an image using a Dockerfile located in the current directory and tag it as “my-image:latest”, you would use the following command:

```
docker build -t my-image:latest .
```

The “.” indicates that the Dockerfile is in the current directory.

If you have a Dockerfile in a different location, you can specify its path using the “-f” option:

```
docker build -t my-image:latest -f /path/to/Dockerfile .
```

During the build process, Docker reads the instructions in the Dockerfile, executes each step, and creates intermediate images as needed. The final resulting image is then tagged with the specified name and optional tag.

Additional instructions, such as copying files, installing dependencies, setting environment variables, or exposing ports, can be defined in the Dockerfile to customize the image as per your requirements.

5.8. docker exec

The “docker exec” command is used to execute a command inside a running Docker container. It allows you to run commands interactively or in a detached mode. Here’s the basic syntax:

```
docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

- **OPTIONS**: Additional options that can be used to customize the execution behavior. Some commonly used options include “-i” or “—interactive” to keep STDIN open for interactive commands, “-t” or “—tty” to allocate a pseudo-TTY, and “-d” or “—detach” to run the command in the background.
- **CONTAINER**: The name or ID of the container where the command should be executed.
- **COMMAND**: The command to be executed inside the container.
- **ARG**: (Optional) Arguments passed to the command inside the container.

For example, to execute the “ls” command inside a container named “my-container”, you would use the following command:

```
docker exec my-container ls
```

This will run the “ls” command inside the specified container and display the list of files and directories. If you want to run an interactive command, such as starting a shell inside the container, you can use the “-it” options together:

```
docker exec -it my-container bash
```

This will start an interactive shell session inside the container, allowing you to execute multiple commands interactively. Please note that the container must be running for the “docker exec” command to work. If you need to execute a command in a stopped or exited container, you can use the “docker start” command to start the container first, and then use “docker exec” to run the command.

5.9. docker pull

The “docker pull” command is used to download Docker images from a Docker registry, such as Docker Hub. It retrieves the specified image or images and saves them to your local image cache. Here’s the basic syntax:

```
docker pull [OPTIONS] IMAGE[:TAG]
```

- **OPTIONS**: Additional options that can be used to customize the pull process. Some commonly used options include “—all-tags” to pull all available tags for an image, “—platform” to specify the platform for which to pull the image, and “—quiet” to suppress the progress output.
- **IMAGE**: The name of the image to pull from the Docker registry. It can be in the format “repository/image” or “repository/image:tag”. If the tag is not specified, “latest” is used by default.

For example, to pull the latest version of the “ubuntu” image from Docker Hub, you would use the following command:

```
docker pull ubuntu
```

If you want to pull a specific tagged version of the image, you can specify the tag:

```
docker pull ubuntu:20.04
```

The specified image will be downloaded from the Docker registry and saved to your local image cache. Once the image is pulled, you can use it to create and run containers on your Docker host.

5.10. docker push

The “docker push” command is used to upload Docker images to a Docker registry, such as Docker Hub or a private registry. It allows you to share your locally built or modified images with others. Here’s the basic syntax:

```
docker push [OPTIONS] NAME[:TAG]
```

- **OPTIONS**: Additional options that can be used to customize the push process. Some commonly used options include "—all-tags" to push all tags for an image, "—disable-content-trust" to skip content trust verification, and "—quiet" to suppress the progress output.
- **NAME**: The name of the image to push. It should include the repository and image name. For example, "username/repository:image".
- **TAG**: (Optional) The tag of the image to push. If not specified, the "latest" tag is used by default.

Before pushing an image, you need to ensure that you are authenticated to the Docker registry. You can log in to the registry using the "docker login" command, providing your username, password, and registry URL if necessary.

For example, to push an image named "my-image" with the "latest" tag to Docker Hub, assuming you are logged in to Docker Hub, you would use the following command:

```
docker push username/my-image:latest
```

The specified image will be uploaded to the Docker registry and made available for others to download and use.

Afterword

Thanks for purchasing and reading my book! If you have any questions, feedback or praise, you can reach me at: **Youssef.Hosni95@outlook.com**

You can check my other books on my **website**. I would be happy if you connect with me personally on **LinkedIn**. If you liked my writings, make sure to follow me on **Medium**. You are also welcomed to subscribe to my **newsletter To Data & Beyond** to never miss any of my writings.

What's inside the book?

- Docker for Data Science Projects: A Beginner-Friendly Introduction
- 11 Docker Container Images for Generative AI & ML Projects
- Building your first Docker container for Machine Learning Applications
- Best Practices For Using Docker for Data Science Projects
- 10 Docker Commands for 90% of Containerization Tasks

About the Author

Youssef Hosni is a data scientist and machine learning researcher who has worked in machine learning and AI for over half a decade.
In addition to being a researcher and data science practitioner, Youssef has a strong passion for education. He is known for his leading data science and AI blog, newsletter, and eBooks on data science and machine learning.

Youssef is a senior data scientist at Ment focusing on building Generative AI features for Ment Products. He is also an AI applied researcher at Aalto University working on AI agents and their applications. Before that, he worked as a researcher in which he applied deep learning and computer vision techniques to medical images.