

CHAPTER 4

Infrastructure for MLOps

This chapter is about infrastructure. You might think of buildings and roads when you hear the word infrastructure, but in MLOps, infrastructure refers to the most fundamental services we need to build more complex systems like training, inference, and model deployment pipelines. For example, we need a way to create data stores that can store features for model training and servers with compute and memory resources for hosting training pipelines. In the next section, we will look at a way we can simplify the process of creating infrastructure by using containers to package up software that can easily be maintained, deployed, and reproduced.

Containerization for Data Scientists

Containers have had a profound impact on the way data scientists code; in particular, it makes it possible to quickly and easily spin up infrastructure or run code inside a container that has all of the software, runtimes, tools, and packages you need to do data science bundled inside.

Why is this a big deal? As you've probably experienced in the previous chapter where we used Python environments to isolate packages and dependencies, a lot of problems with configuring and managing multiple packages become manageable with containerization. With simple

environments like Conda, you could manage multiple versions and with package managers like Pipenv, you had access to a Pipfile which contained all of the configuration you needed to manage your environment.

Now imagine you need more than just Python; you might have different runtime requirements. For example, maybe parts of your data science workflow require R packages and so you need the R runtime. Maybe you also have to manage multiple binaries from CRAN and have your code “talk” to a database which itself has dependencies like a JVM (Java virtual machine) or specific configuration that needs to be set.

Unless you have a strong background in IT, managing all of these configurations, runtimes, toolchains, compilers, and other supporting software becomes tedious and takes away from time you can spend on data science.

There’s another problem: portability. Imagine you have a package that requires Mac but you’re on Windows. Do you install an entire OS just to run that software? Containers solve this problem by allowing you to build once and run anywhere. They also make it pull new containers and swap out components of your machine learning system with ease. Let’s take a deep dive into one of the most popular container technologies: Docker.

Introduction to Docker

Docker is a platform as a service that uses OS-level virtualization to encapsulate software as packages called containers. The software that hosts the containers is called the Docker Engine and is available for a number of platforms including Linux, Windows, and MacOS with both free and paid licensing. Figure 4-1 shows how containers run on top of the Docker Engine using OS-level virtualization.

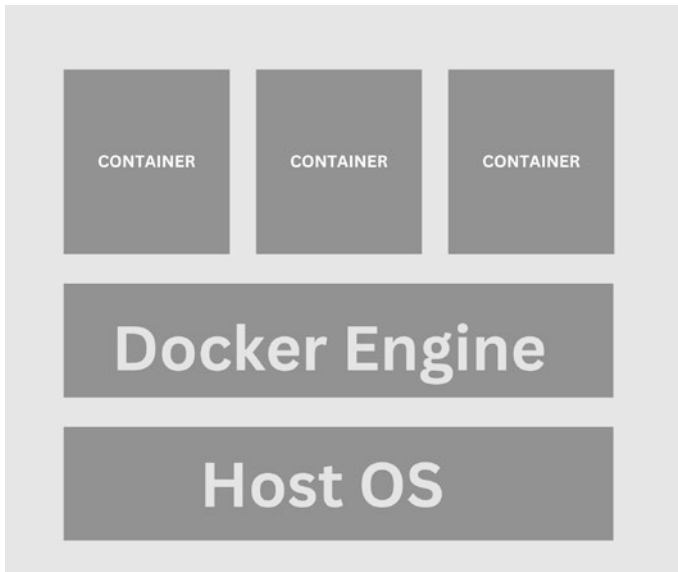


Figure 4-1. How containers run using OS-level virtualization

Anatomy of the Docker File

Okay, so we know what Docker is but how do we use it? Let's say you want to create our own Jupyter based data science lab. You've probably installed Jupyter before, but could you write down a recipe that is reproducible? You might start by noting what operating system (OS) you're using, installing dependencies like Python and pip, and then using pip to install Jupyter lab. If you've read the Jupyter lab documentations, then you probably also know you need to expose some default ports so you can launch and access your notebook from a web browser. If you wanted to do a deep learning workflow using GPU, you might consider installing NVIDIA drivers as well.

This is a lot of work but we can write it as a series of steps:

- From your host OS, install specific software packages.
- Install drivers and low level package managers.

- Install Python and Python package managers.
- Use package managers to install Python packages.
- Run Jupyter lab.
- Expose ports so we can access Notebooks in our web browser.

In Docker, we can encode these steps as a sequence of instructions or commands in a text file called a *Docker File*. Each instruction or command gets executed in the Docker environment in the order it's read starting from the first instruction. The first instruction usually looks something like the following:

```
FROM nvidia/cuda:12.0.1-base-ubuntu20.04
```

This creates what is known in Docker as a *layer* containing the Ubuntu OS with NVIDIA's cuda drivers in case we need GPU support (if you only have a CPU on your laptop, you can still build this docker container).

Other *layers* get installed on top of this layer. In our example of installing a deep learning library, we would need to install Cuda and Nvidia drivers to have GPU accelerated training (covered in the next section). Fortunately, in Ubuntu, these are available in the form of Ubuntu packages. Next, we might want to create a dedicated working dir for all of our notebooks. Docker comes with a WORKDIR instruction. We'll call our directory /lab/

```
WORKDIR /lab/
```

We need to install data science specific Python packages for our lab environment and most important the Jupyter lab packages. We can combine this step into a single command with the RUN instruction.

```
RUN pip install \  
    numpy \  
    pandas \  
    jupyterlab
```

```
tensorflow \
torch \
Jupyterlab
```

Finally we'll need to launch our Jupyter server and expose port 8080 so we can access our notebook in a browser. It's good practice to change the default port, but ensure it's not one that is reserved by the operating system. These steps can be accomplished using the CMD and EXPOSE instructions:

```
CMD ["jupyter", "lab", "--ip=0.0.0.0", "--port=8080",
"--allow-root", "--no-browser"]
EXPOSE 8080
```

In the next section, we will apply this theoretical knowledge of Docker by packaging all of these steps into a Docker file in the next lab and build the image. Once we build the image (a binary file) we can then run the image, creating a *container*. This distinction between an image and container might be confusing if it's the first time you've encountered the terms, but you should understand the difference before proceeding to the lab. We summarize the difference in the following since it is very important for understanding containers.

Docker file: A docker file is a *blueprint* for a Docker image; it contains a series of instructions in plain text describing how a docker image should be built. You need to first build the docker file.

Docker image: A docker image is a binary file that can be stored in the cloud or on disk. It is a lightweight, self-contained (hence the name container), executable piece of software that includes everything needed to run an application including the application code, application runtime, interpreters, compilers, system tools, package managers, and libraries.

Docker container: A docker container is a live piece of software; it's a *runtime instance* of a Docker image created when you run the image through the Docker run command.

Now that we've clarified the difference between a Docker image and a Docker container, we're ready to start building some containers. In this lab, you'll go through the previous steps in detail to create your own data science lab environment, an important addition to any MLOps engineer's toolkit.

Lab 1: Building a Docker Data Science Lab for MLOps

Step 1. We first need to install the Docker engine. Proceed to [Install Docker Desktop on Windows](#) and select your platform. We'll be using Windows 10. Download the Docker Desktop Installer for Windows. We recommend the latest version but we'll use 4.17.1.

Step 2. Right-click the Docker Desktop Installer run as admin. Ensure to check the install WSL and Desktop shortcut options in the first menu, and click next. Figure 4-2 shows Docker Desktop for Windows.

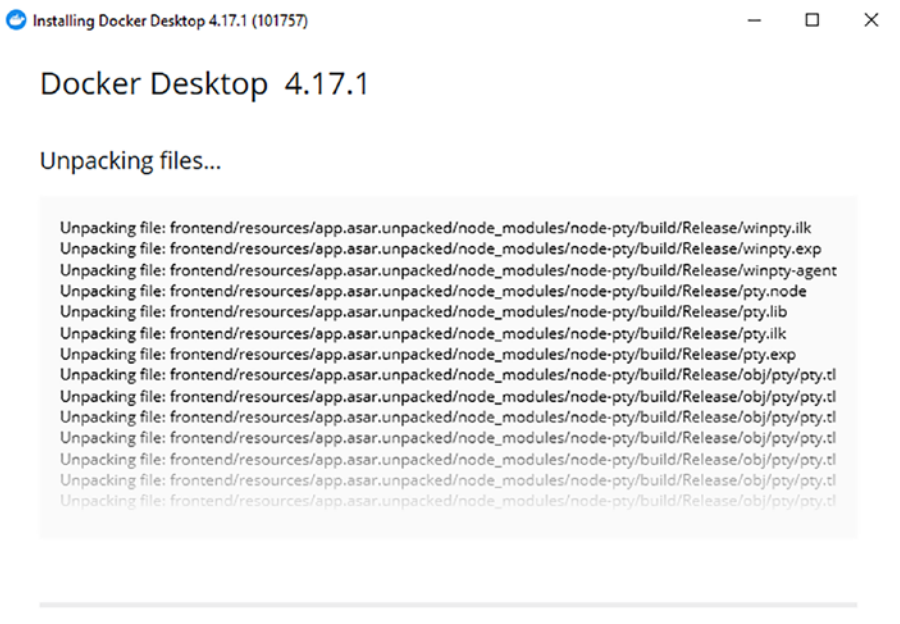


Figure 4-2. Docker Desktop for Windows

Step 3. Launch Docker Desktop from the Desktop icon and accept the service level agreement. Figure 4-3 shows the Docker license agreement.

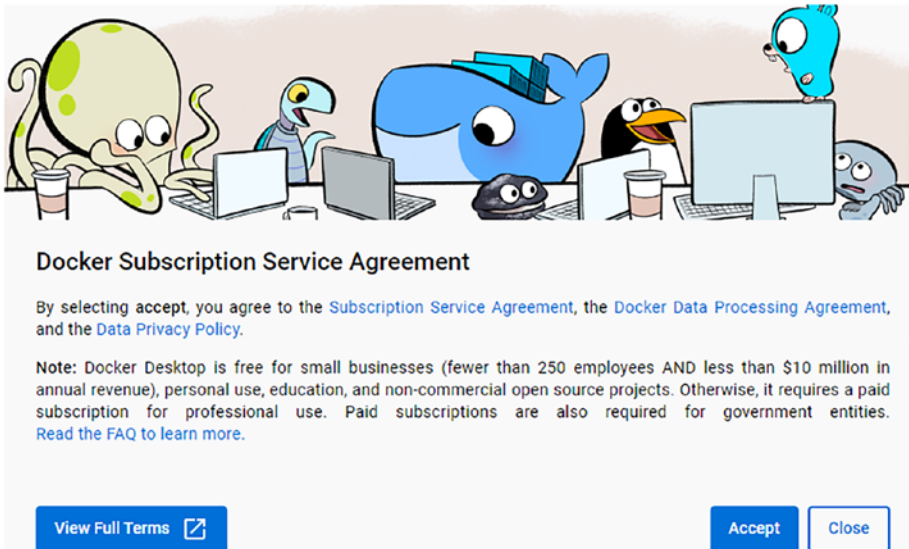


Figure 4-3. Docker license agreement

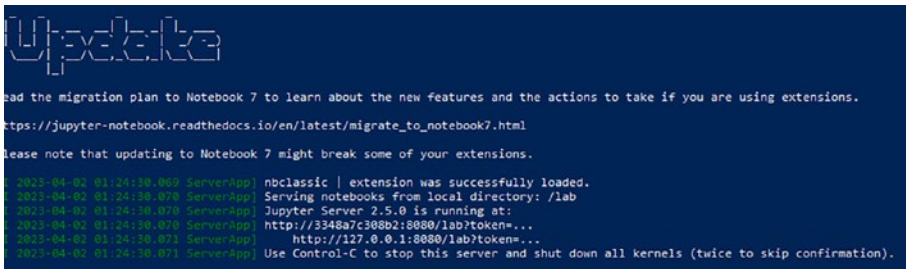
Step 4. Use the Git clone command to clone the repo provided along with the supplementary resources. Start a new terminal session in vs code and cd into Chapter 4 Labs where you will find a file called Dockerfile (this is where you'll find the sequence of plain text instructions or recipe for building your data science lab environment).

Step 5. Run `docker build -t data_science_lab .` inside the directory with Dockerfile. The period at the end is important; it's called the *Docker context*.

Step 6. Build your image. Assign it the name `jupyter_lab` with the `-t` option and run the container. You can also pass in a token (we used `mlops_toolkit`) which will be your password for authenticating with Jupyter notebook.

```
docker build -t jupyter_lab .
docker run --rm -it -p 8080:8080 -e JUPYTER_TOKEN=mlops_toolkit
jupyter_lab
```

Did you notice anything? You should see the following splash screen. Figure 4-4 shows a general view of what you can expect to see, but note that your splash screen may look slightly different especially if you are not using Powershell.



```

JupyterLab

Read the migration plan to Notebook 7 to learn about the new features and the actions to take if you are using extensions.
https://jupyter-notebook.readthedocs.io/en/latest/migrate_to_notebook7.html

Please note that updating to Notebook 7 might break some of your extensions.

2023-04-02 01:24:30.860 ServerApp] nbclassic | extension was successfully loaded.
2023-04-02 01:24:30.870 ServerApp] Serving notebooks from local directory: /lab
2023-04-02 01:24:30.870 ServerApp] Jupyter Server 2.5.0 is running at:
2023-04-02 01:24:30.870 ServerApp] http://3348a7c308b2:8080/lab?token=...
2023-04-02 01:24:30.871 ServerApp] http://127.0.0.1:8080/lab?token=...
2023-04-02 01:24:30.871 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).

```

Figure 4-4. *Splash screen for Jupyter Lab*

Note The PyTorch and TensorFlow wheels are around 620 and 586 MB, respectively, at the time of writing, so these are pretty large. Sometimes this can be a problem if disk space is limited. Although we won't cover it in this lab, optimizing the size of a Docker image is an interesting problem and an area of specialization within MLOps especially when working with deep learning frameworks.

Step 7. Navigate to localhost:8080/lab in a browser (note we exposed port 8080 in the Dockerfile; this is where the number comes from). Enter your token (“mlops_toolkit”) and you should be redirected to the lab environment pictured in Figure 4-5.

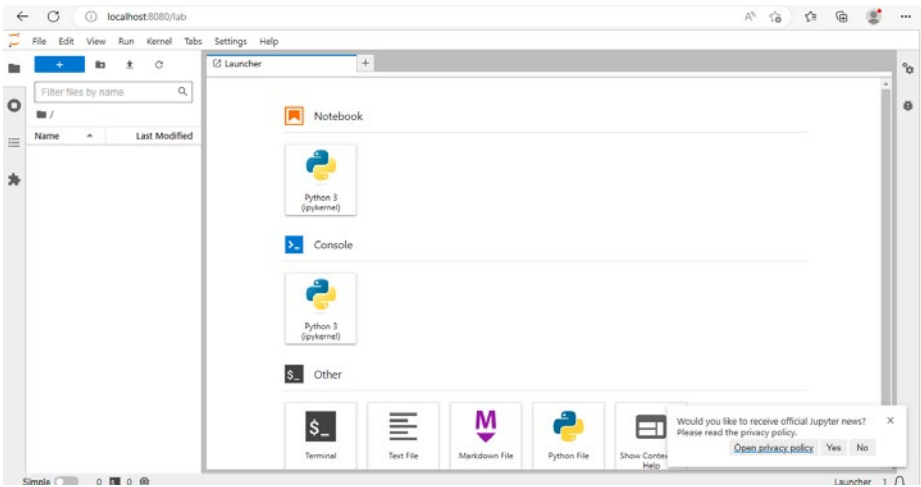


Figure 4-5. *The Jupyter lab environment*

Finally, click Python Kernel in your Lab environment to launch a Jupyter notebook. We'll use this environment in subsequent labs if you need a Jupyter notebook environment. Optionally you can also use Google Colab (Figure 4-6).

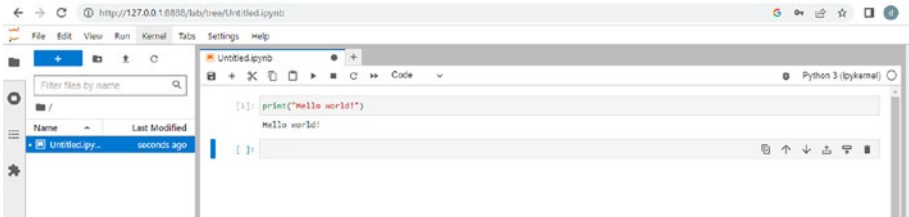


Figure 4-6. *Examples of cells in a Jupyter notebook*

You can now run notebooks inside docker and you have a reproducible data science lab environment you can use and share. We'll talk about how you can deploy this environment to the cloud in future chapters enabling you to have a shared lab environment and collaborate on projects. This is an amazing first step toward mastering data science infrastructure, and

we can now talk about particular kinds of data infrastructure used by data scientists. In the next section, we'll look at the feature store pattern, a pattern for data infrastructure used for supporting robust, scalable feature engineering pipelines.

The Feature Store Pattern

Going back to the MLOps lifecycle, after data collection and basic data cleansing, our goal is to build features from this data. In the real world, you'll frequently deal with 100s of features. It is not uncommon to have data science projects where 100, 200, or even 1000 or more features are constructed. These features eventually will be fed into a feature selection algorithm, for example, when we have a prediction problem using a supervised data, we can reduce these hundreds of features to a reasonable number in many ways, for example, using Lasso or a bagging algorithms like random forest to rank features by importance for our particular problem, filtering out the ones that have little predictive value.

The feature selection process, unlike most other parts of the machine learning lifecycle, may not be automated. One reason for this is feature selection is dependent on what you're trying to model, and there may be specific features like demographic data and PII that need to be excluded even if those features have predictive value.

Although feature selection can reduce the number of features used in a model, we still need to maintain the complete gambit of features for future problems. Additionally, model accuracy deteriorates over time, the business definitions can change, and you may have to periodically rerun feature selection as you add new data sources.

So how do we store all of these features: manage different versions of features to support feature selection, hyper-parameter tuning, model retraining, and future modeling tasks that might make use of these hundreds of features at different points in the lifecycle? This is where the concept of a feature store comes into play.

Feature store: A feature store is a design pattern in MLOps that is used to centralize the storage, processing, and access to features. Features in a feature store are organized into logical groups called *feature groups*, ensuring the features are reusable and experiments are reproducible.

Implementing Feature Stores: Online vs. Offline Feature Stores

A feature store and feature groups may be implemented using a variety of data infrastructure. A model is trained on features which typically involve joining multiple normalized, disparate data sources together. These joins are expensive and, sometimes since data is not well-defined, may involve semi-joints, range joins, or window analytic functions in the mix. These queries, which are executed on remote data store infrastructure, need to support both low latency queries at prediction time and high throughput queries on years of historical data at training time.

To make matters more complex, features may not be available at prediction time or may need to be computed on the fly, possibly using the same code as in the training pipeline. How do we keep these two processes in sync and have data infrastructure support both online and offline workflows requiring low latency and high throughput?

This is a hard problem in MLOps but understanding the types of *data infrastructure* used to implement a feature store. Let's look at some of this data infrastructure we can use for implementing feature stores.

Lab: Exploring Data Infrastructure with Feast

Feast is an open source project (Apache License 2.0 free for commercial use) that can be used to quickly set up a feature store. Feast supports both model training and online inference and allows you to decouple data from

ML infrastructure, ensuring you can move from model training to model serving without rewriting code. Feast also supports both online and offline feature stores.

You can also build your own feature store using docker and installing the underlying database services yourself. At the end of this lab, there is also an exercise so you aren't just following instructions, but first run the following commands:

Step 1. Open code and start a new terminal session in PowerShell or Bash. Install Feast:

```
pipenv install feast  
pipenv shell
```

Step 2. Run the following command to initialize the Feast project. We will call our feature store `pystore`:

```
feast init pystore  
cd pystore/feature_repo
```

Step 3. Look at the files created:

`data/` are parquet files used for training pipeline.

`example_repo.py` contains demo feature definitions.

`feature_store.yaml` contains data source configuration.

`test_workflow.py` showcases how to run all key Feast commands, including defining, retrieving, and pushing features.

Step 4. You can run this with `python test_workflow.py`. Note on Windows we had to convert our paths to raw strings to get this to work (see the code for Chapter 4). Figure 4-7 shows the result of running the test script.

```

Created entity driver
Created feature view driver_hourly_stats
Created feature view driver_hourly_stats_fresh
Created on demand feature view transformed_conv_rate
Created on demand feature view transformed_conv_rate_fresh
Created feature service driver_activity_v2
Created feature service driver_activity_v1
Created feature service driver_activity_v3

```

Figure 4-7. Running the test script locally

Step 5. Run Feat apply (inside pystore directory); this will register entities with Feat. Figure 4-8 shows the result of running this command.

```

Updated feature view sensor
  stream_source: -> name: "sensor_push_source"
  type: PUSH_SOURCE
  data_source_class_type: "feast.data_source.PushSource"
  batch_source {
    name: ".\\data\\sensor.parquet"
    type: BATCH_FILE
    timestamp_field: "timestamp"
    file_options {
      uri: ".\\data\\sensor.parquet"
    }
  }

```

Figure 4-8. Running Feat apply command

Note, you should see two components with temperature and pressure measurements generated in your final feature store pictured in the following. That's it! You've created your first feature store for an IoT data set. Figure 4-9 shows the expected output.

| component_id | pressure | temperature |
|--------------|---------------|-------------|
| 0 | 1 1545.093262 | 1973.979370 |
| 1 | 2 1795.720947 | 1719.880371 |

Figure 4-9. Expected output for pressure and temperature readings

Now as promised, here is an exercise you can do to get a feel for a real MLOps workflow.

Exercise

Being able to iterate and make changes to feature definitions is a part of MLOps since features rarely stay static. In a production environment, these types of anomalies should be caught automatically.

Exercise 1. Modify the notebook and rerun the lab to fix the pressure and temperature features so that they're in a more reasonable range for pressure (measured in Kilopascals) and temperature (measured on the Kelvin scale).

Hint You may need to do some research on what the right range looks like and figure out where in the code you should make the change.

Dive into Parquet Format

You also may have noticed the format we are storing our data. We used the parquet extension as opposed to the more common csv which you're probably already familiar with. So what is the difference between a parquet and a csv and why might we prefer to store files in parquet format at all?

The difference is in the size and efficiency of the format. While parquet is highly efficient at data compression (it is a binary file) meaning the file sizes are much smaller, unlike csv, parquet format encodes the data and schema for fast storage and retrieval. You might also use Parquet format with libraries like Apache Arrow which can make reading a large csv file several times faster. There is also a difference in how the data is stored.

In parquet format, data is stored in a columnar format, whereas csv is row oriented. For data science code, columnar data store is preferred since only a small subset of columns are used for filtering, grouping, or aggregating the data.

Although knowledge of every possible data format isn't required, you should be aware as an MLOps engineer that you can optimize your code for speed and efficiency simply by changing the format to one that better matches your workflow. In the next section, we'll take a look at another way to optimize for speed: hardware accelerated training.

We just took a deep dive into containers and data infrastructure, but if you're a pure data scientist without a background in IT, then you might be wondering do I really need to know how to work with low level data infrastructure and become an expert in containers to do MLOps for my own projects?

The answer depends on the use case, but in general, there are cloud services available for each stage of the MLOps lifecycle. For example, you can use Databricks if you want an end-to-end machine learning platform and add components as needed by integrating with other cloud services, for example, PowerBI, if you need a reporting solution, Azure DevOps if you need to build CI/CD pipelines to deploy your code, and maybe even an external data storage like AWS or Azure data lake to store your models, artifacts, and training data sets. You technically should know about parquet, but in this example, you could use Delta table format which in uses Parquet under the hood for storing data but also gives you a delta log and APIs for working with this format, so the low level details are abstracted for you, leaving more time for data science. In the next section, we'll take a deeper dive into some of the cloud services available while trying to remain agnostic about specific platforms like AWS, Azure, and Google Cloud.

Hardware Accelerated Training

Many times in data science, we are dealing with big data sets. Training sets can total gigabytes, terabytes, and with the rise of IoT data even petabytes of data. In addition to big data, many workflows, especially ones requiring deep learning like transfer learning, can be extremely intensive and require GPU accelerated training.

Training or even fine-tuning a large language model like BERT on commodity hardware using only a CPU can take days. Even if you're not training a large language model from scratch, some model architectures like recurrent neural networks take a long time to train. How do we accelerate this training? We have two options: distributed training and GPU accelerated training. First, let's discuss some of the major cloud service providers before jumping into distributed training.

Cloud Service Providers

There are several major cloud service providers. The big 3 are Azure, Amazon Web Services, and Google Cloud. Each of the three has machine learning service offerings and provides compute, networking, and data storage services. For example, Amazon Web Services has s3 buckets and Azure has blob storage. For end-to-end machine learning, Amazon Web Services offer SageMaker, while Azure has Azure Machine Learning service. There are other services for end-to-end machine learning as well and distributed training like Databricks which is offered in all three of the cloud service providers. There are differences between the different services, for example, Databricks integrates with MLFlow, whereas SageMaker has its own Model registry, but there is a difference in the platform: not the cloud service provider. You can also deploy your own containers in all three cloud service providers. For example, if you want to deploy your own Airflow instance to Kubernetes, all three offer their own version of Kubernetes with differences in cost for compute, storage, and tooling. In the next section, we'll take a look at distributed computing in some of these cloud service providers.

Distributed Training

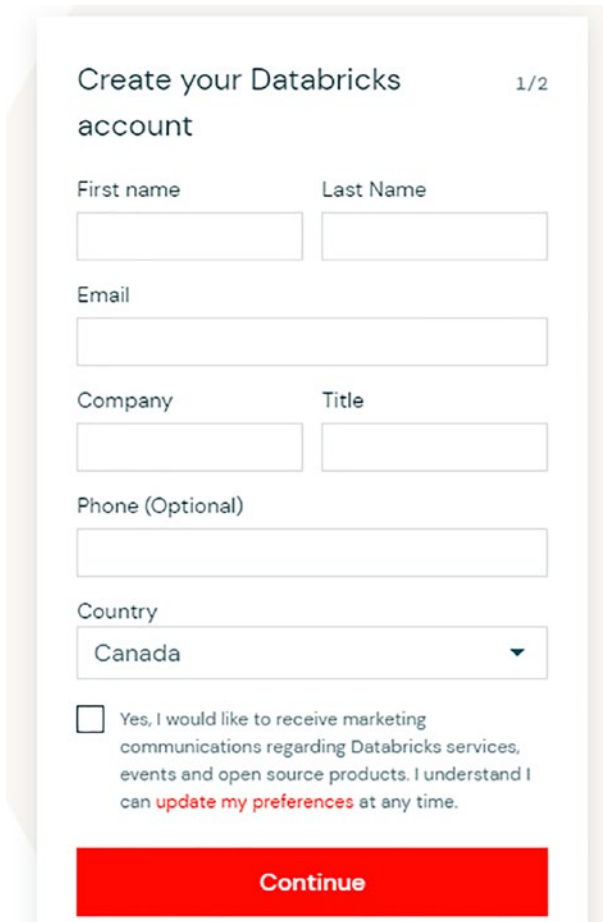
All of the code we've run so far has been executed on a single machine. If you're using a laptop or workstation, you can interact with the physical hardware, but if you're running inside a cloud environment like Google Cloud, Azure, AWS (Amazon Web Services), Google Colab, or Databricks, the hardware infrastructure on the backend may not be so obvious and may actually be hidden from you. For example, in Databricks, you can configure a cluster, a collection of worker nodes and driver nodes which are themselves individual virtual machines complete with their own CPU or GPU for compute and a certain configurable amount of working memory and disk space.

The advantage of using multiple VMs when training is straightforward: More VMs mean more CPU or GPUs available which means model training can be accelerated. If you've ever written Pandas code that attempted to read in a large csv file and experienced out of memory errors, then you've probably already thought about increasing the memory available through out of core (spilling to disk) like Dask, but another option is to run your code on a distributed environment like Databricks.

You can take a look at the supplementary code provided with this chapter for an example of configuring Horovod for distributed training.

You can make a free account on Databricks community edition to try out Databricks, but we recommend you use an Azure cloud subscription for full functionality. The steps to get a Databricks account (which you can later convert to a full featured account) are as follows:

1. In a browser, navigate to <https://community.cloud.databricks.com/login.html>.
2. Click sign up and create a free account. Figure 4-10 shows how to register a Databricks account.



The image shows a registration form for a Databricks account. The title is "Create your Databricks account" with a progress indicator "1/2" in the top right corner. The form contains several input fields: "First name" and "Last Name" (two separate boxes), "Email" (one wide box), "Company" and "Title" (two separate boxes), "Phone (Optional)" (one wide box), and "Country" (a dropdown menu currently showing "Canada"). Below the country field is a checkbox with the text: "Yes, I would like to receive marketing communications regarding Databricks services, events and open source products. I understand I can **update my preferences** at any time." At the bottom of the form is a prominent red button labeled "Continue".

Figure 4-10. *Registering a Databricks account*

Click continue and make sure to select community edition at the bottom; otherwise, choose a cloud provider (AWS, Azure, or Google Cloud as shown in Figure 4-11):

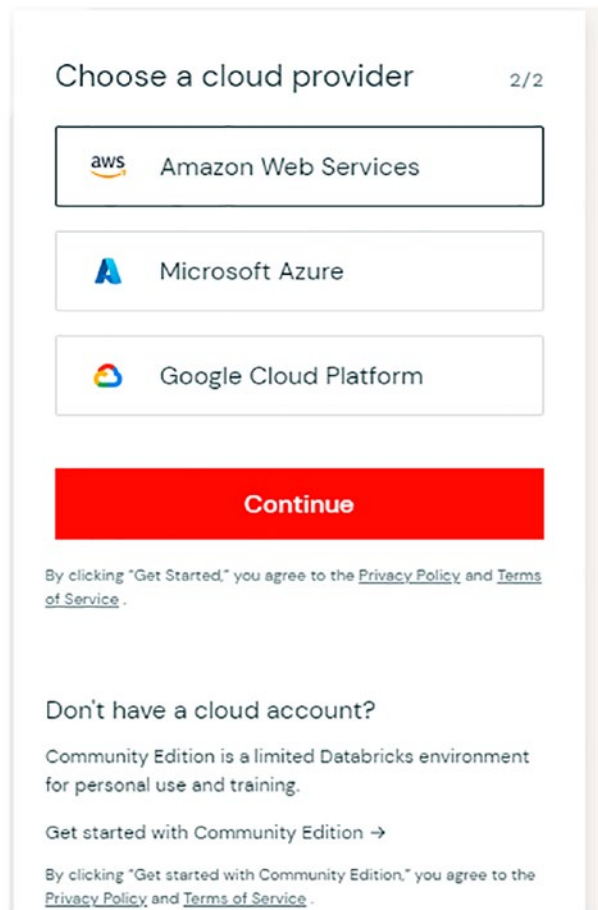


Figure 4-11. *Choosing a cloud provider*

3. In the workspace, create a job cluster. Databricks distinguishes between two types of clusters: all purpose (interactive) and job clusters.
4. Click the cluster creation and edit page; select the Enable autoscaling checkbox in the Autopilot Options box (Figure 4-12).

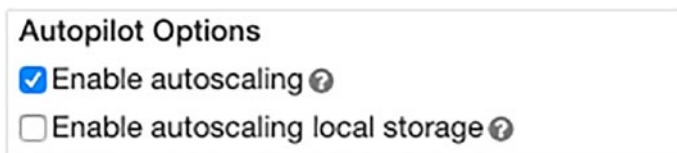


Figure 4-12. *Enable autoscaling is an option for elastic workflows*

Note This step is similar for all-purpose clusters except you will want to include a terminate clause after 120 minutes (or a timeout that fits your use case) to force the cluster to terminate after a period of inactivity. Forgetting this step can be costly since like many cloud services you are charged for what you use, and this detail is an important consideration when choosing to use cloud services. The timeout option is shown in Figure 4-13.

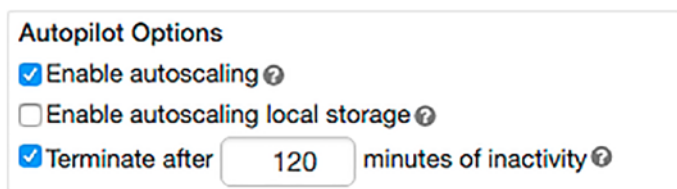


Figure 4-13. *Enabling timeout after 2 hours of cluster inactivity*

To attach a cluster to a notebook in Databricks, follow these steps:

1. Create a new notebook in your workspace.
2. Click the “Connect” button in the top-right corner of the notebook (Figure 4-14).

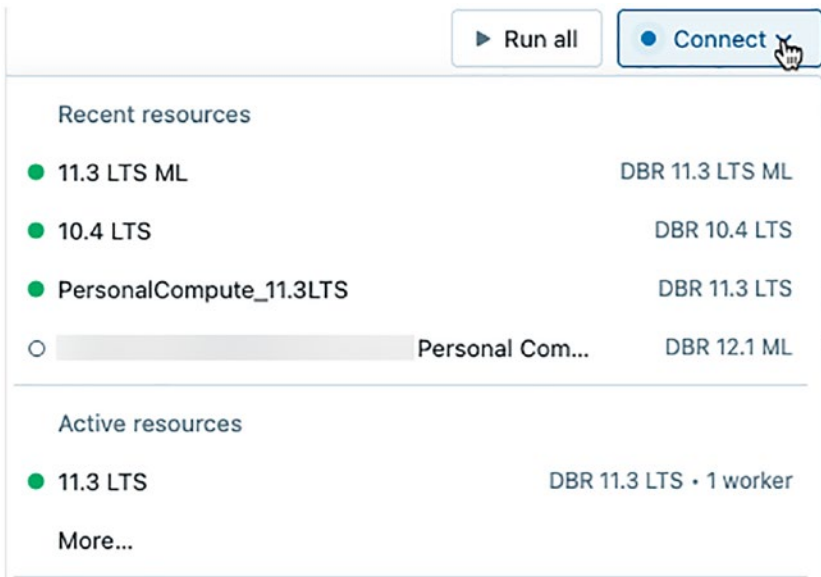


Figure 4-14. *Connect button to attach a notebook to a cluster*

Select the cluster you just created in the previous step.

Once the cluster is attached, you are able to run your code on the cluster, taking advantage of the many workers available for distributed workflows. You can configure the number of workers in your cluster and enable autoscaling for larger workflows. The notebook will connect to the cluster automatically. You can also detach the cluster from the notebook by clicking the “Detach” button in the top-right corner of the notebook. You can optionally copy paste code provided in the next section if you want to try this out.

Optional Lab: PaaS Feature Stores in the Cloud Using Databricks

You may have noticed when using Feast there were a lot of steps and you had to dive deep into the gritty details of data engineering infrastructure and even understand different types of data formats like parquet vs. csv.

If you're a data scientist who wants some of those details abstracted from you, you may consider a Platform as a Service for building your feature store.

Databricks provides a machine learning workspace where feature stores are available without having to configure infrastructure. These feature stores use delta tables in the backend which rely on the open source Parquet format, a column oriented format for big data. Delta tables also come with a delta log that can keep track of transactions on the data, bringing atomicity, consistency, isolation, and durability to machine learning workflows (so-called ACID properties). You can build a feature store by creating a cluster with the ML runtime¹ (12.1 is the latest at the time of writing).

The feature store client allows you to interact with the feature store, register data frames as feature tables, and create training sets consisting of labeled data and training data for use in training pipelines. Databricks also has online feature stores for low latency inference pipelines.

Scaling Pandas Code with a Single Line

If you use Pandas regularly for data wrangling tasks, you may have encountered memory errors. Typically dataframes blow up in memory up at least 2x and sometimes more compared to their size on disk which means if you have a very large csv file, reading that csv file may trigger some out of memory errors if your workflow relies on Pandas. Fortunately, the Pandas on Spark library (formerly Koalas) allows you to write Pandas code to create Spark dataframes and register them in the feature store without having to learn the Spark API. You can import this library in Databricks with the following line (called a drop-in solution).

```
from pyspark import pandas as ps
```

¹Databricks ML runtime documentation can be found at <https://docs.databricks.com/runtime/mlruntime.html>

We've provided an option notebook lab for you called Chapter 4 Lab: Scaling Pandas Workflows provided with this chapter. You can import your notebook into your Databricks workspace and execute the code to get hands-on experience with scaling Pandas code.

Since Databricks requires a cloud subscription, you don't need to complete this lab to understand the rest of the material in this chapter or the rest of the book; however, many organizations use Databricks for MLOps, and knowledge of PySpark, the Pandas on Spark library, clusters, and notebooks may be valuable in an MLOps role. You can import a notebook by clicking Workspace or a user folder and selecting Import as pictured (Figure 4-15):

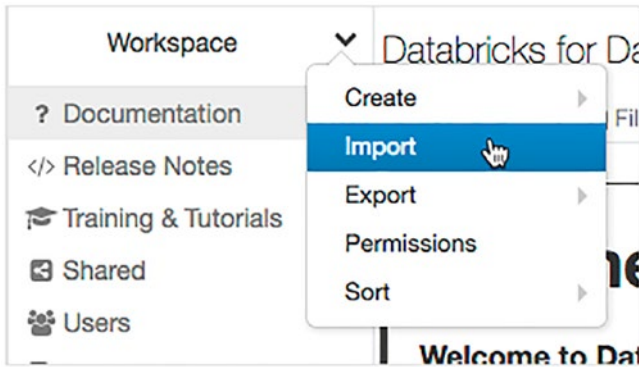


Figure 4-15. Importing a notebook in your workspace

GPU Accelerated Training

GPU accelerated training means using a GPU to help reduce the runtime in training deep learning algorithms. While CPUs are latency optimized, GPUs are bandwidth optimized. Since deep learning involves tensor operations and matrix multiplications which can be implemented on a GPU, these operations can be sped up by using a framework that is GPU aware both because of the data parallelism and the higher bandwidth afforded by a GPU.

One exciting change to the TensorFlow package is that since version 2.1, TensorFlow and TensorFlow-gpu have merged. If you require a version of the TensorFlow package with version ≤ 2.1 , then you can install TensorFlow-gpu as per the following otherwise you may substitute TensorFlow-gpu with TensorFlow.

In your Jupyter lab environment, you can make your notebook GPU aware by using TensorFlow's TensorFlow-gpu package (other deep learning frameworks such as PyTorch require code changes). The steps for configuring GPU awareness in TensorFlow are listed in the following:

1. Uninstall your old TensorFlow.
2. Edit your Dockerfile and add the TensorFlow package to the RUN pip install command (note if for backward compatibility, you require TensorFlow < 2.1 , and then use the older TensorFlow-gpu package instead).

Figure 4-16 shows the informational message.

```

=====
The "tensorflow-gpu" package has been removed!

Please install "tensorflow" instead.

Other than the name, the two packages have been identical
since TensorFlow 2.1, or roughly since Sep 2019. For more
information, see: pypi.org/project/tensorflow-gpu
=====

```

Figure 4-16. *Deprecated packages can cause problems in workflows*

3. Run the docker image with GPU support using docker run.
4. Finally in a Jupyter notebook in your lab environment, you can check install using

```
import tensorflow as tf
tf.config.list_physical_devices('GPU').
```


You should now be able to run GPU accelerated code in TensorFlow without additional code changes. In the following chapter, we'll look at a detailed example of GPU accelerated training using the MNIST data set and the GPU enabled lab environment we just built (optionally, you can use Google Colab if you don't have a physical GPU device). Okay, so we have talked about using hardware to accelerate training, but what about processing large amounts of data? In the next section, we will look at how we can coordinate the processing of massive amounts of data using multiple processors in parallel. These types of databases are called massively parallel analytic databases or MPP.

Databases for Data Science

The distinction between an analytical system and a transactional system is an important one in data science. Transactional systems, also called “online” or operational systems, are designed to handle a large number of very small transactions (e.g., update one row in a table based on a primary key). These types of systems may support business processes like point of sales systems or other operationally critical parts of the business where speed and precision are nonnegotiable.

In contrast, analytical systems are designed to support offline workloads, large volumes of data, and queries over the entire historical data set. These analytical systems are usually implemented as a MPP (massively parallel processing) database. The types of queries that these databases can handle include large CTEs (common table expressions), window analytical functions, and range joins for point in time data sets. Snowflake is one such choice of MPP database. An example of a complex query that uses common table expressions and window analytic functions is given in Listing 4-1.

Listing 4-1. A Common table expression with a window analytic function

```
-- Use a common table expression to deduplicate data
WITH cte AS (
  SELECT id, component, date, value, ROW_NUMBER() OVER
    (PARTITION BY id, component ORDER BY date DESC) AS rn
  FROM sensor_data
)
SELECT id, component, date, value
FROM cte
WHERE rn = 1;
```

In this example, we first create a mock sensor data table “sensor_data” with four columns: id, component, date, and value. We then insert some sample data into this table.

Next, we define a common table expression (CTE) and give it a name. This code is available as part of Chapter 4 (see `example_deduplicate_data.sql`). You can optionally run it by creating a cloud service account on Snowflake. Similar to the Databricks community edition, you can get a free trial using the self-service form on the Snowflake website; however, this is optional and the query will likely run with some modification on most MPP database that supports ANSI SQL since window analytic functions are a part of the standard since 2003. Let’s break this query apart into its component pieces to understand how to write a query:

The SELECT statement is used to select all four columns from the sensor_data table, and we use ROW_NUMBER() window analytic function to assign a unique row number for each row. The PARTITION BY clause ensures that each row number gets reset for each combination of id and component.

Finally, the other SELECT statement selects four columns from our CTE but filters only on rows where the row number is equal to 1. This has the effect of de-duplicating our sensor_data table. You may find queries

like this or even more complex CTEs in typical data science workflows which, for example, in this case may be used to de-duplicate a data set prior to running a train-test-split algorithm, avoiding data leakage. Of course, this is only a simple example.

Snowflake (a type of MPP database) SQL supports a wide range of window analytic and statistical functions for data science tasks such as ranking rows within a partition, calculating running totals, and finding the percentiles of a set of values.

Here are some examples of the types of functions that are commonly used in feature engineering.

- *Ranking functions:* ROW_NUMBER(), RANK(), DENSE_RANK()
- *Aggregate functions:* SUM(), AVG(), MIN(), MAX(), COUNT()
- *Lead and lag functions:* LEAD(), LAG()
- *Percentile functions:* PERCENT_RANK(), PERCENTILE_CONT(), PERCENTILE_DISC()
- *Cumulative distribution functions:* CUME_DIST()
- *Window frame functions:* ROWS BETWEEN, RANGE BETWEEN
- *Date and time functions:* DATE_TRUNC(), DATE_PART(), DATEDIFF()
- *String functions:* CONCAT(), SUBSTRING(), REGEXP_REPLACE()

In the next section, we will briefly detail patterns for enterprise grade database projects so we can get familiar with common architectural patterns.

Patterns for Enterprise Grade Projects

Data lake: A data lake is centralized repository, typically separated into bronze, silver, and gold layers (called the medallion architecture²). The central repository allows you to store both structured and unstructured data, contrasting with a traditional relational database. If you use a cloud storage account like blob storage or s3 buckets, the bronze, silver, and gold layers can map to containers or buckets where you can administer permissions and assign users or service principals access to each container or bucket. The bronze layer is the ingestion layer and should be as close to the raw data sources as possible (you can, e.g., organize raw data sources in folders, but it is important to have a consistent naming convention across the data lake). The silver layer is most important for data science and contains cleaned and conformed data that is still close enough to the source that it can be used for predictive modeling and other data science activities. The gold layer is used for enterprise grade reporting and should contain business-level aggregates.

Data warehouses: Data warehouses are an older pattern and are a centralized repository of data. Data can be integrated from a variety of sources and is loaded using ELT or ETL patterns. The data warehouse can contain dimensional data (slowly changing dimensions) and other tables. This architectural pattern is not well-suited for data science workflows which require flexibility and have to handle schema drift but can be used as a valuable data source for many projects.

Data mesh: A data mesh is a decentralized approach to building data stores that uses self-service design and borrows from domain-oriented design and software development practices. Each domain is responsible for their own data sources, requiring a shift in responsibility, while the data platform team provides a domain-agnostic data platform.

²www.databricks.com/glossary/medallion-architecture

Databases are not only used for feature stores (to organize features for model training) but also for model versioning and artifact storage; in fact, MLFlow also uses a database. Databases are also used for logging and monitoring. This is an important fact that is often overlooked in MLOps. In the next section, we will look at No-SQL databases and how we leverage meta-data in our data science workflows to adapt to change.

No-SQL Databases and Metastores

Relational databases can represent structured data in tables with relationships (foreign keys) between tables. However, not all data can be forced into this pattern. Some data, especially web data (JSON and XML), are semi-structured having nested hierarchies, and text-based data common in NLP problems are unstructured. There is also binary data (common when you have to deal with encrypted columns), and having to store, process, and define the relationships between structured, semi-structured, and unstructured data can be cumbersome and inefficient in a relationship database creating technical complexity. This complexity is compounded by schema evolution common to data science workflows. Hence, there is a need for an efficient way to represent, store, and process semi-structured and unstructured data while meeting nonfunctional requirements like availability, consistency, and other criteria important to the data model. In this section, we will introduce both No-SQL and relational databases that you can use to build data models and meet nonfunctional requirements without having to pigeonhole your solution into a relational database.

- *Cassandra*: Cassandra is a No-SQL distributed database that supports high availability and scalability which makes it ideal as an online feature store.

- *Hive*: Hive is a distributed, fault-tolerant data warehouse system for data analytics at scale. Essentially a data warehouse is a central store of data that you can run queries against. Behind the scenes, these SQL queries are converted into MapReduce jobs, so Hive is an abstraction over MapReduce and is not itself a database.
- *Hive metastore*: Hive metastore is a component you can add to your feature store. It contains names about features such as names of features, data types (called a “schema”). It is also a commonly used component in cloud services like Databricks delta tables, so even if you aren’t building your own feature store directly, you should have some knowledge of this important piece of data infrastructure.

Relational Databases

- *Postgresql*: Postgres is a relational database system that can support gigabytes, terabytes, and even petabytes of data. We can also configure Postgres to work with Hive metastore. In Feast (in version greater than 0.21.0), Postgres is supported as a registry, online and offline feature store.

Introduction to Container Orchestration

We learned about Docker and even created a Dockerfile which was a series of instructions used to build an image. The image, a binary containing layers of software, could be run like a lightweight virtual machine on our host operating system. But what if we have to run multiple services, each with their own Docker images? For example, we might have a service that

hosts a Jupyter notebook where we type in our Python code, but we might want to have another service for storing data in a database and have our notebook interact with this database.

One subtlety you will encounter is networking. How can we get these two services to “talk” to each other and create the network infrastructure to support this communication between services?

Also since containers are ephemeral in nature, how do we spin these services up when we need them and spin them down when they’re no longer needed while persisting the data we need? This is what container orchestration deals with, and the standard tool for orchestrating containers is Docker Compose.

We’ll be using Docker Compose in the next chapter to set up MLFlow and get it to “talk” to our Jupyter lab which we will need to set up experiment tracking for our training pipeline. You can look at the `docker-compose.yml` file included with this chapter (but don’t run any commands just yet). Figure 4-17 shows an example YAML file.

```

1  version: '3.7'
2
3  services:
4    jupyter:
5      build:
6        context: .
7        dockerfile: ./lab/Dockerfile
8      container_name: jupyter
9      restart: always
10     ports:
11       - "8080:8888"
12     environment:
13       - JUPYTER_ENABLE_LAB=yes
14     command: start.sh jupyter lab --LabApp.token=''
15     volumes:
16       - ./notebooks:/home/jovyan/work
17     user: $UID:$GID
18

```

Figure 4-17. A Docker Compose YAML file

Commands for Managing Services in Docker Compose

Container orchestration is a large topic, and as we mentioned, you will be using it in the next chapter to set up MLFlow and build a training pipeline. We'll cover MLFlow in depth, but Docker Compose has commands for managing the entire lifecycle of services and applications.

Here are some important commands useful for managing services:

- *Start services:* `docker-compose up`
- *Stop services:* `docker-compose down`
- *Start a specific service:* `docker-compose up <service name>`
- *Check the version of Docker Compose:* `docker-compose -v`
- *Build all services:* `docker-compose up --build`

While Docker Compose simplifies the process of creating services, you still need to define multi-container applications in a single file. Imagine a scenario where you have infrastructure that spans across different cloud providers or is multitenant in nature. This kind of multitenancy contrasts with multi instance architectures and having a tool that can completely describe infrastructure as code can help with the complexity in these environments.

- **Infrastructure as Code**

Infrastructure as code (IaC) is a DevOps methodology for defining and deploying infrastructure as if it were source code. We have already seen an example of this when we spun up our data science lab environment by defining the

image, binaries, and runtime needed inside the Dockerfile. Since the Dockerfile itself is a series of instructions that can be source controlled and treated like any other source code, we can use it to generate the exact same environment every time we build the image and run the container from the image. The ability to have the same environment each time is called *reproducibility* and is an essential component for data science because experiments need to be reproducible.

It's worth mentioning that there are specific tools and specialties within DevOps for managing infrastructure as code. One tool that is widely used in industry is Terraform. Terraform is an open source infrastructure-as-code tool for provisioning and managing cloud infrastructure such as Databricks. It works with multiple cloud providers and allows MLOps professionals to codify infrastructure in source code that describes the desired end state of the system. An example configuration file is given in the following, but these files can get very complex, and you can use Terraform and similar tools to configure and manage notebooks, clusters, and jobs within Databricks. Figure 4-18 shows a very simple example of infrastructure as code in Terraform.



```
resource "databricks_repo" "this" {  
  url = "https://github.com/user/demo.git"  
}
```

Figure 4-18. *Infrastructure can be described as code*

Making Technical Decisions

We've come a long way in this chapter from introducing Docker, applying what we learned to create our own data science lab environment complete with Jupyter notebook, and getting our hands dirty with Feast, creating our own feature store from an IoT data set.

We've also talked about the philosophy of having infrastructure as code and why it's important for the reproducibility of data science experiments. The final piece of the puzzle is how we can use our knowledge of infrastructure to make better technical decisions. Here are a few key points you should consider when making decisions around infrastructure in your own projects:

- Solve problems using a divide-and-conquer strategy, breaking services and parts of applications into functional components.
- Ask yourself if there is a cloud service or a docker container you might want to use for each functional component in your system.
- Understand the performance requirements for your workload. Do you need a lot of memory? Or do you need dedicated CPUs and GPUs for model training? Understanding the hardware requirements for different models can help you decide.
- Run code profilers on your code to identify bottlenecks in a data-driven way. A great profiler that comes with Python is cProfiler. It's often not enough to "guess"; you should strive to make data-driven decisions by *performance testing* your code.

- Strive to make your experiments reproducible, and deploying by using Docker and adopting a mentality of infrastructure as code can help to manage changes and different versions of infrastructure.
- Decide between PaaS (Platform as a Service) and Infrastructure as a Service. Sometimes, spinning up your own dedicated server and worrying about upgrades, updates, and security batches can be overkilled when a good PaaS meets infrastructure requirements.

Summary

In this chapter, we've learned the fundamentals of infrastructure for MLOps. We've covered sufficient prerequisites for understanding containerization, cloud services, hardware accelerated training, and container orchestration and how we can use our knowledge to become better technical decision-makers on data science projects. At this point, you should understand what a container is and how to build containers and be able to define what container orchestration means and why it is useful for MLOps. Although this chapter covered a lot of ground and you're not expected to know everything about containerization, we hope this chapter has peaked your curiosity as we start to build on these fundamentals and apply what we learned to some real data science problems in the coming chapters. Here is a summary of the topics we've covered:

- Containerization for Data Scientists
- Hardware Accelerated Training
- Feature Store Pattern and Feast

CHAPTER 4 INFRASTRUCTURE FOR MLOPS

- GPU Accelerated Training
- MPP Databases for Data Science
- Introduction to Container Orchestration
- Cloud Services and Infrastructure as Code
- Making Technical Decisions