# Chapter 10. Infrastructure and Tooling for MLOps

In Chapters 4 to 6, we discussed the logic for developing ML systems. In Chapters 7 to 9, we discussed the considerations for deploying, monitoring, and continually updating an ML system. Up until now, we've assumed that ML practitioners have access to all the tools and infrastructure they need to implement that logic and carry out these considerations. However, that assumption is far from being true. Many data scientists have told me that they know the right things to do for their ML systems, but they can't do them because their infrastructure isn't set up in a way that enables them to do so.

ML systems are complex. The more complex a system, the more it can benefit from good infrastructure. Infrastructure, when set up right, can help automate processes, reducing the need for specialized knowledge and engineering time. This, in turn, can speed up the development and delivery of ML applications, reduce the surface area for bugs, and enable new use cases. When set up wrong, however, infrastructure is painful to use and expensive to replace. In this chapter, we'll discuss how to set up infrastructure right for ML systems.

Before we dive in, it's important to note that every company's infrastructure needs are different. The infrastructure required for you depends on the number of applications you develop and how specialized the applications are. At one end of the spectrum, you have companies that use ML for ad hoc business analytics such as to project the number of new users they'll have next year to present at their quarterly planning meeting. These companies probably won't need to invest in any infrastructure—Jupyter Notebooks, Python, and Pandas would be their best friends. If you have only one simple ML use case, such as an Android app for object detection to show your friends, you probably won't need any infrastructure either—you just need an Android-compatible ML framework like TensorFlow Lite.

At the other end of the spectrum, there are companies that work on applications with unique requirements. For example, self-driving cars have unique accuracy and latency requirements—the algorithm must be able to respond within milliseconds and its accuracy must be near-perfect since a wrong prediction can lead to serious accidents. Similarly, Google Search has a unique scale requirement since most companies don't process 63,000 search queries a second, which translates to 234 million search queries an hour, like Google does.[1] These companies will likely need to develop their own highly specialized infrastructure. Google developed a large part of their internal infrastructure for search; so did self-driving car companies like Tesla and Waymo.[2] It's common that part of specialized infrastructure is later made public and adopted by other companies. For example, Google extended their internal cloud infrastructure to the public, resulting in Google Cloud Platform.

In the middle of the spectrum are the majority of companies, those who use ML for multiple common applications—a fraud detection model, a price optimization model, a churn prediction model, a recommender system, etc.—at reasonable scale. "Reasonable scale" refers to companies that work with data in the order of gigabytes and terabytes, instead of petabytes, a day. Their data science team might range from 10 to hundreds of engineers.[3] This category might include any company from a 20-person startup to a company at Zillow's scale, but not at FAAAM scale.[4] For example, back in 2018, Uber was adding tens of terabytes of data a day to their data lake, and Zillow's biggest dataset was bringing in 2 terabytes of uncompressed data a day.[5] In contrast, even back in 2014, Facebook was generating *4 petabytes* of data a day.[6]

Companies in the middle of the spectrum will likely benefit from generalized ML infrastructure that is being increasingly standardized (see Figure 10-1). In this book, we'll focus on the infrastructure for the vast majority of ML applications at a reasonable scale.

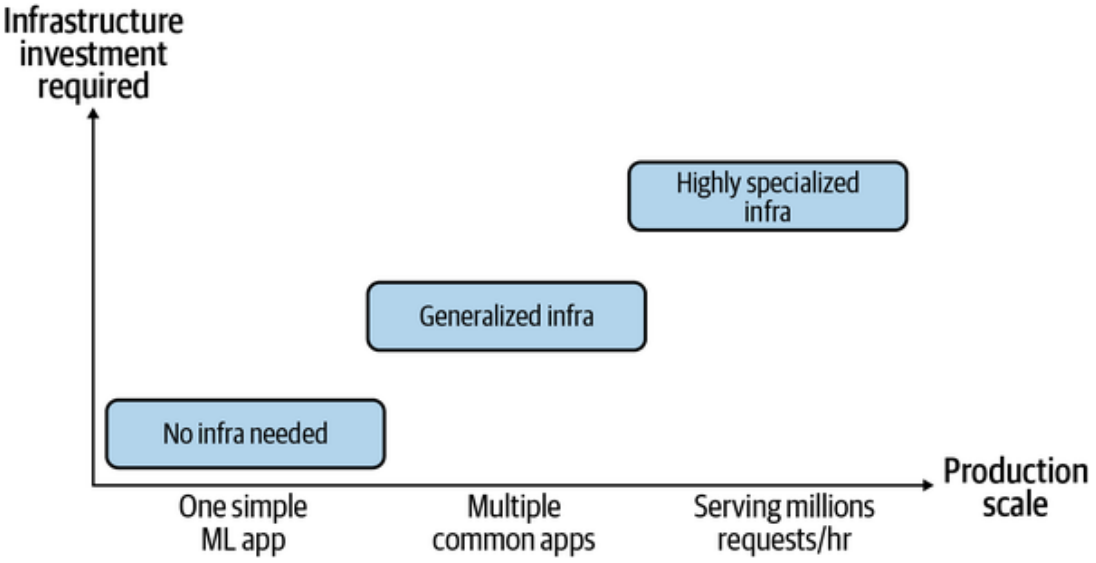


Figure 10-1. Infrastructure requirements for companies at different production scales

In order to set up the right infrastructure for your needs, it's important to understand exactly what infrastructure means and what it consists of. According to Wikipedia, in the physical world, "infrastructure is the set of fundamental facilities and systems that support the sustainable functionality of households and firms."[7] In the ML world, infrastructure is the set of fundamental facilities that support the development and maintenance of ML systems. What should be considered the "fundamental facilities" varies greatly from company to company, as discussed earlier in this chapter. In this section, we will examine the following four layers:

*Storage and compute*

The storage layer is where data is collected and stored. The compute layer provides the compute needed to run your ML workloads such as training a model, computing features, generating features, etc.

*Resource management*

Resource management comprises tools to schedule and orchestrate your workloads to make the most out of your available compute resources. Examples of tools in this category include Airflow, Kubeflow, and Metaflow.

*ML platform*

This provides tools to aid the development of ML applications such as model stores, feature stores, and monitoring tools. Examples of

tools in this category include SageMaker and MLflow.

*Development environment*

This is usually referred to as the dev environment; it is where code is written and experiments are run. Code needs to be versioned and tested. Experiments need to be tracked.

These four different layers are shown in Figure 10-2. Data and compute are the essential resources needed for any ML project, and thus the *storage and compute layer* forms the infrastructural foundation for any company that wants to apply ML. This layer is also the most abstract to a data scientist. We'll discuss this layer first because these resources are the easiest to explain.

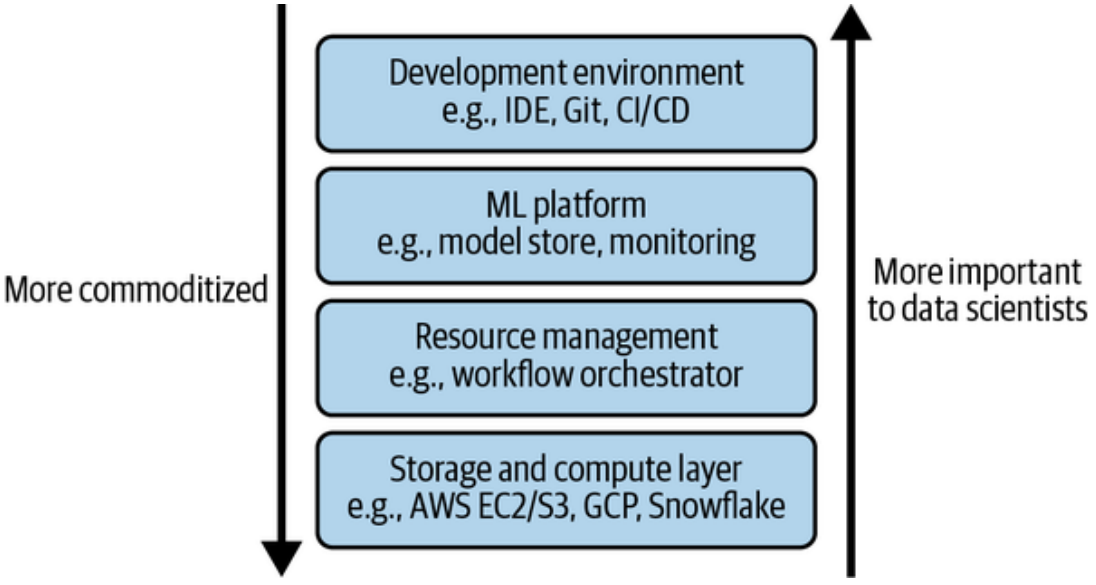


Figure 10-2. Different layers of infrastructure for ML

The dev environment is what data scientists have to interact with daily, and therefore, it is the least abstract to them. We'll discuss this category next, then we'll discuss resource management, a contentious topic among data scientists—people are still debating whether a data scientist needs to know about this layer or not. Because "ML platform" is a relatively new concept with its different components still maturing, we'll discuss this category last, after we've familiarized ourselves with all other categories. An ML platform requires up-front investment from a company, but if it's done right, it can make the life of data scientists across business use cases at that company so much easier.

Even if two companies have the exact same infrastructure needs, their resulting infrastructure might look different depending on their approaches to build versus buy decisions—i.e., what they want to build in-house versus what they want to outsource to other companies. We'll discuss the build versus buy decisions in the last part of this chapter, where we'll also discuss the hope for standardized and unified abstractions for ML infrastructure.

Let's dive in!

## Storage and Compute

ML systems work with a lot of data, and this data needs to be stored somewhere. The *storage layer* is where data is collected and stored. At its simplest form, the storage layer can be a hard drive disk (HDD) or a solid state disk (SSD). The storage layer can be in one place, e.g., you might have all your data in Amazon S3 or in Snowflake, or spread out over multiple locations.[8] Your storage layer can be on-prem in a private data center or on the cloud. In the past, companies might have tried to manage their own storage layer. However, in the last decade, the storage layer has been mostly commoditized and moved to the cloud. Data storage has become so cheap that most companies just store all the data they have without the cost.[9] We've covered the data layer intensively in Chapter 3, so in this chapter, we'll focus on the compute layer.

The *compute layer* refers to all the compute resources a company has access to and the mechanism to determine how these resources can be used. The amount of compute resources available determines the scalability of your workloads. You can think of the compute layer as the engine to execute your jobs. At its simplest form, the compute layer can just be a single CPU core or a GPU core that does all your computation. Its most common form is cloud compute managed by a cloud provider such as AWS Elastic Compute Cloud (EC2) or GCP.

The compute layer can usually be sliced into smaller compute units to be used concurrently. For example, a CPU core might support two concurrent threads; each thread is used as a compute unit to execute its own job. Or multiple CPU cores might be joined together to form a larger compute

unit to execute a larger job. A compute unit can be created for a specific short-lived job such as an AWS Step Function or a GCP Cloud Run—the unit will be eliminated after the job finishes. A compute unit can also be created to be more "permanent," aka without being tied to a job, like a virtual machine. A more permanent compute unit is sometimes called an "instance."

However, the compute layer doesn't always use threads or cores as compute units. There are compute layers that abstract away the notions of cores and use other units of computation. For example, computation engines like Spark and Ray use "job" as their unit, and Kubernetes uses "pod," a wrapper around containers, as its smallest deployable unit. While you can have multiple containers in a pod, you can't independently start or stop different containers in the same pod.

To execute a job, you first need to load the required data into your compute unit's memory, then execute the required operations—addition, multiplication, division, convolution, etc.—on that data. For example, to add two arrays, you will first need to load these two arrays into memory, and then perform addition on the two arrays. If the compute unit doesn't have enough memory to load these two arrays, the operation will be impossible without an algorithm to handle out-of-memory computation. Therefore, a compute unit is mainly characterized by two metrics: how much memory it has and how fast it runs an operation.

The memory metric can be specified using units like GB, and it's generally straightforward to evaluate: a compute unit with 8 GB of memory can handle more data in memory than a compute unit with only 2 GB, and it is generally more expensive.[10] Some companies care not only how much memory a compute unit has but also how fast it is to load data in and out of memory, so some cloud providers advertise their instances as having "high bandwidth memory" or specify their instances' I/O bandwidth.

The operation speed is more contentious. The most common metric is FLOPS—floating point operations per second. As the name suggests, this metric denotes the number of float point operations a compute unit can run per second. You might see a hardware vendor advertising that their GPUs or TPUs or IPUs (intelligence processing units) have teraFLOPS (one trillion FLOPS) or another massive number of FLOPS.

However, this metric is contentious because, first, companies that measure this metric might have different ideas on what is counted as an operation. For example, if a machine fuses two operations into one and executes this fused operation,[11] does this count as one operation or two? Second, just because a compute unit is capable of doing a trillion FLOPS doesn't mean you'll be able to execute your job at the speed of a trillion FLOPS. The ratio of the number of FLOPS a job can run to the number of FLOPs a compute unit is capable of handling is called utilization.[12] If an instance is capable of doing a million FLOPs and your job runs with 0.3 million FLOPS, that's a 30% utilization rate. Of course, you'd want to have your utilization rate as high as possible. However, it's near impossible to achieve 100% utilization rate. Depending on the hardware backend and the application, the utilization rate of 50% might be considered good or bad. Utilization also depends on how fast you can load data into memory to perform the next operations—hence the importance of I/O bandwidth.[13]

When evaluating a new compute unit, it's important to evaluate how long it will take this compute unit to do common workloads. For example, MLPerf is a popular benchmark for hardware vendors to measure their hardware performance by showing how long it will take their hardware to train a ResNet-50 model on the ImageNet dataset or use a BERT-large model to generate predictions for the SQuAD dataset.

Because thinking about FLOPS is not very useful, to make things easier, when evaluating compute performance, many people just look into the number of cores a compute unit has. So you might use an instance with 4 CPU cores and 8 GB of memory. Keep in mind that AWS uses the concept of vCPU, which stands for virtual CPU and which, for practical purposes, can be thought of as half a physical core.[14] You can see the number of cores and memory offered by some AWS EC2 and GCP instances in Figure 10-3.

Some GPU instances on AWS

| Instance | GPUs | vCPU | Mem (GiB) | GPU Mem (GiB) |
|---|---|---|---|---|
| p3.2xlarge | 1 | 8 | 61 | 16 |
| p3.8xlarge | 4 | 32 | 244 | 64 |
| p3.16xlarge | 8 | 64 | 488 | 128 |
| p3dn.24xlarge | 8 | 96 | 768 | 256 |

Some TPU instances on GCP

| TPU type (v2) | v2 cores | Total memory |
|---|---|---|
| v2-8 | 8 | 64 GiB |
| **TPU type (v3)** | **v3 cores** | **Total memory** |
| v3-8 | 8 | 128 GiB |

Figure 10-3. Examples of GPU and TPU instances available on AWS and GCP as of February 2022. Source: Screenshots of AWS and GCP websites

## Public Cloud Versus Private Data Centers

Like data storage, the compute layer is largely commoditized. This means that instead of setting up their own data centers for storage and compute, companies can pay cloud providers like AWS and Azure for the exact amount of compute they use. Cloud compute makes it extremely easy for companies to start building without having to worry about the compute layer. It's especially appealing to companies that have variable-sized workloads. Imagine if your workloads need 1,000 CPU cores one day of the year and only 10 CPU cores the rest of the year. If you build your own data centers, you'll need to pay for 1,000 CPU cores up front. With cloud compute, you only need to pay for 1,000 CPU cores one day of the year and 10 CPU cores the rest of the year. It's convenient to be able to just add more compute or shut down instances as needed—most cloud providers even do that automatically for you—reducing engineering operational overhead. This is especially useful in ML as data science workloads are bursty. Data scientists tend to run experiments a lot for a few weeks during development, which requires a surge of compute power. Later on, during production, the workload is more consistent.

Keep in mind that cloud compute is elastic but not magical. It doesn't actually have infinite compute. Most cloud providers offer limits on the compute resources you can use at a time. Some, but not all, of these limits can be raised through petitions. For example, as of writing this book, AWS EC2's largest instance is X1e with 128 vCPUs and almost 4 TB of memory.[15] Having a lot of compute resources doesn't mean that it's always easy to use them, especially if you have to work with spot instances to save cost.[16]

Due to the cloud's elasticity and ease of use, more and more companies are choosing to pay for the cloud over building and maintaining their own storage and compute layer. Synergy Research Group's research shows that in 2020, "enterprise spending on cloud infrastructure services [grew] by 35% to reach almost $130 billion" while "enterprise spending on data [centers] dropped by 6% to under $90 billion,"[17] as shown in Figure 10-4.

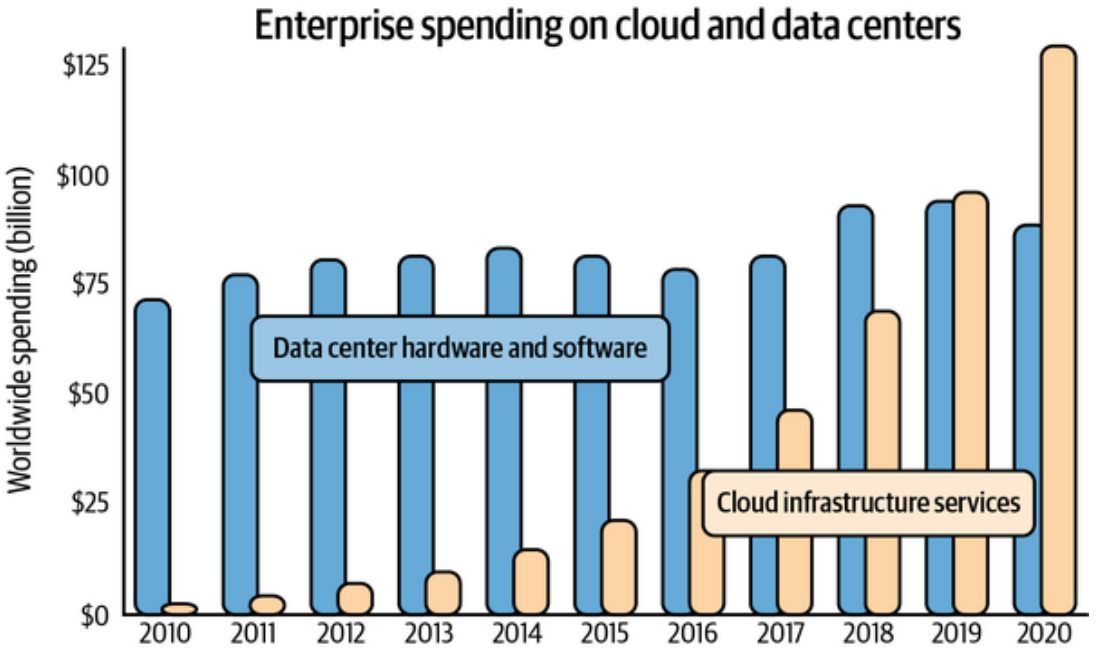


Figure 10-4. In 2020, enterprise spending on cloud infrastructure services grew by 35% while spending on data centers dropped by 6%. Source: Adapted from an image by Synergy Research Group

While leveraging the cloud tends to give companies higher returns than building their own storage and compute layers early on, this becomes less defensible as a company grows. Based on disclosed cloud infrastructure spending by public software companies, the venture capital firm a16z shows that cloud spending accounts for approximately 50% cost of revenue of these companies.[18]

The high cost of the cloud has prompted companies to start moving their workloads back to their own data centers, a process called "cloud repatriation." Dropbox's S-1 filing in 2018 shows that the company was able to save $75M over the two years prior to IPO due to their infrastructure optimization overhaul—a large chunk of it consisted of moving their workloads from public cloud to their own data centers. Is the high cost of cloud unique to Dropbox because Dropbox is in the data storage business? Not quite. In the aforementioned analysis, a16z estimated that "across 50 of

the top public software companies currently utilizing cloud infrastructure, an estimated $100B of market value is being lost among them due to cloud impact on margins—relative to running the infrastructure themselves."[19]

While getting started with the cloud is easy, moving away from the cloud is hard. Cloud repatriation requires nontrivial up-front investment in both commodities and engineering effort. More and more companies are following a hybrid approach: keeping most of their workloads on the cloud but slowly increasing their investment in data centers.

ON MULTICLOUD STRATEGY

Another way for companies to reduce their dependence on any single cloud provider is to follow a multicloud strategy: spreading their workloads on multiple cloud providers.[20] This allows companies to architect their systems so that they can be compatible with multiple clouds, enabling them to leverage the best and most cost-effective technologies available instead of being stuck with the services provided by a single cloud provider, a situation known as vendor lock-in. A 2019 study by Gartner shows that 81% of organizations are working with two or more public cloud providers.[21] A common pattern that I've seen for ML workloads is to do training on GCP or Azure, and deployment on AWS.

The multicloud strategy doesn't usually happen by choice. As Josh Wills, one of our early reviewers, put it: "Nobody in their right mind intends to use multicloud." It's incredibly hard to move data and orchestrate workloads across clouds.

Often, multicloud just happens because different parts of the organization operate independently, and each part makes their own cloud decision. It can also happen following an acquisition—the acquired team is already on a cloud different from the host organization, and migrating hasn't happened yet.

In my work, I've seen multicloud happen due to strategic investments. Microsoft and Google are big investors in the startup ecosystem, and several companies that I work with that were previously on AWS have moved to Azure/GCP after Microsoft/Google invested in them.

# Development Environment

The dev environment is where ML engineers write code, run experiments, and interact with the production environment where champion models are deployed and challenger models evaluated. The dev environment consists of the following components: IDE (integrated development environment), versioning, and CI/CD.

If you're a data scientist or ML engineer who writes code daily, you're probably very familiar with all these tools and might wonder what there is to say about them. In my experience, outside of a handful of tech companies, the dev environment is severely underrated and underinvested in at most companies. According to Ville Tuulos in his book *Effective Data Science Infrastructure*, "you would be surprised to know how many companies have well-tuned, scalable production infrastructure but the question of how the code is developed, debugged, and tested in the first place is solved in an ad-hoc manner."[22]

He suggested that "if you have time to set up only one piece of infrastructure well, make it the development environment for data scientists." Because the dev environment is where engineers work, improvements in the dev environment translate directly into improvements in engineering productivity.

In this section, we'll first cover different components of the dev environment, then we'll discuss the standardization of the dev environment before we discuss how to bring your changes from the dev environment to the production environment with containers.

## Dev Environment Setup

The dev environment should be set up to contain all the tools that can make it easier for engineers to do their job. It should also consist of tools for *versioning*. As of this writing, companies use an ad hoc set of tools to version their ML workflows, such as Git to version control code, DVC to version data, Weights & Biases or Comet.ml to track experiments during development, and MLflow to track artifacts of models when deploying them. Claypot AI is working on a platform that can help you version and track all your ML workflows in one place. Versioning is important for any software engineering projects, but even more so for ML projects because of both the sheer number of things you can change (code, parameters, the data itself, etc.) and the need to keep track of prior runs to reproduce later on. We've covered this in the section "Experiment Tracking and Versioning".

The dev environment should also be set up with a *CI/CD* test suite to test your code before pushing it to the staging or production environment.

Examples of tools to orchestrate your CI/CD test suite are GitHub Actions and CircleCI. Because CI/CD is a software engineering concern, it's beyond the scope of this book.

In this section, we'll focus on the place where engineers write code: the IDE.

### IDE

The *IDE* is the editor where you write your code. IDEs tend to support multiple programming languages. IDEs can be native apps like VS Code or Vim. IDEs can be browser-based, which means they run in browsers, such as AWS Cloud9.

Many data scientists write code not just in IDEs but also in notebooks like Jupyter Notebooks and Google Colab.[23] Notebooks are more than just places to write code. You can include arbitrary artifacts such as images, plots, data in nice tabular formats, etc., which makes notebooks very useful for exploratory data analysis and analyzing model training results.

Notebooks have a nice property: they are stateful—they can retain states after runs. If your program fails halfway through, you can rerun from the failed step instead of having to run the program from the beginning. This is especially helpful when you have to deal with large datasets that might take a long time to load. With notebooks, you only need to load your data once—notebooks can retain this data in memory—instead of having to load it each time you want to run your code. As shown in Figure 10-5, if your code fails at step 4 in a notebook, you'll only need to rerun step 4 instead of from the beginning of your program.

```
In [1]: import pandas as pd

In [2]: fname = "large-dataset.csv"

In [3]: df = pd.read_csv(fname)

In [4]: features = df["Timestamp", "Cost"]

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~/miniconda3/envs/stove39/lib/python3.9/site-packages/pandas/core/indexes/base.py
ance)
   3360             try:
-> 3361                 return self._engine.get_loc(casted_key)
   3362             except KeyError as err:
```

Figure 10-5. In Jupyter Notebooks, if step 4 fails, you only need to run step 4 again, instead of having to run steps 1 to 4 again

Note that this statefulness can be a double-edged sword, as it allows you to execute your cells out of order. For example, in a normal script, cell 4 must run after cell 3 and cell 3 must run after cell 2. However, in notebooks, you can run cell 2, 3, then 4 or cell 4, 3, then 2. This makes notebook reproducibility harder unless your notebook comes with an instruction on the order in which to run your cells. This difficulty is captured in a joke by Chris Albon (see Figure 10-6).

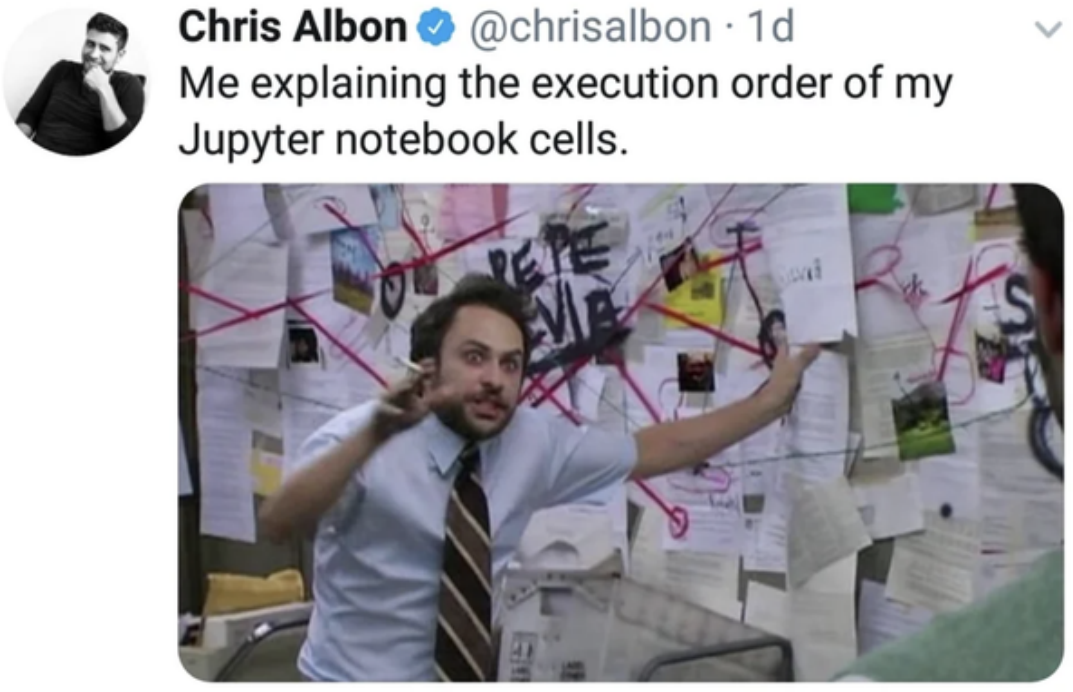


Figure 10-6. Notebooks' statefulness allows you to execute cells out of order, making it hard to reproduce a notebook

Because notebooks are so useful for data exploration and experiments, notebooks have become an indispensable tool for data scientists and ML. Some companies have made notebooks the center of their data science infrastructure. In their seminal article, "Beyond Interactive: Notebook

Innovation at Netflix," Netflix included a list of infrastructure tools that can be used to make notebooks even more powerful.[24] The list includes:

*Papermill*

For spawning multiple notebooks with different parameter sets—such as when you want to run different experiments with different sets of parameters and execute them concurrently. It can also help summarize metrics from a collection of notebooks.

*Commuter*

A notebook hub for viewing, finding, and sharing notebooks within an organization.

Another interesting project aimed at improving the notebook experience is nbdev, a library on top of Jupyter Notebooks that encourages you to write documentation and tests in the same place.

## Standardizing Dev Environments

The first thing about the dev environment is that it should be standardized, if not company-wide, then at least team-wide. We'll go over a story to understand what it means to have the dev environment standardized and why that is needed.

In the early days of our startup, we each worked from our own computer. We had a bash file that a new team member could run to create a new virtual environment—in our case, we use conda for virtual environments—and install the required packages needed to run our code. The list of the required packages was the good old *requirements.txt* that we kept adding to as we started using a new package. Sometimes, one of us got lazy and we just added a package name (e.g., `torch`) without specifying which version of the package it was (e.g., `torch==1.10.0+cpu`). Occasionally, a new pull request would run well on my computer but not another coworker's computer,[25] and we usually quickly figured out that it was because we used different versions of the same package. We resolved to always specify the package name together with the package version when adding a new package to the *requirements.txt*, and that removed a lot of unnecessary headaches.

One day, we ran into this weird bug that only happened during some runs and not others. I asked my coworker to look into it, but he wasn't able to reproduce the bug. I told him that the bug only happened some of the time, so he might have to run the code around 20 times just to be sure. He ran the code 20 times and still found nothing. We compared our packages and everything matched. After a few hours of hair-pulling frustration, we discovered that it was a concurrency issue that is only an issue for Python version 3.8 or earlier. I had Python 3.8 and my coworker had Python 3.9, so he didn't see the bug. We resolved to have everyone on the same Python version, and that removed some more headaches.

Then one day, my coworker got a new laptop. It was a MacBook with the then new M1 chip. He tried to follow our setup steps on this new laptop but ran into difficulty. It was because the M1 chip was new, and some of the tools we used, including Docker, weren't working well with M1 chips yet. After seeing him struggling with setting the environment up for a day, we decided to move to a cloud dev environment. This means that we still standardize the virtual environment and tools and packages, but now everyone uses the virtual environment and tools and packages on the same type of machine too, provided by a cloud provider.

When using a cloud dev environment, you can use a cloud dev environment that also comes with a cloud IDE like AWS Cloud9 (which has no built-in notebooks) and Amazon SageMaker Studio (which comes with hosted JupyterLab). As of writing this book, Amazon SageMaker Studio seems more widely used than Cloud9. However, most engineers I know who use cloud IDEs do so by installing IDEs of their choice, like Vim, on their cloud instances.

A much more popular option is to use a cloud dev environment with a local IDE. For example, you can use VS Code installed on your computer and connect the local IDE to the cloud environment using a secure protocol like Secure Shell (SSH).

While it's generally agreed upon that tools and packages should be standardized, some companies are hesitant to standardize IDEs. Engineers can get emotionally attached to IDEs, and some have gone to great length to defend their IDE of choice,[26] so it'll be hard forcing everyone to use the same IDE. However, over the years, some IDEs have emerged to be the

most popular. Among them, VS Code is a good choice since it allows easy integration with cloud dev instances.

At our startup, we chose GitHub Codespaces as our cloud dev environment, but an AWS EC2 or a GCP instance that you can SSH into is also a good option. Before moving to cloud environments, like many other companies, we were worried about the cost—what if we forgot to shut down our instances when not in use and they kept charging us money? However, this worry has gone away for two reasons. First, tools like GitHub Codespaces automatically shut down your instance after 30 minutes of inactivity. Second, some instances are pretty cheap. For example, an AWS instance with 4 vCPUs and 8 GB of memory costs around $0.1/hour, which comes to approximately $73/month if you never shut it down. Because engineering time is expensive, if a cloud dev environment can help you save a few hours of engineering time a month, it's worth it for many companies.

Moving from local dev environments to cloud dev environments has many other benefits. First, it makes IT support so much easier—imagine having to support 1,000 different local machines instead of having to support only one type of cloud instance. Second, it's convenient for remote work—you can just SSH into your dev environment wherever you go from any computer. Third, cloud dev environments can help with security. For example, if an employee's laptop is stolen, you can just revoke access to cloud instances from that laptop to prevent the thief from accessing your codebase and proprietary information. Of course, some companies might not be able to move to cloud dev environments also because of security concerns. For example, they aren't allowed to have their code or data on the cloud.

The fourth benefit, which I would argue is the biggest benefit for companies that do production on the cloud, is that having your dev environment on the cloud reduces the gap between the dev environment and the production environment. If your production environment is in the cloud, bringing your dev environment to the cloud is only natural.

Occasionally, a company has to move their dev environments to the cloud not only because of the benefits, but also out of necessity. For the use cases where data can't be downloaded or stored on a local machine, the

only way to access it is via a notebook in the cloud (SageMaker Studio) that can read the data from S3, provided it has the right permissions.

Of course, cloud dev environments might not work for every company due to cost, security, or other concerns. Setting up cloud dev environments also requires some initial investments, and you might need to educate your data scientists on cloud hygiene, including establishing secure connections to the cloud, security compliance, or avoiding wasteful cloud usage. However, standardization of dev environments might make your data scientists' lives easier and save you money in the long run.

## From Dev to Prod: Containers

During development, you might usually work with a fixed number of machines or instances (usually one) because your workloads don't fluctuate a lot—your model doesn't suddenly change from serving only 1,000 requests an hour to 1 million requests an hour.

A production service, on the other hand, might be spread out on multiple instances. The number of instances changes from time to time depending on the incoming workloads, which can be unpredictable at times. For example, a celebrity tweets about your fledgling app and suddenly your traffic spikes 10x. You will have to turn on new instances as needed, and these instances will need to be set up with required tools and packages to execute your workloads.

Previously, you'd have to spin up and shut down instances yourself, but most public cloud providers have taken care of the autoscaling part. However, you still have to worry about setting up new instances.

When you consistently work with the same instance, you can install dependencies once and use them whenever you use this instance. In production, if you dynamically allocate instances as needed, your environment is inherently stateless. When a new instance is allocated for your workload, you'll need to install dependencies using a list of predefined instructions.

A question arises: how do you re-create an environment on any new instance? Container technology—of which Docker is the most popular—is

designed to answer this question. With Docker, you create a Dockerfile with step-by-step instructions on how to re-create an environment in which your model can run: install this package, download this pretrained model, set environment variables, navigate into a folder, etc. These instructions allow hardware anywhere to run your code.

Two key concepts in Docker are image and container. Running all the instructions in a Dockerfile gives you a Docker image. If you run this Docker image, you get back a Docker container. You can think of a Dockerfile as the recipe to construct a mold, which is a Docker image. From this mold, you can create multiple running instances; each is a Docker container.

You can build a Docker image either from scratch or from another Docker image. For example, NVIDIA might provide a Docker image that contains TensorFlow and all necessary libraries to optimize TensorFlow for GPUs. If you want to build an application that runs TensorFlow on GPUs, it's not a bad idea to use this Docker image as your base and install dependencies specific to your application on top of this base image.

A container registry is where you can share a Docker image or find an image created by other people to be shared publicly or only with people inside their organizations. Common container registries include Docker Hub and AWS ECR (Elastic Container Registry).

Here's an example of a simple Dockerfile that runs the following instructions. The example is to show how Dockerfiles work in general, and might not be executable.

1. Download the latest PyTorch base image.
2. Clone NVIDIA's apex repository on GitHub, navigate to the newly created *apex* folder, and install apex.
3. Set *fancy-nlp-project* to be the working directory.
4. Clone Hugging Face's transformers repository on GitHub, navigate to the newly created *transformers* folder, and install transformers.

```
FROM pytorch/pytorch:latest
RUN git clone https://github.com/NVIDIA/apex
RUN cd apex && \
```

```
    python3 setup.py install && \
    pip install -v --no-cache-dir --global-option="--cpp_ext" \
    --global-option="--cuda_ext" ./

WORKDIR /fancy-nlp-project
RUN git clone https://github.com/huggingface/transformers.git && \
    cd transformers && \
    python3 -m pip install --no-cache-dir.
```

If your application does anything interesting, you will probably need more than one container. Consider the case where your project consists of the featurizing code that is fast to run but requires a lot of memory, and the model training code that is slow to run but requires less memory. If you run both parts of the code on the same GPU instances, you'll need GPU instances with high memory, which can be very expensive. Instead, you can run your featurizing code on CPU instances and the model training code on GPU instances. This means you'll need one container for featurizing and another container for training.

Different containers might also be necessary when different steps in your pipeline have conflicting dependencies, such as your featurizer code requires NumPy 0.8 but your model requires NumPy 1.0.

If you have 100 microservices and each microservice requires its own container, you might have 100 containers running at the same time. Manually building, running, allocating resources for, and stopping 100 containers might be a painful chore. A tool to help you manage multiple containers is called container orchestration. Docker Compose is a lightweight container orchestrator that can manage containers on a single host.

However, each of your containers might run on its own host, and this is where Docker Compose is at its limits. Kubernetes (K8s) is a tool for exactly that. K8s creates a network for containers to communicate and share resources. It can help you spin up containers on more instances when you need more compute/memory as well as shutting down containers when you no longer need them, and it helps maintain high availability for your system.

K8s was one of the fastest-growing technologies in the 2010s. Since its inception in 2014, it's become ubiquitous in production systems today. Jeremy Jordan has a great introduction to K8s for readers interested in learning more. However, K8s is not the most data-scientist-friendly tool, and there have been many discussions on how to move data science workloads away from it.[27] We'll go more into K8s in the next section.

## Resource Management

In the pre-cloud world (and even today in companies that maintain their own data centers), storage and compute were finite. Resource management back then centered around how to make the most out of limited resources. Increasing resources for one application could mean decreasing resources for other applications, and complex logic was required to maximize resource utilization, even if that meant requiring more engineering time.

However, in the cloud world where storage and compute resources are much more elastic, the concern has shifted from how to maximize resource utilization to how to use resources cost-effectively. Adding more resources to an application doesn't mean decreasing resources for other applications, which significantly simplifies the allocation challenge. Many companies are OK with adding more resources to an application as long as the added cost is justified by the return, e.g., extra revenue or saved engineering time.

In the vast majority of the world, where engineers' time is more valuable than compute time, companies are OK using more resources if this means it can help their engineers become more productive. This means that it might make sense for companies to invest in automating their workloads, which might make using resources less efficient than manually planning their workloads, but free their engineers to focus on work with higher returns. Often, if a problem can be solved by either using more non-human resources (e.g., throwing more compute at it) or using more human resources (e.g., requiring more engineering time to redesign), the first solution might be preferred.

In this section, we'll discuss how to manage resources for ML workflows. We'll focus on cloud-based resources; however, the discussed ideas can also be applicable for private data centers.

## Cron, Schedulers, and Orchestrators

There are two key characteristics of ML workflows that influence their resource management: repetitiveness and dependencies.

In this book, we've discussed at length how developing ML systems is an iterative process. Similarly, ML workloads are rarely one-time operations but something repetitive. For example, you might train a model every week or generate a new batch of predictions every four hours. These repetitive processes can be scheduled and orchestrated to run smoothly and cost-effectively using available resources.

Scheduling repetitive jobs to run at fixed times is exactly what *cron* does. This is also all that cron does: run a script at a predetermined time and tell you whether the job succeeds or fails. It doesn't care about the dependencies between the jobs it runs—you can run job A after job B with cron but you can't schedule anything complicated like run B if A succeeds and run C if A fails.

This leads us to the second characteristic: dependencies. Steps in an ML workflow might have complex *dependency* relationships with each other. For example, an ML workflow might consist of the following steps:

1. Pull last week's data from data warehouses.
2. Extract features from this pulled data.
3. Train two models, A and B, on the extracted features.
4. Compare A and B on the test set.
5. Deploy A if A is better; otherwise deploy B.

Each step depends on the success of the previous step. Step 5 is what we call conditional dependency: the action for this step depends on the outcome of the previous step. The order of execution and dependencies among these steps can be represented using a graph, as shown in Figure 10-7.

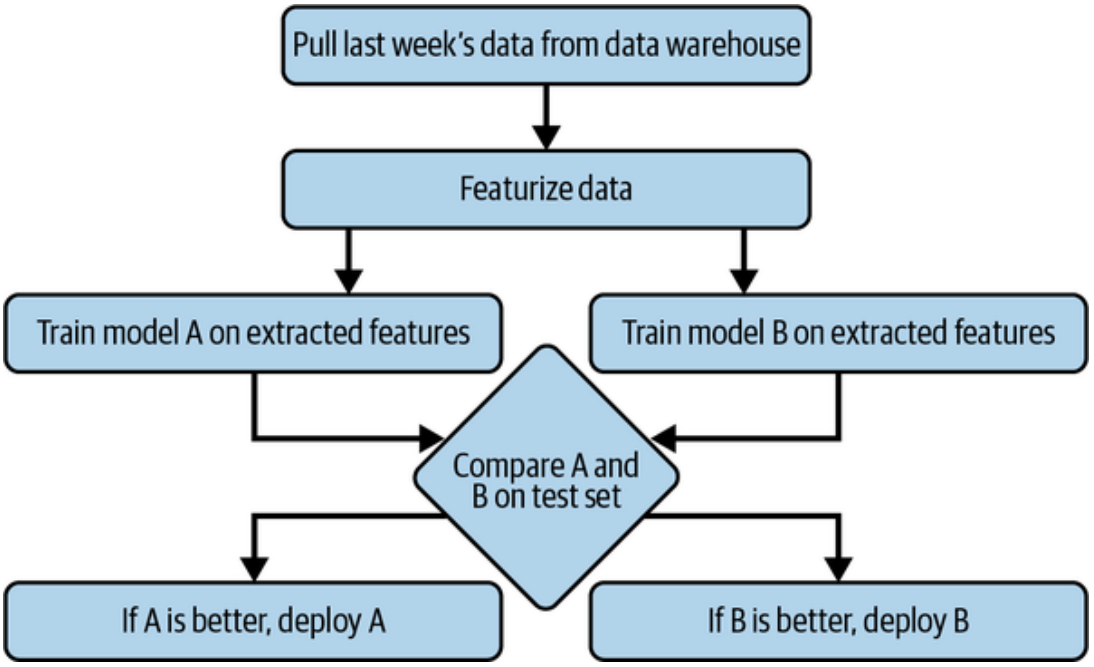


Figure 10-7. A graph that shows the order of execution of a simple ML workflow, which is essentially a DAG (directed acyclic graph)

Many readers might recognize that Figure 10-7 is a DAG: directed acyclic graph. It has to be directed to express the dependencies among steps. It can't contain cycles because, if it does, the job will just keep on running forever. DAG is a common way to represent computing workflows in general, not just ML workflows. Most workflow management tools require you to specify your workflows in a form of DAGs.

*Schedulers* are cron programs that can handle dependencies. It takes in the DAG of a workflow and schedules each step accordingly. You can even schedule to start a job based on an event-based trigger, e.g., start a job whenever an event X happens. Schedulers also allow you to specify what to do if a job fails or succeeds, e.g., if it fails, how many times to retry before giving up.

Schedulers tend to leverage queues to keep track of jobs. Jobs can be queued, prioritized, and allocated resources needed to execute. This means that schedulers need to be aware of the resources available and the resources needed to run each job—the resources needed are either specified as options when you schedule a job or estimated by the scheduler. For instance, if a job requires 8 GB of memory and two CPUs, the scheduler needs to find among the resources it manages an instance with 8 GB of memory and two CPUs and wait until the instance is not executing other jobs to run this job on the instance.

Here's an example of how to schedule a job with the popular scheduler Slurm, where you specify the job name, the time when the job needs to be executed, and the amount of memory and CPUs to be allocated for the job:

```
#!/bin/bash
#SBATCH -J JobName
#SBATCH --time=11:00:00       # When to start the job
#SBATCH --mem-per-cpu=4096   # Memory, in MB, to be allocated per CPU
#SBATCH --cpus-per-task=4          # Number of cores per task
```

Schedulers should also optimize for resource utilization since they have information on resources available, jobs to run, and resources needed for each job to run. However, the number of resources specified by users is not always correct. For example, I might estimate, and therefore specify, that a job needs 4 GB of memory, but this job only needs 3 GB of memory or needs 4 GB of memory at peak and only 1–2 GB of memory otherwise. Sophisticated schedulers like Google's Borg estimate how many resources a job will actually need and reclaim unused resources for other jobs,[28] further optimizing resource utilization.

Designing a general-purpose scheduler is hard, since this scheduler will need to be able to manage almost any number of concurrent machines and workflows. If your scheduler is down, every single workflow that this scheduler touches will be interrupted.

If schedulers are concerned with *when* to run jobs and what resources are needed to run those jobs, orchestrators are concerned with *where* to get those resources. Schedulers deal with job-type abstractions such as DAGs, priority queues, user-level quotas (i.e., the maximum number of instances a user can use at a given time), etc. Orchestrators deal with lower-level abstractions like machines, instances, clusters, service-level grouping, replication, etc. If the orchestrator notices that there are more jobs than the pool of available instances, it can increase the number of instances in the available instance pool. We say that it "provisions" more computers to handle the workload. Schedulers are often used for periodical jobs, whereas orchestrators are often used for services where you have a long-running server that responds to requests.

The most well-known orchestrator today is undoubtedly Kubernetes, the container orchestrator we discussed in the section "From Dev to Prod: Containers". K8s can be used on-prem (even on your laptop via minikube). However, I've never met anyone who enjoys setting up their own K8s clusters, so most companies use K8s as a hosted service managed by their cloud providers, such as AWS's Elastic Kubernetes Service (EKS) or Google Kubernetes Engine (GKE).

Many people use schedulers and orchestrators interchangeably because schedulers usually run on top of orchestrators. Schedulers like Slurm and Google's Borg have some orchestrating capacity, and orchestrators like HashiCorp Nomad and K8s come with some scheduling capacity. But you can have separate schedulers and orchestrators, such as running Spark's job scheduler on top of Kubernetes or AWS Batch scheduler on top of EKS. Orchestrators such as HashiCorp Nomad and data science–specific orchestrators including Airflow, Argo, Prefect, and Dagster have their own schedulers.

## Data Science Workflow Management

We've discussed the differences between schedulers and orchestrators and how they can be used to execute workflows in general. Readers familiar with workflow management tools aimed especially at data science like Airflow, Argo, Prefect, Kubeflow, Metaflow, etc. might wonder where they fit in this scheduler versus orchestrator discussion. We'll go into this topic here.

In its simplest form, workflow management tools manage workflows. They generally allow you to specify your workflows as DAGs, similar to the one in Figure 10-7. A workflow might consist of a featurizing step, a model training step, and an evaluation step. Workflows can be defined using either code (Python) or configuration files (YAML). Each step in a workflow is called a task.

Almost all workflow management tools come with some schedulers, and therefore, you can think of them as schedulers that, instead of focusing on individual jobs, focus on the workflow as a whole. Once a workflow is defined, the underlying scheduler usually works with an orchestrator to allocate resources to run the workflow, as shown in Figure 10-8.

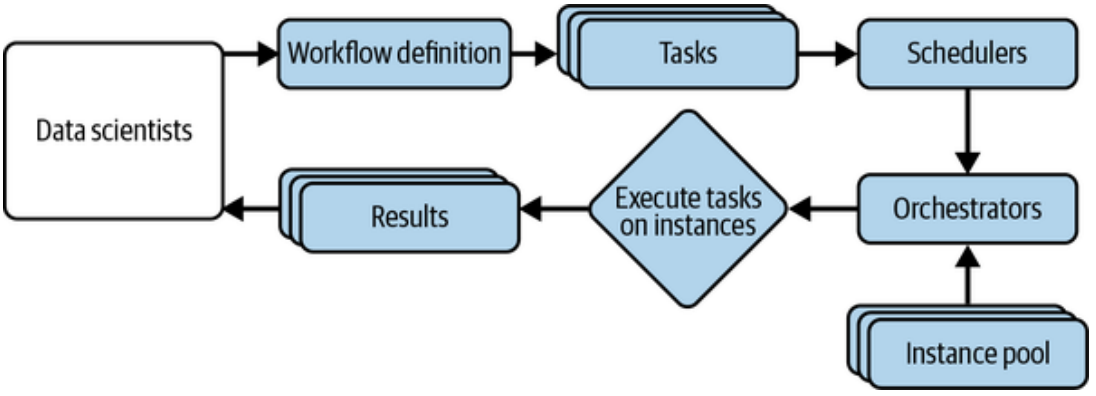


Figure 10-8. After a workflow is defined, the tasks in this workflow are scheduled and orchestrated

There are many articles online comparing different data science workflow management tools. In this section, we'll go over five of the most common tools: Airflow, Argo, Prefect, Kubeflow, and Metaflow. This section isn't meant to be a comprehensive comparison of those tools, but to give you an idea of different features a workflow management tool might need.

Originally developed at Airbnb and released in 2014, Airflow is one of the earliest workflow orchestrators. It's an amazing task scheduler that comes with a huge library of operators that makes it easy to use Airflow with different cloud providers, databases, storage options, and so on. Airflow is a champion of the "configuration as code" principle. Its creators believed that data workflows are complex and should be defined using code (Python) instead of YAML or other declarative language. Here's an example of an Airflow workflow, drawn from the platform's GitHub repository:

```
from datetime import datetime, timedelta

from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.providers.docker.operators.docker import DockerOperator

dag = DAG(
    'docker_sample',
    default_args={'retries': 1},
    schedule_interval=timedelta(minutes=10),
    start_date=datetime(2021, 1, 1),
    catchup=False,
)
```

```
t1 = BashOperator(task_id='print_date', bash_command='date', dag=dag)
t2 = BashOperator(task_id='sleep', bash_command='sleep 5', retries=3, dag=dag)
t3 = DockerOperator(
    docker_url='tcp://localhost:2375',  # Set your docker URL
    command='/bin/sleep 30',
    image='centos:latest',
    network_mode='bridge',
    task_id='docker_op_tester',
    dag=dag,
)

t4 = BashOperator(
    task_id='print_hello',
    bash_command='echo "hello world!!!"',
    dag=dag
)

t1 >> t2
t1 >> t3
t3 >> t4
```

However, because Airflow was created earlier than most other tools, it had no tool to learn lessons from and suffers from many drawbacks, as discussed in detail in a blog post by Uber Engineering. Here, we'll go over only three to give you an idea.

First, Airflow is monolithic, which means it packages the entire workflow into one container. If two different steps in your workflow have different requirements, you can, in theory, create different containers for them using Airflow's `DockerOperator`, but it's not that easy to do so.

Second, Airflow's DAGs are not parameterized, which means you can't pass parameters into your workflows. So if you want to run the same model with different learning rates, you'll have to create different workflows.

Third, Airflow's DAGs are static, which means it can't automatically create new steps at runtime as needed. Imagine you're reading from a database and you want to create a step to process each record in the database

(e.g., to make a prediction), but you don't know in advance how many records there are in the database. Airflow won't be able to handle that.

The next generation of workflow orchestrators (Argo, Prefect) were created to address different drawbacks of Airflow.

Prefect's CEO, Jeremiah Lowin, was a core contributor of Airflow. Their early marketing campaign drew intense comparison between Prefect and Airflow. Prefect's workflows are parameterized and dynamic, a vast improvement compared to Airflow. It also follows the "configuration as code" principle so workflows are defined in Python.

However, like Airflow, containerized steps aren't the first priority of Prefect. You can run each step in a container, but you'll still have to deal with Dockerfiles and register your docker with your workflows in Prefect.

Argo addresses the container problem. Every step in an Argo workflow is run in its own container. However, Argo's workflows are defined in YAML, which allows you to define each step and its requirements in the same file. The following code sample, drawn from the Argo GitHub repository, demonstrates how to create a workflow to show a coin flip:

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: coinflip-
  annotations:
    workflows.argoproj.io/description: |
      This is an example of coin flip defined as a sequence of conditional steps
      You can also run it in Python:
      https://couler-proj.github.io/couler/examples/#coin-flip
spec:
  entrypoint: coinflip
  templates:
  - name: coinflip
    steps:
    - - name: flip-coin
        template: flip-coin
    - - name: heads
        template: heads
        when: "{{steps.flip-coin.outputs.result}} == heads"
```

```
      - name: tails
        template: tails
        when: "{{steps.flip-coin.outputs.result}} == tails"

    - name: flip-coin
      script:
        image: python:alpine3.6
        command: [python]
        source: |
          import random
          result = "heads" if random.randint(0,1) == 0 else "tails"
          print(result)
    - name: heads
      container:
        image: alpine:3.6
        command: [sh, -c]
        args: ["echo \"it was heads\""]

    - name: tails
      container:
        image: alpine:3.6
        command: [sh, -c]
        args: ["echo \"it was tails\""]
```

The main drawback of Argo, other than its messy YAML files, is that it can only run on K8s clusters, which are only available in production. If you want to test the same workflow locally, you'll have to use minikube to simulate a K8s on your laptop, which can get messy.

Enter Kubeflow and Metaflow, the two tools that aim to help you run the workflow in both dev and prod environments by abstracting away infrastructure boilerplate code usually needed to run Airflow or Argo. They promise to give data scientists access to the full compute power of the prod environment from local notebooks, which effectively allows data scientists to use the same code in both dev and prod environments.

Even though both tools have some scheduling capacity, they are meant to be used with a bona fide scheduler and orchestrator. One component of Kubeflow is Kubeflow Pipelines, which is built on top of Argo, and it's meant to be used on top of K8s. Metaflow can be used with AWS Batch or K8s.

Both tools are fully parameterized and dynamic. Currently, Kubeflow is the more popular one. However, from a user experience perspective, Metaflow is superior, in my opinion. In Kubeflow, while you can define your workflow in Python, you still have to write a Dockerfile and a YAML file to specify the specs of each component (e.g., process data, train, deploy) before you can stitch them together in a Python workflow. Basically, Kubeflow helps you abstract away other tools' boilerplate by making you write Kubeflow boilerplate.

In Metaflow, you can use a Python decorator `@conda` to specify the requirements for each step—required libraries, memory and compute requirements—and Metaflow will automatically create a container with all these requirements to execute the step. You save on Dockerfiles or YAML files.

Metaflow allows you to work seamlessly with both dev and prod environments from the same notebook/script. You can run experiments with small datasets on local machines, and when you're ready to run with the large dataset on the cloud, simply add `@batch` decorator to execute it on AWS Batch. You can even run different steps in the same workflow in different environments. For example, if a step requires a small memory footprint, it can run on your local machine. But if the next step requires a large memory footprint, you can just add `@batch` to execute it on the cloud.

```
# Example: sketch of a recommender system that uses an ensemble of two models.
# Model A will be run on your local machine and model B will be run on AWS.

class RecSysFlow(FlowSpec):
    @step
    def start(self):
        self.data = load_data()
        self.next(self.fitA, self.fitB)

    # fitA requires a different version of NumPy compared to fitB
    @conda(libraries={"scikit-learn":"0.21.1", "numpy":"1.13.0"})
    @step
    def fitA(self):
        self.model = fit(self.data, model="A")
        self.next(self.ensemble)
```

```
@conda(libraries={"numpy":"0.9.8"})
# Requires 2 GPU of 16GB memory
@batch(gpu=2, memory=16000)
@step
def fitB(self):
    self.model = fit(self.data, model="B")
    self.next(self.ensemble)

@step
def ensemble(self, inputs):
    self.outputs = (
                (inputs.fitA.model.predict(self.data) +
                 inputs.fitB.model.predict(self.data)) / 2
                for input in inputs
    )
    self.next(self.end)

def end(self):
    print(self.outputs)
```

# ML Platform

The manager of the ML platform team at a major streaming company told me the story of how his team got started. He originally joined the company to work on their recommender systems. To deploy their recommender systems, they needed to build out tools such as feature management, model management, monitoring, etc. Last year, his company realized that these same tools could be used by other ML applications, not just recommender systems. They created a new team, the ML platform team, with the goal of providing shared infrastructure across ML applications. Because the recommender system team had the most mature tool, their tools were adopted by other teams, and some members from the recommender system team were asked to join the new ML platform team.

This story represents a growing trend since early 2020. As each company finds uses for ML in more and more applications, there's more to be gained by leveraging the same set of tools for multiple applications instead of supporting a separate set of tools for each application. This shared set of tools for ML deployment makes up the ML platform.

Because ML platforms are relatively new, what exactly constitutes an ML platform varies from company to company. Even within the same company, it's an ongoing discussion. Here, I'll focus on the components that I most often see in ML platforms, which include model development, model store, and feature store.

Evaluating a tool for each of these categories depends on your use case. However, here are two general aspects you might want to keep in mind:

*Whether the tool works with your cloud provider or allows you to use it on your own data center*

You'll need to run and serve your models from a compute layer, and usually tools only support integration with a handful of cloud providers. Nobody likes having to adopt a new cloud provider for another tool.

*Whether it's open source or a managed service*

If it's open source, you can host it yourself and have to worry less about data security and privacy. However, self-hosting means extra engineering time required to maintain it. If it's managed service, your models and likely some of your data will be on its service, which might not work for certain regulations. Some managed services work with virtual private clouds, which allows you to deploy your machines in your own cloud clusters, helping with compliance. We'll discuss this more in the section "Build Versus Buy".

Let's start with the first component: model deployment.

## Model Deployment

Once a model is trained (and hopefully tested), you want to make its predictive capability accessible to users. In Chapter 7, we talked at length on how a model can serve its predictions: online or batch prediction. We also discussed how the simplest way to deploy a model is to push your model and its dependencies to a location accessible in production then expose your model as an endpoint to your users. If you do online prediction, this endpoint will provoke your model to generate a prediction. If you do batch prediction, this endpoint will fetch a precomputed prediction.

A deployment service can help with both pushing your models and their dependencies to production and exposing your models as endpoints. Since deploying is the name of the game, deployment is the most mature among all ML platform components, and many tools exist for this. All major cloud providers offer tools for deployment: AWS with SageMaker, GCP with Vertex AI, Azure with Azure ML, Alibaba with Machine Learning Studio, and so on. There are also a myriad of startups that offer model deployment tools such as MLflow Models, Seldon, Cortex, Ray Serve, and so on.

When looking into a deployment tool, it's important to consider how easy it is to do both online prediction and batch prediction with the tool. While it's usually straightforward to do online prediction at a smaller scale with most deployment services, doing batch prediction is usually trickier.[29] Some tools allow you to batch requests together for online prediction, which is different from batch prediction. Many companies have separate deployment pipelines for online prediction and batch prediction. For example, they might use Seldon for online prediction but leverage Databricks for batch prediction.

An open problem with model deployment is how to ensure the quality of a model before it's deployed. In Chapter 9, we talked about different techniques for test in production such as shadow deployment, canary release, A/B testing, and so on. When choosing a deployment service, you might want to check whether this service makes it easy for you to perform the tests that you want.

## Model Store

Many companies dismiss model stores because they sound simple. In the section "Model Deployment", we talked about how, to deploy a model, you have to package your model and upload it to a location accessible in production. Model store suggests that it stores models—you can do so by uploading your models to storage like S3. However, it's not quite that simple. Imagine now that your model's performance dropped for a group of inputs. The person who was alerted to the problem is a DevOps engineer, who, after looking into the problem, decided that she needed to inform

the data scientist who created this model. But there might be 20 data scientists in the company; who should she ping?

Imagine now that the right data scientist is looped in. The data scientist first wants to reproduce the problems locally. She still has the notebook she used to generate this model and the final model, so she starts the notebook and uses the model with the problematic sets of inputs. To her surprise, the outputs the model produces locally are different from the outputs produced in production. Many things could have caused this discrepancy; here are just a few examples:

- The model being used in production right now is not the same model that she has locally. Perhaps she uploaded the wrong model binary to production?
- The model being used in production is correct, but the list of features used is wrong. Perhaps she forgot to rebuild the code locally before pushing it to production?
- The model is correct, the feature list is correct, but the featurization code is outdated.
- The model is correct, the feature list is correct, the featurization code is correct, but something is wrong with the data processing pipeline.

Without knowing the cause of the problem, it'll be very difficult to fix it. In this simple example, we assume that the data scientist responsible still has access to the code used to generate the model. What if that data scientist no longer has access to that notebook, or she has already quit or is on vacation?

Many companies have realized that storing the model alone in blob storage isn't enough. To help with debugging and maintenance, it's important to track as much information associated with a model as possible. Here are eight types of artifacts that you might want to store. Note that many artifacts mentioned here are information that should be included in the model card, as discussed in the section "Create model cards".

*Model definition*

This is the information needed to create the shape of the model, e.g., what loss function it uses. If it's a neural network, this includes

how many hidden layers it has and how many parameters are in each layer.

*Model parameters*

These are the actual values of the parameters of your model. These values are then combined with the model's shape to re-create a model that can be used to make predictions. Some frameworks allow you to export both the parameters and the model definition together.

*Featurize and predict functions*

Given a prediction request, how do you extract features and input these features into the model to get back a prediction? The featurize and predict functions provide the instruction to do so. These functions are usually wrapped in endpoints.

*Dependencies*

The dependencies—e.g., Python version, Python packages—needed to run your model are usually packaged together into a container.

*Data*

The data used to train this model might be pointers to the location where the data is stored or the name/version of your data. If you use tools like DVC to version your data, this can be the DVC commit that generated the data.

*Model generation code*

This is the code that specifies how your model was created, such as:

- What frameworks it used
- How it was trained
- The details on how the train/valid/test splits were created
- The number of experiments run
- The range of hyperparameters considered
- The actual set of hyperparameters that final model used

Very often, data scientists generate models by writing code in notebooks. Companies with more mature pipelines make their data sci-

entists commit the model generation code into their Git repos on GitHub or GitLab. However, in many companies, this process is ad hoc, and data scientists don't even check in their notebooks. If the data scientist responsible for the model loses the notebook or quits or goes on vacation, there's no way to map a model in production to the code that generated it for debugging or maintenance.

*Experiment artifacts*

These are the artifacts generated during the model development process, as discussed in the section "Experiment Tracking and Versioning". These artifacts can be graphs like the loss curve. These artifacts can be raw numbers like the model's performance on the test set.

*Tags*

This includes tags to help with model discovery and filtering, such as owner (the person or the team who is the owner of this model) or task (the business problem this model solves, like fraud detection).

Most companies store a subset, but not all, of these artifacts. The artifacts a company stores might not be in the same place but scattered. For example, model definitions and model parameters might be in S3. Containers that contain dependencies might be in ECS (Elastic Container Service). Data might be in Snowflake. Experiment artifacts might be in Weights & Biases. Featurize and prediction functions might be in AWS Lambda. Some data scientists might manually keep track of these locations in, say, a README, but this file can be easily lost.

A model store that can store sufficient general use cases is far from being a solved problem. As of writing this book, MLflow is undoubtedly the most popular model store that isn't associated with a major cloud provider. Yet three out of the six top MLflow questions on Stack Overflow are about storing and accessing artifacts in MLflow, as shown in Figure 10-9. Model stores are due for a makeover, and I hope that in the near future a startup will step up and solve this.


Figure 10-9. MLflow is the most popular model store, yet it's far from solving the artifact problem. Three out of the six top MLflow questions on Stack Overflow are about storing and accessing artifacts in MLflow. Source: Screenshot of Stack Overflow page

Because of the lack of a good model store solution, companies like Stitch Fix resolve to build their own model store. Figure 10-10 shows the artifacts that Stitch Fix's model store tracks. When a model is uploaded to their model store, this model comes with the link to the serialized model, the dependencies needed to run the model (Python environment), the Git commit where the model code generation is created (Git information), tags (to at least specify the team that owns the model), etc.

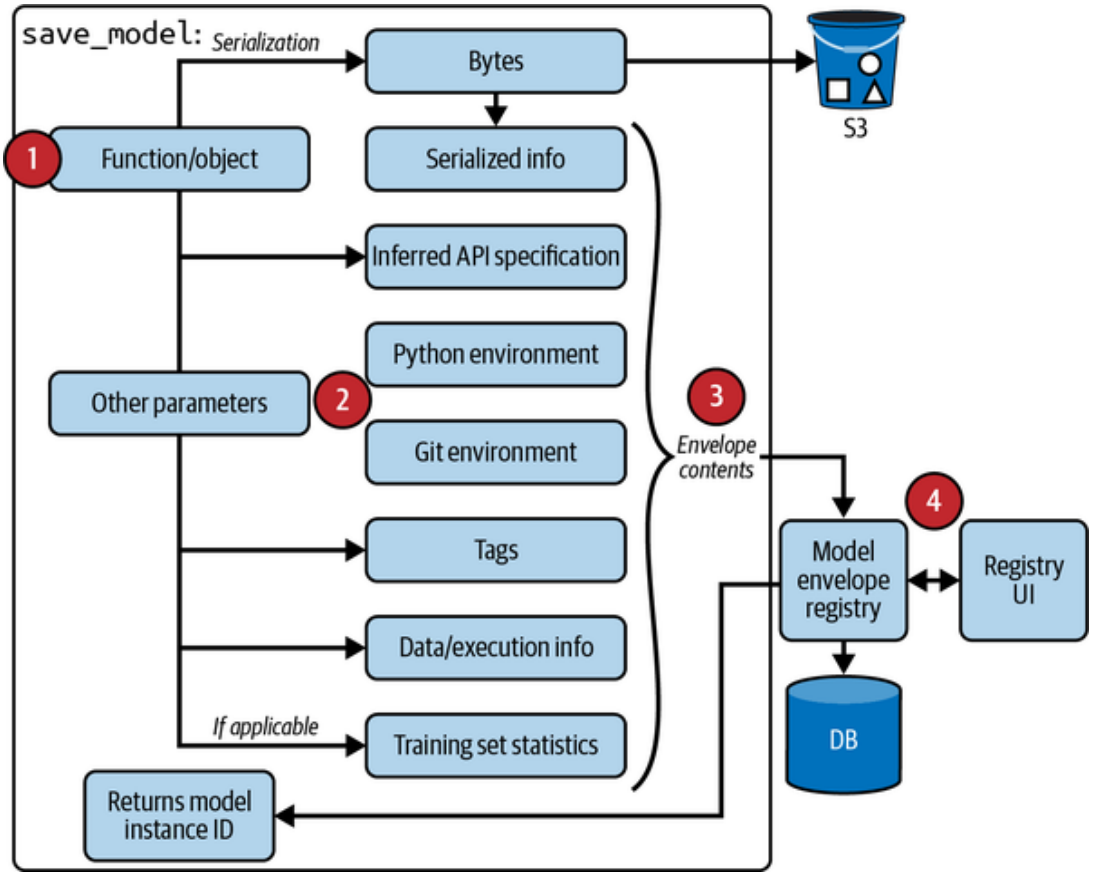


Figure 10-10. Artifacts that Stitch Fix's model store tracks. Source: Adapted from a slide by Stefan Krawczyk for CS 329S (Stanford).

## Feature Store

"Feature store" is an increasingly loaded term that can be used by different people to refer to very different things. There have been many attempts by ML practitioners to define what features a feature store should have.[30] At its core, there are three main problems that a feature store can help address: feature management, feature transformation, and feature consistency. A feature store solution might address one or a combination of these problems:

*Feature management*

A company might have multiple ML models, each model using a lot of features. Back in 2017, Uber had about 10,000 features across teams![31] It's often the case that features used for one model can be useful for another model. For example, team A might have a model to predict how likely a user will churn, and team B has a model to predict how likely a free user will convert into a paid user. There are many features that these two models can share. If team A dis-

covers that feature X is super useful, team B might be able to leverage that too.

A feature store can help teams share and discover features, as well as manage roles and sharing settings for each feature. For example, you might not want everyone in the company to have access to sensitive financial information of either the company or its users. In this capacity, a feature store can be thought of as a feature catalog. Examples of tools for feature management are Amundsen (developed at Lyft) and DataHub (developed at LinkedIn).

*Feature computation*[32]

Feature engineering logic, after being defined, needs to be computed. For example, the feature logic might be: use the average meal preparation time from yesterday. The computation part involves actually looking into your data and computing this average.

In the previous point, we discussed how multiple models might share a feature. If the computation of this feature isn't too expensive, it might be acceptable computing this feature each time it is required by a model. However, if the computation is expensive, you might want to execute it only once the first time it is required, then store it for feature uses.

A feature store can help with both performing feature computation and storing the results of this computation. In this capacity, a feature store acts like a data warehouse.

*Feature consistency*

In Chapter 7, we talked about the problem of having two separate pipelines for the same model: the training pipeline extracts batch features from historical data and the inference pipeline extracts streaming features. During development, data scientists might define features and create models using Python. Production code, however, might be written in another language, such as Java or C, for performance.

> This means that feature definitions written in Python during development might need to be converted into the languages used in production. So you have to write the same features twice, once for training and once for inference. First, it's annoying and time-consuming. Second, it creates extra surface for bugs since one or more features in production might differ from their counterparts in training, causing weird model behaviors.
>
> A key selling point of modern feature stores is that they unify the logic for both batch features and streaming features, ensuring the consistency between features during training and features during inference.

Feature store is a newer category that only started taking off around 2020. While it's generally agreed that feature stores should manage feature definitions and ensure feature consistency, their exact capacities vary from vendor to vendor. Some feature stores only manage feature definitions without computing features from data; some feature stores do both. Some feature stores also do feature validation, i.e., detecting when a feature doesn't conform to a predefined schema, and some feature stores leave that aspect to a monitoring tool.

As of writing this book, the most popular open source feature store is Feast. However, Feast's strength is in batch features, not streaming features. Tecton is a fully managed feature store that promises to be able to handle both batch features and online features, but their actual traction is slow because they require deep integration. Platforms like SageMaker and Databricks also offer their own interpretations of feature stores. Out of 95 companies I surveyed in January 2022, only around 40% of them use a feature store. Out of those who use a feature store, half of them build their own feature store.

## Build Versus Buy

At the beginning of this chapter, we discussed how difficult it is to set up the right infrastructure for your ML needs. What infrastructure you need depends on the applications you have and the scale at which you run these applications.

How much you need to invest into infrastructure also depends on what you want to build in-house and what you want to buy. For example, if you want to use fully managed Databricks clusters, you probably need only one engineer. However, if you want to host your own Spark Elastic MapReduce clusters, you might need five more people.

At one extreme, you can outsource all your ML use cases to a company that provides ML applications end-to-end, and then perhaps the only piece of infrastructure you need is for data movement: moving your data from your applications to your vendor, and moving predictions from that vendor back to your users. The rest of your infrastructure is managed by your vendor.

At the other extreme, if you're a company that handles sensitive data that prevents you from using services managed by another company, you might need to build and maintain all your infrastructure in-house, even having your own data centers.

Most companies, however, are in neither of these extremes. If you work for one of these companies, you'll likely have some components managed by other companies and some components developed in-house. For example, your compute might be managed by AWS EC2 and your data warehouse managed by Snowflake, but you have your own feature store and your own monitoring dashboards.

Your build versus buy decisions depend on many factors. Here, we'll discuss three common ones that I often encounter when talking with heads of infrastructures on how they evaluate these decisions:

*The stage your company is at*

> In the beginning, you might want to leverage vendor solutions to get started as quickly as possible so that you can focus your limited resources on the core offerings of your product. As your use cases grow, however, vendor costs might become exorbitant and it might be cheaper for you to invest in your own solution.

*What you believe to be the focus or the competitive advantages of your company*

Stefan Krawczyk, manager of the ML platform team at Stitch Fix, explained to me his build versus buy decision: "If it's something we want to be really good at, we'll manage that in-house. If not, we'll use a vendor." For the vast majority of companies outside the technology sector—e.g., companies in retail, banking, manufacturing—ML infrastructure isn't their focus, so they tend to bias toward buying. When I talk to these companies, they prefer managed services, even point solutions (e.g., solutions that solve a business problem for them, like a demand forecasting service). For many tech companies where technology is their competitive advantage, and whose strong engineering teams prefer to have control over their stacks, they tend to bias toward building. If they use a managed service, they might prefer that service to be modular and customizable, so that they can plug and play with any component.

*The maturity of the available tools*

For example, your team might decide that you need a model store, and you'd have preferred to use a vendor, but there's no vendor mature enough for your needs, so you have to build your own feature store, perhaps on top of an open source solution.

This is what happens in the early days of ML adoption in the industry. Companies that are early adopters, i.e., big tech companies, build out their own infrastructure because there are no solutions mature enough for their needs. This leads to the situation where every company's infrastructure is different. A few years later, solution offerings mature. However, these offerings find it difficult to sell to big tech companies because it's impossible to create a solution that works with the majority of custom infrastructure.

As we're building out Claypot AI, other founders have actually advised us to avoid selling to big tech companies because, if we do, we'll get sucked into what they call "integration hell"—spending more time integrating our solution with custom infrastructure instead of building out our core features. They advised us to focus on startups with much cleaner slates to build on.

Some people think that building is cheaper than buying, which is not necessarily the case. Building means that you'll have to bring on more engi-

neers to build and maintain your own infrastructure. It can also come with future cost: the cost of innovation. In-house, custom infrastructure makes it hard to adopt new technologies available because of the integration issues.

The build versus buy decisions are complex, highly context-dependent, and likely what heads of infrastructure spend much time mulling over. Erik Bernhardsson, ex-CTO of Better.com, said in a tweet that "one of the most important jobs of a CTO is vendor/product selection and the importance of this keeps going up rapidly every year since the infrastructure space grows so fast."[33] There's no way that a small section can address all its nuances. But I hope that this section provides you with some pointers to start the discussion.

# Summary

If you've stayed with me until now, I hope you agree that bringing ML models to production is an infrastructural problem. To enable data scientists to develop and deploy ML models, it's crucial to have the right tools and infrastructure set up.

In this chapter, we covered different layers of infrastructure needed for ML systems. We started from the storage and compute layer, which provides vital resources for any engineering project that requires intensive data and compute resources like ML projects. The storage and compute layer is heavily commoditized, which means that most companies pay cloud services for the exact amount of storage and compute they use instead of setting up their own data centers. However, while cloud providers make it easy for a company to get started, their cost becomes prohibitive as this company grows, and more and more large companies are looking into repatriating from the cloud to private data centers.

We then continued on to discuss the development environment where data scientists write code and interact with the production environment. Because the dev environment is where engineers spend most of their time, improvements in the dev environment translate directly into improvements in productivity. One of the first things a company can do to improve the dev environment is to standardize the dev environment for

data scientists and ML engineers working on the same team. We discussed in this chapter why standardization is recommended and how to do so.

We then discussed an infrastructural topic whose relevance to data scientists has been debated heavily in the last few years: resource management. Resource management is important to data science workflows, but the question is whether data scientists should be expected to handle it. In this section, we traced the evolution of resource management tools from cron to schedulers to orchestrators. We also discussed why ML workflows are different from other software engineering workflows and why they need their own workflow management tools. We compared various workflow management tools such as Airflow, Argo, and Metaflow.

ML platform is a team that has emerged recently as ML adoption matures. Since it's an emerging concept, there are still disagreements on what an ML platform should consist of. We chose to focus on the three sets of tools that are essential for most ML platforms: deployment, model store, and feature store. We skipped monitoring of the ML platform since it's already covered in Chapter 8.

When working on infrastructure, a question constantly haunts engineering managers and CTOs alike: build or buy? We ended this chapter with a few discussion points that I hope can provide you or your team with sufficient context to make those difficult decisions.

1 Kunal Shah, "This Is What Makes SEO Important for Every Business," *Entrepreneur India*, May 11, 2020, *https://oreil.ly/teQlX*.

2 For a sneak peek into Tesla's compute infrastructure for ML, I highly recommend watching the recording of Tesla AI Day 2021 on YouTube.

3 The definition for "reasonable scale" was inspired by Jacopo Tagliabue in his paper "You Do Not Need a Bigger Boat: Recommendations at Reasonable Scale in a (Mostly) Serverless and Open Stack," *arXiv*, July 15, 2021, *https://oreil.ly/YNRZQ*. For more discussion on reasonable scale, see "ML and MLOps at a Reasonable Scale" by Ciro Greco (October 2021).

4 FAAAM is short for Facebook, Apple, Amazon, Alphabet, Microsoft.

[5] Reza Shiftehfar, "Uber's Big Data Platform: 100+ Petabytes with Minute Latency," *Uber Engineering*, October 17, 2018, *https://oreil.ly/6Ykd3*; Kaushik Krishnamurthi, "Building a Big Data Pipeline to Process Clickstream Data," Zillow, April 6, 2018, *https://oreil.ly/SGmNe*.

[6] Nathan Bronson and Janet Wiener, "Facebook's Top Open Data Problems," Meta, October 21, 2014, *https://oreil.ly/p6QjX*.

[7] Wikipedia, s.v. "Infrastructure," *https://oreil.ly/YaIk8*.

[8] I've seen a company whose data is spread over Amazon Redshift and GCP BigQuery, and their engineers are not very happy about it.

[9] We only discuss data storage here since we've discussed data systems in Chapter 2.

[10] As of writing this book, an ML workload typically requires between 4 GB and 8 GB of memory; 16 GB of memory is enough to handle most ML workloads.

[11] See operation fusion in the section "Model optimization".

[12] "What Is FLOP/s and Is It a Good Measure of Performance?," Stack Overflow, last updated October 7, 2020, *https://oreil.ly/M8jPP*.

[13] For readers interested in FLOPS and bandwidth and how to optimize them for deep learning models, I recommend the post "Making Deep Learning Go Brrrr From First Principles" (He 2022).

[14] According to Amazon, "EC2 instances support multithreading, which enables multiple threads to run concurrently on a single CPU core. Each thread is represented as a virtual CPU (vCPU) on the instance. An instance has a default number of CPU cores, which varies according to instance type. For example, an m5.xlarge instance type has two CPU cores and two threads per core by default—four vCPUs in total" ("Optimize CPU Options," Amazon Web Services, last accessed April 2020, *https://oreil.ly/eeOtd*).

[15] Which costs $26.688/hour.

[16] On-demand instances are instances that are available when you request them. Spot instances are instances that are available when nobody else is using them. Cloud providers tend to offer spot instances at a discount compared to on-demand instances.

17 Synergy Research Group, "2020—The Year That Cloud Service Revenues Finally Dwarfed Enterprise Spending on Data Centers," March 18, 2021, *https://oreil.ly/uPx94*.

18 Sarah Wang and Martin Casado, "The Cost of Cloud, a Trillion Dollar Paradox," a16z, *https://oreil.ly/3nWU3*.

19 Wang and Casado, "The Cost of Cloud."

20 Laurence Goasduff, "Why Organizations Choose a Multicloud Strategy," Gartner, May 7, 2019, *https://oreil.ly/ZiqzQ*.

21 Goasduff, "Why Organizations Choose a Multicloud Strategy."

22 Ville Tuulos, *Effective Data Science Infrastructure* (Manning, 2022).

23 As of writing this book, Google Colab even offers free GPUs for their users.

24 Michelle Ufford, M. Pacer, Matthew Seal, and Kyle Kelley, "Beyond Interactive: Notebook Innovation at Netflix," *Netflix Technology Blog*, August 16, 2018, *https://oreil.ly/EHvAe*.

25 For the uninitiated, a new pull request can be understood as a new piece of code being added to the codebase.

26 See editor war, the decade-long, heated debate on Vim versus Emacs.

27 Chip Huyen, "Why Data Scientists Shouldn't Need to Know Kubernetes," September 13, 2021, *https://huyenchip.com/2021/09/13/data-science-infrastructure.html*; Neil Conway and David Hershey, "Data Scientists Don't Care About Kubernetes," Determined AI, November 30, 2020, *https://oreil.ly/FFDQW*; I Am Developer on Twitter (@iamdevloper): "I barely understand my own feelings how am I supposed to understand kubernetes," June 26, 2021, *https://oreil.ly/T2eQE*.

28 Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes, "Large-Scale Cluster Management at Google with Borg," *EuroSys '15: Proceedings of the Tenth European Conference on Computer Systems* (April 2015): 18, *https://oreil.ly/9TeTM*.

29 When doing online prediction at a smaller scale, you can just hit an endpoint with payloads and get back predictions. Batch prediction requires setting up batch jobs and storing predictions.

30 Neal Lathia, “Building a Feature Store,” December 5, 2020, *https://oreil.ly/DgsvA*; Jordan Volz, “Why You Need a Feature Store,” *Continual*, September 28, 2021, *https://oreil.ly/kQPMb*; Mike Del Balso, “What Is a Feature Store?” *Tecton*, October 20, 2020, *https://oreil.ly/pzy0I*.

31 Jeremy Hermann and Mike Del Balso, “Meet Michelangelo: Uber’s Machine Learning Platform,” *Uber Engineering*, September 5, 2017, *https://oreil.ly/XteNy*.

32 Some people use the term “feature transformation.”

33 Erik Bernhardsson on Twitter (@bernhardsson), September 29, 2021, *https://oreil.ly/GnxOH*.