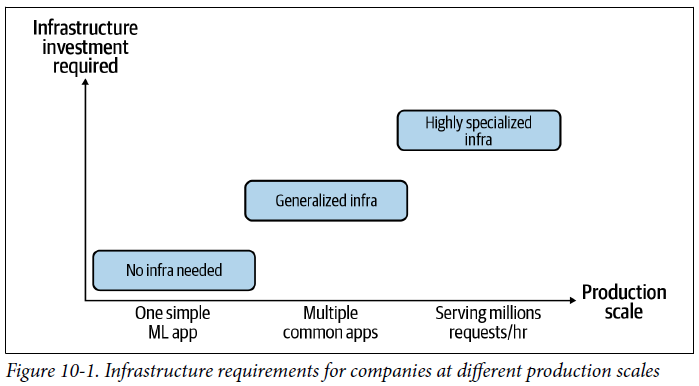
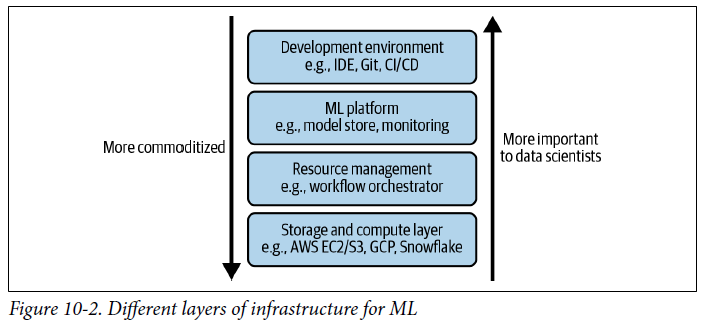
# Designing Machine Learning Systems - Chip Huyen

## Chapter 10 – Infrastructure and Tooling for MLOps

* Up until now, we’ve assumed ML practitioners have access to all the tools + infrastructure they need to implement the logic for developing ML systems and carry out the considerations for deploying, monitoring, and continually updating an ML system.
* However, **that assumption is far from being true**
* Many data scientists know the right things to do for their ML systems, but can’t do them because their infrastructure isn’t set up in a way that enables them to do so
* **ML systems are complex**, and **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, which, in turn, can speed up 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**
* **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**
* **One end of the spectrum = 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 a quarterly planning meeting)**
* **These companies probably won’t need to invest in any infrastructure**, and 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 friends), you **probably won’t need any infrastructure either**, you just need an Android-compatible ML framework like TensorFlow Lite
* **Other end of the spectrum = companies that work on applications with unique requirements**
* Ex: 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/second, which translates to 234 million search queries/hour
* 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
* It’s **common that part of specialized infrastructure is later made public and adopted by other companies**
* Ex: Google extended their internal cloud infrastructure to the public, resulting in GCP
* **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 GB and TB, instead of PB, a day**
* Their data science team might range from **10 to hundreds of engineers**
* This category might include any company from a **20-person startup to a company at Zillow’s scale, but not at FAAAM scale**
* Ex: Back in 2018, Uber was adding tens of TB of data a day to their data lake, and Zillow’s biggest dataset was bringing in 2 TB of uncompressed data a day
* In contrast, even back in 2014, Facebook was generating *4 PB* of data a day
* **Companies in this middle of the spectrum will likely benefit from *generalized* ML infrastructure that is being increasingly standardized** (see below)



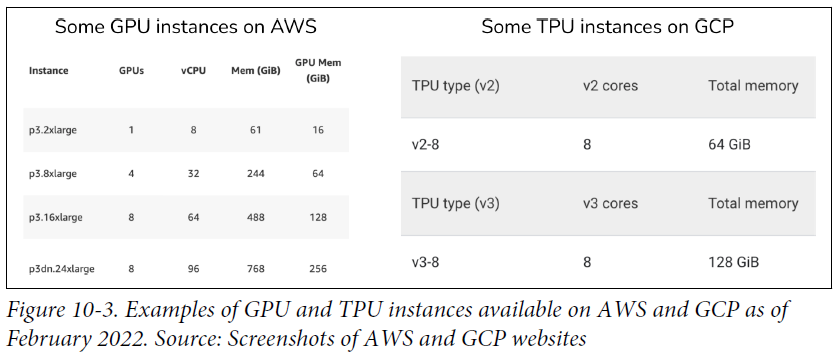
* **In order to set up the right infrastructure for your needs, it’s important to understand exactly what infrastructure means and what it consists of**
* Wikipedia, *in the physical world*, “infrastructure is the set of fundamental facilities and systems that support the sustainable functionality of households and firms.”
* **In the ML world, infrastructure is the set of fundamental facilities that support the development and maintenance of ML systems**
* What should be considered **“fundamental facilities” varies greatly from company to company**
* There are **4 infrastructure layers:**
* **1) Storage and compute**
* **Storage layer is where data is collected + stored**, **compute layer provides the compute needed to run ML workloads** (training models, computing and/or generating features, etc.)
* **2) Resource management**
* **Tools to schedule and orchestrate workloads to make the most out of available compute**
* **3) ML platform**
* **Provides tools to aid the development of ML applications (model stores, feature stores, monitoring tools, etc.)**
* Examples of tools in this category include SageMaker and MLflow.
* **4) Development (DEV) environment**
* **Where code is written and experiments are run**
* **Code** needs to be **versioned and tested**, and **experiments** need to be **tracked**
* These four different layers with **importance and commoditization** are shown below



* **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**
* The **DEV environment is what data scientists have to interact with daily**, and therefore, is the **least abstract to them**
* **Resource management** is a contentious topic among data scientists (people are still debating **whether a data scientist needs to know about this layer or not**)
* **“ML platform” is a relatively new concept** **with its different components still maturin**g,
* An ML platform **requires up-front investment from a company, but if done right, it can make the life of data scientists across business use cases at that company so much easier**
* **Even if 2 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)

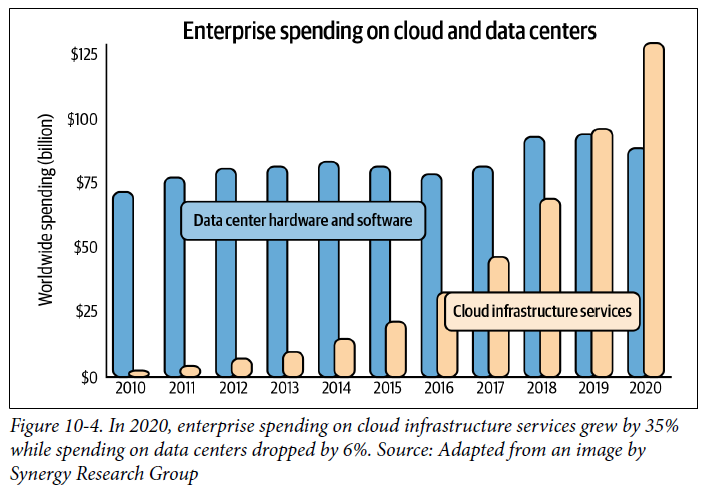
### Storage and Compute

* 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**, this can be a **hard drive disk (HDD) or a solid state disk (SSD)**
* The storage layer **can be in one place** (might have all data in Amazon S3 or in Snowflake), **or spread out over multiple locations**
* 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**
* 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**
* 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**
* Ex: A CPU core might support 2 concurrent threads, each 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 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 **VM** (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
* Ex: 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**
* Ex: To add 2 arrays, you first need to load these 2 arrays into memory, *then* perform addition
* If the compute unit doesn’t have enough memory to load these 2 arrays, the operation will be impossible *without an algorithm to handle out-of-memory computation*
* Therefore, a **compute unit is mainly characterized by 2 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**
* Nowadays, an ML workload typically requires between 4-8 GB of memory, and 16 GB of memory is enough to handle most ML workloads
* 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)**, which denotes the number of float point operations a compute unit can run per second
* Might see hardware vendors 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**:
* **1) Companies that measure this metric might have different ideas on what is counted as an “operation”**
* Ex: If a machine fuses 2 operations into 1 and executes this fused operation, does this count as 1 operation or 2?
* **2) 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**
* 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)**
* **When evaluating a new compute unit, it’s important to evaluate how long it will take this compute unit to do common workloads**
* Ex: MLPerf = a popular benchmark for hardware vendors to measure 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**
* Ex: You might use an instance with 4 CPU cores and 8 GB of memory
* Keep in mind that AWS uses the concept of vCPU (virtual CPU) which, for practical purposes, can be thought of as *half* a physical core
* According to 2020 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, varying according to instance type
* For example, an m5.xlarge instance type has 2 CPU cores and 2 threads per core by default, 4 vCPUs in total
* You can see the number of cores and memory offered by some AWS EC2 and GCP instances below



#### Public Cloud Vs. Private Data Center

* Like data storage, the **compute layer is largely commoditized**, which 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**
* **Especially appealing to companies that have variable-sized workloads**
* Ex: Workloads need 1,000 CPU cores 1 day of the year, and only 10 CPU cores for the rest
* 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 1 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,* reducing engineering operational overhead**
* **Most cloud providers even do that automatically for you**
* **Especially useful in ML, as data science workloads are burst-y**
* 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 **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.
* **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**
* **On-demand instances = available when you request them**
* **Spot instances = available when nobody else is using them**
* Providers tend to offer spot instances at a discount compared to on-demand instances
* **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 2020 research: “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,” as shown in below



* 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, VC firm a16z shows that cloud spending accounts for approximately 50% cost of revenue of these companies
* They estimated: “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”
* **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 2018 S-1 filing shows the company was able to save $75M over the 2 years prior to IPO due to an infrastructure optimization overhaul, a large chunk of which consisted of moving their workloads from public cloud to their own data centers
* **While getting started with the cloud is easy, moving away from the cloud is hard**
* **Cloud repatriation** **= nontrivial up-front investment in both commodities + 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 Multi-cloud Strategy

* Another way for companies to reduce their dependence on any single cloud provider is to follow a **multi-cloud strategy: spreading their workloads on multiple cloud providers**
* 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 (**vendor lock-in**)
* 2019 Gartner study: 81% of organizations are working with 2 or more public cloud providers
* Common pattern for ML workloads = do training on GCP or Azure, and deployment on AWS
* The **multi-cloud strategy doesn’t usually happen by choice**
* It’s **incredibly hard to move data and orchestrate workloads across clouds**
* Often, **multi-cloud 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** (acquired team is already on a cloud different from the host organization, and migrating hasn’t happened yet)
* Multi-cloud **can also happen due to strategic investments**
* Microsoft and Google = big investors in the startup ecosystem, and several companies that were previously on AWS have moved to Azure/GCP after Microsoft/Google invested in them

### Development Environment

* This is where MLE’s **write code, run experiments, and interact with the PROD environment where champion models are deployed and challenger models evaluated**
* It consists of the following components: **IDE, versioning, and CI/CD**
* Data scientists or MLE’s who writes code daily are probably very already familiar with all these tools
* Outside of a handful of tech companies, the DEV environment is severely underrated and underinvested in at most companies.
* Because the DEV environment is where engineers work, **improvements in the DEV environment translate directly into improvements in engineering productivity**

#### 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 and also consist of tools for versioning
* As of this writing, **companies use an ad hoc set of tools to version 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 version + track all ML workflows in 1 place
* **Versioning** is **important** for *any* SWE 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**
* The DEV environment should also be set up with a **CI/CDtest suite** **to test your code before pushing it to the staging or PROD environment**
* **CI/CD is a SWE concern**
* Some tools to orchestrate a CI/CD test suite: **GitHub Actions** and **CircleCI**
* **IDE = the editor where you write code, which tend to support multiple programming languages**.
* Can be native apps like VS Code or Vim, and can be browser-based, such as AWS Cloud9
* Many data scientists write code not just in IDEs but also in **notebooks** (Jupyter Notebooks, Google Colab)
* Notebooks are more than just places to write code
* **Can include arbitrary artifacts such as images, plots, data in nice tabular formats, etc., which makes notebooks very useful for EDA and analyzing model training results**.
* **Notebooks** have a **nice property**: they are **stateful** **(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**
* Especially helpful when dealing 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**
* Note that **stateful-ness can be a double-edged sword** = **allows *out-of-order* cell execution**
* This **makes notebook reproducibility harder unless your notebook comes with an instruction on the order in which to run your cells**
* Because notebooks are so useful for EDA 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 “Beyond Interactive: Notebook Innovation at Netflix,” Netflix included a list of infrastructure tools that can be used to make notebooks even more powerful
* 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***
* Can also help **summarize metrics from a collection of notebooks**
* **Commuter:** A notebook hub for viewing, finding, + sharing notebooks w/in an org.
* 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**
* Ex: In the early days of a startup, everyone worked from their own computer
* They had a bash file that a new team member could run to create a new virtual environment and install the required packages needed to run our code contained in requirements.txt that was added to as they started using a new package
* Sometimes, someone got lazy and 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 PR would run well on one computer but not another coworker’s computer, and it was usually because they used different versions of the same package
* They resolved to always specify package name together with package version when adding a new package to requirements.txt, and that removed a lot of unnecessary headaches
* One day, they ran into this weird bug that only happened during some runs and not others
* A coworker wasn’t able to reproduce the bug, and since the bug only happened some of the time, he ran the code 20 times but still found nothing
* They compared packages and everything matched
* After a few hours of hair-pulling frustration, they discovered that it was a concurrency issue that is only an issue for Python version 3.8 or earlier, and one person I had Python 3.8 and the coworker had Python 3.9, so he didn’t see the bug
* They resolved to have everyone on the same Python version, and that removed some more headaches
* Then one day, a coworker got a new laptop, a MacBook with the then *new* M1 chip
* He tried to follow the setup steps on this new laptop but ran into difficulty because the M1 chip was new, and some of the tools used, including Docker, weren’t working well with M1 chips yet
* After struggling with setting the environment up for a day, they decided to move to a cloud DEV environment.
* They still standardized 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
* Can use a cloud DEV environment that *also* comes with a cloud IDE, such as **AWS Cloud9 (**no built-in notebooks) and **Amazon SageMaker Studio** (comes with hosted JupyterLab)
* Nowadays, Amazon SageMaker Studio seems more widely used than Cloud9
* **However, most engineers 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**
* Ex: 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, 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
* VS Code is a good choice since it allows easy integration with cloud DEV instances
* At this startup, they chose **GitHub Codespaces as the 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, they were **worried about the cost** (what if they forgot to shut down our instances when not in use and they kept charging us money?)
* However, this worry went away for 2 reasons.
* **1) Tools like GitHub Codespaces automatically shut down instances after 30 minutes of inactivity**
* **2) Some instances are pretty cheap**
* Ex: 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**
* **1) Makes IT support so much easier**
* imagine having to support 1,000 different local machines instead of having to support only 1 type of cloud instance
* **2) Convenient for remote work**
* Can just SSH into your DEV environment wherever you go from any computer
* **3) Can help with security**
* Ex: 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**
* Ex: They aren’t allowed to have their code or data on the cloud
* **4) Potentially the biggest benefit for companies that do production on the cloud, is having your DEV environment in a cloud reduces the gap between the DEV and PROD environments**
* If your PROD 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, 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/instances (usually 1) 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**
* Ex: A celebrity tweets about your fledgling app and suddenly 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 PROD, 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** (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**
* **2 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**
* 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, and each is a Docker container**
* **Can build a Docker image either from scratch or from *another* Docker image**
* Ex: 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)
* **Example of a simple Dockerfile** that runs the following instructions:
* 1. Download the latest PyTorch base image
* 2. Clone NVIDIA’s apex GitHub repo, navigate to 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



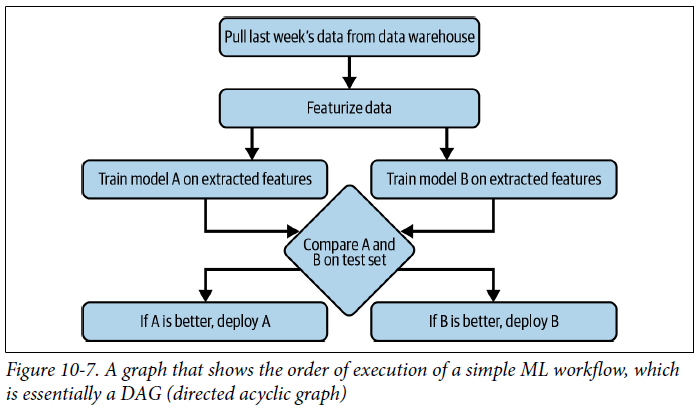
* **If your application does anything interesting, you will probably need more than one container**
* Consider a case where a **project consists of featurizing code that is fast to run but requires a lot of memory + model training code that is slow to run but requires less memory**
* If you run both 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 (ex: featurizer code requires NumPy 0.8 but model requires v1.0)**
* **If you have 100 microservices and each requires its own container, you might have 100 containers running at the same time**
* **Manually building, running, allocating resources for, + stopping 100 containers = a chore**
* A **tool to help you manage *multiple* containers is called container orchestration**
* **Docker Compose** = lightweight orchestrator that **can manage containers on a single host**
* ***However*, each container might run on its *own* host, where Docker Compose is at its limits**
* **Kubernetes (K8s) is a tool for exactly that**
* **Creates a network for containers to communicate and share resources**
* **Can help 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, and since its inception in 2014, it’s become ubiquitous in production systems today
* Good introduction to K8s: <https://www.jeremyjordan.me/kubernetes/>
* **However, K8s is *not* the most data-scientist-friendly tool, with many discussions on how to move data science workloads away from it**
* “Why Data Scientists Shouldn’t Need to Know Kubernetes” (2021)
* <https://huyenchip.com/2021/09/13/data-science-infrastructure.html>
* “Data Scientists Don’t Care About Kubernetes” (2020)
* <https://www.determined.ai/blog/data-scientists-dont-care-about-kubernetes>

### 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 + compute resources are much more *elastic*, 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 it means it can help engineers become more productive**
* This means that **it might make sense for companies to invest in automating workloads, which might make using resources less efficient than *manually* planning workloads, but free 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**

#### Cron, Schedulers, and Orchestrators

* **2 key characteristics of ML workflows that influence their resource management:** **repetitiveness** and **dependencies**.
* **Repetitiveness**: **Developing ML systems is an iterative process, and similarly, ML workloads are rarely one-time operations but something repetitive**.
* Ex: Might train a model every week or generate a new batch of predictions every 4 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 crondoes**
* 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 ***can’t* schedule anything *complicated* like: run B if A succeeds and run C if A fails”**
* **Dependencies:** **Steps in an ML workflow might have complex dependencyrelationships with each other**
* Ex: 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 steps can be represented using a graph:



* This is a **DAG**
* 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**.
* **A DAG is a common way to represent computing workflows in general, not just ML workflows**
* **Most workflow management tools require you to specify workflows in a form of DAGs**
* **Schedulers****= cron programs that can handle dependencies**
* It **takes in the DAG of a workflow and schedules each step accordingly**
* **Can even schedule to start a job based on an event-based trigger** (e.g., start a job whenever an event X happens)
* 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)
* **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**
* Ex: If a job requires 8 GB of memory and 2 CPUs, the scheduler needs to find an instance with 8 GB of memory and 2 CPUs among the resources it manages and wait until the instance is not executing other jobs to run this job on the instance
* Ex: How to schedule a job with Slurm, where you specify job name, time when the job needs to be executed, + 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*

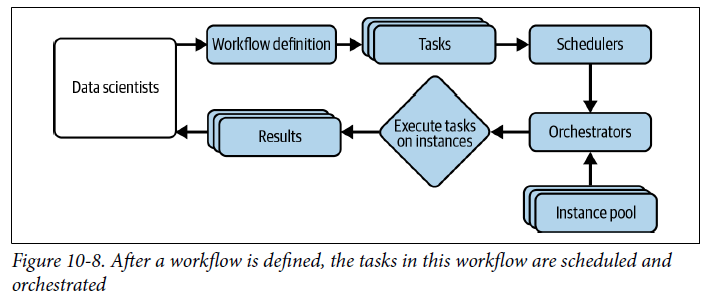
*#SBATCH --mem-per-cpu=4096 # Memory, in MB, to be allocated per CPU*

*#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***
* Ex: You might estimate, *and therefore specify*, that a job needs 4 GB of memory, but it 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,** **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., 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**
* **Most well-known orchestrator today = undoubtedly K8s**
* Can be used on-prem (even on your laptop via minikube)
* But few enjoy 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

* Those 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 vs. orchestrator talk
* **In its simplest form, workflow management tools manage workflows**
* They **generally allow you to specify your workflows as DAGs**
* **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 a 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 below:**



* There are many articles online comparing different data science workflow management tools
* We’ll go over 5 of the most common tools: **Airflow**, **Argo**, **Prefect**, **Kubeflow**, and **Metaflow**
* **Airflow: originally developed at Airbnb and released in 2014, one of the earliest workflow orchestrators**
* **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**
* A champion of the **“configuration as code”** **principle**
* <https://airflow.apache.org/docs/apache-airflow/stable/>
* Creators believed **data workflows are complex + should be defined using code (Python) instead of YAML or other declarative language**
* Ex: An Airflow workflow, drawn from the platform’s GitHub repo:

**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**
* <https://www.uber.com/blog/managing-data-workflows-at-scale/>
* Here, we’ll go over only **3 to give you an idea**.
* **1) Airflow is monolithic, which** **means it packages the *entire* workflow into 1 container**
* If 2 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
* **2) Airflow’s DAGs are not parameterized** = you **can’t pass parameters into workflows**
* So, if you want to run the same model with different learning rates, you’ll have to create different workflows
* **3) Airflow’s DAGs are static** = **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:** CEO Jeremiah Lowin was a core contributor of Airflow
* Early marketing campaign drew intense comparison between Prefect and Airflow
* <https://medium.com/the-prefect-blog/why-not-airflow-4cfa423299c4>
* **Prefect workflows are *parameterized* + *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**
* **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**
* Code sample from the Argo GitHub repo that demonstrates how to create a workflow to show a coin flip:
* <https://github.com/argoproj/argo-workflows/blob/master/examples/coinflip.yaml>
* 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**
* **To test the same workflow locally, you’ll have to use minikube to simulate a K8s on your laptop, which can get messy**
* **Kubeflow** and **Metaflow** **= 2** **tools that aim to help you run a 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 PROD 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 UX perspective, many say Metaflow is superior**
* 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 requirements for each step (required libraries, memory + compute requirements, etc.) and Metaflow will *automatically* create a container with *all* these requirements to execute the step, saving you from Dockerfiles or YAML files**
* Also **allows you to work seamlessly with both DEV and PROD environments from the same notebook/script**
* Can run experiments with small datasets on local machines, and when ready to run with the large dataset on the cloud, simply add @batch decorator to execute it on AWS Batch
* **Can even run *different* steps in the *same* workflow in *different* environments**
* Ex: If a **step requires a small memory footprint, it can run on your local**
* 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

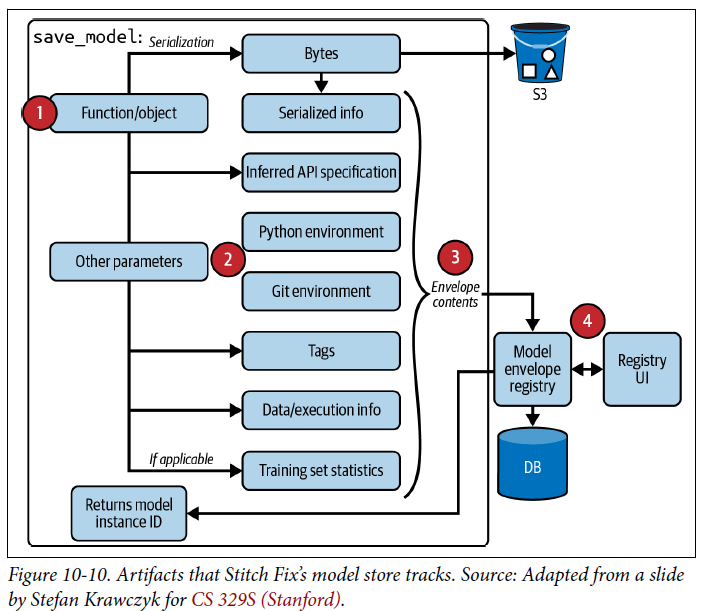
* The manager of the ML platform team at a major streaming company originally joined the company to work on their recommender systems
* To deploy their recommender systems, they needed to build out tools such as feature management, model management, monitoring, etc.
* Last year, his company realized 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 companies 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***
* Components most often sees in ML platforms include **model development + deployment, model store, and feature store**
* Evaluating a tool for each of these categories depends on your use case
* However, here are 2 general aspects you might want to keep in mind:
* **1) 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
* **2) 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 a 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

#### Model Deployment

* Once a model is trained (and hopefully tested), you want to make its predictive capability accessible to users
* A model can serve its predictions via online or batch prediction, and the simplest way to deploy a model is to push your model + its dependencies to a location accessible in production, and then expose the model as an endpoint to 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 + 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**
* **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**
* 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**
* Ex: 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**
* Different techniques for test in production: 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
* To deploy a model, you have to package it and upload it to a location accessible in production
* Model store suggests that it stores models, and that 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:
* 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**
* **8 types of artifacts that you might want to store**
* **Note: Many artifacts mentioned here = information that should be included in the model card**
* **1) Model definition**
* The information needed to create the shape of the model (e.g., what loss function it uses)
* If it’s a NN, this includes how many hidden layers it has and how many parameters are in each layer
* **2) 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 + the model definition together
* **3) 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
* **4) Dependencies**
* The dependencies (e.g., Python version, Python packages) needed to run your model are usually packaged together into a container
* **5) 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
* **6) 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 scientists commit the model generation code into their Git repos
* 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
* **7) Experiment artifacts**
* These are artifacts generated during the model development process, as discussed
* Can be graphs like the loss curve, or raw numbers like a model’s performance on the test set
* **8) 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**
* Ex: Model definitions and model parameters might be in S3, containers that contain dependencies in ECS (Elastic Container Service), data in Snowflake, experiment artifacts in Weights & Biases, and featurize and prediction functions 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 today (Spring 2023), MLflow is undoubtedly the most popular model store that isn’t associated with a major cloud provider
* Yet 3/6 top MLflow questions on Stack Overflow are about storing and accessing artifacts in MLflow
* **Model stores are due for a makeover**
* **Because of the lack of a good model store solution, companies like Stitch Fix resolve to build their *own* model store**
* Below 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.



#### 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
* At its core, there are **3 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:**
* **1) 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
* It’s often the case that features used for one model can be useful for another model
* Ex: 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 discovers 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**
* Ex: 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)
* **2) Feature computation (or “feature transformation”)**
* **Feature engineering logic, after being defined, needs to be computed**
* Ex: The feature logic might be: “use the average meal prep time from yesterday”
* The computation part involves actually looking into data + 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**
* **3) Feature consistency**
* We’ve talked about the **problem of having 2 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
* *But production code might be written in another language, like 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 now (Spring 2023), 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 surveyed in January 2022, only ~40% of use a feature store**
* **Out of those who use a feature store, half of them build their own feature store**

### Build Vs. Buy

* **It’s difficult 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**
* Ex: If you want to use fully managed Databricks clusters, you probably need only 1 engineer
* However, if you want to host your own Spark Elastic MapReduce clusters, you might need 5 more people
* **One extreme: you outsource all 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), while the rest of your infrastructure is managed by your vendor**
* **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 + 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**
* Ex: 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 vs. buy decisions depend on many factors**
* **3 common ones** when talking with heads of infrastructures on how they evaluate these decisions:
* **1) 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
* **2) 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: “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 tech sector (e.g., retail, banking, manufacturing), ML infrastructure isn’t their focus, so they tend to bias toward buying**
* 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 tech is the competitive advantage, + 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
* **3) The maturity of the available tools**
* Ex: 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
* “**Integration hell**” = spending more time integrating a solution with custom infrastructure instead of building out core features
* **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 engineers to build and maintain your own infrastructure**
* 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**
* **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: “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.”

### Summary

* **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
* We covered different layers of infrastructure needed for ML systems, starting 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 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**
* **DEV environment = where data scientists write code and interact with the PROD 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 MLE’s working on the same team**
* **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**
* Evolution of resource management tools: from cron to schedulers to orchestrators
* **ML workflows are different from other SWE workflows, and they need their own workflow management tools** (ex: Airflow, Argo, Metaflow, etc.)
* **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
* **3 sets of tools essential for most ML platforms: deployment, model store, feature store**
* When working on infrastructure, a question constantly haunts engineering managers and CTOs alike: **build or buy?**
* We went over a few discussion points that hopefully provide you/your team with sufficient context to make those difficult decisions