**Why Do People Say It’s So Hard To Deploy A ML Model To Production?**

**Multiple dimensions of change create specialized challenges in ML services**



Photo by [Dan Lohmar](https://unsplash.com/@dlohmar) on [Unsplash](https://unsplash.com/photos/WajTuzeanUk)

You’ve heard it said, you’ve heard it written and you’ve heard it sung from the rooftops by analysts and vendors alike: [87% of data science projects never make it into production](https://venturebeat.com/2019/07/19/why-do-87-of-data-science-projects-never-make-it-into-production/).

Why is that?

Well, of course, it always depends, but the slightly more precise answer is that building ML services is more complex than building other types of software. While ML applications share many of the same complexities as traditional software and data engineering projects, ML introduces a new artifact that adds a layer of complexity: the model itself.

Here’s how I’ve started think of it:

* In traditional software engineering, your main concern is the **code**.
* In data engineering, your main concerns are the **code** and the **data**.
* In ML engineering, your main concerns are the **code,** the **data,** and the **model**.

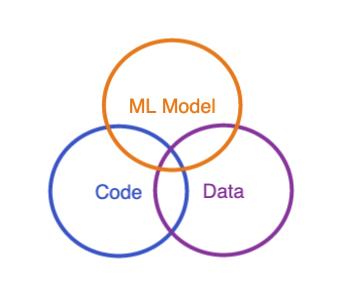


Photo by [author](https://www.linkedin.com/in/timliu9/)

It’s not just me, either. In a [recent article](https://martinfowler.com/articles/cd4ml.html), software thought leader, Martin Fowler, shares why ML services are so complex:

“Besides the code, changes to ML models and the data used to train them are another type of change that needs to be managed and baked into the software delivery process.”

In this article, I’ll dig into what you need to know about each of these three components and how they interact with each other, as well as offer some tips and recommendations for managing them to deploy your models.

**Dimension 1: The code**

The role of code in an ML application is essentially the “glue” that:

* Exposes an API
* Transforms the data to extract appropriate features
* Uses the model for inference
* Incorporates business logic

For the most part, it is a fairly known quantity and has a deterministic quality about it. In other words, given the same input, it will produce the same output. Even if your code is stateful (not recommended), I’d argue that the “state” is part of the input.

Below are several challenges you may face in managing the code for your ML service:

**Updates to code can result in regressions**

The feature extraction code is commonly written by the data scientist. When it comes time to create the API, developers will need to migrate the code from the training environment to the prediction service. Additionally, they may have to transform the input to the service (perhaps a REST request) into usable data. Developers and data scientists will absolutely want to update this code as they iterate on the service which can cause regressions.

**Tips:**

* Unless there’s a compelling reason not to, create the model serving service in Python. Most data scientists use Python. It makes the deployment process way more streamlined. In my experience, users go from “weeks/months” to days when redeploying a new model. Some teams need ridiculously high performance and prefer to convert all the code into Rust/Go. Most do not. By the way, AWS Sagemaker is written in Python.
* Remember to implement good coding practices even in Jupyter notebooks. You’ll be surprised what code ends up going to production.

**Dependency updates can have unforeseen consequences**

Dependencies in a prediction service are very important because if they are not precisely configured, not only could the code behave differently than expected, the model itself could make different predictions than in the training environment. The right versions of the ML library, the runtime, and the dependencies should all be used in order for the service to run correctly. That said, dependencies must be updated from time to time to keep up with security patches at the very least.

Tip: Make sure you use the same version of Python and the ML library that you used to train the model. You can use “pip freeze” to determine which versions you’re using.

**The deployment environment can be unstable**

A scalable prediction service needs the proper environment in order to run the code correctly. For example, particular ML libraries can only run on particular OS distributions. Docker is the standard for configuring a reproducible environment, but configuring Docker for an ML use case can be extremely challenging. If configured correctly, it will give you a clean artifact from which you can deploy to a number of different services, as well as provide the ability to run the service locally so that you can debug it if there is an issue.

**Tips:**

* Make sure the Docker image (or associated runtime) exposes health, documentation and monitoring endpoints for your service.
* For GPU support, use [Nvidia’s docker image](https://hub.docker.com/r/nvidia/cuda), it provides a good base out of the box to GPU workloads. Believe me, you do not want to end up in “cuda hell”

**Dimension 2: The data**

Once you’ve addressed the key challenges with regards to managing the code, it’s time to look at the data.

The **data** is deterministic in a way, but having spent years building data pipelines, it’s difficult to know what types of data and at what volumes you will receive it. You will always be surprised. For this reason, I would say the data is the most variable piece of the puzzle.

Once your service is in production, there is no telling how the data will change over time — whether that means receiving bad data all of a sudden or simply experiencing changes in the data as users change their behavior. While data changes are often impossible to predict, you can still manage it with certain strategies.

**Data could drift and cause service degradation**

Changes in the data over time can result in a variety of issues: the model may produce less accurate predictions or fail to perform at all. Data drift, or variations in production input data, can happen as time passes or as seasons change. It’s important to know if it’s happening so that you can retrain your model if necessary.

**Tips:**

* Deploying often and at speed will give you more confidence in your pipeline. Not only that, it has added benefits like preventing problems due to input data drift.
* Use ML specific monitoring tools like [WhyLabs](https://whylabs.ai/), [Evidently.ai](https://evidentlyai.com/) or [Aporia](https://www.aporia.com/) can help detect data drift and monitor performance.

Model performance can also change as time passes, but can be much more difficult to monitor because it requires you to join predictions with downstream outcomes. For example, connecting the dots between an ML-driven product recommendation and if the product was actually purchased by the consumer.

**Tip:** Monitoring performance often includes joining with downstream metrics. Make sure you’re sending your inference results to the same place (likely a data warehouse) as the metrics you want to join with.

**Bad data can break the service**

APIs are a powerful construct because it exposes functionality to the world. That means it’s unlikely you can control what’s being sent to your service. Unexpected data is just as bad as data that is incorrect and it’s likely your service will receive both at some point. Try to make sure this doesn’t cripple your system.

**Tip:** Use a data validation framework like [Pydantic](https://pydantic-docs.helpmanual.io/) to validate incoming data so that it doesn’t unexpectedly break your service.

**Fresh data is needed to validate new models**

As data changes in production, you may not necessarily want to risk deploying a new model. For this reason, it can be helpful to have ways to test production data with models that you’re still validating. Getting that production data can be difficult because the production environment is usually deployed in a secure environment.

**Tip:** Using a service mesh like Istio or AWS App Mesh can help fan out production traffic and help validate new models using [shadow pipelines](https://istio.io/latest/docs/tasks/traffic-management/mirroring/).

**Dimension 3: The model**

The last dimension is of course, the model itself.

I think of the **model** as a deterministic function, albeit the most blackbox of the three dimensions. It’s similar to the experience of reading “spaghetti code” that even the most experienced engineers can’t figure out: a trained model is composed of complex vectors, which by itself are very difficult to understand, but given a particular input, the output should always be the same for a given version of a model.

Models may change because a data scientist is iterating on model performance, or is automatically retraining them to keep the model trained with the latest data. There are many opportunities for a model to break on its way to production.

**Models can be hard to reproduce**

ML frameworks have different methods of saving a model to ensure the reproducibility of the prediction. Depending on how you save the model, it could be serializing the Python class or just saving the weights of the model. Both methods could fail to reproduce the model depending on the context of the training environment. If the model is not saved correctly, then it could produce erratic predictions in production.

**Tip:** Do your research when saving your model and make sure it’s the recommended way of saving. For example, [Pytorch has documentation](https://pytorch.org/tutorials/beginner/saving_loading_models.html) that explains the various methods used to save the model depending on your context.

**The hand-off between people and systems can be complicated**

Saving the model may result in multiple files containing the model’s weights and associated metadata. All of these are required to instantiate the model again. You’ll want to move these artifacts to a centralized location where the automated deployment pipeline can ship the model to production.

The problem is that production CI/CD pipelines are likely located in secure environments in the cloud. Designing a secure, streamlined system that goes from a data scientist’s laptop (or a remote training environment) to production can be complicated.

**Tip:** Try to find a standard which you can use to coordinate the model training with the model service. Tools like [MLFlow](https://github.com/mlflow/mlflow), [BentoML](https://github.com/bentoml/BentoML), [TFServing](https://github.com/tensorflow/serving) and [Cortex](https://github.com/cortexproject/cortex) help to coordinate the model into a model registry and ultimately to a deployed destination. For example, use MLFlow to manage your experimentation and training and BentoML for model serving and deployment.

**You could accidentally lose good models**

Persisting various versions of the model is extremely important in order to rollback bad updates or debug working models. You may even want to run multiple model versions in production (known as A/B testing) in order to gauge which model runs the best. More sophisticated deployments can run model version experiments like “[multi-arm bandit](https://en.wikipedia.org/wiki/Multi-armed_bandit)” that can score the models in real time and direct more traffic to the best model.

**Tip:** Tools like S3 or Google Cloud Storage can be an elegant solution as a shared model repository that data scientists and developers can use to share models.

**Conclusion**

The complexity introduced by having three different and shifting dimensions is not trivial, especially given that the development process often involves multiple stakeholders. We’re still in the early days of developing best practices, because operationalizing ML projects is such a new field. With more tools and practitioners entering this market, there’s no doubt that best practices will begin to emerge just like they did in traditional software, DevOps, and even data engineering.

[As Andrew Ng puts it](https://www.deeplearning.ai/wp-content/uploads/2021/06/MLOps-From-Model-centric-to-Data-centric-AI.pdf), the goal of these MLOps tools and processes is to make “AI an efficient and systematic process.” The more we can streamline these processes, the better companies can begin to benefit from ML.