**Deploy, Version Control, and Manage ML Models Right From Your Jupyter Notebook with Modelbit**

Deployment has possibly never been so simple.

* [](https://www.dailydoseofds.com/author/avi/)

[Avi Chawla](https://www.dailydoseofds.com/author/avi/)



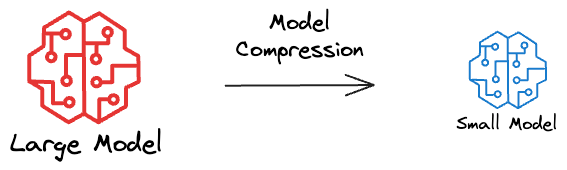
TABLE OF CONTENTS

1. [Introduction](https://www.dailydoseofds.com/deploy-version-control-and-manage-ml-models-right-from-your-jupyter-notebook-with-modelbit/#introduction)
2. [Post-deployment considerations](https://www.dailydoseofds.com/deploy-version-control-and-manage-ml-models-right-from-your-jupyter-notebook-with-modelbit/#post-deployment-considerations)
3. [Why is Deployment Challenging (Typically)](https://www.dailydoseofds.com/deploy-version-control-and-manage-ml-models-right-from-your-jupyter-notebook-with-modelbit/#why-is-deployment-challenging-typically)
4. [Pain Points with Traditional Hosting Services](https://www.dailydoseofds.com/deploy-version-control-and-manage-ml-models-right-from-your-jupyter-notebook-with-modelbit/#pain-points-with-traditional-hosting-services)
5. [Basic model deployment with Modelbit](https://www.dailydoseofds.com/deploy-version-control-and-manage-ml-models-right-from-your-jupyter-notebook-with-modelbit/#basic-model-deployment-with-modelbit)
   1. [Process workflow](https://www.dailydoseofds.com/deploy-version-control-and-manage-ml-models-right-from-your-jupyter-notebook-with-modelbit/#process-workflow)

**Introduction**

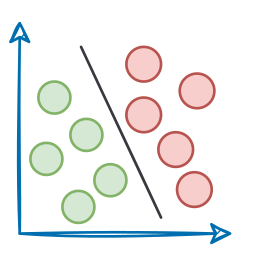
In recent deep dives, we’ve primarily focused on cultivating skills that can help us develop large machine learning (ML) projects.

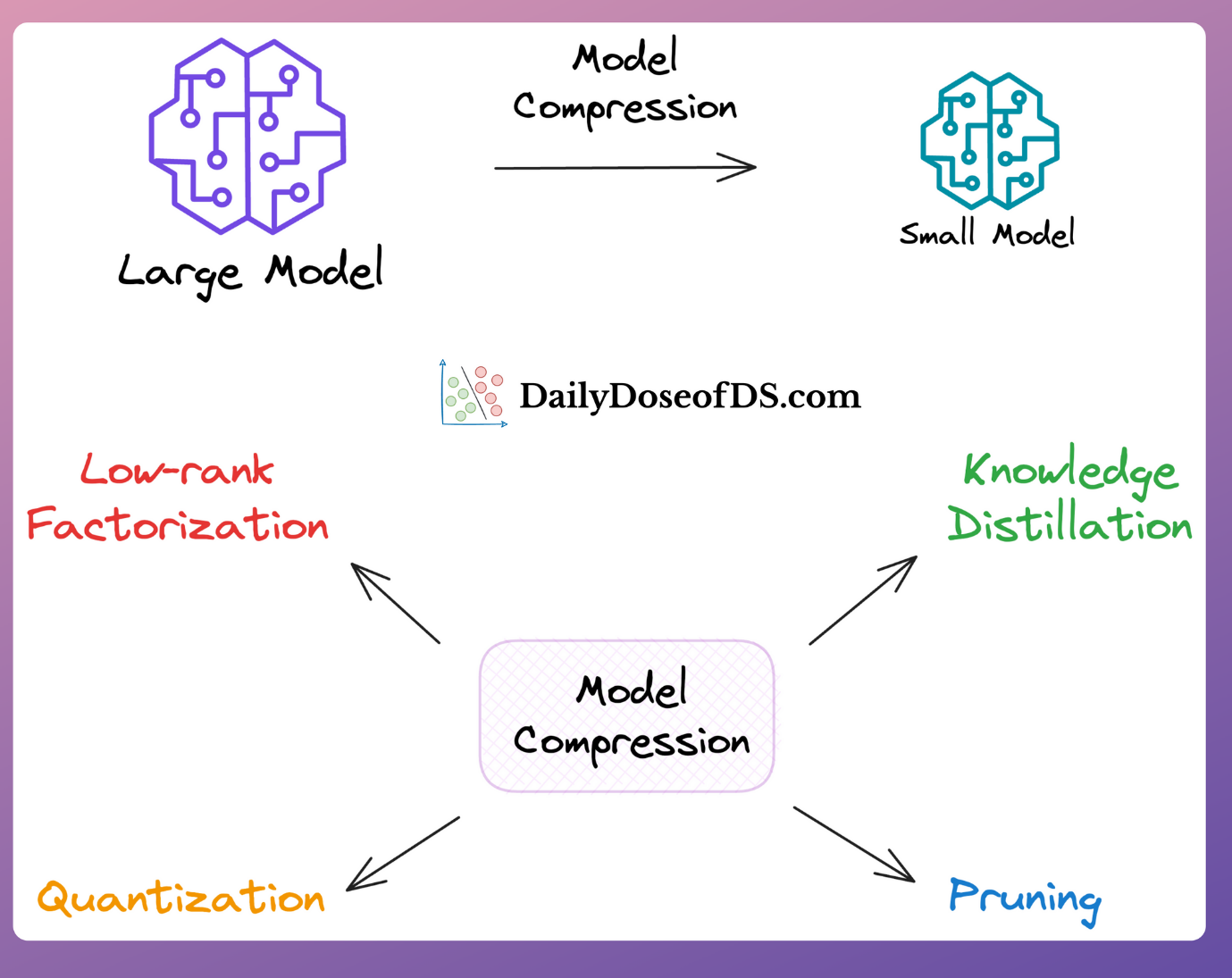
For instance, in the most recent deep dive on “**Model Compression**”, we learned many techniques to drastically reduce the size of a model to make it more production-friendly.



**[Model Compression: A Critical Step Towards Efficient Machine Learning](https://www.dailydoseofds.com/model-compression-a-critical-step-towards-efficient-machine-learning/)**

[Four critical ways to reduce model footprint and inference time.](https://www.dailydoseofds.com/model-compression-a-critical-step-towards-efficient-machine-learning/)

[[](https://www.dailydoseofds.com/model-compression-a-critical-step-towards-efficient-machine-learning/)Daily Dose of Data ScienceAvi Chawla](https://www.dailydoseofds.com/model-compression-a-critical-step-towards-efficient-machine-learning/)

[[](https://www.dailydoseofds.com/model-compression-a-critical-step-towards-efficient-machine-learning/)](https://www.dailydoseofds.com/model-compression-a-critical-step-towards-efficient-machine-learning/)

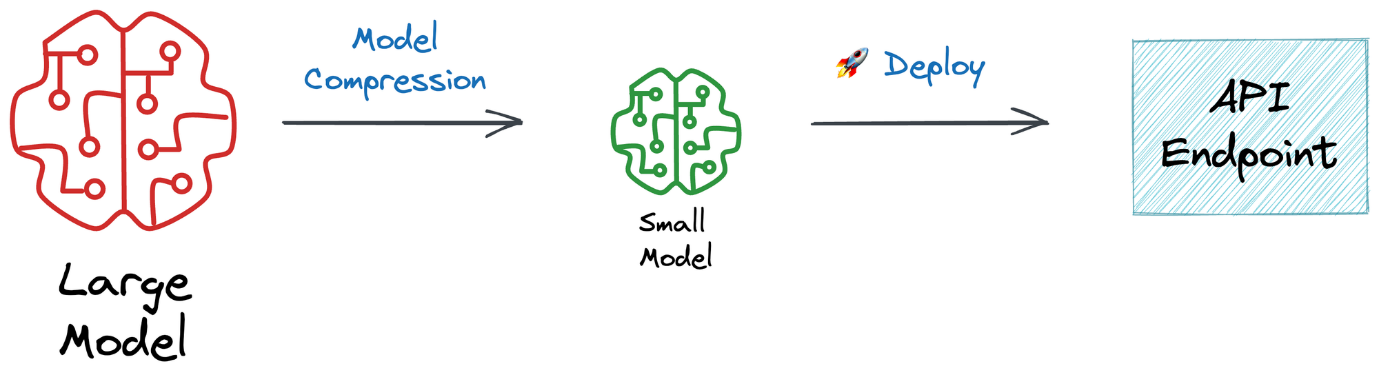
In the above article, we saw how these techniques allow us to reduce both the latency and size of the original model, which directly helps in:

* Lowering computation costs.
* Reducing model footprint.
* Improving user experience due to low latency…

**…all of which are critical metrics for businesses.**

However, learning about model compression techniques isn’t sufficient.

In most cases, we would only proceed with model compression when the model is intended to serve an end-user.



And that is only possible when we know how to deploy and manage machine learning in production.

Thus, after learning about model compression techniques, we are set to learn the next critical skill—**deployment.**

In my opinion, many think about deployment as just “deployment” — host the model somewhere, obtain an API endpoint, integrate it into the application, and you are done!

**But that is almost NEVER the case.**

This is because, in reality, plenty of things must be done post-deployment to ensure the model’s reliability and performance.

What they are? Let’s understand!

**Post-deployment considerations**

Deployment is a pivotal stage in the ML project lifecycle. It’s that stage where a real user will rely on your model’s predictions.

Yet, it’s important to recognize that deployment is not the final destination.

After deploying a model, several critical considerations must be addressed to ensure its reliability and performance.

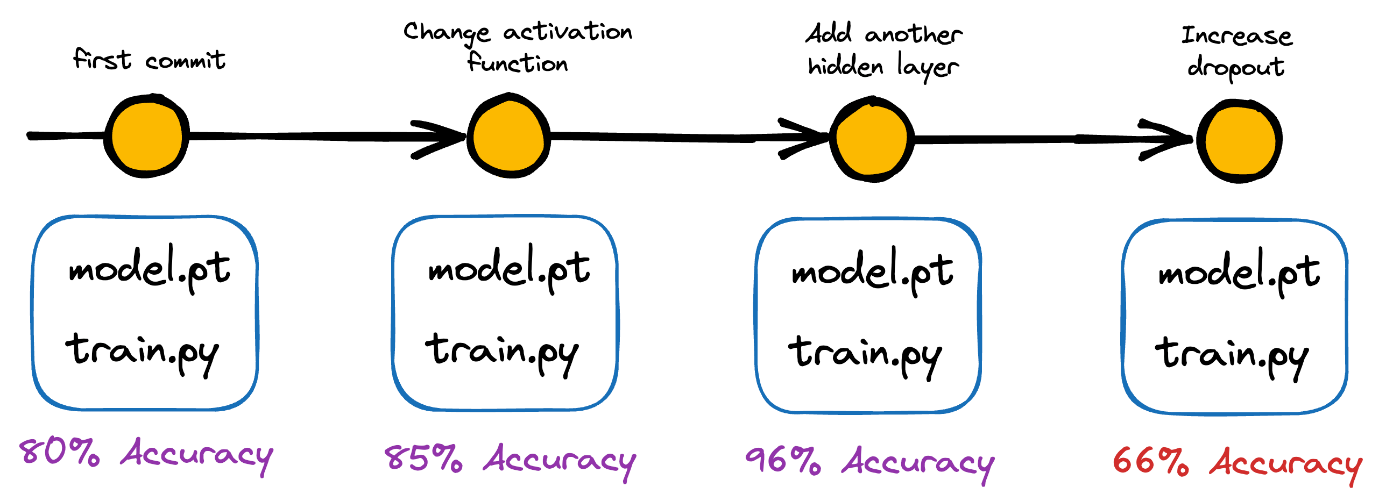
Let’s understand them.

**#1) Version Control**

Version control is critical to all development processes. It allows developers to track software changes (code, configurations, data, etc.) over time.

In the context of data teams, version control can be especially crucial when deploying models.

For instance, with version control, one can precisely identify what changed, when it changed, and who changed it — which is crucial information when trying to diagnose and fix issues that arise during the deployment process or if models start underperforming post-deployment.



This goes back to what we discussed in a recent deep dive — “*Machine learning deserves the rigor of any software engineering field.*”

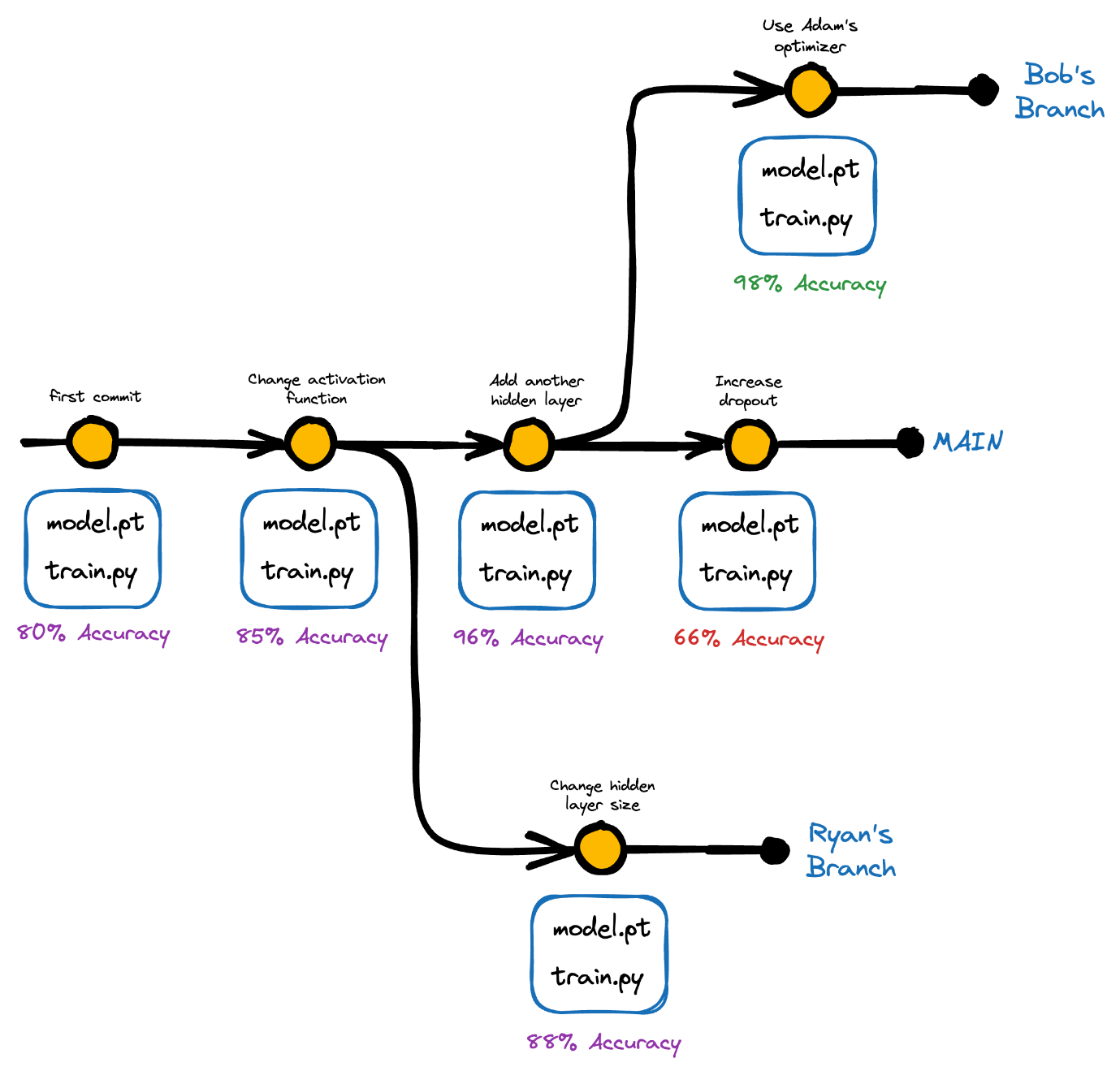
If the model starts underperforming, git-based functionality allows us to quickly roll back to previous versions of the model.

There are many other benefits too.

**#1.1) Collaboration**

Effective collaboration becomes increasingly important as data science projects get bigger and bigger.

Someone in the team might be working on identifying better features for the model, and someone else might be responsible for fine-tuning hyperparameters or optimizing the deployment infrastructure.



And it is well known that with version control, teams can work on the same codebase/data and improve the same models without interfering with each other’s work.

Moreover, one can easily track changes, review each other’s work, and resolve conflicts (if any).

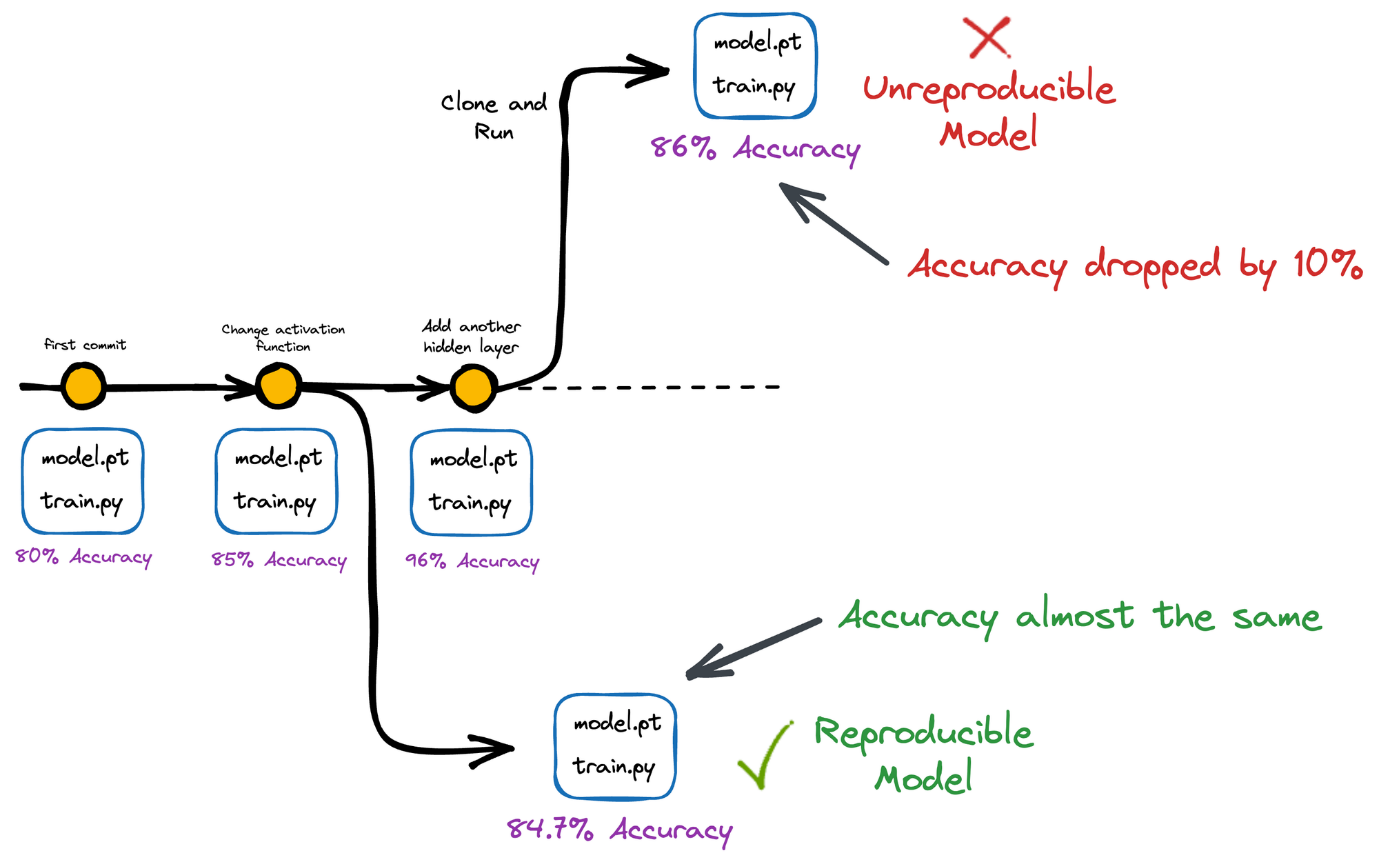
**#1.2) Reproducibility**

Reproducibility is one of the critical aspects of building reliable machine learning.

Imagine this: Something that one works on one system but does not work on another reflects bad reproducibility practices.

Why it’s important, you may wonder?

It ensures that results can be replicated and validated by others, which improves the overall credibility of our work.



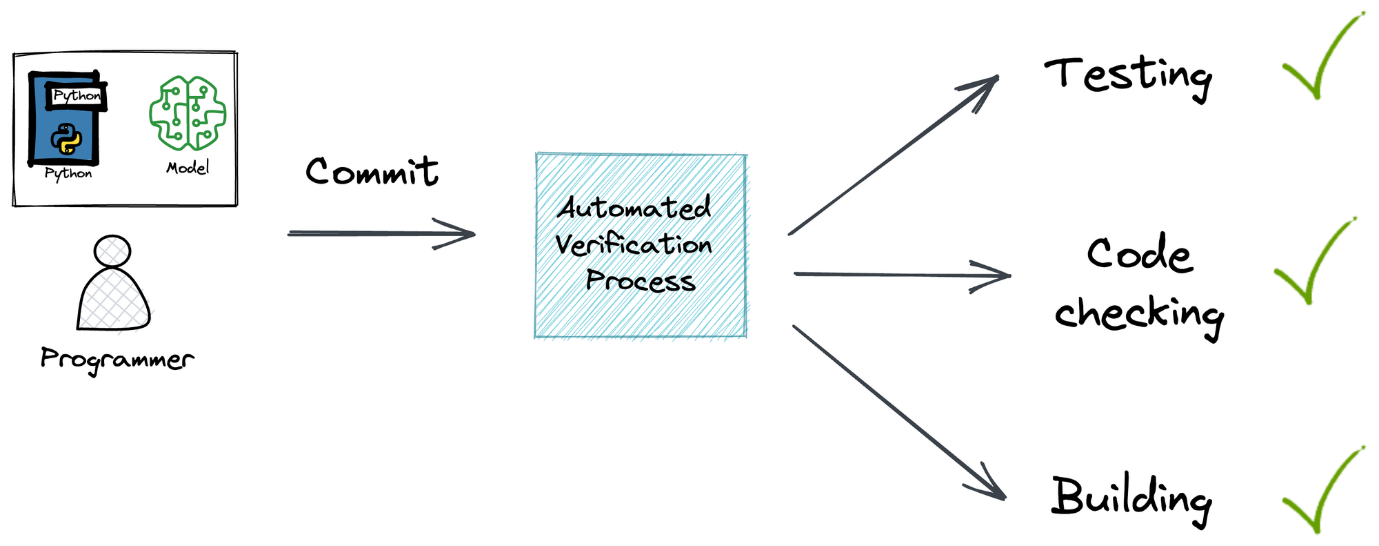
Version control allows us to track the exact code version and configurations used to produce a particular result, making it easier to reproduce results in the future.

This becomes especially useful for open-source data projects that many may use.

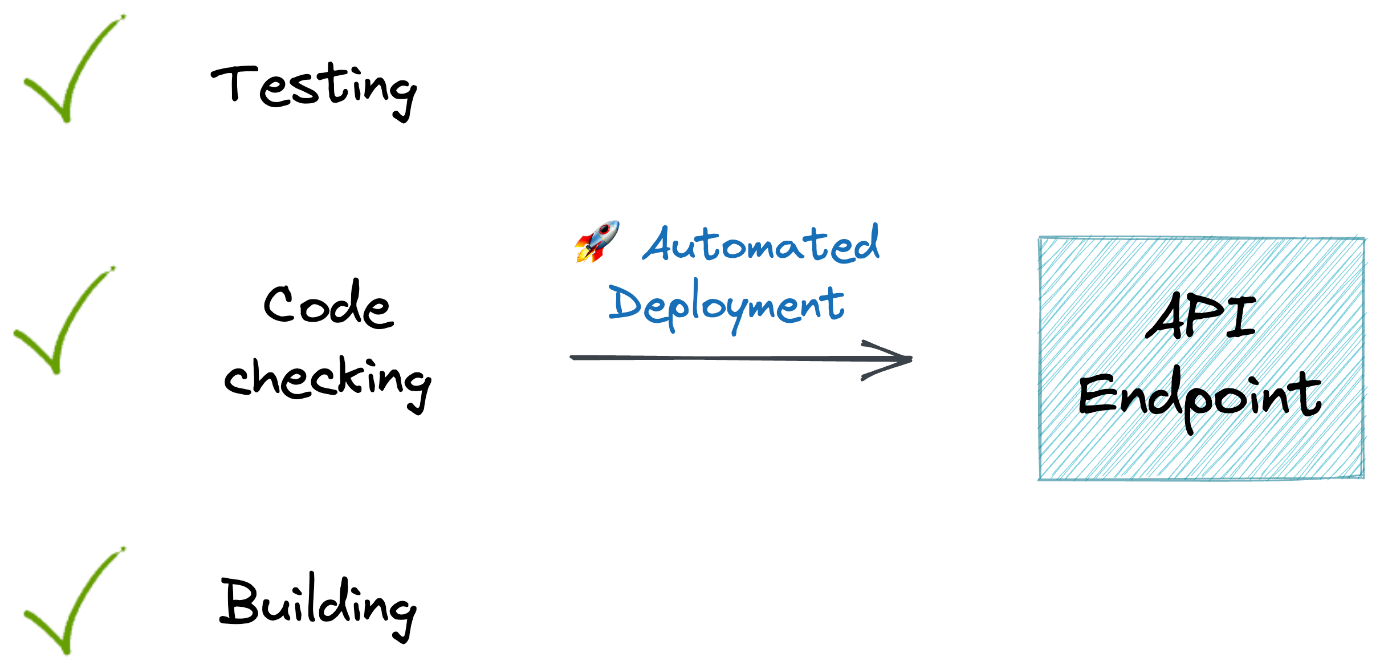
**#1.3) Continuous Integration and Continuous Deployment (CI/CD)**

CI/CD enables teams to build, test, and deploy code quickly and efficiently.

In machine learning, Continuous Integration (CI) may involve building and testing changes automatically to ML models as soon as they are committed to a code repository.



In Continuous Deployment (CD), the objective can be to reflect new changes to the model once they have passed testing.

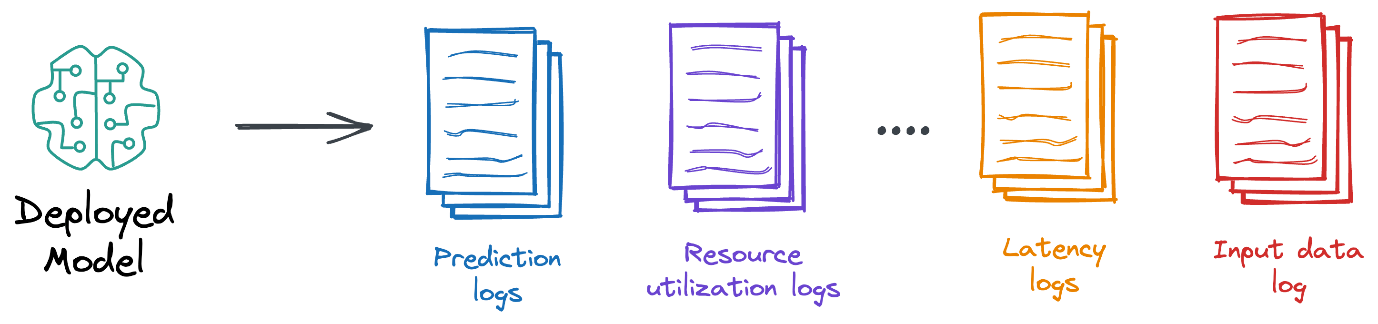


Consequently, it should seamlessly update the changes to production, making the latest version of the model available to end users.

**#2) Model Logging**

Model logging is another crucial aspect of post-deployment ML operations.

As the name suggests, logging involves capturing and storing relevant information about model performance, resource utilization, predictions, input data, latency, etc.



There are various reasons why model logging is important and why it’s something that should NEVER be overlooked.

To understand better, imagine you have already deployed a model, and it is serving end-users.

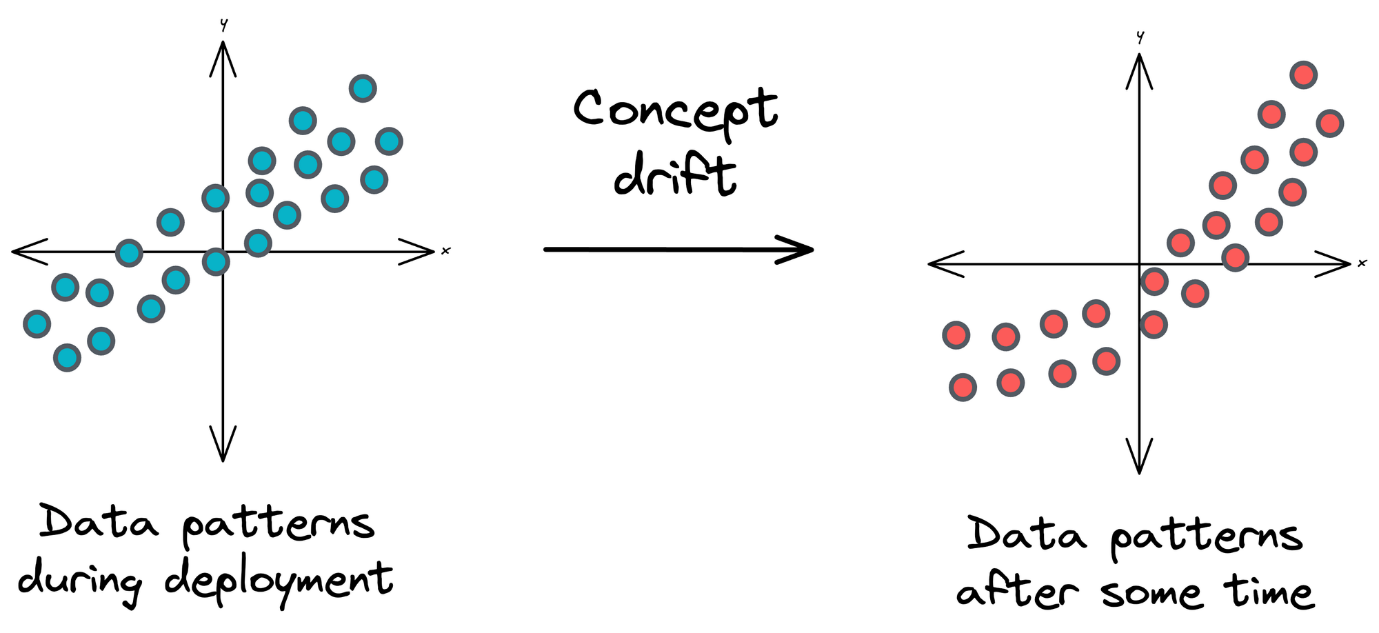
Once deployed, there’s little possibility that nothing will go wrong in production, **especially on the data front**!

Let’s understand in detail.

**#2.1) Concept drift**

**Concept drift** happens when the statistical properties of the target variable or the input features sent as input to the model change over time.

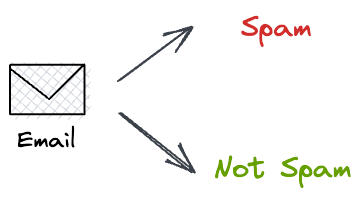
In simpler terms, the relationship between your model's inputs and outputs evolves, making your model less accurate over time if not addressed.



Concept drift can occur due to various reasons, such as:

* Changes in user behavior
* Shifts in the data source
* Alterations in the underlying data-generating process.

For instance, imagine you are building a spam email classifier. You train the model on a dataset collected over several months.



Initially, the model performs well and accurately classifies spam and non-spam emails.

However, over time, email spamming techniques evolve.

New types of spam emails emerge with different keywords, structures, and techniques.

This change in the underlying concept of “spam” represents concept drift.

That is why it is important to have periodic retraining or continuous training strategies in place.

If your model isn't regularly retrained with up-to-date data, it may start misclassifying the new types of spam emails, leading to decreased performance.

**#2.2) Covariate shift**

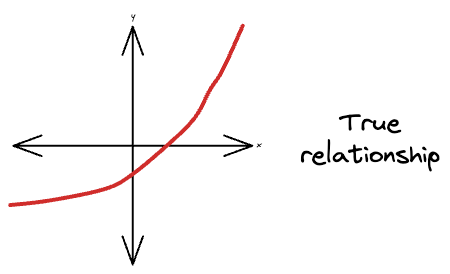
💡

The term ‘Covariates’ refers to the features of your model.

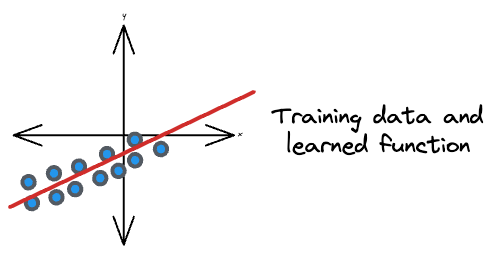
**Covariate shift** is a specific type of concept drift that occurs when the distribution of the input features (covariates) in your data changes over time, but the true relationship between the target variable and the inpute remains the same.

In other words, the true (or natural) relationships between the input features and the target variable stay constant, but the distribution of the input features shifts.

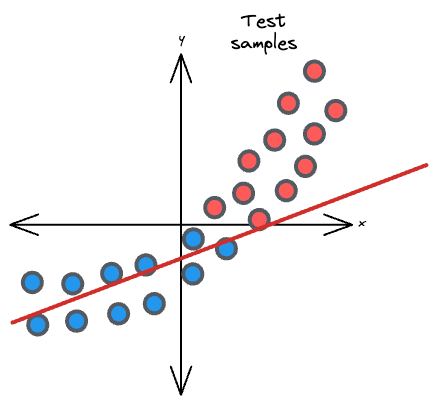
For instance, consider this is the true relationship, which is non-linear:



Based on the observed training data, we ended up learning a linear relationship:



However, at the time of inference post-deployment, the distribution of input samples was different from that of the observed distribution:



It leads to poor model performance because the model was trained on one distribution of the data, but now, it is being tested or deployed on a different distribution.

Methods for addressing covariate shifts include reweighting the training data or using domain adaptation techniques to align the source and target distributions.

💡

To an extent, batch normalization is a remedy for covariate shifts in neural networks.

For instance, suppose you are building a weather forecasting model. You train the model using historical weather data from a specific region, and the training data includes features like temperature, humidity, and wind speed.

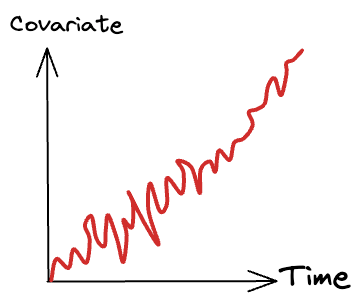
However, when you deploy the model to a different region with a distinct climate, the distribution of these features can shift significantly.

For instance, temperature ranges and humidity levels in the new region might be quite different from those in the training data. This covariate shift can cause your model to make inaccurate predictions in the new environment.

**#2.3) Non-stationarity**

When building statistical models, we typically assume that the samples are identically distributed.

**Non-stationarity** refers to the situation where the **probability distribution** of the samples evolves over time in a non-systematic or unpredictable manner.



This can encompass various aspects, including changes in data distributions, trends, seasonality, or other patterns.

**Non-stationarity can be challenging for machine learning models, as they are typically trained while assuming that the data distribution remains constant.**

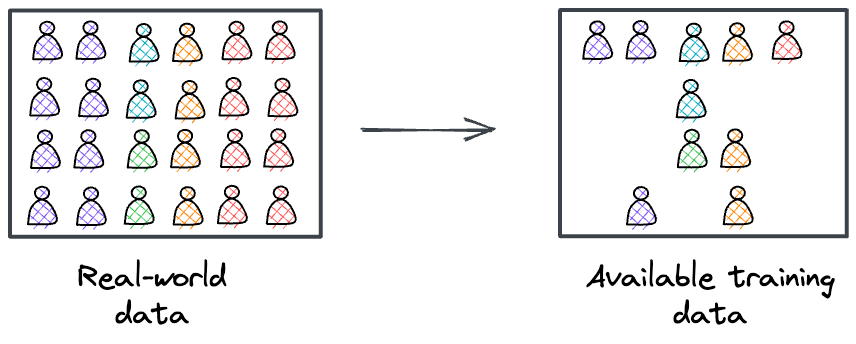
For instance, assume you are building some wealth predictor. Using your currency amount feature will typically not serve as a good feature because currency values get affected due to inflation.

Models deployed in non-stationary environments may need regular updates or adaptive learning strategies to cope with changing data patterns.

**#2.4) Unrepresentative training data**

**Unrepresentative training data** is a situation where the data used to train a machine learning model does not adequately represent the real-world conditions or the diversity of scenarios that the model will encounter in production.

When training data is not representative, the model may perform well on the training data but poorly on new, unseen data.



This issue can lead to bias and poor generalization.

For instance, suppose you are building a speech recognition system for a voice assistant.

You collect training data primarily from young adults with clear accents and no speech impairments.

However, in real-world usage, the voice assistant will be used by people of all ages who speak various languages and have different speech patterns.

If your training data is unrepresentative and biased towards a specific demographic, the model may struggle to understand and accurately transcribe speech from a more diverse user base, leading to poor performance in production.

The above four problems that we typically face in production systems highlight the importance of model logging.

As discussed above, addressing these issues often involves continuous monitoring of model performance in the deployment environment, collecting and labeling new data when necessary, and retraining the model to adapt to changing conditions.

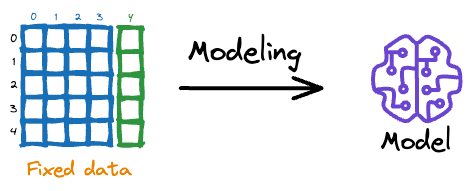
Traditionally, in industry, advanced techniques like adaptive learning are employed to update the model with the new data and potentially mitigate the impact of concept drift, covariate shift, non-stationarity, and unrepresentative training data.

👉

Below, we shall discuss a bit about Adaptive learning (mostly conceptual ideas), but we will do a full practical article on this soon.

**Adaptive learning**

Traditionally, ML models are trained on some gathered fixed/static dataset and then used to make predictions on unseen data.

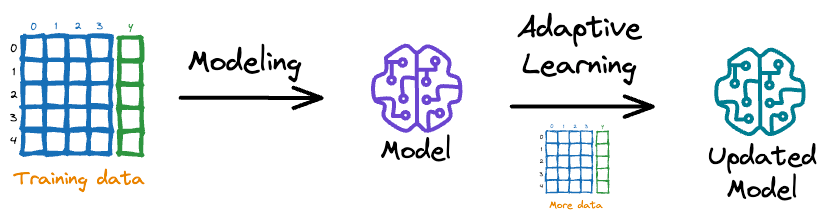


While this is how machine learning has been typically (and successfully) approached so far, it gets infeasible to train a new model from scratch every time we get some new data.

This makes intuitive sense as well.

Adaptive learning is a remedy to this problem.

Adaptive models can adapt and improve their performance as they are exposed to more data, leading to better accuracy and utility.



In situations where the data distribution is constantly changing, adaptive models can adapt and continue to perform well, while non-adaptive models may struggle.

A major advantage of adaptive learning is that since the model isn’t trained from scratch, for every update, the additional computational cost is small if the previously trained model is used for the start of the next retraining iteration.

In the realm of solving real-life problems by deploying machine learning models, it is inevitable that, with time, data distribution will change.

As a result, the models trained on old data will likely provide little value going forward.

Adaptive models are a great solution in such situations, as they consistently adapt to the incoming data streams.

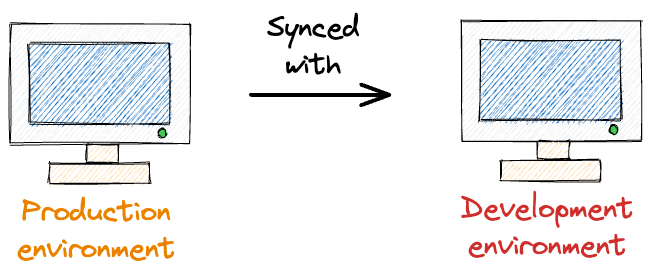
**Why is Deployment Challenging (Typically)**

**#1) Consistency challenges**

In almost all ML use cases, the algorithm is never coded from scratch.

Instead, one uses open-source implementations offered by libraries like PyTorch, Sklearn, and many more.

To ensure reproducibility in production, the production environment should be consistent with the environment in which it was trained.



This involves installing similar versions of libraries used, software dependencies, OS configurations, and many more.

Of course, achieving this consistency is not a painstaking process. All you should do is maintain an environment configuration.

Yet, it does require careful environment configuration and management.

This involves documenting and tracking the versions of all software components, libraries, and dependencies used during model development and deployment.

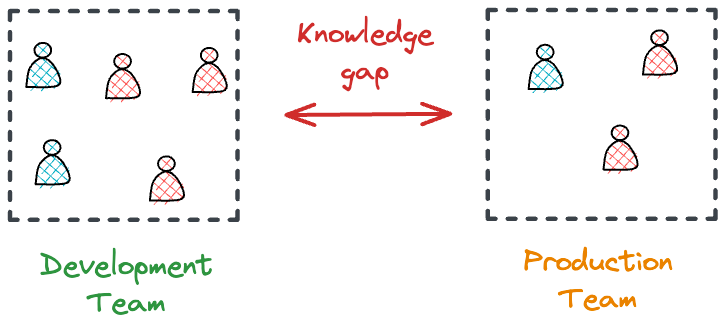
To address these consistency challenges, organizations often use containerization technologies like Docker.

Containers encapsulate the entire environment, including software dependencies, libraries, and configurations, ensuring that the same environment is replicated in both the development and production stages.

**#2) Inadequate Expertise (or Knowledge Gap)**

ML engineers may not have experience with deployment. They may not have the necessary expertise in areas such as software engineering, MLOps, and infrastructure management.

This can make it difficult for them to effectively deploy and scale models in production environments.



In such cases, organizations hire specialized talents.

However, engineers hired specifically for deployment may not have an in-depth understanding of ML algorithms and techniques.

This makes it difficult for them to understand the code and make necessary optimizations, leading to issues with scaling, performance, and reliability, and can ultimately impact the effectiveness of the model in production.

The above pain points, along with the data challenges we discussed above, highlight the **necessity for a data scientist to have the necessary deployment expertise**.

**Pain Points with Traditional Hosting Services**

Traditional hosting services like Google Cloud, AWS, and Heroku have been go-to options for deploying machine learning models.

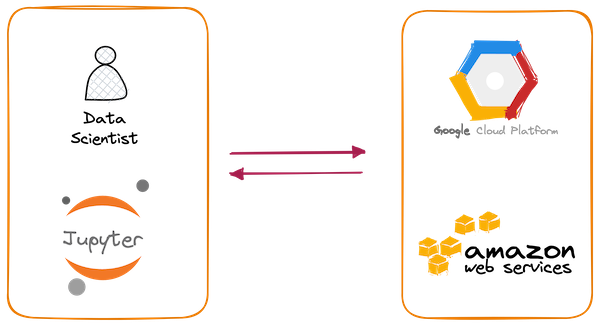
However, the process can be challenging and time-consuming, requiring specialized expertise in infrastructure and DevOps.

For data scientists without these skills, deploying models to production can be a significant pain point.

There are several challenges associated with traditional hosting services.

**#1) No Jupyter Notebook Support**

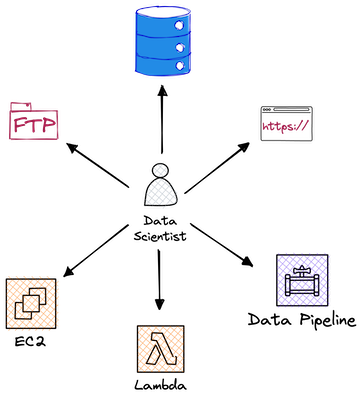
First, data scientists often switch between different tools and environments to manage deployments. This means leaving the comfort of their Jupyter notebooks, where they spend most of their time developing and refining models.



The process can be jarring, and the need to learn new tools and interfaces can slow down productivity.

**#2) Specialized Expertise**

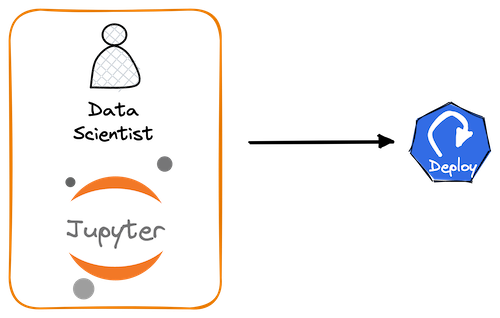
Second, deploying machine learning models to production environments demands plenty of configuration and management of infrastructure resources, including servers, networking, and security.



This is a specialized area that many data scientists may need to become more familiar with, and it can take a lot of time and effort to get right.

The above pain points highlight a need for a simple and elegant way to deploy machine learning models that doesn’t require specialized expertise and can be done entirely from a Jupyter Notebook.

Modelbit is a deployment service that specifically addresses all these challenges, allowing data scientists to deploy models with just a single command from their notebooks.



With Modelbit, there’s no need to worry about infrastructure, security, or server management — the service takes care of everything, allowing data scientists to focus on what they are supposed to do — building and improving models.

Thus, in this article, let’s understand how to use Modelbit for machine learning model deployment.

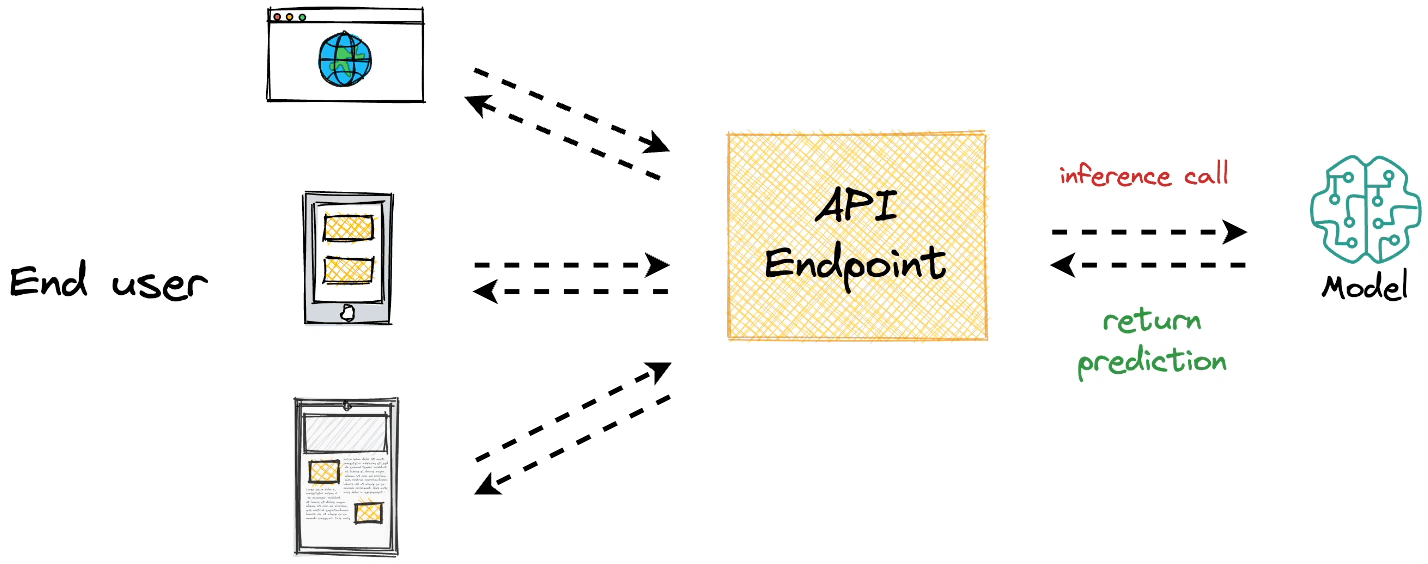
Let’s begin!

🗒️

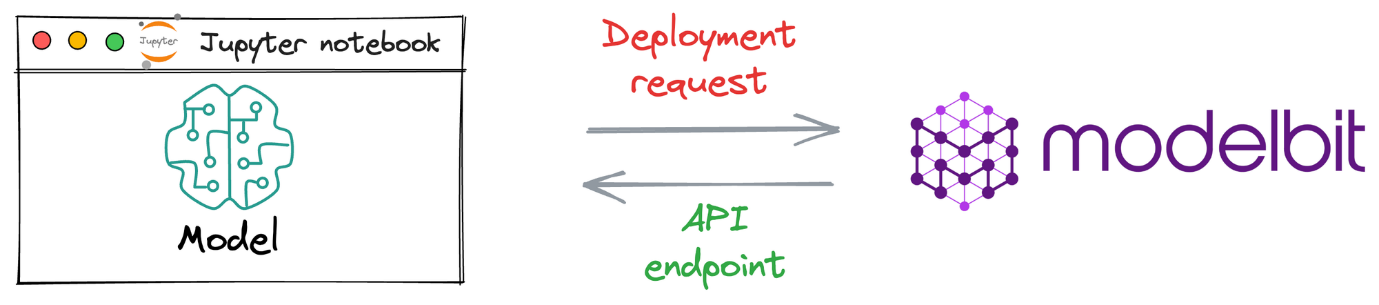
**Note:** Of course, Modelbit is a commercial product. However, this article is not influenced by any external motivation (or sponsorships) to portray them. I have personally found Modelbit’s deployment procedures to be immensely simple and unconvoluted. Everything demonstrated in this article can be done with just a Modelbit account for free.

**Basic model deployment with Modelbit**

The core objective behind model deployment is to obtain an API endpoint for our deployed model, which can be later used for inference purposes:



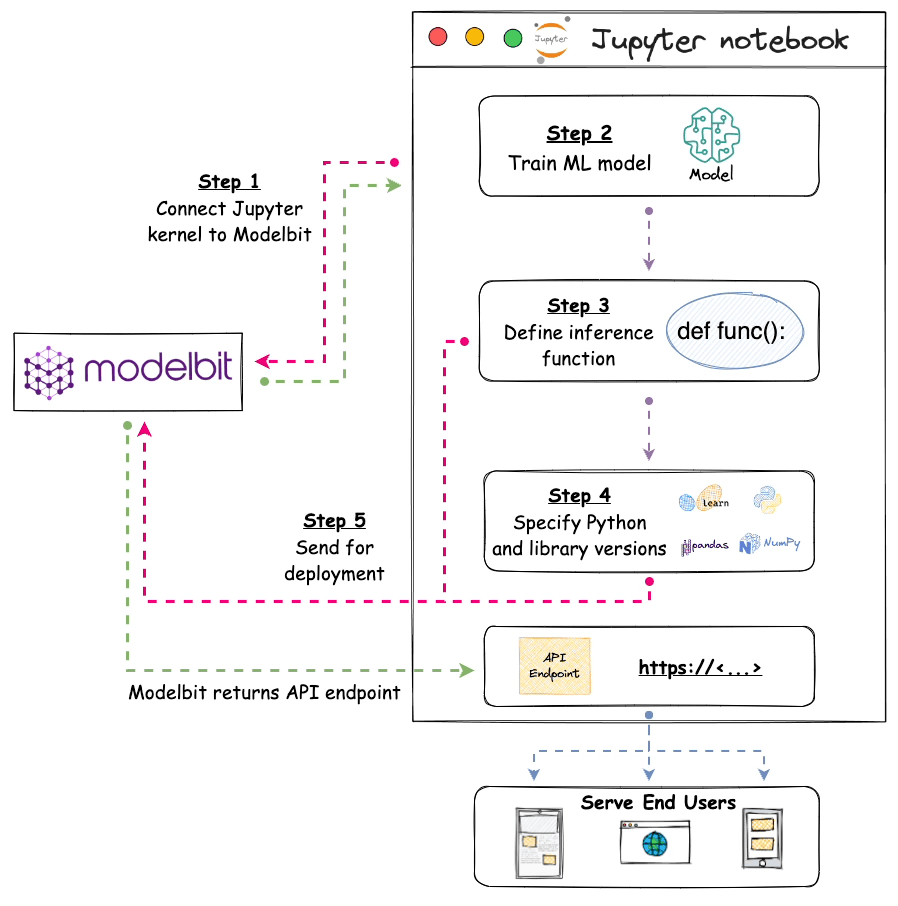
Modelbit lets us seamlessly deploy ML models directly from our Python notebooks (or Git, as we would see ahead in this article) and obtain a REST API.



**Process workflow**

Since Modelbit is a relatively new service, let’s understand the general workflow to generate an API endpoint when deploying a model with Modelbit.

The image below depicts the steps involved in deploying models with Modelbit:



* Step 1) We connect the Jupyter kernel to Modelbit.
* Step 2) Next, we train the ML model.
* Step 3) We define the inference function. Simply put, this function contains the code that will be executed at inference. Thus, it will be responsible for returning the prediction.
* Step 4) [OPTIONAL] Here, we specify the version of Python and other open-source libraries we used while training the model.
* Step 5) Lastly, we send it for deployment.

Once done, Modelbit returns the API endpoint, which we can integrate into any of the applications and serve end-users with.

Let’s implement this!