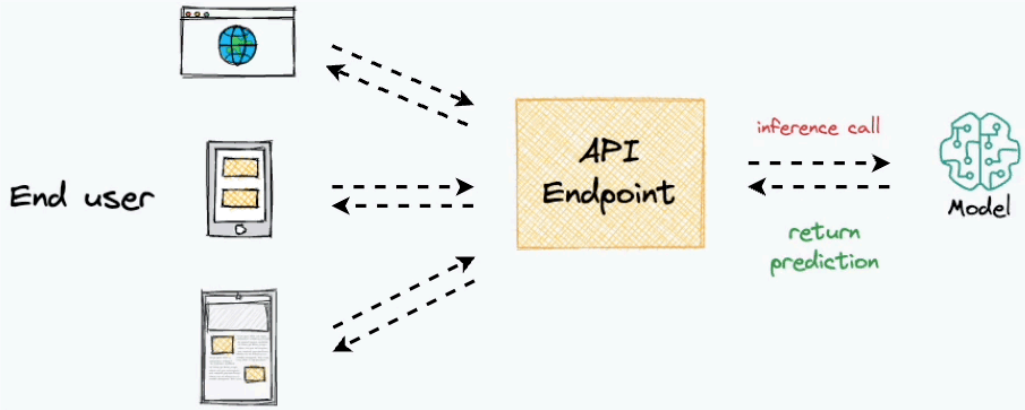
Deployment

Deploy ML Models from Jupyter Notebook

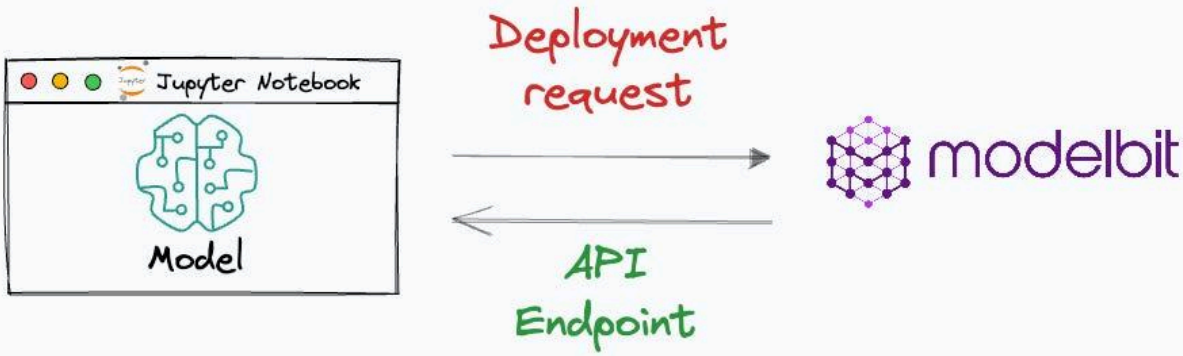
The core objective of model deployment is to obtain an API endpoint that can be used for inference purposes:



While this sounds simple, deployment is typically quite a tedious and time-consuming process. One must maintain environment files, configure various settings, ensure all dependencies are correctly installed, and many more.

So, in this chapter, I want to help you simplify this process. More specifically, we shall learn how to deploy any ML model right from a Jupyter Notebook in just three simple steps using the Modelbit API.

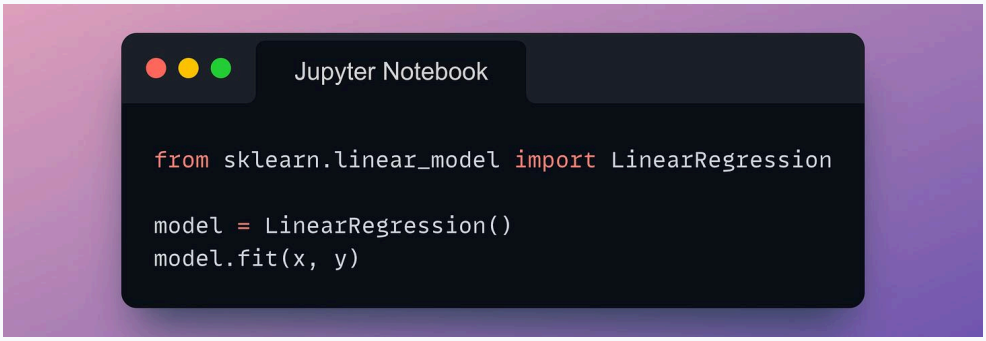
Modelbit lets us seamlessly deploy ML models directly from our Python notebooks (or git) to Snowflake, Redshift, and REST.



Deployment with Modelbit

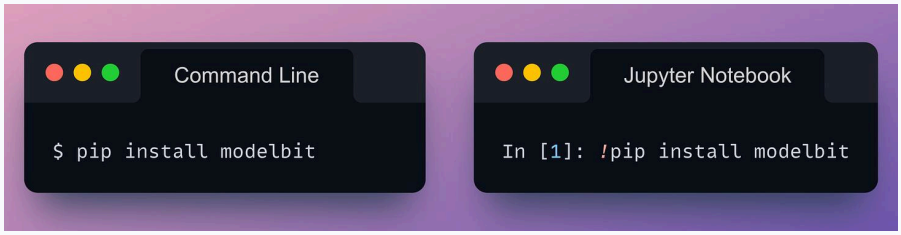
Assume we have already trained our model.

For simplicity, let's assume it to be a linear regression model trained using sklearn, but it can be any other model as well:

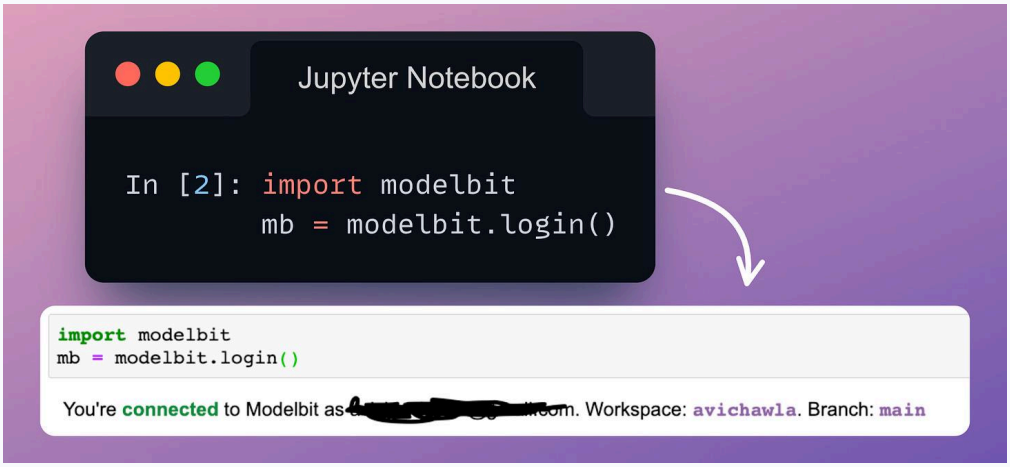


Let's see how we can deploy this model with Modelbit!

- First, we install the Modelbit package via pip:



- Next, we log in to Modelbit from our Jupyter Notebook (make sure you have created an account here: Modelbit)



- Finally, we deploy it, but here's an important point to note:

To deploy a model using Modelbit, we must define an inference function.

Simply put, this function contains the code that will be executed at inference. Thus, it will be responsible for returning the prediction.

```
def inference_function():

    # prediction code
```

A function that returns model predictions

We must specify the input parameters required by the model in this method. Also, we can name it anything we want.

For our linear regression case, the inference function can be as follows:

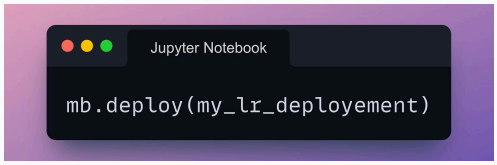
Jupyter Notebook

```
def my_lr_deployement(input_x):

    if isinstance(input_x, (int, float)):    #  check input type
        return model.predict([[input_x]])[0] # prediction

    else:
        return None
```

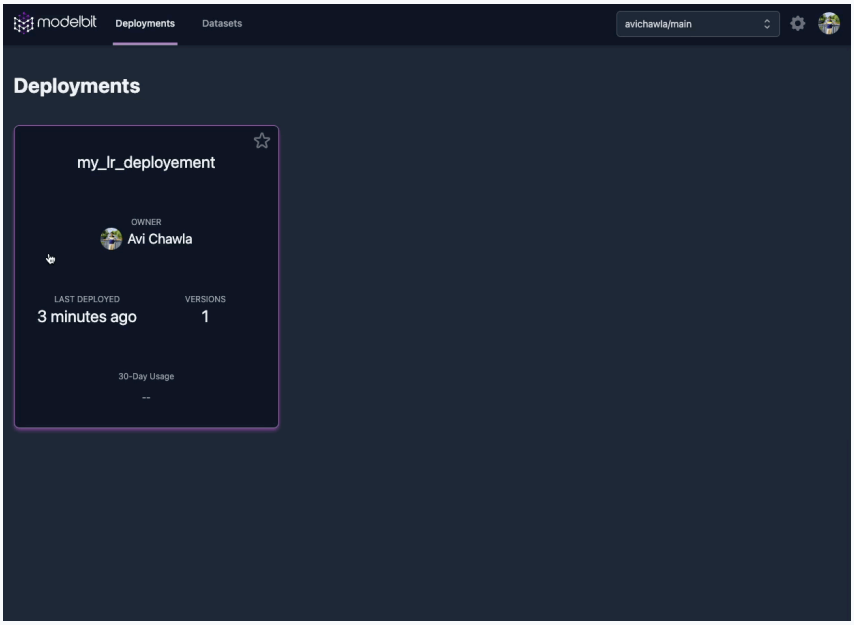
- We define a function `my_lr_deployment()`.
- Next, we specify the input of the model as a parameter of this method.
- We validate the input for its data type.
- Finally, we return the prediction.

One good thing about Modelbit is that every dependency of the function (the model object in this case) is pickled and sent to production automatically along

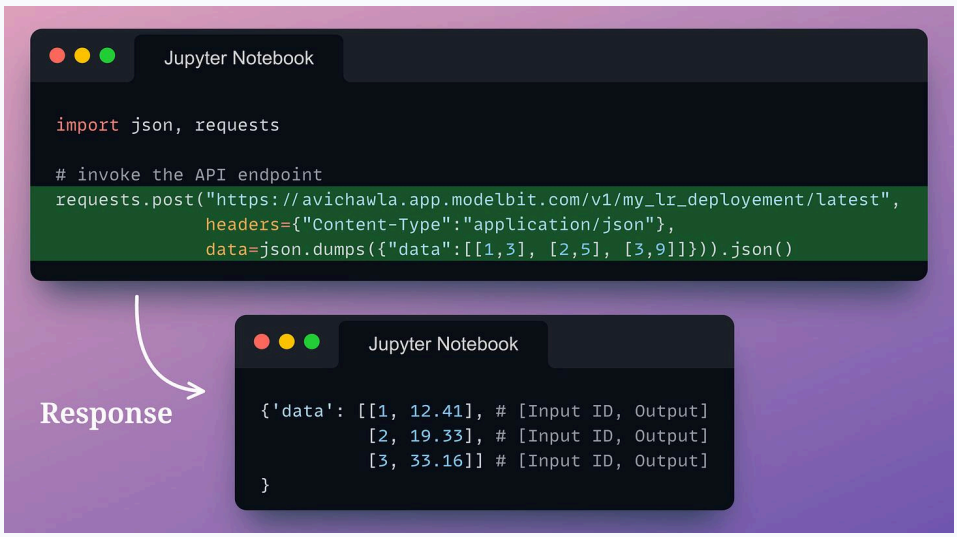
with the function. Thus, we can reference any object in this method. Once we have defined the function, we can proceed with deployment as follows:



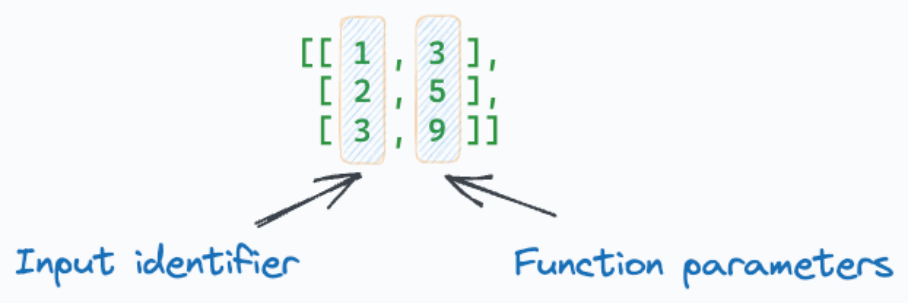
We have successfully deployed the model in three simple steps, that too, right from the Jupyter Notebook! Once our model has been successfully deployed, it will appear in our Modelbit dashboard.



As shown above, Modelbit provides an API endpoint. We can use it for inference purposes as follows:



In the above request, data passed to the endpoint is a list of lists.



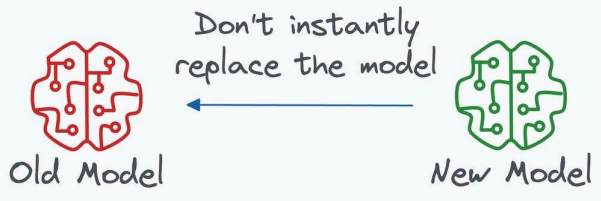
The first number in the list is the input ID. All entries following the ID in a list are the function parameters.

Lastly, we can also specify specific versions of the libraries or Python used while deploying our model. This is depicted below:

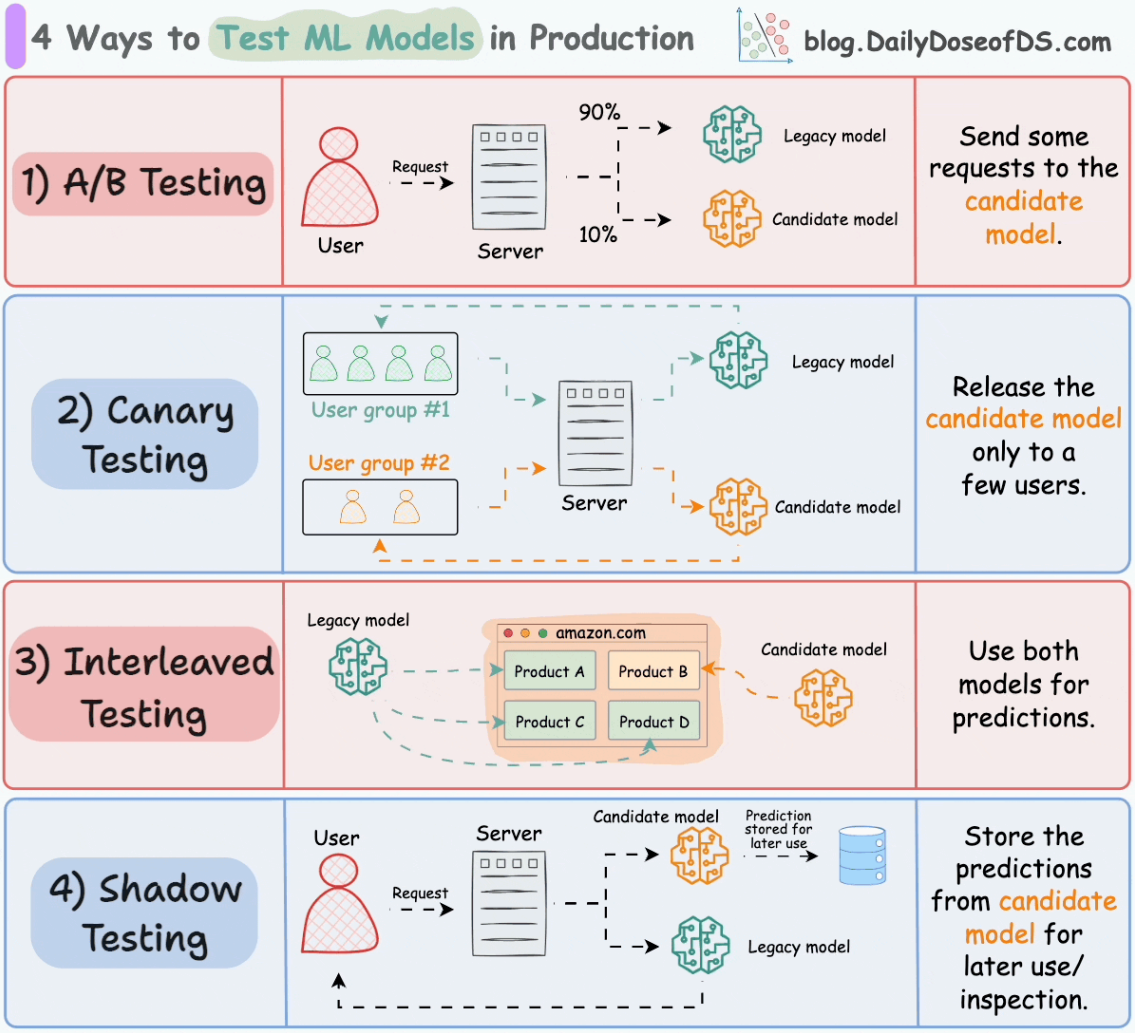
Isn't that cool, simple, and elegant over traditional deployment approaches?

4 Ways to Test ML Models in Production

Despite rigorously testing an ML model locally (on validation and test sets), it could be a terrible idea to instantly replace the previous model with a new model.

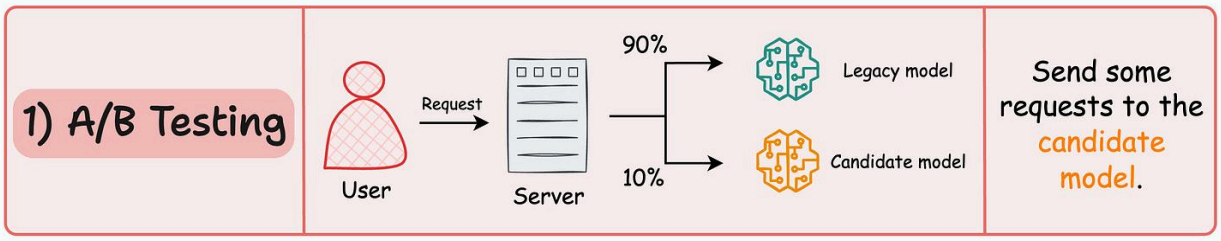


A more reliable strategy is to test the model in production (yes, on real-world incoming data). While this might sound risky, ML teams do it all the time, and it isn't that complicated. The following visual depicts 4 common strategies to do so:



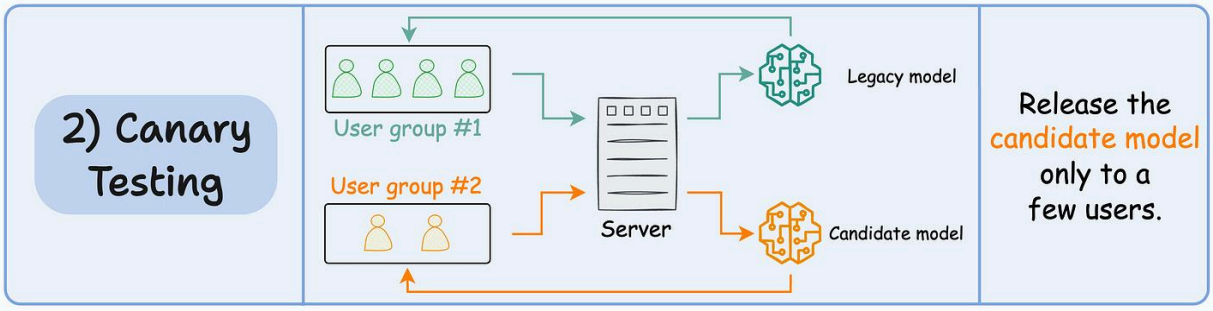
- The current model is called the legacy model.
- The new model is called the candidate model.

#1) A/B testing



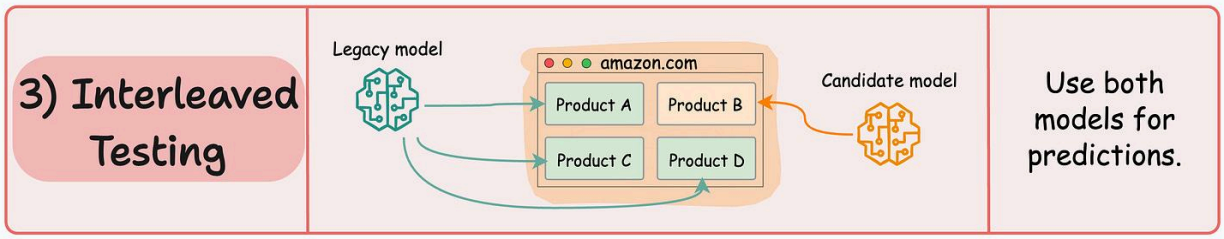
- Distribute the incoming requests non-uniformly between the legacy model and the candidate model.
- Intentionally limit the exposure of the candidate model to avoid any potential risks. Thus, the number of requests sent to the candidate model must be low.

#2) Canary testing



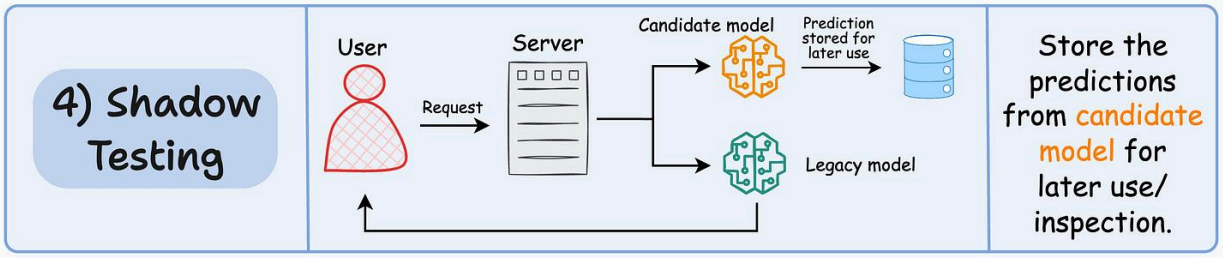
- In A/B testing, since traffic is randomly redirected to either model irrespective of the user, it can potentially affect all users.
- In canary testing, the candidate model is released to a small subset of users in production and gradually rolled out to more users.

#3) Interleaved testing



- This involves mixing the predictions of multiple models in the response.
- Consider Amazon's recommendation engine. In interleaved deployments, some product recommendations displayed on the homepage can come from the legacy model, while some can come from the candidate model.

#4) Shadow testing



- All of the above techniques affect some (or all) users.
- Shadow testing (or dark launches) lets us test a new model in a production environment without affecting the user experience.
- The candidate model is deployed alongside the existing legacy model and serves requests like the legacy model. However, the output is not sent back to the user. Instead, the output is logged for later use to benchmark its performance against the legacy model.
- We explicitly deploy the candidate model instead of testing offline because the production environment is difficult to replicate offline.

Shadow testing offers risk-free testing of the candidate model in a production environment.

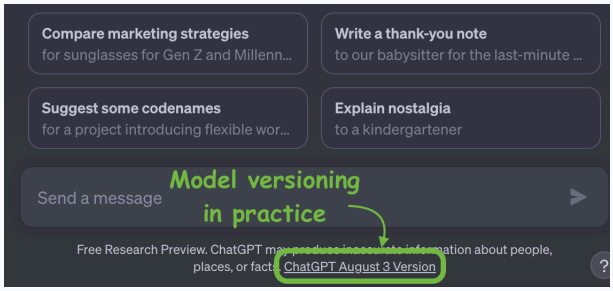
Version Controlling and Model Registry

Real-world ML deployment is never just about "deployment" — host the model somewhere, obtain an API endpoint, integrate it into the application, and you are done!

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

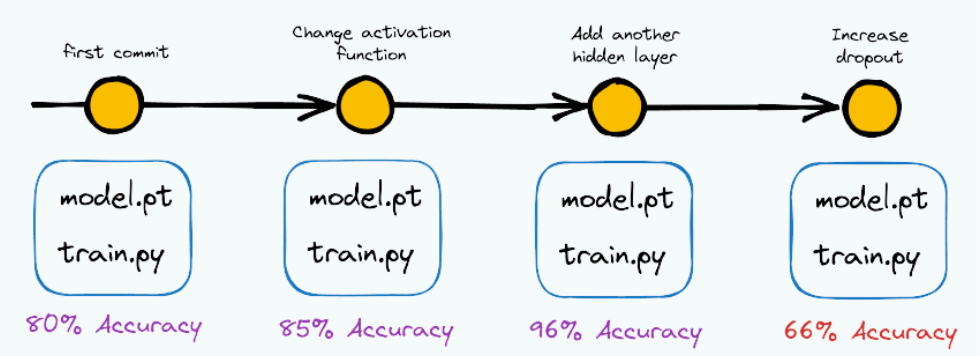
#1) Version control

To begin, it is immensely crucial to version control ML deployments. You may have noticed this while using ChatGPT, for instance.



But updating does not simply mean overwriting the previous version.

Instead, ML models are always version-controlled (using git tools).



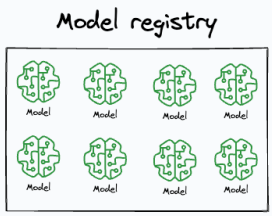
The advantages of version-controlling ML deployments are pretty obvious:

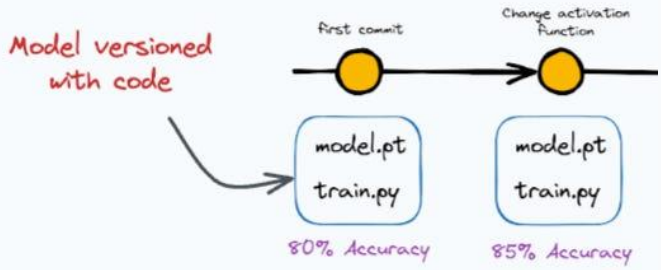
- In case of sudden mishaps post-deployment, we can instantly roll back to an older version.
- We can facilitate parallel development with branching, and many more.

#2) Model registry

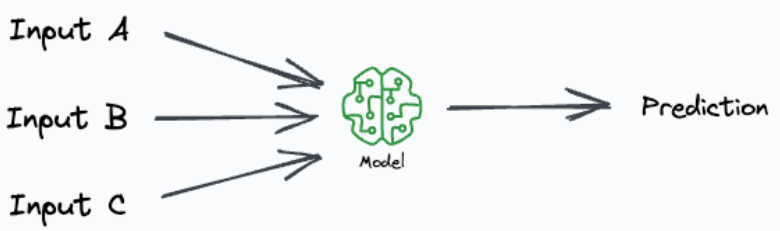
Another practical idea is to maintain a model registry for deployments. Let's understand what it is.

Simply put, a model registry can be considered repository of models. See, typically, we might be inclined to version our code and the ML model together:

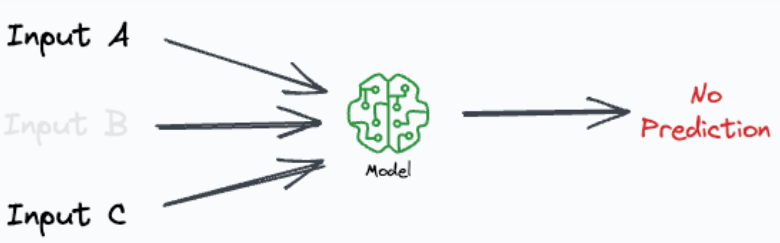




However, when we use a model registry, we version models separately from the code. Let me give you an intuitive example to understand this better. Imagine our deployed model takes three inputs to generate a prediction:



While writing the inference code, we overlooked that, at times, one of the inputs might be missing. We realized this by analyzing the model's logs.



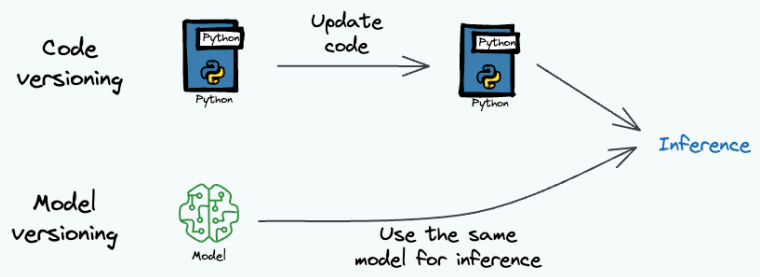
We may want to fix this quickly (at least for a while) before we decide on the next steps more concretely. Thus, we may decide to update the inference code by assigning a dummy value for the missing input.

This will allow the model to still process the incoming request.

Let me ask you a question: "Did we update the model?"

No, right?

Here, we only need to update the inference code. The model will remain the same.



But if we were to version the model and code together, it would lead to a redundant model and take up extra space.

However, by maintaining a model registry:

- We can only update the inference code.
- Avoid pushing a new (yet unwanted) model to deployment.

This makes intuitive sense as well, doesn't it?