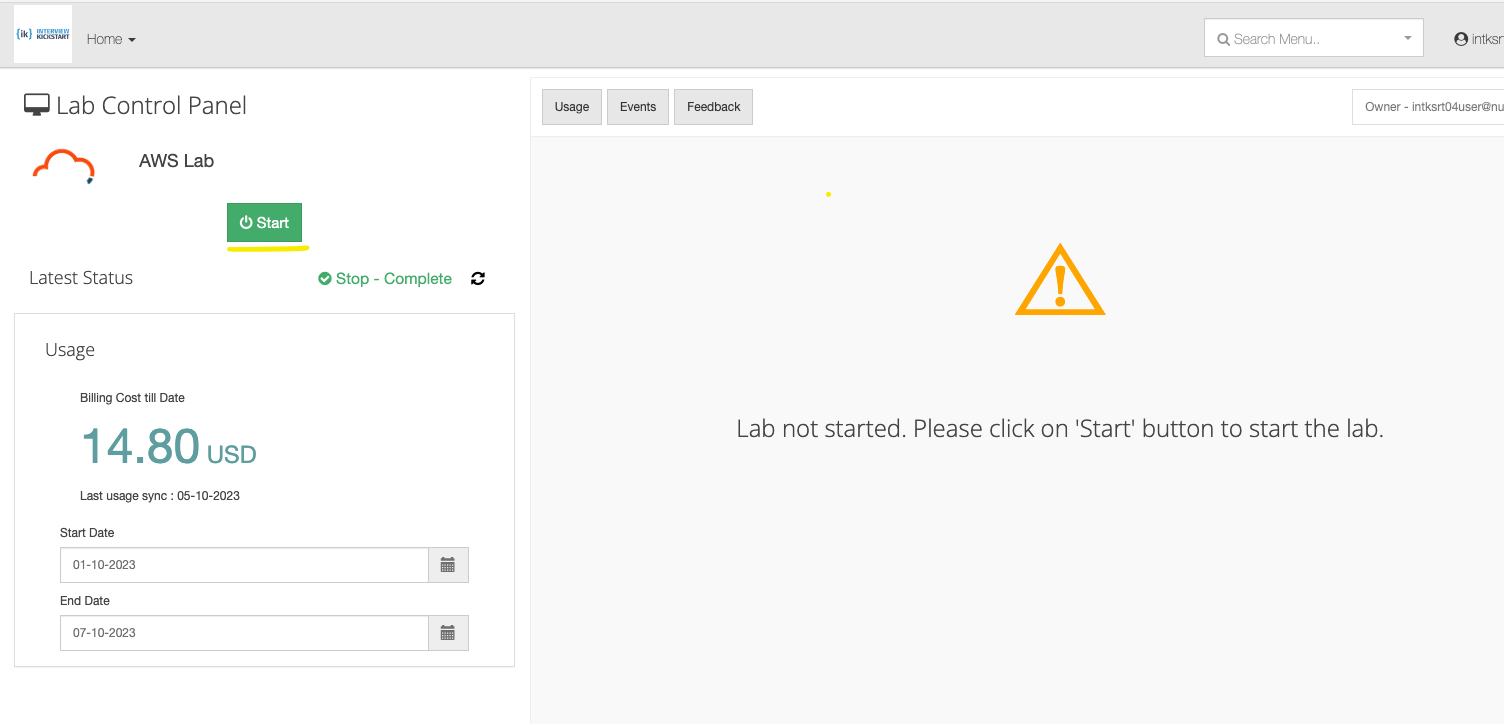
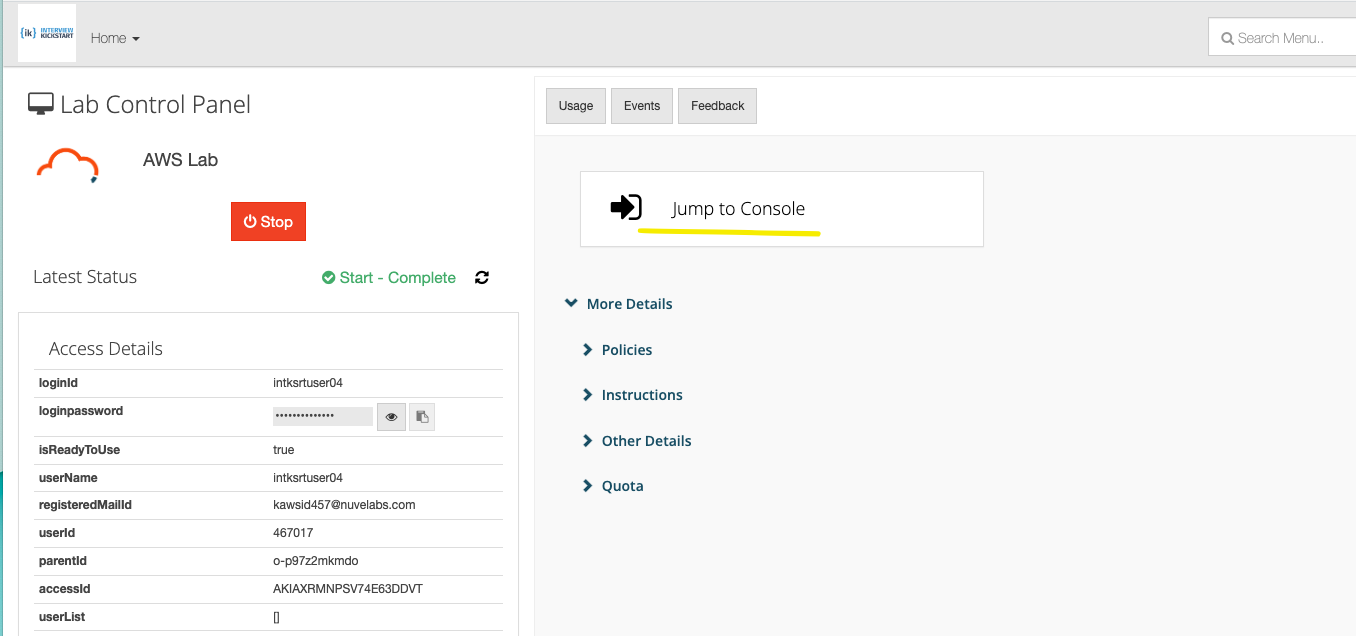
# Model Training Demo Setup and Delivery Instructions

## Creating and Setting up the VM:

1. Login to your Nuvepro account and Start the Nuvepro Lab.



1. After the Lab session starts, launch the AWS console.



1. Now we need to create an EC2 instance in the AWS console. Search for the EC2 service and create a VM with the following specifications

**Operating System** - *Ubuntu*

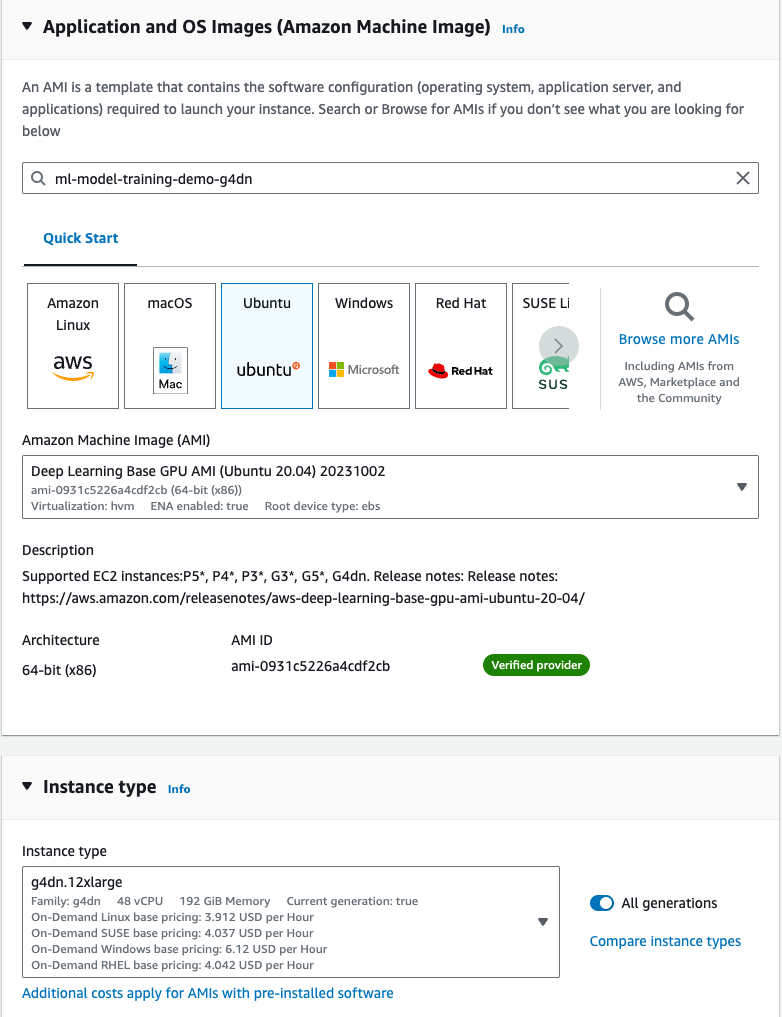
**Amazon Machine Image (AMI)** - *Deep Learning Base GPU AMI*

**Instance Type** - *g4dn.12xlarge*

**Storage -** *100 GiB gp3*

All other settings can remain the same.

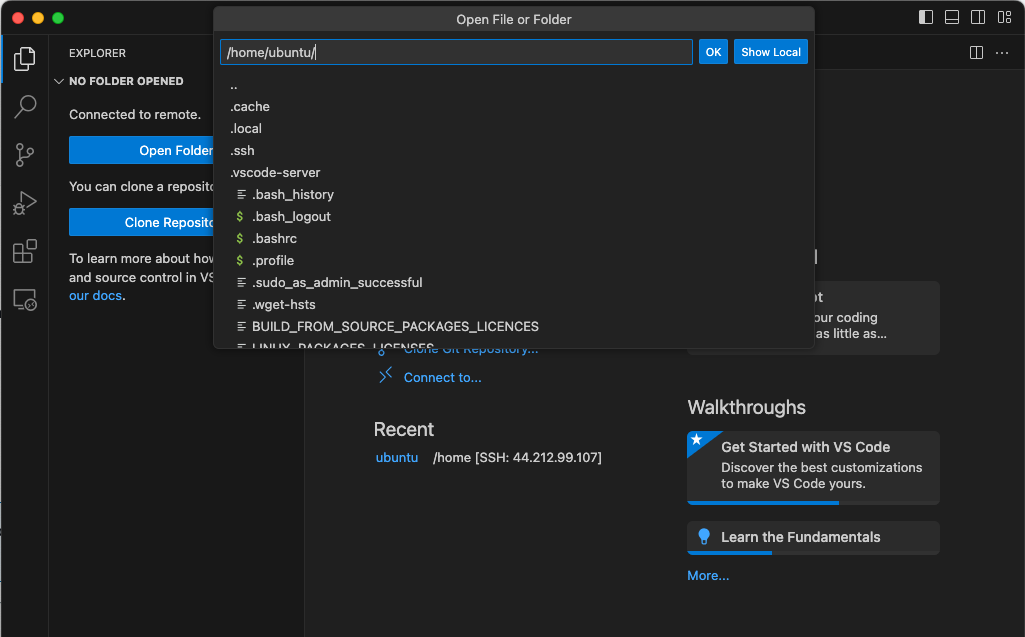
Create the instance with a Key Pair Login Mechanism.



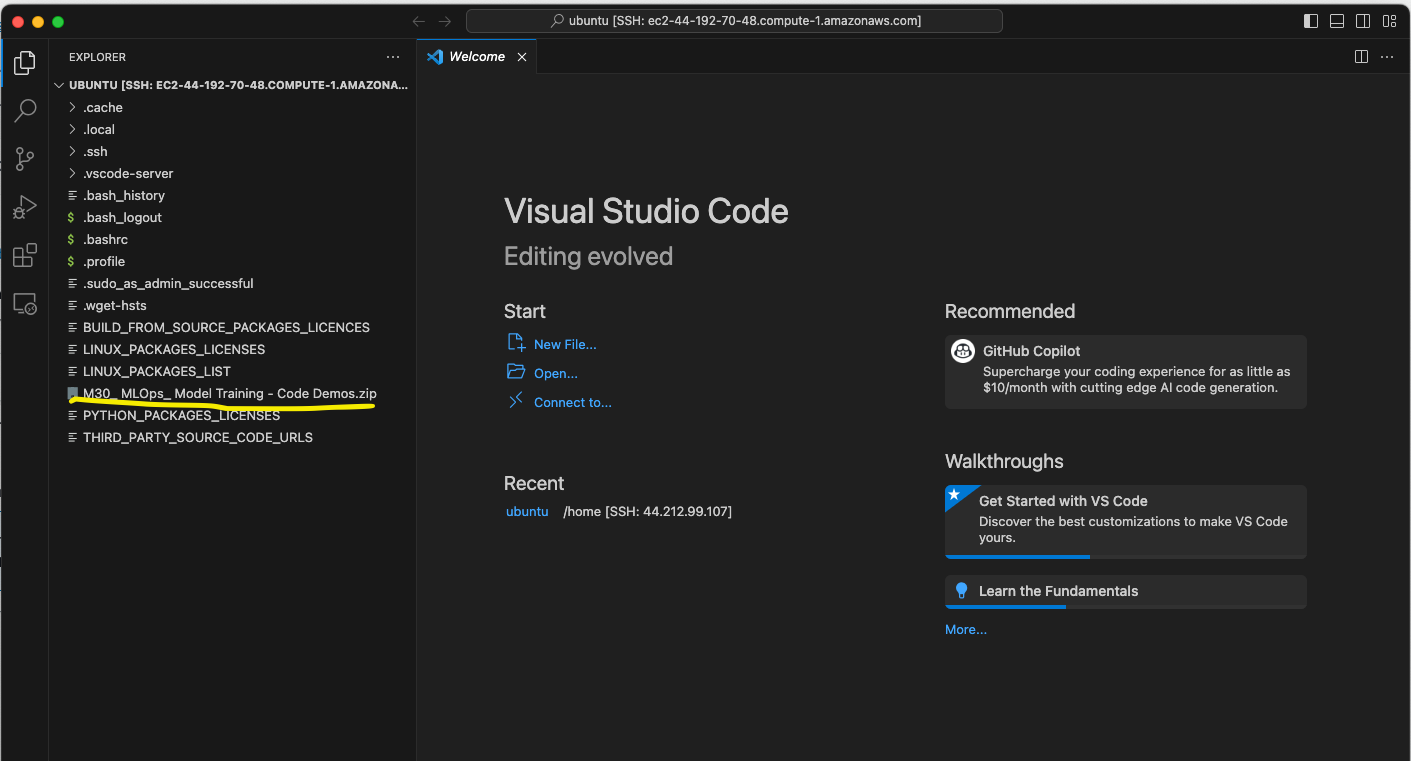
5. Modify the permissions on the pem file just downloaded using the following command. The command is valid for Linux/Mac Operating System. If you have Windows, follow the instructions [here](https://superuser.com/questions/1296024/windows-ssh-permissions-for-private-key-are-too-open).

| chmod 400 <path of the downloaded key file> |
| --- |

6. Setup VS Code to connect to and execute code on this remote machine using the instructions given here: <https://code.visualstudio.com/docs/remote/ssh>. After the setup, connect to the remote machine. Click on Open Folder and open the Home Directory of the VM.

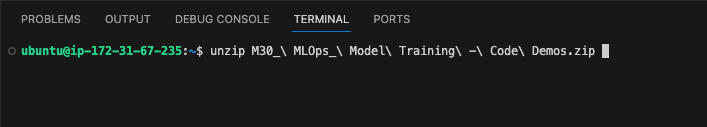


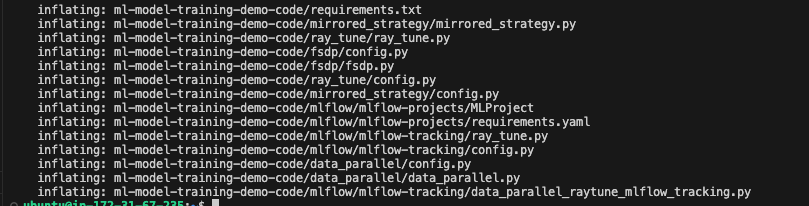
7. Now Drag and Drop the Demo code Zip File from your File Explorer to the VSCode’s File Explorer. This would copy the code file on the remote server.

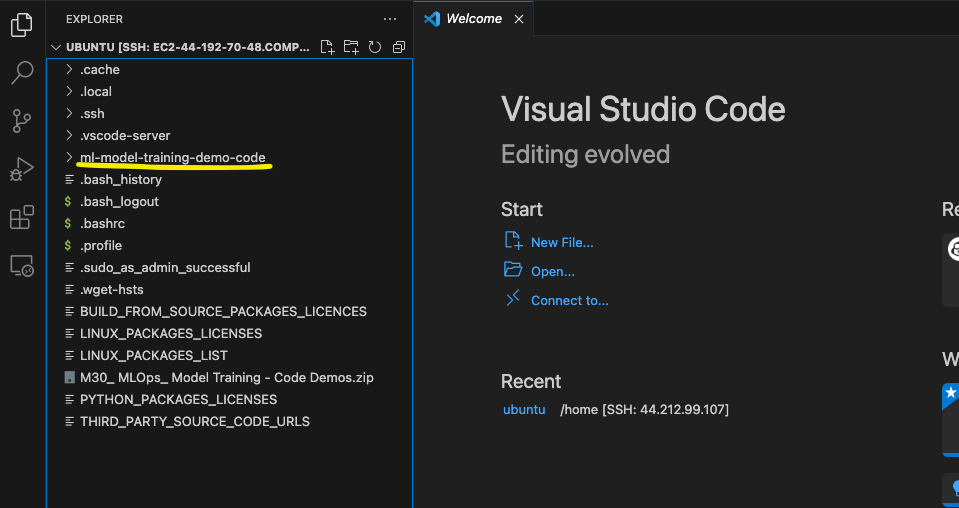


8. Launch a new Terminal in VS Code (this would open a remote shell terminal) and unzip the code file. The unzipped code directory should be visible in the VSCode’s Explorer after this.

| unzip M30\_\ MLOps\_\ Model\ Training\ -\ Code\ Demos.zip |
| --- |

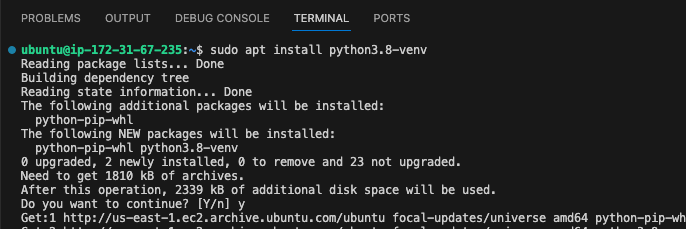


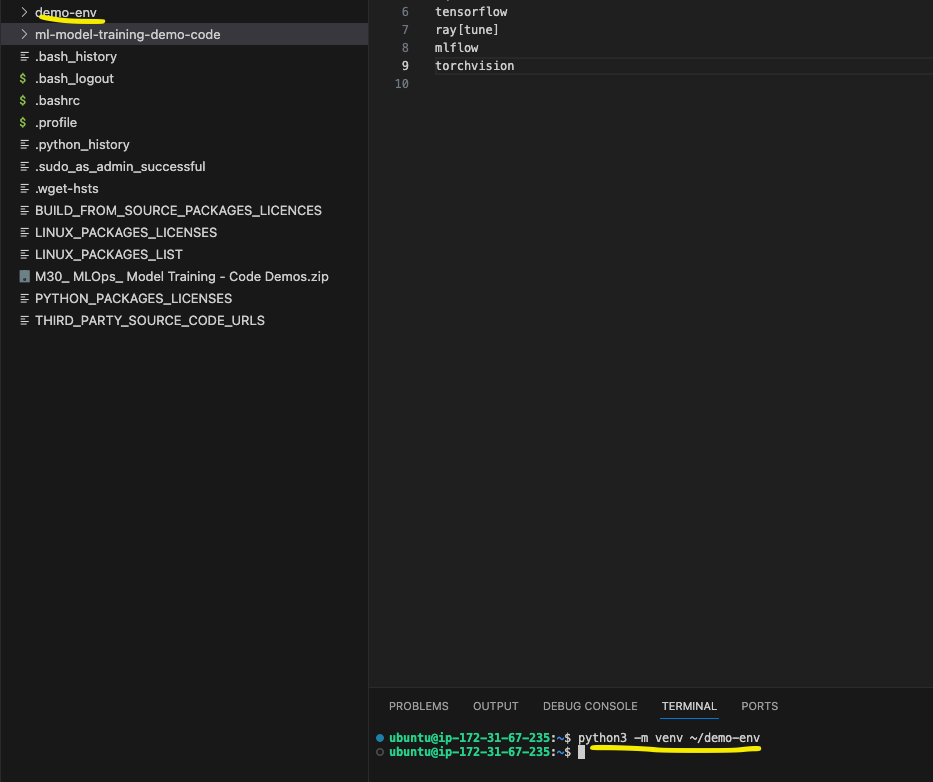




9. In the same terminal, install virtual-env package from apt and then create a virtual environment for the demo dependencies.

| sudo apt install python3.8-venv python3 -m venv ~/demo-env |
| --- |





10. Activate the virtual env and install the dependencies from the requirements.txt file

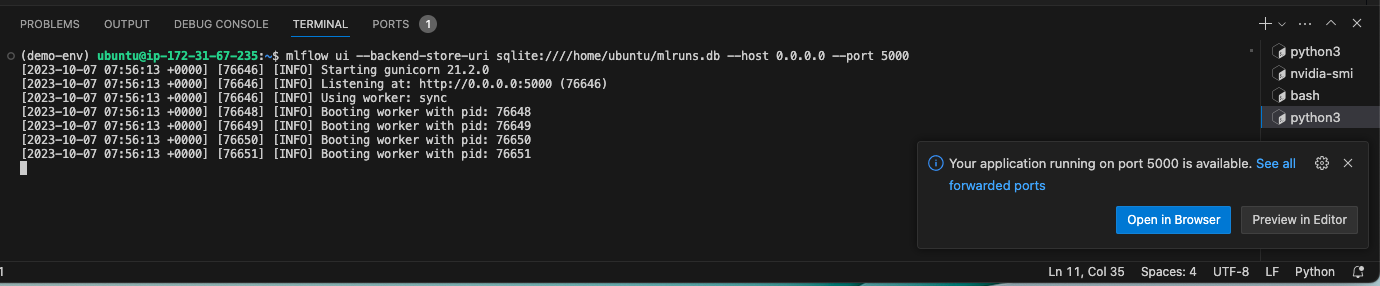
| source ~/demo-env/bin/activate pip3 install -r ~/ml-model-training-demo-code/requirements.txt |
| --- |

11. Verify the installation using the following commands

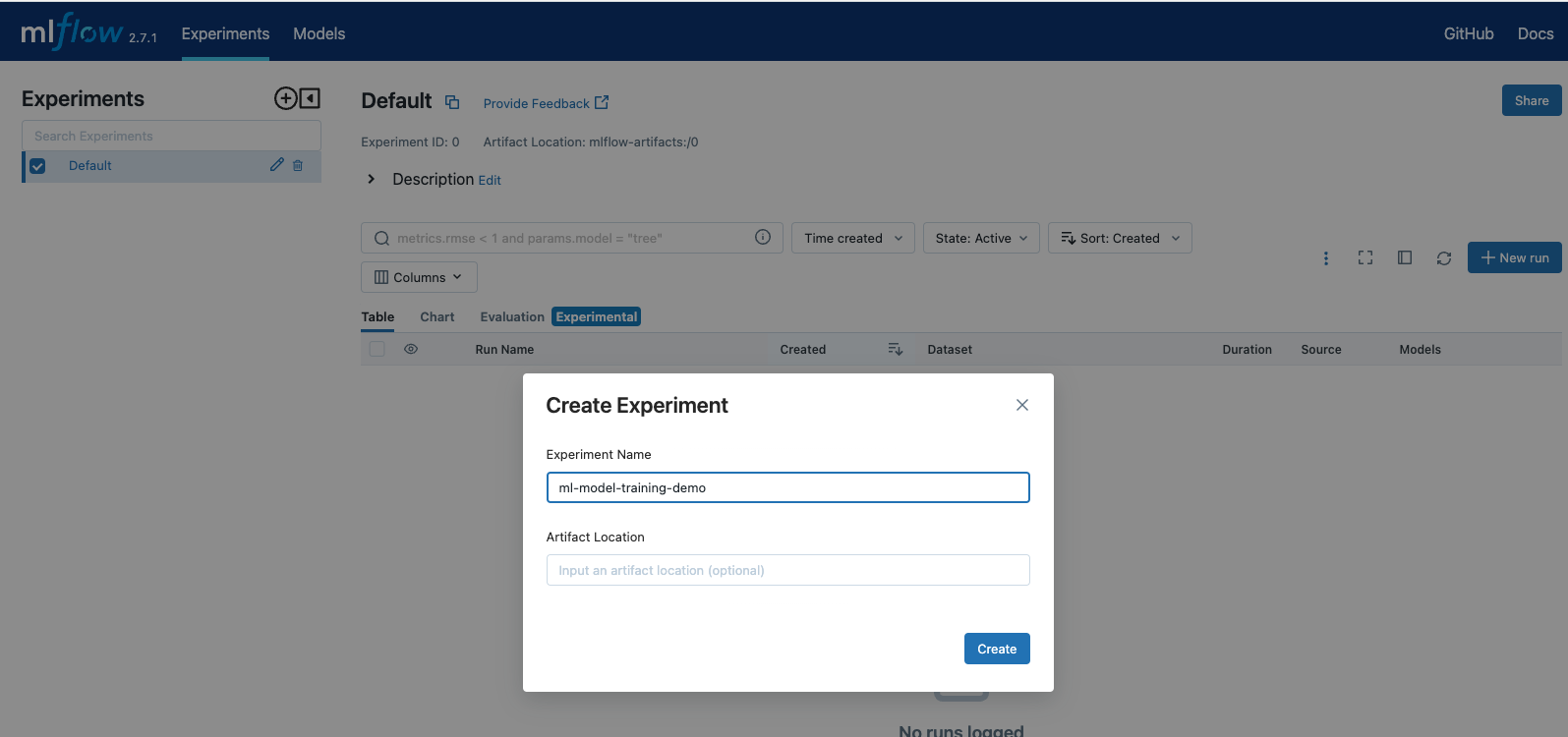
| nvidia-smi  python3 -c "import torch; print(torch.cuda.device\_count())" python3 -c "import tensorflow as tf  print(tf.config.list\_physical\_devices('GPU'))" |
| --- |

12. Run MFlow Tracking using the following command in a separate terminal. As soon as you start MLflow, VSCode will create a port forwarding tunnel, so you can access the remote MLflow server on your local machine.

| mlflow ui --backend-store-uri sqlite:////home/ubuntu/mlruns.db --host 0.0.0.0 --port 5000 |
| --- |



13. Access MLflow on the localhost and create a new experiment



14. Now install pyenv using the following command. Pyenv is required by Mflow for virtual environment management

| curl https://pyenv.run | bash |
| --- |

Add pyenv to the Path in bashrc file

| echo "export PATH=$PATH:$HOME/.pyenv/bin" >> ~/.bashrc |
| --- |

The Setup is now complete and we can move forward with the demonstrations

## Delivering the Demonstrations

All the demonstrations can/should be executed using the Terminal in VSCode.

### 1. Pytorch’s Data Parallel

This Demo uses Pytorch’s Data Parallel to train a large model on multiple GPUs on a single machine. The intention is to estimate the difference in the size of the batch we can train on, with and without using data parallel.

#### Demo Directory

| /home/ubuntu/ml-model-training-demo-code/data\_parallel |
| --- |

#### Flow of Demonstration

1. Walk through the code.
2. Start by setting (in config.py) do\_data\_parallel = False and per\_device\_batch\_size to 6. You will notice that the training proceeds with ease but the total effective batch size is only 6, since there is no parallelism.
3. Notice the time taken per epoch by the last training run and set that aside as a baseline.
4. Now Enable Data Parallel by setting do\_data\_parallel = True and Rerun the training. This will enable an effective batch size of 4 x 6 = 24.
5. You will notice that the training proceeds without any issue. The amount of time taken to process the epoch has reduced to almost 1/4th of what was taken without data parallel, since the effective batch size is now 4 x 6 = 24
6. Now try to train the model with per\_device\_batch\_size = 24 and set do\_data\_parallel = False. The training will run out of memory, since the GPU cannot handle such a large batch.
7. Highlight the observations - a) We were able to train the model with a larger batch size using Data Parallel which otherwise wasn't possible with a single device. b) Since the batch size quadrupled, the training time reduced by almost 4 times
8. Make sure to mention that the focus in the demo is not on modelling or model type or the dataset or statistical performance. The focus is on Training Parallelism.
9. Now execute another round of training with per\_device\_batch\_size set to 1. Notice the amount of memory per device consumed and set this aside as a Baseline for FSDP Demo.

#### Execution Command(s)

| python3 /home/ubuntu/ml-model-training-demo-code/data\_parallel/data\_parallel.py |
| --- |

### 2. Pytorch’s FSDP

The demo here used Pytorch’s Full Sharded Data Parallel, which shards the model parameters, gradients and optimizer states across GPUs. The intention is to compare the amount of memory used by Data Paralel and FSDP when using the same model. In the last demo, we executed a training with batch size = 1 using Data Parallel, therefore we negated the effect of a batched activations. We will run the same demo using FSDP to compare the amount of memory consumed.

#### Demo Directory

| /home/ubuntu/ml-model-training-demo-code/fsdp |
| --- |

#### Flow of Demonstration

1. Start by setting per\_device\_batch\_size = 1. Notice the amount of memory consumed by the training.
2. Compare this memory consumption with that of Data Parallel with batch size = 1
3. Now run through the following calculations
4. BERT Large has around 340 million parameters.
5. Amount of Memory taken by the model:
   1. Memory consumed by parameters = 340 M x 4 bytes = 1.36 GB
   2. Memory consumed by gradients = 340 M x 4 bytes = 1.36 GB
   3. Memory taken by optimizer states = 340M x 8 bytes = 2.72 GB
   4. Total Memory consumed by the model = 1.36 + 1.36 + 2.72 = 5.44 GB
6. Additional Memory is consumed by activations (for batch size = 1) which is not accounted for here.
7. For Data Parallel, these components are not sharded. Therefore the effective memory is as per the calculations here. The run would have showed close to 6.73 GB memory
8. For FSDP, we can divide the consumed memory by 4 (num of devices). 5.44 / 4 = 1.36 GB. Therefore effective memory consumed by FSDP should be 4.08 GB (5.44 GB - 1.36 GB) less than Data Parallel.
9. The run should indicate around 2.89 GB memory consumed. 6.73 - 2.89 = 3.84, which is close to our calculation above.
10. Important to mention - Since the consumed memory is significantly low in FSDP, we can train our model with a batch size even larger than what Data Parallel allows. More importantly, since the entire model isn’t replicated on each device, we can train a model even larger than what Data Parallel allows.

#### Execution Command(s)

| torchrun --nnodes 1 --nproc\_per\_node 4 home/ubuntu/ml-model-training-demo-code/fsdp/fsdp.py |
| --- |

### 3. Tensorflow’s Mirrored Strategy

This is exactly similar to Pytorch’s Data Parallel. We use the same model (the tensorflow version of sequence classifier) and the same dataset. So just do a code walkthrough here. You can choose to ignore the execution since data sharding takes a few minutes in tensorflow.

#### Demo Directory

| /home/ubuntu/ml-model-training-demo-code/mirrored-strategy |
| --- |

#### Execution Command(s)

| python3 /home/ubuntu/ml-model-training-demo-code/mirrored\_strategy/mirrored\_strategy.py |
| --- |

### 4. Scikit Opt

This demo is meant to showcase the capability of scikit opt for Bayesian Optimization. The code for this is present in a notebook. Since no GPU is needed here, this should be run using colab.

Run through the code available here (Make a choice whether to execute or not): <https://drive.google.com/file/d/10uYJ42jOmlFWQhwOpvETO8301CeZaZ88/view?usp=drive_link>

### 4. Ray Tune

This demonstration uses Ray Tune to parallelize Hyperparameter Optimization runs on multiple GPUs on the node. This utilizes the same data parallel training as Demo 1. In the interest of time, we will just do a Grid Search on just the batch size. For the same reason, we use a smaller model. These configurations are available in config.py

#### Demo Directory

| /home/ubuntu/ml-model-training-demo-code/ray\_tune |
| --- |

#### Flow of Demonstration

1. Simply run the given code using the command specified below. The code will execute two parallel runs - with the configured batch sizes, and will display the best configuration.
2. Explain the impact of modifying num\_gpu and num\_samples configuration.

#### Execution Command(s)

| PYTHONPATH=/home/ubuntu/ml-model-training-demo-code/data\_parallel python3 /home/ubuntu/ml-model-training-demo-code/ray\_tune/ray\_tune.py |
| --- |

### 5. MLflow Tracking

This demonstration is a follow up of the last demonstration. Along with performing distributed hyperparameter tuning with raytune, we also log each of the tuning experiments with MLflow tracking. The code used is exactly the same, but has additional mlflow loggers embedded.

#### Demo Directory

| /home/ubuntu/ml-model-training-demo-code/mlflow/mlflow-tracking |
| --- |

#### Flow of Demonstration

1. Execute the raytune code using the given command. This would invoke two parallel hyperparameter optimization runs and log the metrics, parameters, models and predictions to MLflow.
2. Navigate to MLflow UI, and demo the various components in the logged run - metrics, parameters, models, charts, and evaluation (for the logged predictions)
3. Also highlight the grouping of runs under a parent run as shown below.

#### 

#### 

#### 

#### 

#### 

#### Execution Command(s)

| python3 /home/ubuntu/ml-model-training-demo-code/mlflow/mlflow-tracking/ray\_tune.py |
| --- |

### 6. MLflow Projects

This demo executes the same code as MLflow Tracking, but the execution is done through MLflow Projects. The intention is to highlight how MLflow Projects can bundle the training project and dependencies in the form of a package, and help execute training projects in a standardized manner.

#### Demo Directory

| /home/ubuntu/ml-model-training-demo-code/mlflow/mlflow-projects |
| --- |

#### Flow of Demonstration

1. Walkthrough the MLProject.yaml file. Highlight the use of entrypoints and python\_env as an environment isolation mechanism.
2. Highlight the use of Python 3.8 in the base virtual environment, while the mention of Python 3.11 in this file. This showcases environment isolation. Also, this would allow a data scientist to execute the training without worrying about the base environment and the dependencies it needs, since everything is packaged up as one.
3. Execute the code using the given command. MLflow projects execution requires MLFLOW\_TRACKING\_URI and MLFLOW\_EXPERIMENT\_ID environment variables explicitly set.

#### Execution Command(s)

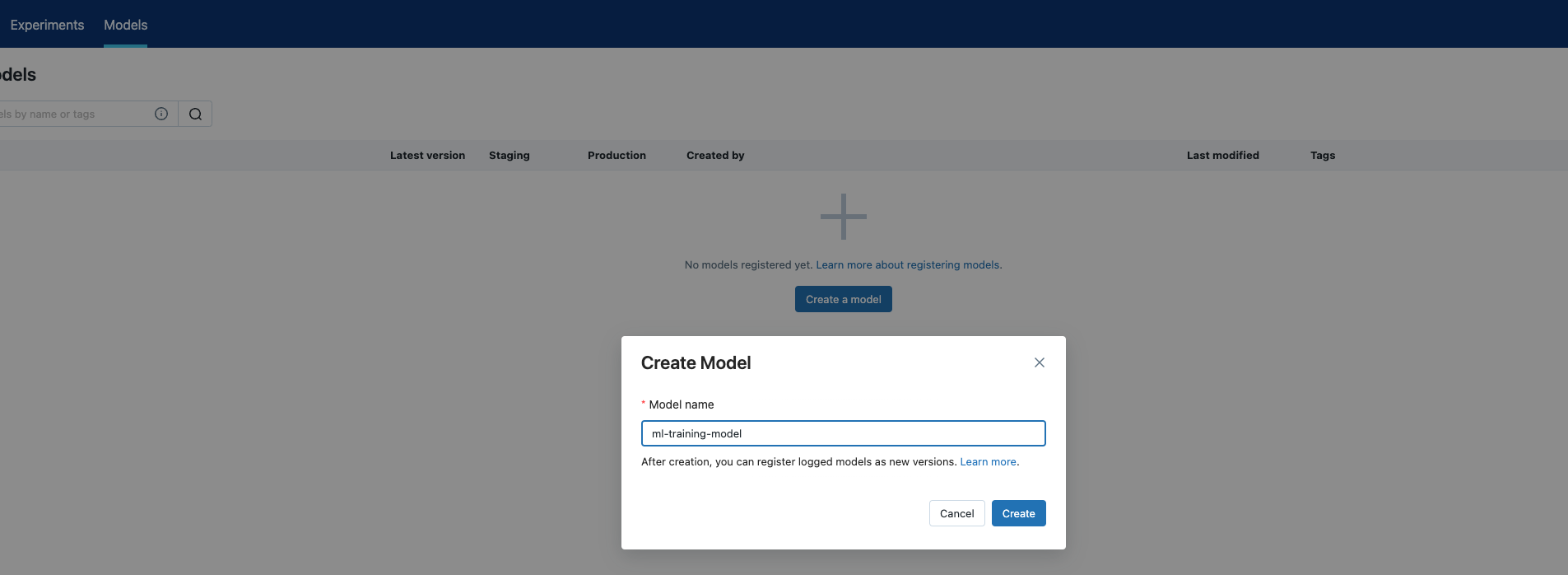
| export MLFLOW\_TRACKING\_URI="http://localhost:5000" export MLFLOW\_EXPERIMENT\_ID='1' mlflow run /home/ubuntu/ml-model-training-demo-code/mlflow/mlflow-projects -e train |
| --- |

### 7. MLflow Registry

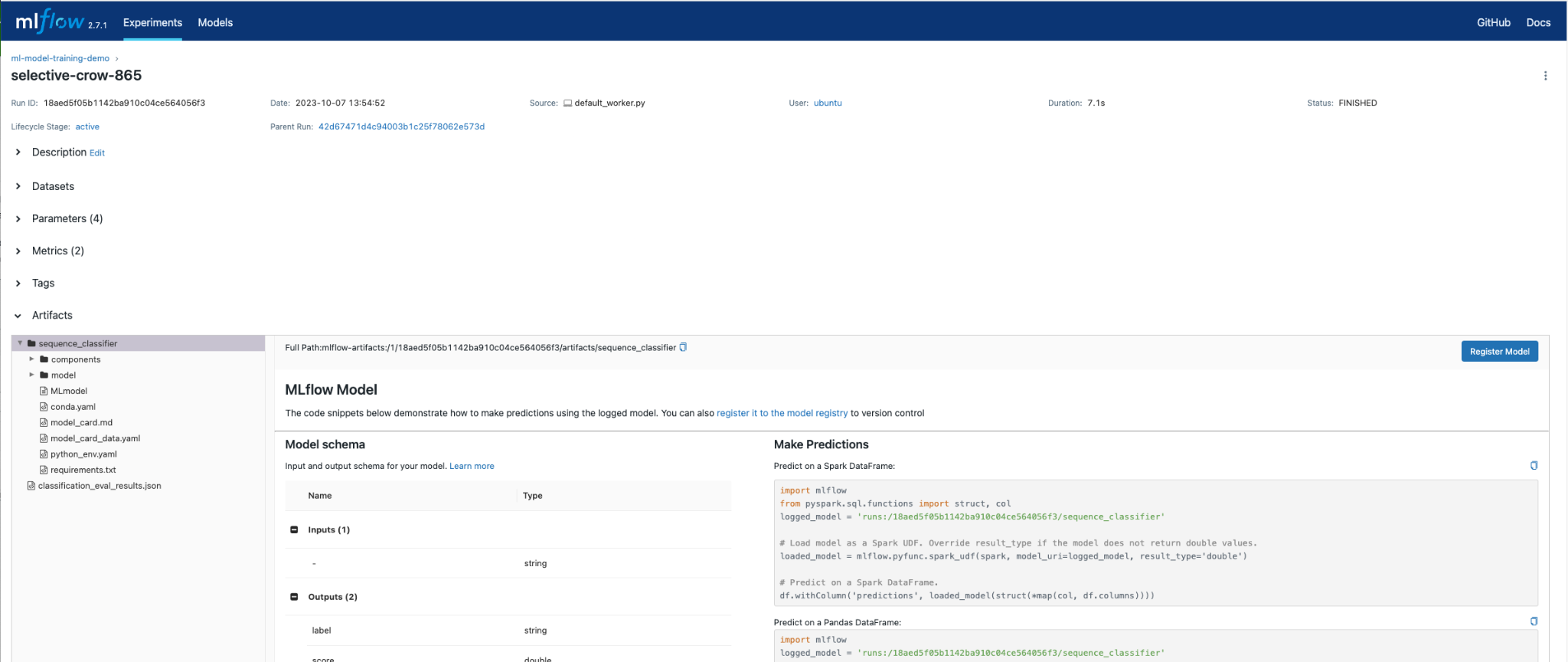
The purpose of the demo is to register the logged models in the previous demo in MLflow registry and transition their stage. We will avoid using MLflow APIs in this and demonstrate using the UI (using the APIs will be given as an assignment).

#### Flow of Demonstration

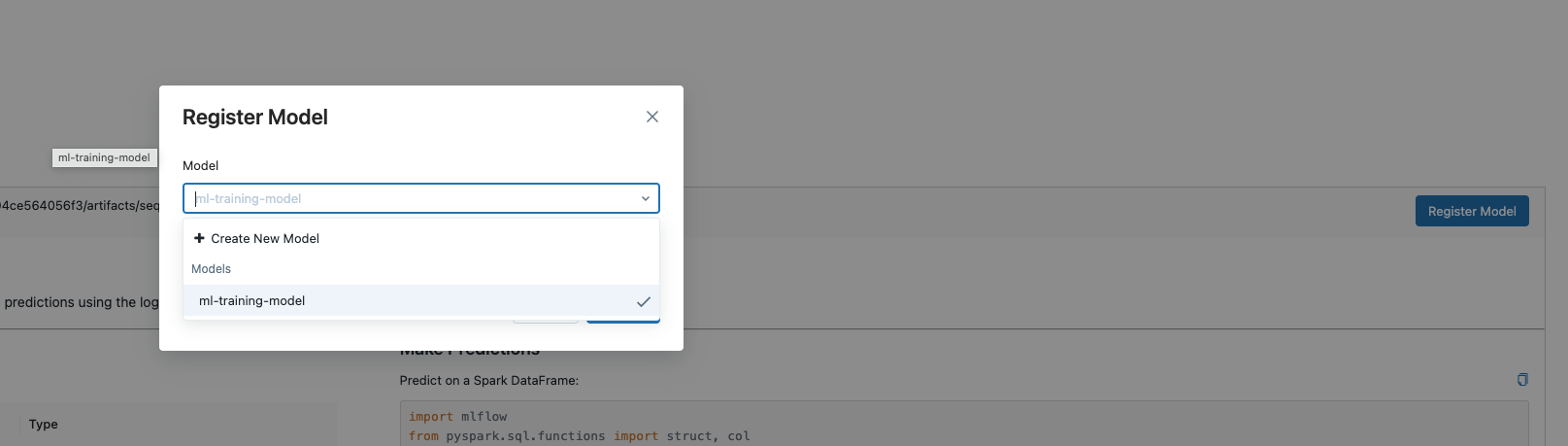
1. Navigate to the MLflow Tracking Server.
2. Click on the “Models” tab on top-left. Now click on “Create a Model, and provide the name for the new model in the registry, and click on Create. This will create a new Model in the registry.



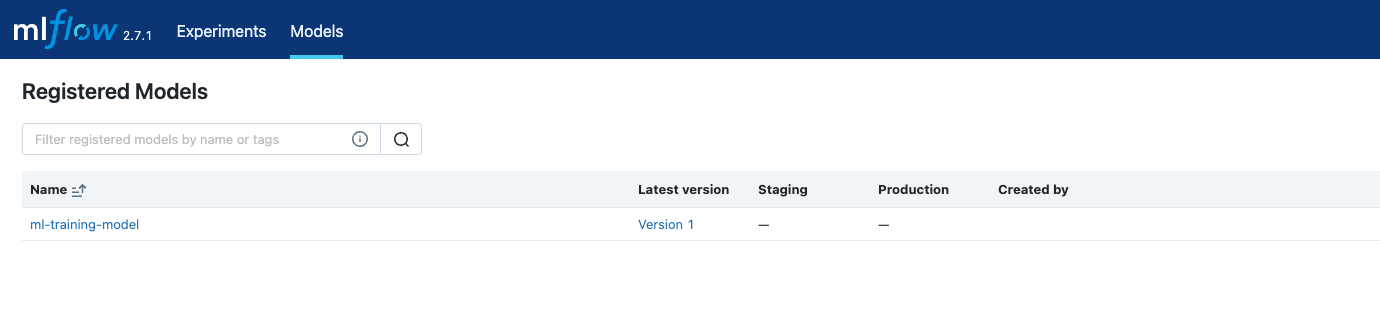
1. Now click on the “Experiments” tab on top-left, navigate to the last demoed run in the tracking server, and open the logged models.



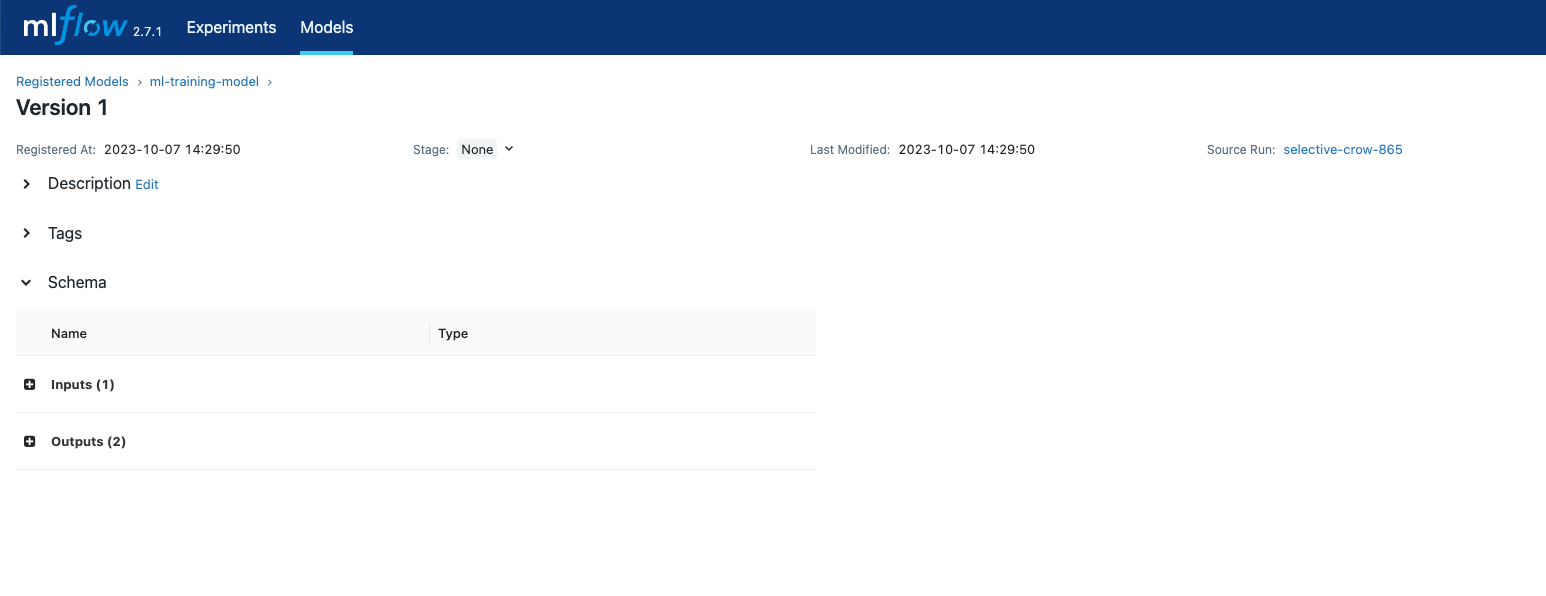
1. Click on the Register Model button, and select the model created in step 2.



1. Navigate back to “Models” through Models tab



1. Click on “Version 1”. This will open the model we just registered.



1. Transition the Stage from ‘None’ to ‘Production’.

