

ML Model into Production using MLFlow, DVC, Evidently, Grafana and Prometheus Waiter Tips Prediction

A Case from Tipping the Waiter: Use of Algorithms, Tools, and Technologies to Turn an Idea into a Beneficial Application,

Part-1: Training with XGBoost, Hyperparameter Optimization, and Experiment Tracking

Launching an ML model in the production environment is much more challenging than developing it, as an MLOps journey should involve several compatible tools and considerations.

We will discuss one of many different ways to achieve how to productionize an ML model. However, all cases have more or less in common regarding optimization, automation, workflow management, drift check, experiment tracking, visualization, testing, and cloud deployment.

We'll look into this topic in a series of articles. While moving from one scene to another, I'll also try to unfold the mistakes I made and the problems I faced.

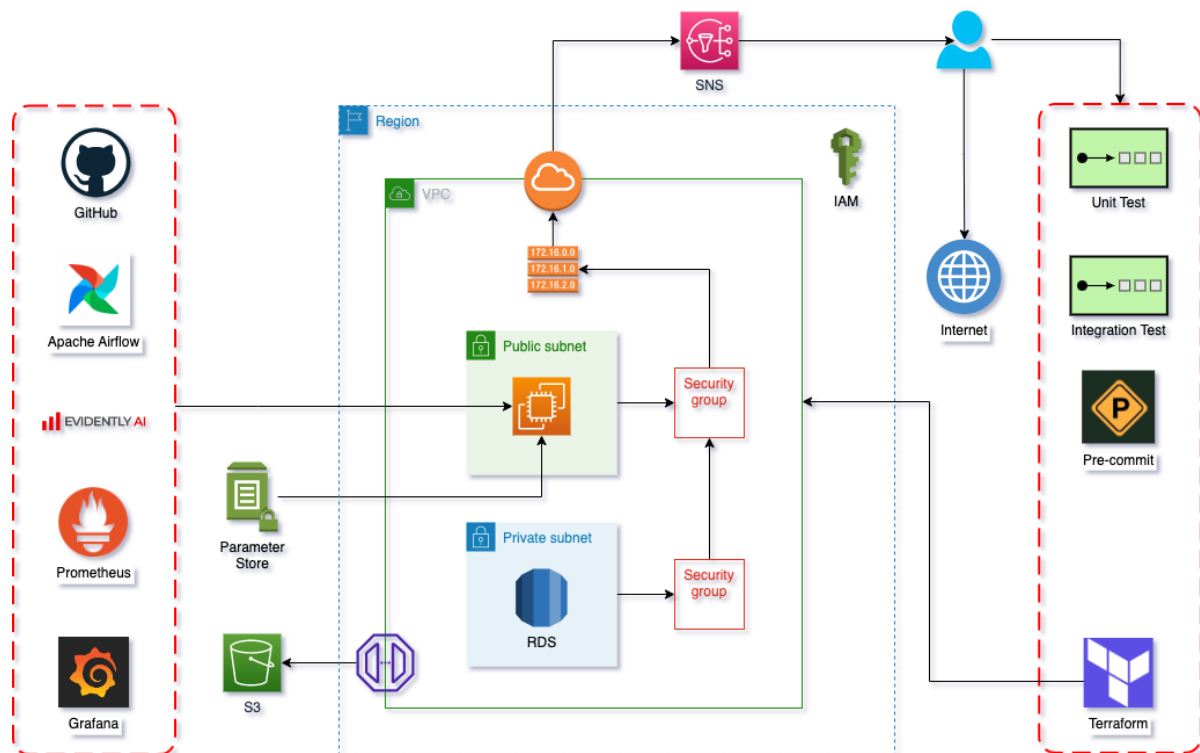


Exhibit-1: MLOps Project Diagram (Image by Author)

Our example ML model is to **predict waiter tips**. Any model goes fine since the main focus is on the deployment within the production environment.

We deploy the cloud infrastructure on AWS with *Terraform*. After performing the tests, we commit the code to GitHub, which eventually loads it into the AWS server through *Actions*. *Apache Airflow*, every month, retrieves the latest data from S3, retrains the model, and stores the best one in the bucket. *Flask* allows the user to find the outcome based on the inputs entered through a web interface.

During the process, we can track the experiments with *MLFlow* and check for any data drift with *Evidently*, *Prometheus* and *Grafana*. The app notifies the user of the training results and data metrics via AWS SNS email notification. This description is just an overview. Toward building the whole structure piece by piece, we may start with a short and straightforward regression prediction.

According to the legend(!), a restaurant waiter keeps a record of [information](#) about the tips he received over a few months[1]. The data contains the following information:

1. *total_bill*: Total bill in dollars, including taxes. A continuous variable.
2. *sex*: The gender of the person paying the bill. Male or Female.
3. *smoker*: A categorical variable for whether the payer smoked or not. Yes or No.
4. *day*: Day of the week when the payment occurred. Sunday through Saturday.
5. *time*: Mealtimes. Lunch or Dinner.
6. *size*: The number of people at the table. An integer variable.
7. *tip*: Tip granted to waiters in dollars. Label as a continuous variable.

| | total_bill | tip | sex | smoker | day | time | size |
|----|------------|------|--------|--------|-----|--------|------|
| 0 | 16.99 | 1.01 | Female | No | Sun | Dinner | 2 |
| 1 | 10.34 | 1.66 | Male | No | Sun | Dinner | 3 |
| 2 | 21.01 | 3.5 | Male | No | Sun | Dinner | 3 |
| 3 | 23.68 | 3.31 | Male | No | Sun | Dinner | 2 |
| 4 | 24.59 | 3.61 | Female | No | Sun | Dinner | 4 |
| 5 | 25.29 | 4.71 | Male | No | Sun | Dinner | 4 |
| 6 | 8.77 | 2.0 | Male | No | Sun | Dinner | 2 |
| 7 | 26.88 | 3.12 | Male | No | Sun | Dinner | 4 |
| 8 | 15.04 | 1.96 | Male | No | Sun | Dinner | 2 |
| 9 | 14.78 | 3.23 | Male | No | Sun | Dinner | 2 |
| 10 | 10.27 | 1.71 | Male | No | Sun | Dinner | 2 |

Exhibit-2: Data (Image by Author)

The case is a regression problem, and we will implement XGBoost.

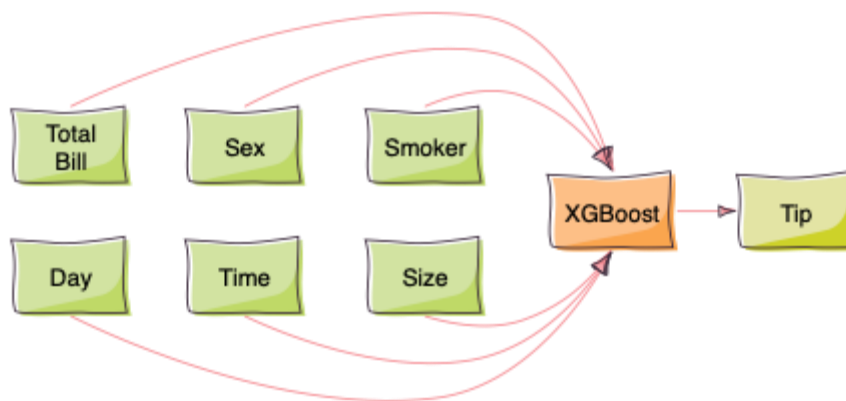


Exhibit-3: Model, Features, and Label (Image by Author)

Data Preparation

We first need to transform the data by hot-encoding categorical values[2] and save it on the local disk. The local disk can be your local machine or EC2 instance.

We import the raw data from the S3 bucket, transform, and save it as a new file, `data_transformed.csv` on the local machine.

Import libraries

```
from airflow.models import Variable
```

```
from utils.airflow_utils import get_vars
```

```
from utils.aws_utils import get_bucket_object, put_object
```

As here and later, some *Airflow* or *AWS* elements exist, we can disregard them for now. We collect and organize the methods we're going to see as specific to *MLflow*, *Airflow*, and *AWS* under separate utility modules: `mlflow_utils.py`, `airflow_utils.py`, and `aws_utils.py`, respectively.

```
def transform_data() -> tuple[dict[str, str], str, int, int, int]:
```

```
    """
```

```
    Imports data from the S3 bucket and converts it into
    a pandas dataframe. Makes necessary transformations.
    Saves it into the local disk.
```

```
    """
```

```
    import json
```

```
    import pandas as pd
```

```
    # Retrieve the s3_dict variable
```

```
    bucket, file_name, local_path, _, _, _ = get_vars()
```

```
    original_file_name = file_name.split("/")[-1]
```

```
    # Read data from S3
```

```

file, _ = get_bucket_object(bucket, file_name)
data = pd.read_csv(file)

# Transform the data
data_transformed = data.copy(deep=True)
data_transformed["sex"] = data_transformed["sex"].map({"Female": 0, "Male": 1})
data_transformed["smoker"] = data_transformed["smoker"].map({"No": 0, "Yes": 1})
data_transformed["day"] = data_transformed["day"].map(
    {"Thur": 0, "Fri": 1, "Sat": 2, "Sun": 3}
)
data_transformed["time"] = data_transformed["time"].map({"Lunch": 0, "Dinner": 1})

# Save it to the local disk as csv file
transformed_file_name = "tips_transformed.csv"
local_data_transformed_filename = f"{local_path}data/{transformed_file_name}"
data_transformed.to_csv(local_data_transformed_filename)

```

Once we have transformed and saved the data, we split it as train and test sets, and save them as separate files. There is a reason for this; we can put it off for now. We also upload the transformed data and training and test datasets in the S3 bucket.

```

def split_data(test_size: float) -> tuple[int, int]:

    """
    Splits the data as training and validation sets.
    Saves them to the local disk and to the S3 bucket.
    """

    import glob

    import numpy as np
    import pandas as pd
    from numpy import savetxt
    from sklearn.model_selection import train_test_split

    # Read data
    data_transformed = pd.read_csv(local_data_transformed_filename)

    # Features and label
    x_features = np.array(
        data_transformed[["total_bill", "sex", "smoker", "day", "time", "size"]]
    )
    y_label = np.array(data_transformed["tip"])

    # Train-test split
    x_train, x_val, y_train, y_val = train_test_split(
        x_features, y_label, test_size=test_size, random_state=42
    )

```

```
# Save the datasets to the local disk
files = {"x_train": x_train, "x_val": x_val, "y_train": y_train, "y_val": y_val}
for key, value in files.items():
    savetxt(f"{local_path}data/{key}.csv", value, delimiter=",")
```

These scripts will run after we deploy the infrastructure on AWS with *Terraform*.

We saved the train and test datasets separately, and now bring them to the local namespace when necessary, and convert them into DMatrix, which is optimized for both memory efficiency and training speed.

```
import numpy as np
import xgboost as xgb
from numpy import loadtxt

# Retrieve variables
_, _, local_path, _, experiment_name, _, _ = get_vars()

# Load data from the local disk
x_train = loadtxt(f"{local_path}data/X_train.csv", delimiter=",")
x_val = loadtxt(f"{local_path}data/X_val.csv", delimiter=",")
y_train = loadtxt(f"{local_path}data/y_train.csv", delimiter=",")
y_val = loadtxt(f"{local_path}data/y_val.csv", delimiter=",")

# Convert to DMatrix data structure for XGBoost
train = xgb.DMatrix(x_train, label=y_train)
valid = xgb.DMatrix(x_val, label=y_val)
```

Our data for training on XGBoost and optimizing the model is ready.

Overview of MLFlow Experiment Tracking Tool

Optimization trials give us some results, but we need to store them so that we can choose the right model (version) later. Thus, we need an experiment tracking tool, and that is going to be *MLFlow*. Therefore, the next step following the data preparation is to start *MLFlow* server. For this, we need to understand its components first.

Our *MLFlow* configuration mainly fits Scenario-4 on *MLFlow* [page](#), which depicts an architecture with a remote *MLFlow* Tracking Server, a Postgres database for backend entity storage, and an S3 bucket for artifact storage[5]. However, one can use a single machine as the tracking server, backend store, and artifact storage. We don't argue about tastes!

Our remote tracking server will be an EC2 instance, yet the word "remote" may sound misleading as we run experiments from within the instance, it is a local host indeed. To record all runs' *MLFlow* entities (experiments, runs, parameters, versions, etc. but no artifacts), the tracking server creates an instance of SQLAlchemy Store and connects to the remote (this is really remote!) PostgreSQL database (RDS in our case) and inserts *MLFlow* entities there.

The tracking server address is 127.0.0.1 as it is the local host. If we exactly set the EC2 as the remote tracking server, then the hostname would be something like `ec2-54-75-5-9.eu-west-1.compute.amazonaws.com`. RDS, as the backend store, bears an endpoint such as `aswtp-rds-instance.cmpdlb9srhwd.eu-west-1.rds.amazonaws.com`. If the tracking server is the local

machine at home or work, RDS should be publicly accessible. If the tracking server is an EC2 instance, then RDS doesn't need to be publicly accessible.

We'll also create a folder in S3 to store the *MLFlow* artifacts which would be output files in any format. For example, we can record images (for example, PNGs), models (for example, a pickled scikit-learn model), and data files (for example, a [Parquet](#) file) as artifacts[6].

The tracking server provides the *MLFlow* client with an artifact store URI location (an S3 storage URI in our case). The *MLFlow* client creates an instance of an *S3ArtifactRepository* on the local host (EC2 in our case), connects to the remote AWS host using the [boto client](#) libraries, and uploads the artifacts to the S3 bucket URI location.

Now, we can put the components together and start the *MLFlow* server.

Starting the MLFlow server

To start *MLFlow*, we need to specify the port (5000), the backend entity storage, and the artifact store on the command line such as:

```
mlflow server --backend-store-uri postgresql://user:password@postgres:5432/mlflowdb --
default-artifact-root s3://bucket_name --host remote_host --no-serve-artifacts
```

As our EC2 instance is the local host, we can type the following line:

```
mlflow server -h 0.0.0.0 -p 5000 --backend-store-uri
postgresql://DB_USER:DB_PASSWORD@DB_ENDPOINT:5432/DB_NAME --default-artifact-root
s3://S3_BUCKET_NAME
```

PostgreSQL runs on port 5432. It might be good to choose one of the earlier versions of Postgres as the RDS engine as I had trouble using the latest one in some other apps. When spinning up the RDS, we will define `DB_USER` and `DB_PASSWORD` and note the database endpoint. They are all necessary in the command line when starting the *MLFlow* server.

After having started the tracking server, backend entity storage, and artifact storage, we should define the *MLFlow* variables that will be called in the script:

```
from mlflow.tracking import MlflowClient
```

```
# We store variables that won't change often in AWS Parameter Store.
```

```
tracking_server_host = get_parameter(
    "tracking_server_host"
) # This can be local: 127.0.0.1 or EC2, e.g.: ec2-54-75-5-9.eu-west-
1.compute.amazonaws.com.
```

```
# Set the tracking server uri
```

```
MLFLOW_PORT = 5000
```

```
mlflow_tracking_uri = f"http://{tracking_server_host}:{MLFLOW_PORT}"
mlflow.set_tracking_uri(mlflow_tracking_uri)
```

```
mlflow_client = MlflowClient(mlflow_tracking_uri)
```

```
# Retrieve the initial path and mlflow artifact path from AWS Parameter Store.
```

```
try:
```

```

mlflow_artifact_path = json.loads(get_parameter("artifact_paths"))[
    "mlflow_model_artifacts_path"
] # models_mlflow
mlflow_initial_path = json.loads(get_parameter("initial_paths"))[
    "mlflow_model_initial_path"
] # s3://s3b-tip-predictor/mlflow/
except:
    mlflow_artifact_path = "models_mlflow"
    mlflow_initial_path = "s3://s3b-tip-predictor/mlflow/"

```

In the above code, we retrieve (`get_parameter`) some variables from *AWS*, where they were created during the *Terraform* deployment. What we need to know essentially by now is only the following list of variables:

- **MLFlow port:** 5000
- **tracking_server_host:** 127.0.0.1. This becomes the local host as we run the entire code from within EC2.
- **mlflow_tracking_uri:** <http://127.0.0.1:5000>. This is the combination of the `tracking_server_host` and the port.

The above three help us create an `MLflowClient` instance. *MLflow* client interacts with the tracking server and artifact storage host.

- **mlflow_artifact_path:** “models_mlflow”
- **mlflow_initial_path:** “s3://s3b-tip-predictor/mlflow/”

We define the last two variables, and they indicate the path to the artifact store on S3, such as:

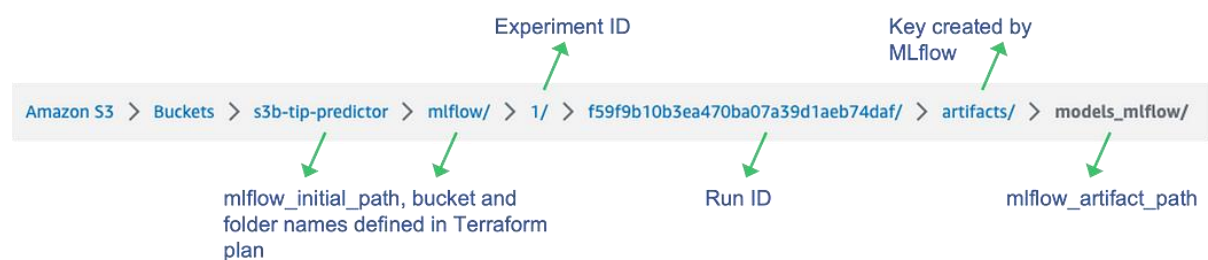


Exhibit-4: Artifact Storage on S3 (Image by Author)

- **experiment_name:** “mlflow-experiment-1”. Again, defined by us.

Flow of Logic

We have completed data preparation and setup of the *MLflow* server, so we can continue with the programmatic execution of optimization and experiment tracking. In the remainder, these tasks will be performed in the flow of logic as follows:

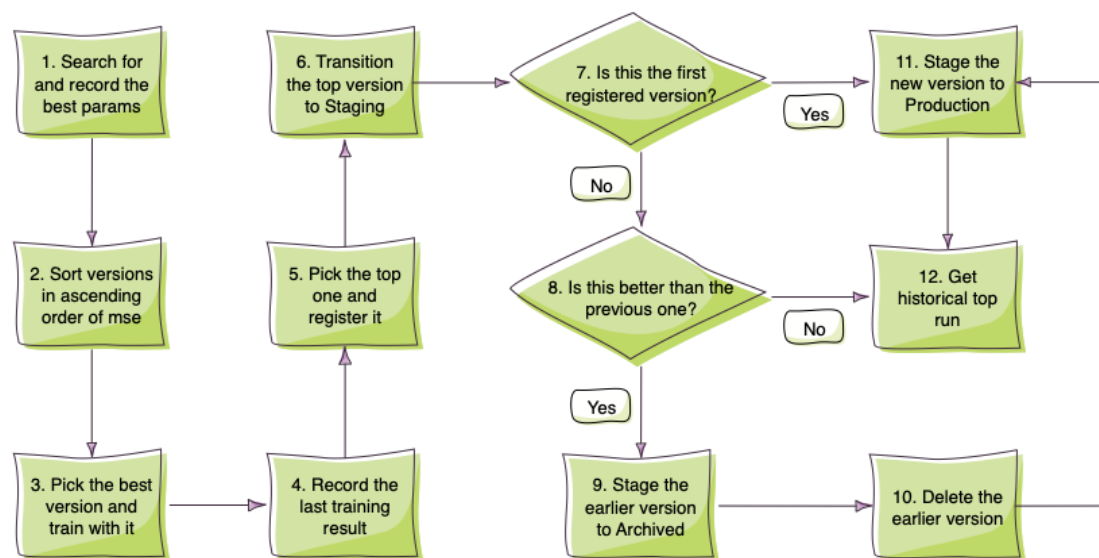


Exhibit-5: Flow of Logic of MLflow Tasks (Image by Author)

We can fulfill many of the above steps from the console. However, to productionize the work, we should execute all tasks programmatically.

Hyperparameter Optimization with Hyperopt

The first task we need to tackle is to search and record the best parameters. Before embedding the optimization snippet in the code, viewing it in isolation should help understand it.

A typical [XGBoost](#) model with parameters is similar to the following:

```
xgboost_model = XGBRegressor(n_estimators=1000, max_depth=7, eta=0.1,
                              subsample=1.0, colsample_bytree=1.0)
```

[Hyperopt](#) is one of the solutions to find the best parameters. Searching a space of selected parameters, *Hyperopt* comes up with a solution set given the constraints of time and computational resources. Unlike *GridSearchCV* which tries all the combinations of the values passed and evaluates the model for each combination using the Cross-Validation method[3], *Hyperopt* implements optimization algorithms to find the combination close to the best but probably not exactly the best[4]. The following is the *Hyperopt* implementation for our ML model:

```
from hyperopt import STATUS_OK, Trials, fmin, hp, tpe
from hyperopt.pyll import scope
```

```
# Search space for parameters
```

```
search_space = {
    "max_depth": scope.int(hp.quniform("max_depth", 4, 100, 1)),
    "learning_rate": hp.loguniform("learning_rate", -3, 0),
    "colsample_bytree": hp.choice("colsample_bytree", np.arange(0.3, 0.8, 0.1)),
    "subsample": hp.uniform("subsample", 0.8, 1),
    "n_estimators": 100,
```



```

"reg_lambda": hp.loguniform("reg_lambda", -6, -1),
"min_child_weight": hp.loguniform("min_child_weight", -1, 3),
"objective": "reg:squarederror",
"seed": 42,
}

best_result = fmin(
    fn=objective,
    space=search_space,
    algo=tpe.suggest,
    max_evals=10,
    trials=Trials(),
)

```

XGBoost parameters are covered in `search_space`. Our objective is to minimize the squared error in the regression. Given previous trials and the domain, the optimization suggests the best-expected hp point according to the TPE-EI algorithm. Trials instance stores the historical records of tested points in the search space and test results, and makes them available to `fmin`. Parameter `max_evals` will try 10 different combinations of hyperparameters, each of which goes through a number of training rounds (`n_estimators`). That means we are up to choosing the set of hyperparameters that resulted in the lowest error score, which was one of the total 10 scores, each produced by an independent XGBoost training cycle. I just set 10 trials to avoid long-lasting combinations, yet, this should be too few; a point to remember!

The `hp.uniform` returns a value uniformly between low and high. For subsample the algorithm will select a value from among equally likely values between 0.8 and 1.0. The `hp.quniform` returns a value like `round(uniform(low, high) / q) * q`. Rounding by `hp.quniform` still returns a whole number of float-type such as 3.0 or 7.0, so we use `scope.int` to convert it to an integer. Increasing `q` allows us to test a lower number of integer values for the parameter. However, `q=1` as stated above will not economize on that population. `hp.loguniform` returns a value drawn according to `exp(uniform(low, high))` so that the logarithm of the return value is uniformly distributed. The `hp.loguniform` gives learning rate and L2 regularization parameters as multiples of power, similar to, 0.1, 0.01, or 0.001. The `hp.choice` randomly chooses from a list of values. For example, we pick `colsample_bytree` from among equally likely values in the list [0.3, 0.4, 0.5, 0.6, 0.7].

Step 1: Searching for the Best

We need to embed the optimization in experiment tracking. For this purpose, we'll create an experiment where we record and observe the results of our trials. The next snippet will create a new experiment if it doesn't exist. If it exists, all future runs will be assigned to it.

```

import mlflow

# Set an mlflow experiment
mlflow.set_experiment(experiment_name)

```

After creating an experiment with `set_experiment`, we define the objective function that *Hyperopt* should minimize. Search space for parameters is the same as we have seen before.

```

# Import libraries
from typing import Any, Literal, Union
from hyperopt import STATUS_OK, Trials, fmin, hp, tpe
from hyperopt.pyll import scope
from sklearn.metrics import mean_squared_error

def search_best_parameters(tag: str) -> dict[str, Union[int, float]]:

    """
    Searches and finds the optimum parameters within the
    defined ranges with Hyperopt. Logs them in MLFlow.
    """

    # Search for best parameters
    def objective(params: dict) -> dict[str, Union[float, Literal["ok"]]]:

        with mlflow.start_run():

            mlflow.set_tag("model", tag)
            mlflow.log_params(params)

            booster = xgb.train(
                params=params,
                dtrain=train,
                num_boost_round=100,
                evals=[(valid, "validation")],
                early_stopping_rounds=50,
            )

            y_pred = booster.predict(valid)
            rmse = mean_squared_error(y_val, y_pred, squared=False)
            mlflow.log_metric("rmse", rmse)

            return {"loss": rmse, "status": STATUS_OK}

    # Search space for parameters
    search_space = {
        "max_depth": scope.int(hp.quniform("max_depth", 4, 100, 1)),
        "learning_rate": hp.loguniform("learning_rate", -3, 0),
        "colsample_bytree": hp.choice("colsample_bytree", np.arange(0.3, 0.8, 0.1)),
        "subsample": hp.uniform("subsample", 0.8, 1),
        "n_estimators": 100,
        "reg_lambda": hp.loguniform("reg_lambda", -6, -1),
        "min_child_weight": hp.loguniform("min_child_weight", -1, 3),
        "objective": "reg:squarederror",
        "seed": 42,
    }

```

```

best_result = fmin(
    fn=objective,
    space=search_space,
    algo=tpe.suggest,
    max_evals=10,
    trials=Trials(),
)

return best_result

```

Under the objective function, we start *MLflow* experiment runs, log parameters that are selected in the search space, and train the data with them. XGBoost sets the maximum number of boosting rounds as 100 with `early_stopping_rounds=50`. `n_estimators` is the number of gradient-boosted trees, and equivalent to the number of boosting rounds. Following the final computation of MSE, STATUS_OK sends the message that optimization is successful. Each of the 10 runs will appear as an individual row in the *MLflow* UI as shown in Exhibit 6.

| Created | Duration | Run Name | User | Source | Version | Models | Metrics | Parameters | Tags |
|----------------|----------|-----------------|--------|---------|---------|-----------------|---------|--|---------|
| 27 minutes ago | 3.5s | righteous-r... | ubuntu | airflow | - | xgboost-mo.../8 | 0.826 | celsample_byt=0.7000000..., learning_rate=0.5433182..., max_depth=92 | xgboost |
| 32 minutes ago | 3.5s | inquisitive-... | ubuntu | airflow | - | xgboost | 0.826 | celsample_byt=0.7000000..., learning_rate=0.5433182..., max_depth=92 | xgboost |
| 32 minutes ago | 478ms | merciful-w... | ubuntu | airflow | - | - | 0.826 | celsample_byt=0.7000000..., learning_rate=0.5433182..., max_depth=92 | xgboost |
| 32 minutes ago | 0.5s | rogue-cub-... | ubuntu | airflow | - | - | 0.845 | celsample_byt=0.6000000..., learning_rate=0.2125745..., max_depth=58 | xgboost |
| 36 minutes ago | 3.5s | salty-bee-97 | ubuntu | airflow | - | xgboost | 0.85 | celsample_byt=0.5, learning_rate=0.7333341..., max_depth=27 | xgboost |
| 40 minutes ago | 3.5s | awesome-c... | ubuntu | airflow | - | xgboost | 0.85 | celsample_byt=0.5, learning_rate=0.7333341..., max_depth=27 | xgboost |
| 44 minutes ago | 3.5s | dazzling-sh... | ubuntu | airflow | - | xgboost | 0.85 | celsample_byt=0.5, learning_rate=0.7333341..., max_depth=27 | xgboost |
| 50 minutes ago | 5.7s | efficient-sh... | ubuntu | airflow | - | xgboost-mo.../3 | 0.85 | celsample_byt=0.5, learning_rate=0.7333341..., max_depth=27 | xgboost |
| 58 minutes ago | 3.4s | unleashed-... | ubuntu | airflow | - | xgboost-mo.../2 | 0.85 | celsample_byt=0.5, learning_rate=0.7333341..., max_depth=27 | xgboost |
| 1 hour ago | 3.7s | unruly-doe... | ubuntu | airflow | - | xgboost | 0.85 | celsample_byt=0.5, learning_rate=0.7333341..., max_depth=27 | xgboost |
| 1 hour ago | 424ms | bright-squ... | ubuntu | airflow | - | - | 0.85 | celsample_byt=0.5, learning_rate=0.7333341..., max_depth=27 | xgboost |
| 1 hour ago | 1.4s | judicious-s... | ubuntu | airflow | - | - | 0.854 | celsample_byt=0.7000000..., learning_rate=0.0928525..., max_depth=16 | xgboost |

Exhibit-6: Records of MLflow Runs (Image by Author)

In addition to artifacts sent to S3, *MLflow* UI provides us with access to the *MLflow* entities recorded in RDS through the web browser while we run experiments. We can open *MLflow* UI at <http://0.0.0.0:5000>. If we reach this URL from our local machine at home or work, we first need to do port forwarding before opening it in our browser. As other tools we will see later have various port numbers, all require port forwarding in the case of outbound local access.

We can see those 10 runs here as well. We are interested in the version that has the lowest mean squared error (MSE). The above image shows which experiment and run have the most optimized version and what parameters are used to train it.

Steps 2–4: Choosing and Recording the Best

Next, we need to get the parameters of the best version from the backend store to run the model with them.

```
from utils.mlflow_utils import search_runs
```

```
def get_best_params(
    client: MlflowClient, experiment_name: str, order_by: str, max_results: int = 10000
) -> tuple[dict[str, Any], str]:
    """
    Gets the parameters of the best model run of a particular experiment.
    """

    # Get the id of the found experiment
    experiment_id = get_experiment_id(experiment_name)

    # Get the pandas data frame of the experiment results in the ascending order by RMSE
    runs = search_runs(client, experiment_id, order_by, max_results)

    # Get the id of the best run
    best_run_id = runs[0].info.run_id

    # Get the best model parameters
    best_params = runs[0].data.params
    rmse = runs[0].data.metrics["rmse"]

    return best_params
```

Here, we input the name of the experiment to find its ID. Then based on the ID, `search_runs` collects all runs of that experiment, and sorts them in ascending order of MSE. Then, we choose the very top run on the table and fetch and store its parameters (`best_params`) and MSE score (`rmse`) (step 2).

As mentioned earlier and like in the above routine, some general use-case functions (such as `get_experiment_id`) that involve *MLflow* methods are part of the user-defined *MLflow* utility module (`mlflow_utils.py`).

Using the best version (`best_params`) we found and fetched, we can run the XGBoost in the same fashion as we did in the `search_best_parameters` function with slight differences (step 3). *MLflow* then records the output (step 4):

```
with mlflow.start_run() as run:
```

```
    # Get the run_id of the best model
    best_run_id = run.info.run_id

    mlflow.set_tag("model", tag)
    mlflow.log_params(best_params)

    # Train the XGBoost model with the best parameters
```

```

booster = xgb.train(
    params=best_params,
    dtrain=train,
    num_boost_round=100,
    evals=[(valid, "validation")],
    early_stopping_rounds=50,
)

y_pred = booster.predict(valid)
rmse = mean_squared_error(y_val, y_pred, squared=False)
mlflow.log_metric("rmse", rmse)

```

This is a single run (best_run_id); one can see it as the 11th row in *MLflow* UI (see Exhibit 6), and its details in Exhibit 7:

The screenshot shows the MLflow UI interface. At the top, there's a navigation bar with 'mlflow 1.30.0', 'Experiments', and 'Models'. Below this, the breadcrumb is 'mlflow-experiment-1 > righteous-rat-791'. The main title is 'righteous-rat-791'. Below the title, there are three columns of information: 'Run ID: 7902d89d6c834aed90fc116efda7990' (highlighted with a red box), 'Date: 2022-11-08 16:06:10', 'Source: ☐ airflow'; 'User: ubuntu', 'Duration: 3.5s', and 'Status: FINISHED'; 'Lifecycle Stage: active'. Below this, there's a 'Description' section with a dropdown arrow and the text 'None'. Then, there's a 'Parameters (9)' section (highlighted with a red box) which contains a table with 9 parameters:

| Name | Value |
|------------------|----------------------|
| colsample_bytree | 0.7000000000000002 |
| learning_rate | 0.5433182151655946 |
| max_depth | 92 |
| min_child_weight | 14.326631966590954 |
| n_estimators | 100 |
| objective | reg:squarederror |
| reg_lambda | 0.005122843691525722 |
| seed | 42 |
| subsample | 0.8095672790689089 |

Exhibit-7: Details of a Run (Image by Author)

When we open up a particular run, we'll see the same set of information: Run ID, Parameters, etc.

The code line below stores the artifacts in S3 as shown in Exhibits 8 and 9.

```

# Save the model (booster) using 'log_model' in the defined artifacts folder/bucket
# This is defined on the CLI and as artifact path parameter on AWS Parameter Store:
# s3://s3b-tip-predictor/mlflow/ ... /models_mlflow/
mlflow.xgboost.log_model(booster, artifact_path=mlflow_artifact_path)

```

Over time each experiment will have several runs. In our case, every time we train our data, *MLflow* will generate 10 distinct runs (max_evals=10). One can set how many runs there should be. Below we see the population of runs under experiment 1.

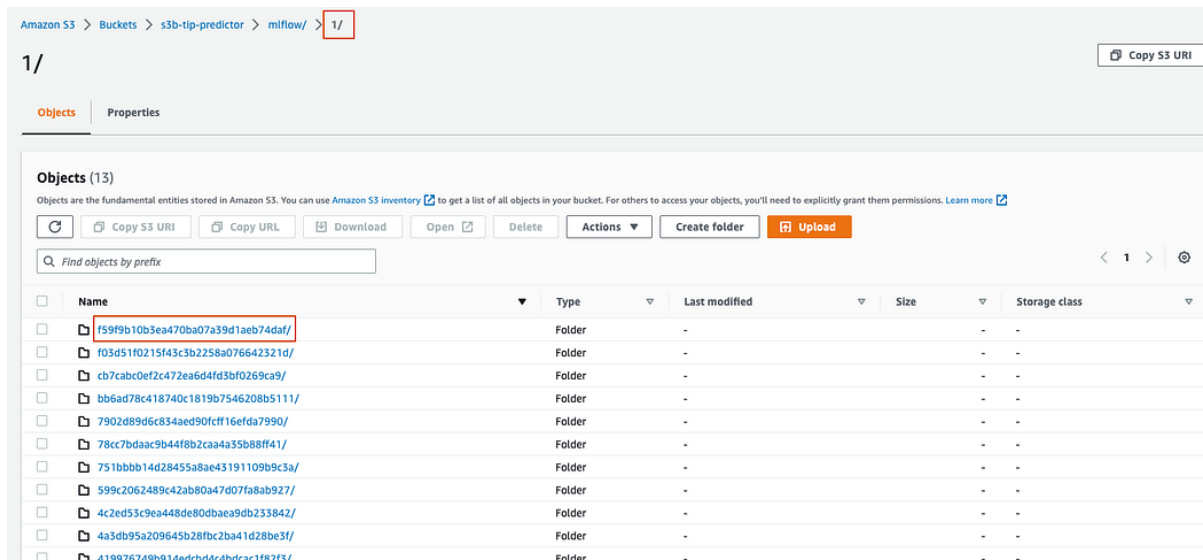


Exhibit-8: Runs in the Experiment Folder in S3 (Image by Author)

And each run stores artifacts as explained earlier:

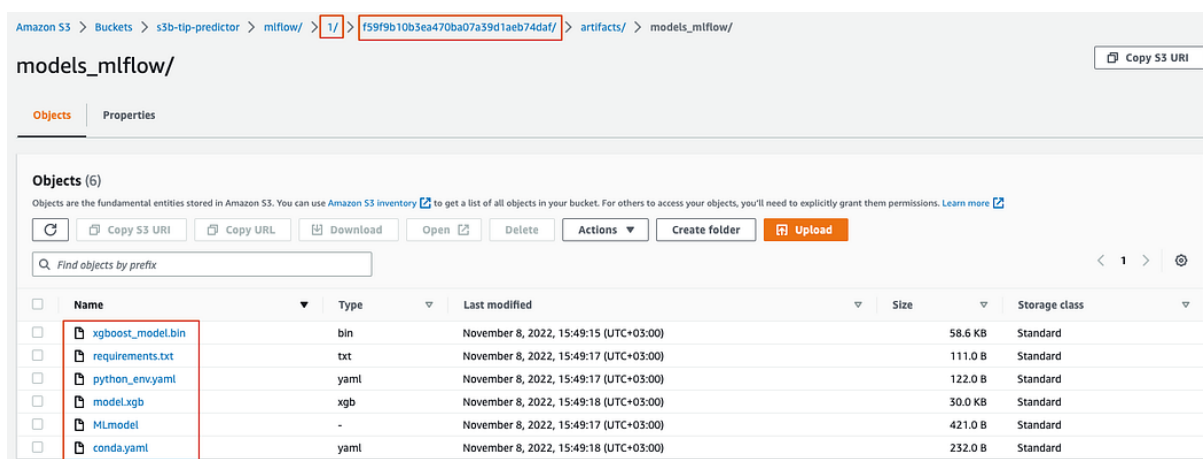


Exhibit-9: Run Folder in S3 (Image by Author)

Step 5: Registering the Best Version

We may now register the best version, describe, and transition it to a stage. Registration helps us use the version later in production. One can see and achieve version registration on the console shown in Exhibit 10:

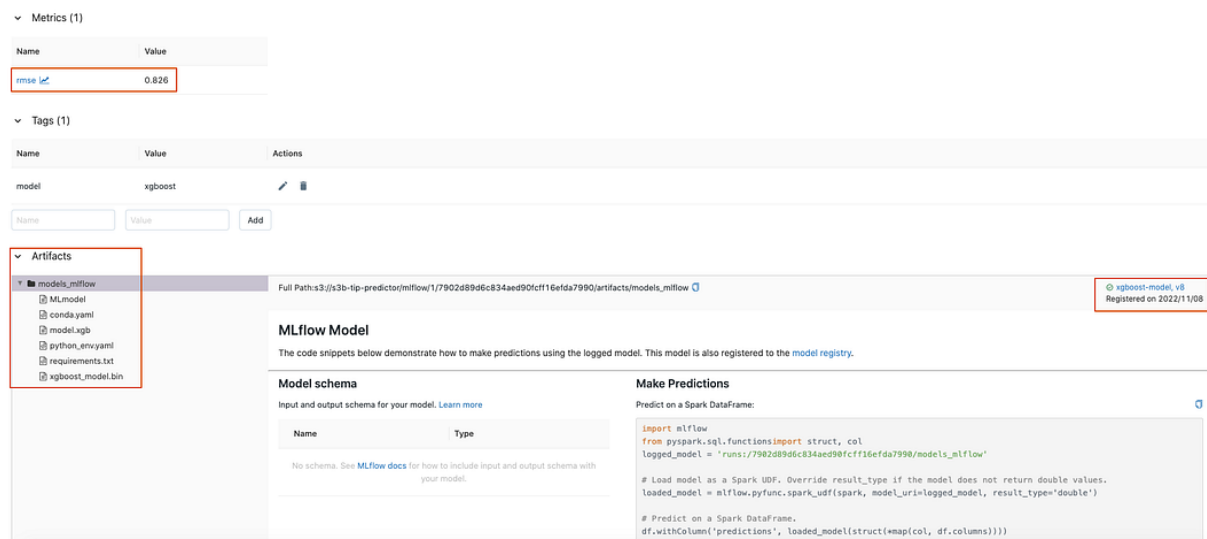


Exhibit-10: Model and Artifacts of a Run on MLflow UI (Image by Author)

However, we'll perform the registration and the following steps programmatically with the aid of `mlflow_utils.py`.

We first retrieve the model name and the best run id from *Airflow*. We had logged this information in *MLflow*. One thing that we have not seen yet was to store the same information in *Airflow XCom*, a concept we'll see later in one of the following articles. For now, we should take it as is.

```
from utils.airflow_utils import get_vars
```

```
# Retrieve variables
```

```
_, _, _, _, _, model_name = get_vars()
```

```
# Get the best run id
```

```
best_run_id = ti.xcom_pull(key="best_run_id", task_ids=["run_best_model"])
```

```
best_run_id = best_run_id[0]
```

We then register the version (`register_model`):

```
from utils.mlflow_utils import register_model
```

```
# Register the best model
```

```
model_details = register_model(best_run_id, mlflow_artifact_path, model_name)
```

After registering a model version, it may take a short time to become ready. Certain operations, such as model stage transitions, require the model to be in the `READY` state. Other procedures, such as adding a description or fetching model details, can be performed before the model version is ready[7]. Just in case, after the registration, `wait_until_ready` gets executed before adding descriptions as well as transitioning.

```
from utils.mlflow_utils import wait_until_ready
```

```
# Wait until the model is ready
wait_until_ready(mlflow_client, model_details.name, model_details.version)
```

We can add two types of descriptions[7]:

1. Registered model description: This high-level description is useful for recording information that applies to multiple model versions (such as a general overview of the modeling problem and dataset). The `update_registered_model` method achieves that goal.

```
from utils.mlflow_utils import update_registered_model
```

```
# Add a high-level description to the registered model, including the machine
# learning problem and dataset
description = """
```

```
    This model predicts the tips given to the waiters for serving the food.
```

```
    Waiter Tips data consists of six features:
```

1. total_bill,
2. sex (gender of the customer),
3. smoker (whether the person smoked or not),
4. day (day of the week),
5. time (lunch or dinner),
6. size (number of people in a table)

```
    """
```

```
update_registered_model(mlflow_client, model_details.name, description)
```

2. Model version description: This low-level description is useful for detailing the unique attributes of a particular model version (such as the methodology and algorithm used to develop the model). The `update_model_version` method performs this task.

```
from utils.mlflow_utils import update_model_version
```

```
# Add a model version description with information about the model architecture and
# machine learning framework
```

```
update_model_version(
    mlflow_client, model_details.name, model_details.version, version_description
)
```

Step 6: Stage Transitioning

Following the registration and description, we'll move the version to the Staging stage. If this is our first registered best model or outperforms the current one, then we may transition the former to the Production stage. The current and weaker version is to end up in the Archived stage for a possible deletion later. We can view the registered and staged versions as shown in Exhibit 11:

| mlflow | | | | | GitHub | Docs |
|-------------------|----------------|-----------|------------|---------------------|-------------------|------|
| Registered Models | | | | | search model name | |
| Name | Latest Version | Staging | Production | Last Modified | | |
| Model A | Version 1 | Version 1 | – | 2019-10-16 22:51:19 | | |
| Model B | Version 1 | – | – | 2019-10-16 22:51:52 | | |

Exhibit-11: MLflow Stage Transition (Taken from <https://www.mlflow.org/docs/latest/model-registry.html>)

Now, we transition the new version to Staging (transition_to_stage).

```
from utils.mlflow_utils import transition_to_stage
```

```
# Transition the model to Staging
transition_to_stage(
    mlflow_client, model_details.name, model_details.version, "staging", False
)
```

And finally, we push the updated model details to *Airflow*.

```
# Push model details to XCom
model_details_dict = {}
model_details_dict["model_name"] = model_details.name
model_details_dict["model_version"] = model_details.version
ti.xcom_push(key="model_details", value=model_details_dict)
```

Steps 7–11: Finding the Historical Best

At this point, we should ask: Can we use the registered version in production? The answer is: If it is our first version ever, then yes, we can. If not, we need to find out if it outperforms the current one we're using in production. Testing and comparing both versions will resolve the issue.

We start pulling the variables from *Airflow* where we pushed them before:

```
# Get model details from XCom
model_details_dict = ti.xcom_pull(
    key="model_details", task_ids=["register_best_model"]
)
model_details_dict = model_details_dict[0]
model_name = model_details_dict["model_name"]
model_version = model_details_dict["model_version"]
```

We load the test datasets we saved earlier:

```
# Load data from the local disk
x_test = loadtxt(f"{local_path}data/X_val.csv", delimiter=",")
y_test = loadtxt(f"{local_path}data/y_val.csv", delimiter=",")
```

Though not fully covered in the image, the bottom right of Exhibit 10 shows the part of UI where *MLflow* provides the python code snippet of how to load a specific model. This is what the `load_models` utility function does.

```
import logging
from utils.mlflow_utils import load_models

# Load the staging model, predict with the new data and calculate its RMSE
model_staging = load_models(model_name, "staging")
model_staging_predictions = model_staging.predict(x_test)
model_staging_rmse = mean_squared_error(
    y_test, model_staging_predictions, squared=False
)
logging.info("model_staging_rmse: %s", model_staging_rmse)

try:
    # Load the production model, predict with the new data and calculate its RMSE
    model_production = load_models(model_name, "production")
    model_production_predictions = model_production.predict(x_test)
    model_production_rmse = mean_squared_error(
        y_test, model_production_predictions, squared=False
    )

except Exception:
    print(
        "It seems that there is not any model in the production stage yet. \
        Then, we transition the current model to production."
    )
    model_production_rmse = None
```

We first load the model in the Staging stage, make a prediction using the test data, and compute the MSE. As you may have realized, I've made a mistake in naming the loss variable `rmse` everywhere in the code so far (sorry!), yet we are not computing the root mean squared error as the loss but the mean squared error.

Then, we repeat the same process for the model in the Production stage. As a final step, we store both error values in *Airflow XCom* to use them later. As of now, we have supplied information to make the two decisions at steps 7 and 8, which are represented with rhombuses in Exhibit 5.

```
rmse_dict = {}
rmse_dict["model_production_rmse"] = model_production_rmse
rmse_dict["model_staging_rmse"] = model_staging_rmse

# Push the RMSEs of the staging and production models to XCom
ti.xcom_push(key="rmse_dict", value=rmse_dict)
```

Once we get the loss values, now we may compare them to see which model performs better and go through steps 7, 8, 9, 10, and 11.

We bring the MSE scores from the *Airflow*:

```
# Get rmse_dict from XCom
rmse_dict = ti.xcom_pull(key="rmse", task_ids=["test_model"])
rmse_dict = rmse_dict[0]
model_production_rmse = rmse_dict["model_production_rmse"]
model_staging_rmse = rmse_dict["model_staging_rmse"]
```

Next, we retrieve the model details (model_name, model_version) from *Airflow XCom* the same way as we did before. Now we're ready to compare both MSEs.

```
from utils.mlflow_utils import get_latest_version, delete_version
```

```
# Compare RMSEs
# If there is a model in production stage already
if model_production_rmse:

    # If the staging model's RMSE is lower than or equal to the production model's
    # RMSE, transition the former model to production stage, and delete the previous
    # production model.
    if model_staging_rmse <= model_production_rmse:
        transition_to_stage(
            mlflow_client, model_name, model_version, "production", True
        )
        latest_stage_version = get_latest_version(
            mlflow_client, model_name, "archived"
        )
        delete_version(mlflow_client, model_name, latest_stage_version)
    else:
        # If there is not any model in production stage already, transition the staging
        # model to production
        transition_to_stage(
            mlflow_client, model_name, model_version, "production", False
        )
```

If a model exists in production (step 8), and the new one has a lower MSE, we send the current one to Archive first (step 9) and then delete it (delete_version, step 10), and transition the new model to the Production stage (transition_to_stage, step 11). If the MSE of the latest version is higher than that of the current one, we do nothing; the new version stays in the Staging stage, and thus we keep using the current one in production. In case we haven't had any model in the Production stage (step 7), then we automatically move the new version to that stage.

Step 12: Historical Top Run

So far, we have dwelled on a single experiment. In the future, one might want to create more experiments, each of which will have several runs. Across all experiments, to find the best run, we execute the following scripts:

```
best_rmse_dict = {}
```

```

# A very high error score as the baseline
best_run_score = 1_000_000

# List of existing experiment ids
experiment_ids = get_experiments_by_id()

for experiment_id in experiment_ids:

    try:
        # Best run of a particular experiment
        runs = search_runs(mlflow_client, experiment_id, metric, 10000)
        best_run_id = runs[0].info.run_id
        best_run_rmse = runs[0].data.metrics["rmse"]

        # If the experiment has the best score, we store it.
        if best_run_rmse <= best_run_score:
            best_run_score = best_run_rmse
            best_rmse_dict["experiment_id"] = experiment_id
            best_rmse_dict["best_run_id"] = best_run_id
            best_rmse_dict["best_run_rmse"] = best_run_rmse

    except Exception:
        print("Experiment by id '%s' has no runs at all.", experiment_id)

```

In the above code segment, we first collect all experiments by their IDs (`get_experiments_by_id`). Then, we go over each of them to get their respective best run. We compare each experiment's best-run MSE with the lowest MSE recorded so far. After we find the run with the lowest MSE across all experiments, we store its score, experiment ID, and run ID in a dictionary (`best_rmse_dict`).

The `get_experiments_by_id` is as follows:

```

from utils.mlflow_utils import list_experiments

def get_experiments_by_id() -> list[str]:

    """
    Lists all existing experiments and finds their ids.
    """

    # List of existing experiments
    experiments = list_experiments(mlflow_client)
    experiment_ids = [experiment.experiment_id for experiment in experiments]

    # Remove default experiment as it has no runs
    experiment_ids.remove("0")

    return experiment_ids

```

The `list_experiments` is part of the `mlflow_utils.py`.

Finally, we create a parameter named `logged_model` on *AWS Parameter Store* assigning a value such as `s3://s3b-tip-predictor/mlflow/1/3b4f17801fed4ae0be3fc6e20efaf44d/artifacts/models_mlflow/` and notify the user of the information stored in the `best_rmse_dict` via email through *AWS SNS*. The notification we'll receive can be seen in Exhibit 12:

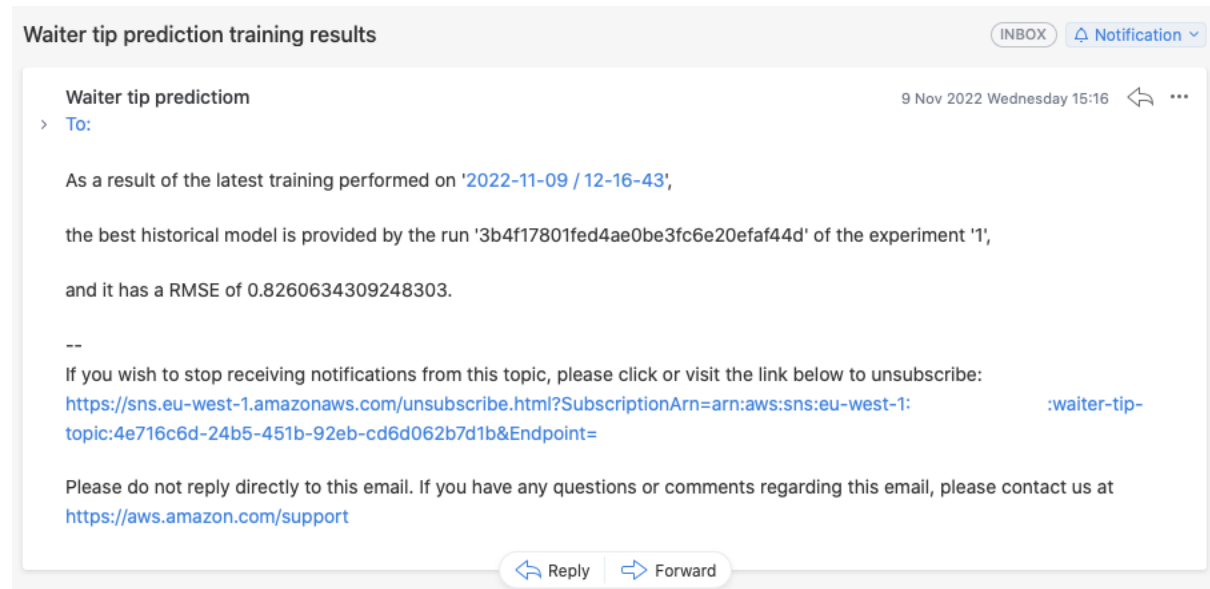


Exhibit-12: Training Results Notified by AWS SNS (Image by Author)

The S3 URI is the logged model we're going to use in our application. We will see this in the *AWS with Terraform* part later.

The cycle depicted in Exhibit 5, with some additional steps that will come out later, works periodically, thanks to *Apache Airflow* which checks and sends the latest data in S3 to training every month. However, over time, the performance of the model may decline, the relevance and correlation of the data may fade, or the overall concept might drift. To see and monitor if all these happen or not, we need a **monitoring tool** such as *Evidently*, which we'll try to discover in the next article.

Troubleshooting

While implementing *MLflow*, I encountered some errors, and could fix them in the following way:

- When you receive this error message:

Detected out-of-date database schema (found version cc1f77228345, but expected 97727af70f4d). Take a backup of your database, then run 'mlflow db upgrade <database_uri>' to migrate your database to the latest schema. NOTE: schema migration may result in database downtime — please consult your database's documentation for more detail.

You need to upgrade your database, and for security, take a backup of your database. Version numbers should be different in your case. You can upgrade your database as follows:

```
mlflow db upgrade postgresql://DB_USER:DB_PASSWORD@DB_ENDPOINT:5432/DB_NAME
```

Not all databases may require an upgrade though.

- After installing and running *MLflow* on Ubuntu on AWS, the following errors might arise:

warnings.warn("urllib3 ({}), or chardet ({}), doesn't match a supported

Then, you may need to install the right versions:

```
pip uninstall chardet
pip install chardet==4.0.0
pip install requests -U
```

A second issue that might come up:

cannot import name 'ParameterSource' from 'click.core'

Then execute:

```
pip install -U click
```

Or, if you see the following message:

ERROR: flask-appbuilder 4.1.2 has requirement Flask-WTF<1.0.0,>=0.14.2, but you'll have flask-wtf 1.0.1 which is incompatible.

ERROR: flask-appbuilder 4.1.2 has requirement WTForms<3.0.0, but you'll have wtforms 3.0.1 which is incompatible.

Install the compatible versions:

```
pip uninstall Flask-WTF -y
pip uninstall WTForms -y
pip install Flask-WTF==0.15.1
pip install WTForms==2.3.3
```

Compatible version installations can be handled during the pip install with requirements.txt or Pipfile. However, at first, I didn't know which versions were compatible before installing and trying them.

In a summary, we have trained and optimized our model and found the best version to use in production. All work is saved on the local machine and uploaded to S3. *Airflow* and *AWS* stored some other outputs as well. Now, we are ready to move on with the data check.

Part 2: Evaluating, Testing, and Monitoring the ML Model, and Measuring the Data Quality and Drift with Evidently

Let us refresh our memory as to what we are trying to build. We are going in the footsteps of Exhibit 1. In the Part2 , we saw how to optimize the *XGBoost* hyperparameters, train our regression model to predict waiter tips, track our experiments on *MLFlow* and save the results locally and on the *AWS* cloud.

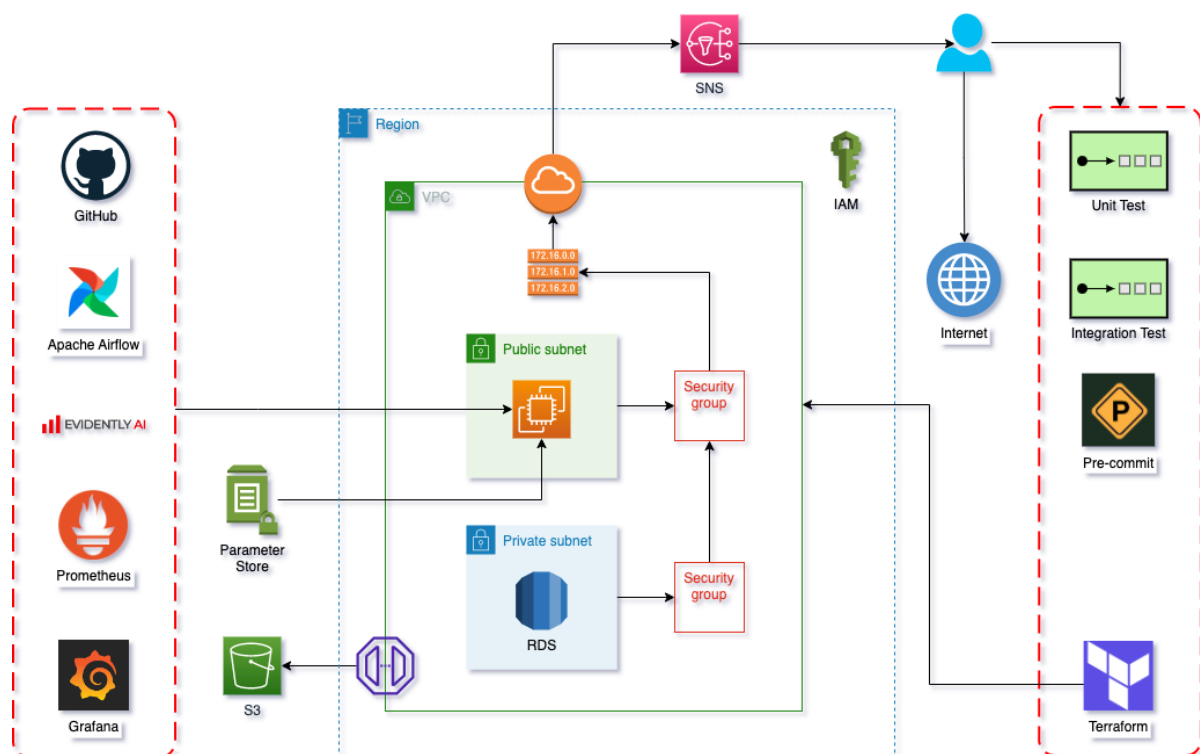


Exhibit-1: MLOps Project Diagram

To get more accurate results in predicting waiter tips, we need to periodically receive new data from S3 to retrain our model. While this may go on forever, the model's success is not guaranteed to hold for good as we usually witnessed during the early phases of development and implementation. Such broken optimism can occur due to changes in the dynamics of the model. In more precise terms, *concept drift* and *data drift* can cause the deterioration of the model.

Concept drift means that the independent variable is not powerful enough anymore to explain the dependent variable, indicating a conceptual flaw in our model. For example, the smoking habit of the person paying the bill might not be as explanatory as before since smoking has declined considerably in the past years if it is so.

Data drift means that data used to train the model has become obsolete. The training data and the actual data now come from different distributions. A photo app, for example, which tells you how many animals there are in the image may have been developed with the dev and test sets compiled from high-quality images. Over time, users upload images with lesser quality, such as being taken with their mobile phones, noisy shots, or

perfunctory angle setting. Such type mismatch between the dev/test set and the actual data calls for retraining.

We need to monitor and be aware of both types of deterioration to take necessary corrective actions, and they need to be engaged regularly as our core model.

In the previous article, our core model was displayed as follows:

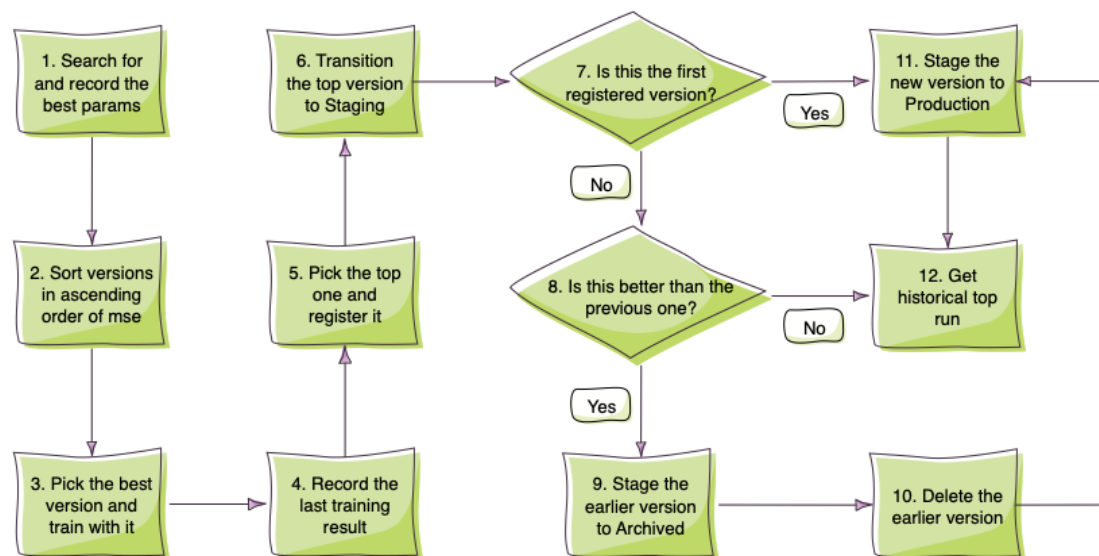


Exhibit-2: Flow of Logic of MLflow Tasks (Image by Author)

Once all tasks in the flow of logic have been accomplished, performance monitoring should start immediately. Avoiding cramming new steps into Exhibit 2, we may depict the extensional functionality starting with step 12 as follows:

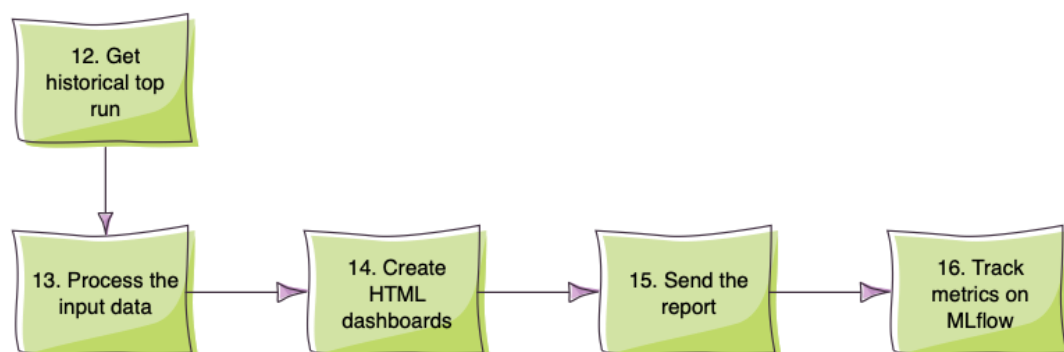


Exhibit-3: Evidently Part of the Flow of Logic (Image by Author)

We're going to use [Evidently](#) to monitor the performance of our model, which covers steps 13 through 16.

Evidently

Evidently helps evaluate, test, and monitor ML models in production[1].

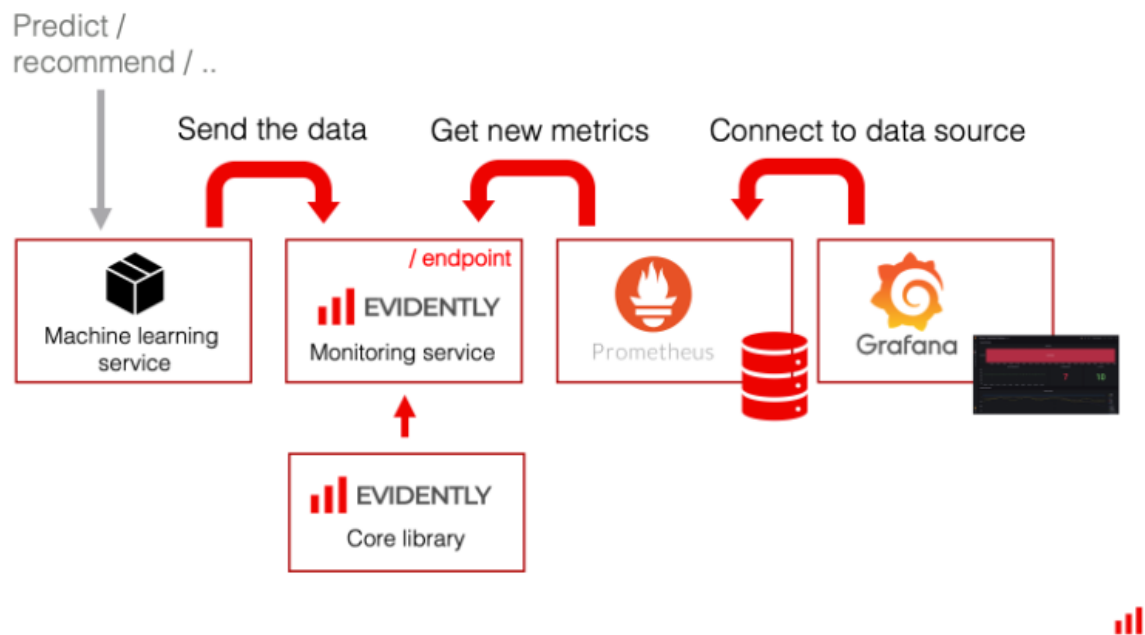


Exhibit-4: High-level Overview of the Integration Architecture

The Evidently treats the benchmark dataset and the current dataset as model application logs and reads these model logs. It receives the input data from a production ML service, and once enough new observations are collected, it calculates the metrics we need. It uses the Analyzers from the core Evidently library to define the way statistical tests and metrics are estimated. The Evidently service then exposes a Prometheus endpoint. Prometheus will check for the new metrics from time to time and log them to the database. Prometheus is then used as a data source to Grafana[2].

Installation

We start by installing *Evidently* (Mac OS and Linux)[3]:

```
$ pip install evidently
```

This installation allows us to generate interactive reports as HTML files or JSON profiles.

To build interactive reports in a Jupyter notebook, we need to install *jupyter nbextension*. After installing *Evidently*, we open the terminal in the *Evidently* directory such as

```
cd /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages/evidently
```

and execute the two following commands in the terminal from the *Evidently* directory.

```
$ jupyter nbextension install --sys-prefix --symlink --overwrite --py evidently
```

To enable it, run

```
$ jupyter nbextension enable evidently --py --sys-prefix
```

I had difficulty generating reports inside the Jupyter Notebook until I realized that this had to do with using Jupyter Lab instead of Jupyter Notebook as I used the former. However, the report generation in separate HTML files worked fine.

Since we are neither interested in playing around with Jupyter Lab nor Jupyter Notebook, keep this information as a side note and continue with the first installation command only.

I suggest exploring the [Evidently](#) website to see alternative ways to achieve monitoring with a focus on different use cases. As this article is not devoted to the topical explanation of *Evidently*, we'll go forward with a specific set of methods and metrics.

Step 13: Processing the Input Data

Evidently expects a particular dataset structure and input column names[4]. Therefore, we need to process the input data such that our monitoring tool can use it to generate reports.

```
import logging
from typing import Any, Dict, List, Literal, Union
```

```
from utils.airflow_utils import get_vars
from utils.aws_utils import get_parameter, put_object, send_sns_topic_message
```

We haven't seen *Airflow* and *AWS* methods yet, so we may disregard them now for further inspection. If anyone is interested, however, he may check out for *Airflow*, and for *AWS*.

Processing data can occur in two ways:

1. Column types: *Evidently* has a default mapping strategy to assign a data type to each column based on the data it has. As recommended, developers should manually do column mapping to avoid any mismatches that might occur such as defining an integer-type column as categorical.
2. Dataset structure: *Evidently* follows a particular dataset structure[4].
 - The column named "id" will be treated as an ID column.
 - The column named "datetime" will be treated as a DateTime column.
 - The column named "target" will be treated as a target function.
 - The column named "prediction" will be treated as a model prediction.

ID, DateTime, target, and prediction are utility columns. If provided, the "datetime" column will be used as an index for some plots. In its absence, each sample observation will be represented with an index number. The "ID" column will be excluded from drift analysis. We should have two columns mapped to "target" and prediction." "Target" will be the actual values (i.e., label), and "prediction" as it suggests, will be the predictions the model made.

We can create a ColumnMapping object to map our column names and feature types.

```
from evidently.pipeline.column_mapping import ColumnMapping
```

```

column_mapping = ColumnMapping()
column_mapping.target = (
    "target" # 'target' is the name of the column with the target function
)
column_mapping.prediction = (
    "prediction" # 'prediction' is the name of the column(s) with model predictions
)
column_mapping.id = None # There is no ID column in the dataset

column_mapping.numerical_features = ["total_bill", "size"] # List of numerical features
column_mapping.categorical_features = [
    "sex",
    "smoker",
    "day",
    "time",
] # List of categorical features

column_mapping.task = 'regression'

```

The “target” column is the label column, i.e., “tip.” In our `tips_transformed.csv` file, the actual value column is named “tip.” Now, we need to map it to “target.” We follow a similar pattern mapping our predictions to the “prediction” column. This way, we tell *Evidently*, which column in our dataset should be treated as “target,” and which column should be treated as “prediction.” “Target” and “prediction” are keys to obtaining metrics. We don’t have any ID or datetime columns. We also explicitly mention which columns in our data are numerical and which are categorical.

The task parameter accepts two values: “regression” and “classification.” If you don’t specify the task, *Evidently* will choose “regression” if the target is of numeric type and the number of unique values is higher than five. In all other cases, it will choose “classification.” In the original code, I didn’t set the task parameter. This is a bad practice! Just imagine a multi-class case where classes are encoded as numbers. *Evidently* might consider it a regression problem. Furthermore, the lack of explicit mention affects the calculation and selection of statistical tests.

Finally, we can perform the whole input data processing programmatically. As you may remember from the previous article, we saved our best model. Now, it’s time to call it again:

```

import mlflow
import numpy as np

# Retrieve variables
bucket, _, local_path, _, _, _ = get_vars()

# Load the model
logged_model = get_parameter("logged_model")
loaded_model = mlflow.pyfunc.load_model(logged_model)

```

We load the datasets:

```
import pandas as pd
```

```
# Load processed datasets from the local storage
x_train = pd.read_csv(f"{local_path}data/X_train.csv")
x_val = pd.read_csv(f"{local_path}data/X_val.csv")
y_train = pd.read_csv(f"{local_path}data/y_train.csv")
y_val = pd.read_csv(f"{local_path}data/y_val.csv")
```

and perform the dataset structure leg by renaming and appending columns:

```
def rename_columns(
    data_frame: pd.DataFrame, old_names: list[str], new_names: list[str]
) -> pd.DataFrame:

    """
    Renames the columns in a dataframe, and changes the data type of columns.
    """

    # Rename the columns
    cols_dict = dict(zip(old_names, new_names))
    data_frame.rename(columns=cols_dict, inplace=True)

    # Change type of the feature columns
    if len(new_names) > 1:
        convert_dict = {x: int for x in new_names[1:]}
        data_frame = data_frame.astype(convert_dict)

    logging.info("Columns are renamed as defined in '%s!'", cols_dict)

    return data_frame


# Rename the target columns of the label datasets as 'target'
y_train_target = y_train.columns.to_list()[0]
y_train = rename_columns(y_train, [y_train_target], ["target"])

y_val_target = y_val.columns.to_list()[0]
y_val = rename_columns(y_val, [y_val_target], ["target"])

# Rename the columns of the feature datasets
columns = ["total_bill", "sex", "smoker", "day", "time", "size"]
x_train_cols = x_train.columns.to_list()
x_val_cols = x_val.columns.to_list()

x_train = rename_columns(x_train, x_train_cols, columns)
x_val = rename_columns(x_val, x_val_cols, columns)
```

```
# Add 'target' column to feature datasets
```

```
x_train["target"] = y_train["target"]
```

```
x_val["target"] = y_val["target"]
```

We renamed the columns because we lost the column headers when we did the train-test split in [Part 1](#), so we needed to restore them and add the “target” column.

Next, we append the “prediction” column, naturally, after computing predictions:

```
def get_prediction(data_frame: pd.DataFrame) -> Any:
```

```
    """
```

```
    Turns the dataframe into a numpy array, and computes  
    the predicted value with the model based on input features.
```

```
    """
```

```
    shape = data_frame.shape[0] # 6
```

```
    arr = data_frame.to_numpy()
```

```
    arr = np.reshape(arr, (-1, shape))
```

```
    prediction = loaded_model.predict(arr)[0]
```

```
    return prediction
```

```
# Add 'prediction' column to feature datasets
```

```
x_train["prediction"] = x_train[columns].apply(get_prediction, axis=1)
```

```
x_val["prediction"] = x_val[columns].apply(get_prediction, axis=1)
```

And as we have always done, we save the updated datasets on the local disk and upload them to our S3 bucket:

```
# Save the dataframes to local disk.
```

```
x_train.to_csv(f"{local_path}data/reference.csv")
```

```
x_val.to_csv(f"{local_path}data/current.csv")
```

```
# Put the dataframes in S3 bucket also
```

```
put_object(
```

```
    f"{local_path}data/reference.csv", bucket, "data/reference.csv", "Name", "data"
```

```
)
```

```
put_object(
```

```
    f"{local_path}data/current.csv", bucket, "data/current.csv", "Name", "data"
```

```
)
```

As you may notice, we have saved the train data as reference.csv and the validation data as current.csv. The “reference” dataset is the data used in model training earlier and serves as a baseline or benchmark. The “current” dataset is the latest data. We’ll compare the current data to the reference data to detect if any drift is occurring. We will find out how the model and data quality have changed by the time new data has arrived.

We can prepare both datasets from a single dataset as long as they have an identical schema. An identical schema in both datasets is a must. It is essential that the reference dataset refers to the values used during the earlier training while the current one involves the recent data points. That way allows us to draw interpretable and comparable conclusions. As the [waiter tips data](#) doesn't have any reference-current segmentation, I used the validation set as the "current" dataset pretending it was the new data.

We have prepared our data and can create reports from it.

Step 14: Creating Dashboards

We will create our reports in HTML. Though *Evidently* provides custom reports, built-in dashboards are enough to monitor the data and model performance. While one or two dashboards sufficiently achieve this task, I see no waste in studying four of them. They are Data Drift, Data Quality, Target Drift, and Regression Performance. Dashboards are created for both reference and current datasets side by side. We will look into each of them as applied to our problem:

- **Data Drift:** Detects if input features in the reference and current datasets come from the same probability distribution. Since our reference data is small, less than or equal to 1000 observations, *Evidently* uses a [two-sample Kolmogorov-Smirnov](#) test for numerical features with a number of unique values higher than five.

Drift is detected for 0.00% of features (0 out of 8). Dataset Drift is NOT detected.

| Feature | Type | Reference Distribution | Current Distribution | Data Drift | Stat Test | Drift Score |
|--------------|------|---|---|--------------|--------------------|-------------|
| > target | num |  |  | Not Detected | K-S p_value | 0.144186 |
| > prediction | num |  |  | Not Detected | K-S p_value | 0.642216 |
| > size | num |  |  | Not Detected | K-S p_value | 1 |
| > time | cat |  |  | Not Detected | Z-test p_value | 0.962887 |
| > smoker | cat |  |  | Not Detected | Z-test p_value | 0.746132 |
| > sex | cat |  |  | Not Detected | Z-test p_value | 0.680869 |
| > day | cat |  |  | Not Detected | chi-square p_value | 0.652283 |
| > total_bill | num |  |  | Not Detected | K-S p_value | 0.30684 |

Exhibit-5: Data Drift Dashboard by Evidently (Image by Author)

Four numerical columns, target, prediction, size, and total_bill fit the bill and are subject to the two-sample KS test.

Evidently applies the [Chi-squared test](#) for the categorical columns with five or fewer unique values and the proportion difference test for independent samples based on Z-score for binary categorical features ($n_{\text{unique}} \leq 2$). Time, smoker, and sex are binary categorical features, subject to Z-test. The day column that has four unique values requires a Chi-squared test.

All tests return `p_value` and use a 0.95 confidence level by default. By default, Dataset Drift is detected if at least 50% of features drift[5].

As can be seen in the summary table in Exhibit 5, our data seems to pass the data drift test. Both datasets show highly comparable feature distributions.

- **Data Quality:** The Data Quality report provides detailed feature statistics and a feature behavior overview. It calculates fundamental statistics for features of all types, displays data distribution and feature behavior over time, and provides visualizations of interactions and correlations between features and the target[6].

The Data Quality report includes three widgets: the Summary Widget, Features Widget, and Correlation Widget.

The Summary Widget gives an overview of the dataset, including missing or empty features and other general information. It also shows the share of almost empty and almost constant features. This applies to cases when 95% or more features are missing or constant[6].

| | reference | | current | |
|--------------------------|-----------|--|----------|--|
| | target | | target | |
| target column | target | | target | |
| date column | None | | None | |
| number of variables | 6 | | 6 | |
| number of observations | 182 | | 60 | |
| missing cells | 0 (0.0%) | | 0 (0.0%) | |
| categorical features | 4 | | 4 | |
| numeric features | 2 | | 2 | |
| datetime features | 0 | | 0 | |
| constant features | 0 | | 0 | |
| empty features | 0 | | 0 | |
| almost constant features | 0 | | 0 | |
| almost empty features | 0 | | 0 | |

Exhibit-6: Data Quality Dashboard Summary Widget by Evidently (Image by Author)

The Features Widget resembles pandas' describe function. There are three components. “Feature overview table” shows relevant statistical summaries for each feature based on its type, and plots feature distribution. In Exhibit 7, we can see the statistics for the Day feature.

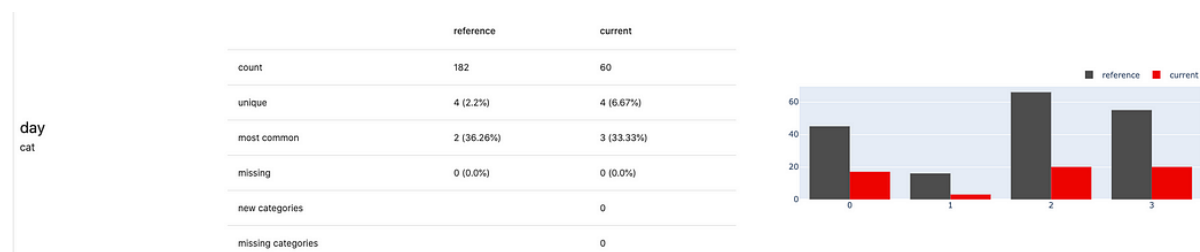


Exhibit-7: Feature Overview Table of Features Widget by Evidently (Image by Author)

“Feature in time” details each feature by including additional visualization to show feature behavior over time. Since we don’t have any datetime feature in our dataset, our report will not display this section.

The last component of the Features Widget is “feature by target” which plots the interaction between a given feature and the target as seen in Exhibit 8.

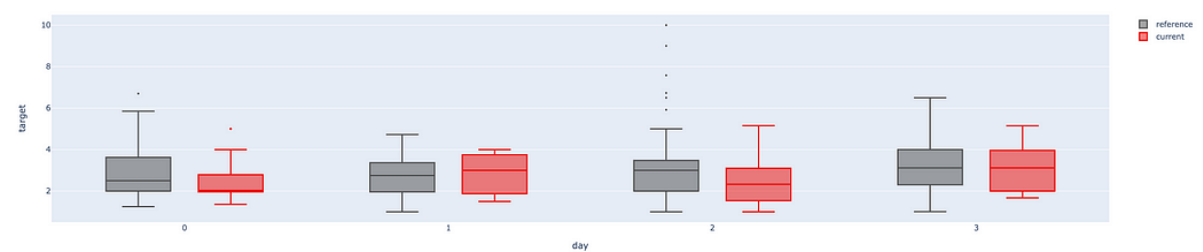


Exhibit-8: Feature by Target Plot of the Features Widget by Evidently (Image by Author)

As the third and last component of the Data Quality report, the Correlation Widget shows the correlations between different features.

For two datasets, it lists the top-5 pairs of variables where correlation changes the most between the reference and current datasets. Similarly, it uses categorical features from Cramer’s V correlation matrix and numerical features from the Spearman correlation matrix[6].

| Correlations | top 5 correlation diff category (Cramer_V) | | | | top 5 correlation diff numerical (Spearman) | | | |
|--------------|--|-----------|------------|------------|---|-----------|------------|------------|
| | | value ref | value curr | difference | | value ref | value curr | difference |
| | day, smoker | 0.308 | 0.398 | -0.09 | size, target | 0.486 | 0.444 | 0.042 |
| | sex, time | 0.191 | 0.262 | -0.071 | size, total_bill | 0.61 | 0.604 | 0.006 |
| | day, time | 0.931 | 1 | -0.069 | - | - | - | - |
| | day, sex | 0.23 | 0.278 | -0.048 | - | - | - | - |
| | smoker, time | 0.048 | 0.095 | -0.047 | - | - | - | - |

Exhibit-9: Data Quality Dashboard Summary of Pairwise Feature Correlations by Evidently (Image by Author)

For example, two categorical values, Day and Smoker have Cramer_V correlations in both datasets, the difference of which is greatest among those of other pairs.

The Correlation Widget also includes four heatmaps. Evidently calculates the Cramer’s V correlation matrix for categorical features and the Pearson, Spearman, and Kendall matrices for numerical features. The matrix will show the target according to its type if it is in the dataset.



Exhibit-10: Correlation Heatmaps of the Correlation Widget by Evidently (Image by Author)

- **Target Drift:** The Target Drift report allows us to detect and visually explore target and prediction drifts. If we include both target and predictions, *Evidently* will generate two sets of plots. If we include only one of them (either target or predictions), *Evidently* will build one set of plots. We have both columns to plot them.

This report has four components: Target (Prediction) Drift, Target (Prediction) Correlations, Target (Prediction) Values, and Target (Prediction) Behavior By Feature[7].

The Target (Prediction) Drift shows the comparison of target (prediction) distributions in the current and reference datasets. You can see the result of the statistical test or the value of a distance metric for the target in Exhibit 11. I won't show the report for predictions as the same procedure applies to them. All statistical tests and algorithms are the same as explained in Data Drift.

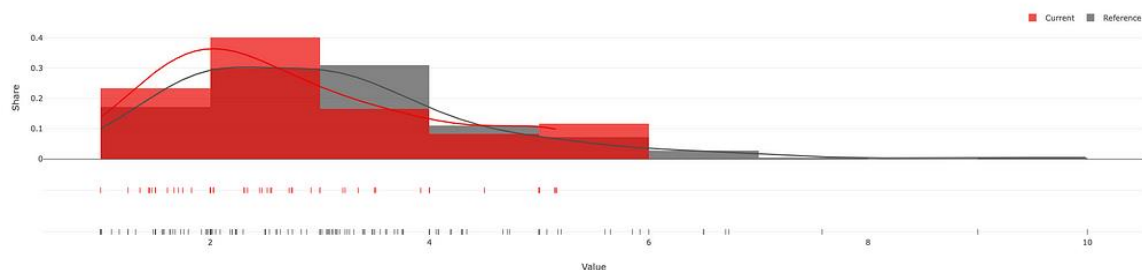


Exhibit-11: The Target Drift by Evidently (Image by Author)

Target (Prediction) Correlations report calculates the Pearson correlation between the target (prediction) and each feature in the reference and current datasets to detect a change in the relationship for numerical targets.



Exhibit-12: The Target Correlations by Evidently (Image by Author)

Target (Prediction) Values report plots the target (prediction) values by index or time (if the datetime column is available or defined in the column_mapping dictionary) for numerical targets. This plot compares the target behavior between the datasets. As we don't have the

datetime column, the report (Exhibit 13) plots the 182 data points of the reference dataset and the 60 data points of the current dataset by index.

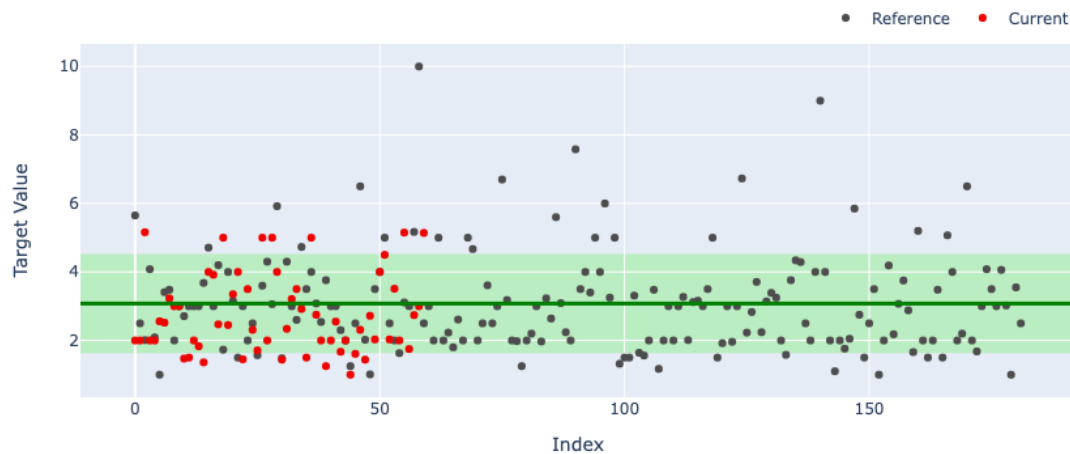


Exhibit-13: The Target Values by Evidently (Image by Author)

Target (Prediction) Behavior By Feature report generates an interactive table with the visualizations of dependencies between the target and each feature. The plot shows how feature values relate to the target (prediction) values and if there are differences between the datasets[7].

Our code didn't produce the Target (Prediction) Behavior By Feature report, however.

- **Regression Performance:** The Regression Performance report evaluates the quality of a regression model. It also compares the performance of the current model to that of the past version or the performance of an alternative model. Regression Performance can compare the two models or work for a single model. It plots performance and errors and helps explore where underestimation and overestimation occur[8].

We need both target and prediction columns and input features as optional to produce this report. Including input features allows us to explore relations between them and the target.

We need the reference and current datasets to analyze the change. In such a comparative report, the reference dataset serves as a benchmark.

The Regression Performance report involves several components. While we briefly go over them, one who is interested in visualizations may refer to the [Regression Performance](#) page, as they are too many to show them here. All metrics and visualizations are presented for both the reference and current datasets. Now, the components are:

1. **Model Quality Summary Metrics:** These are standard model quality metrics such as Mean Error (ME), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE). Each quality metric comes up with one standard deviation of its value (in brackets) to estimate the stability of the performance.

2. **Predicted vs Actual:** A scatter plot of the predicted values against the actual values.
3. **Predicted vs Actual in Time:** A line plot of the predicted and actual values over time or by index, if no datetime is available.
4. **Error (Predicted — Actual):** A line plot of model error values over time or by index, if no datetime is available.
5. **Absolute Percentage Error:** A line plot of absolute percentage error values over time or by index if no datetime is available.
6. **Error Distribution:** Distribution of the model error values.
7. **Error Normality:** Quantile-quantile plot (Q-Q plot) to estimate value normality.
8. **Mean Error per Group:** A summary of the model quality metrics for each of the two segments: Mean Error (ME), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE).
9. **Predicted vs Actual per Group:** A scatter plot that displays the regions where the model underestimates and overestimates the target function.
10. **Error Bias: Mean/Most Common Feature Value per Group:** This table helps see the differences in feature values between the 3 groups:
OVER (top-5% of predictions with overestimation)
UNDER (top-5% of the predictions with underestimation)
MAJORITY (the rest 90%)
 The table shows the mean value per group for the numerical features, and shows the most common value for the categorical features. It also displays the values for both reference and current datasets.

There are two more components according to the documentation but the above tables are those we have for our datasets.

We understand these reports, and now it is time to create them. We first bring the reference and current datasets into the namespace:

```
reference = pd.read_csv(f"{local_path}data/reference.csv")
current = pd.read_csv(f"{local_path}data/current.csv")
```

and then create the dashboards:

```
from evidently.dashboard import Dashboard
from evidently.dashboard.tabs import (
    DataDriftTab,
    DataQualityTab,
    NumTargetDriftTab,
    RegressionPerformanceTab,
)
```

```
def create_evidently_reports() -> list[Any]:
```

```
    """
```

Creates and saves evidently dashboards as html.

"""

Dashboards we need

dashboards = {

 "data_drift_dashboard": DataDriftTab(),

 "data_target_drift_dashboard": NumTargetDriftTab(),

 "regression_model_performance_dashboard": RegressionPerformanceTab(),

 "data_quality_dashboard": DataQualityTab(),

}

Create and save dashboards

for key, value in dashboards.items():

 dashboard = Dashboard(tabs=[value])

 dashboard.calculate(reference, current, column_mapping=column_mapping)

 file_path = f"{local_path}web-flask/templates/{key}.html"

 dashboard.save(file_path)

 put_object(file_path, bucket, f"evidently/reports/{key}.html", "Name", "report")

return list(dashboards.keys())

Here, *Evidently's* Dashboard module helps create all dashboards we mentioned above. The Dashboard instance calculates metrics of each specified dashboard (for example, DataDriftTab()) for reference and current datasets.

After creating the reports, we will save them on the local disk and upload them to the S3 bucket (s3://s3b-tip-predictor/evidently/reports/). We'll create a separate folder in S3 to keep the *Evidently* reports during the *Terraform* deployment.

Viewing dashboards as web pages

Besides keeping dashboards as HTML files on the local machine and cloud, we may want to see them as web pages on the local host. While such an extra piece of work may or may not be necessary, there is nothing wrong with adding it to our reporting repertoire. Let's go with the Data Quality dashboard as an example:

from flask import Flask, render_template

app = Flask(__name__)

@app.route("/data-quality")

def data_quality_report():

 """

 Returns data_quality_dashboard as html

 """

return render_template("data_quality_dashboard.html")

```
if __name__ == "__main__":  
    app.run(debug=True, host="0.0.0.0", port=3600)
```

Thus, if we open the browser at <http://127.0.0.1:3600/data-quality>, we'll see the Data Quality report we generated earlier. Note that we should do port forwarding to open it on our home or work machine as all files and scripts run on EC2. We will see port forwarding later in one of the articles in the series.

One who wants to see the code for all reports on the web may refer to the following file:

Step 15: Sending the Report

We have the reports on our local disk and in the S3 bucket and can see them on the local host web browser. However, we want to send them to our email address also. We'll use AWS SNS for this purpose. However, SNS does not support HTML content while sending notifications to email. That means we cannot email the dashboard reports we have created. Also, there is no point in crowding our email with the same reports that reflect identical content and level of detail. We already have them on our local machine and on S3, ready to see and study. Therefore, we'll produce only the data drift summary and send it to our email as shown in Exhibit 14.

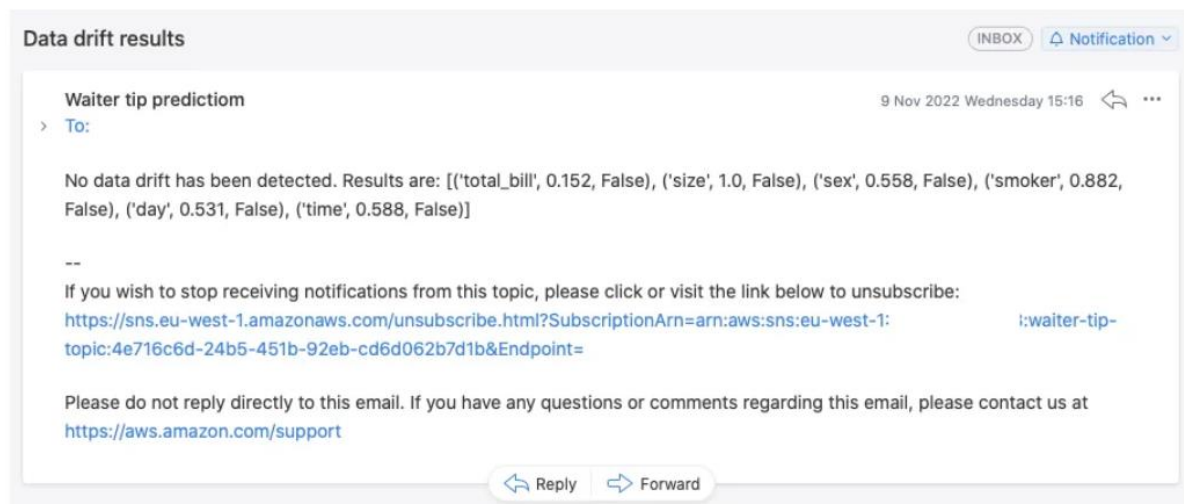


Exhibit-14: Data Drift Summary Results Notified by AWS SNS (Image by Author)

Dashboards are visualizations. In order to produce and send a report like in Exhibit 14, we need to extract the numerical values of data drift. The Profile module of Evidently helps us create and retrieve a profile for any report type in the form of a JSON document that comprises all relevant statistics and numerical values. For example, assuming `X_train` as the reference dataset, and `X_val` as the current dataset, we run the following code to get the subsequent output.

```
from evidently.model_profile import Profile
```

```
from evidently.model_profile.sections import DataDriftProfileSection
```

```
waiter_tip_data_drift_profile = Profile(sections=[DataDriftProfileSection()])
```

```
waiter_tip_data_drift_profile.calculate(X_train, X_val, column_mapping =  
column_mapping)
```

```
waiter_tip_data_drift_profile.json()
```

Output:

```
{  
  "data_drift": {  
    "name": "data_drift", "datetime": "2022-09-11 14:59:55.397685", "data": {  
      "utility_columns": {  
        "date": null, "id": null, "target": "target", "prediction": "prediction",  
        "cat_feature_names": ["sex", "smoker", "day", "time"], "num_feature_names": ["size",  
        "total_bill", "target", "prediction"], "datetime_feature_names": [], "target_names": null,  
        "options": {  
          "confidence": null, "drift_share": 0.5, "nbinsx": 10, "xbins": null, "metrics": {  
            "n_features": 8, "n_drifted_features": 0, "share_drifted_features": 0.0, "dataset_drift":  
            false, "size": {  
              "current_small_hist": [[0.08333333333333333, 0.0, 1.2083333333333333, 0.0,  
              0.3333333333333333, 0.0, 0.2916666666666667, 0.0, 0.04166666666666664,  
              0.04166666666666664], [1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0]], "ref_small_hist":  
              [[0.020618556701030927, 0.0, 1.288659793814433, 0.0, 0.30927835051546393, 0.0,  
              0.30927835051546393, 0.0, 0.041237113402061855, 0.030927835051546393], [1.0, 1.5, 2.0,  
              2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0]], "feature_type": "num", "statstest_name": "K-S p_value",  
              "drift_score": 0.9999999999196367, "drift_detected": false}, "total_bill": {  
              "current_small_hist": [[0.009238728750923873, 0.06005173688100518,  
              0.04619364375461938, 0.05081300813008128, 0.02771618625277163,  
              0.00923872875092387, 0.0, 0.004619364375461938, 0.009238728750923877,  
              0.0046193643754619314], [3.07, 7.58, 12.09, 16.599999999999998, 21.11,  
              25.619999999999997, 30.13, 34.64, 39.15, 43.66, 48.17]], "ref_small_hist":  
              [[0.016015301476610795, 0.048045904429832385, 0.05948540548455438,  
              0.03546245326963819, 0.024022952214916193, 0.017159251582082993,  
              0.010295550949249797, 0.005719750527360999, 0.0022879002109443994,  
              0.003431850316416599], [5.75, 10.256, 14.762, 19.268, 23.774, 28.28, 32.786, 37.292,  
              41.798, 46.304, 50.81]], "feature_type": "num", "statstest_name": "K-S p_value",  
              "drift_score": 0.15224779225692237, "drift_detected": false}, "target": {  
              "current_small_hist": [[0.15024038461538464, 0.5008012820512818, 0.4507211538461537,  
              0.4006410256410257, 0.20032051282051286, 0.1502403846153845, 0.10016025641025643,  
              0.20032051282051286, 0.0, 0.2504006410256408], [1.0, 1.416, 1.832, 2.248, 2.664, 3.08,  
              3.4960000000000004, 3.9120000000000004, 4.328, 4.744, 5.16]], "ref_small_hist":  
              [[0.16036655211912945, 0.3436426116838488, 0.3264604810996562,  
              0.13172966781214213, 0.08018327605956468, 0.028636884306987388,  
              0.022909507445589936, 0.005727376861397484, 0.005727376861397478,  
              0.005727376861397478], [1.0, 1.9, 2.8, 3.7, 4.6, 5.5, 6.4, 7.3, 8.2, 9.1, 10.0]], "feature_type":  
              "num", "statstest_name": "K-S p_value", "drift_score": 0.0717575921409452,  
              "drift_detected": false}, "prediction": {  
              "current_small_hist": [[0.9030840950211592, 0.3612338868980647, 0.12041121266948789, 0.3010280316737197, 0.42143924434320756,  
              0.18061694344903234, 0.3612338868980647, 0.0, 0.0, 0.24082242533897577],  
              [1.6889560222625732, 2.0349924564361572, 2.381028652191162, 2.727065086364746,
```

```

3.07310152053833, 3.419137954711914, 3.765174150466919, 4.111210346221924,
4.457246780395508, 4.803283214569092, 5.149319648742676]], "ref_small_hist":
[[0.5660568348173863, 0.32771734069102776, 0.25323595241830443,
0.3426133473894707, 0.3575095798846651, 0.2830286124149785, 0.2532361268976124,
0.10427362746636064, 0.20854725493272128, 0.1936510224375269],
[1.6889560222625732, 2.0349924564361572, 2.381028652191162, 2.727065086364746,
3.07310152053833, 3.419137954711914, 3.765174150466919, 4.111210346221924,
4.457246780395508, 4.803283214569092, 5.149319648742676]], "feature_type": "num",
"statstest_name": "K-S p_value", "drift_score": 0.3342798027210424, "drift_detected":
false}, "sex": {"current_small_hist": [[19, 29], [0, 1]], "ref_small_hist": [[68, 126], [0, 1]],
"feature_type": "cat", "statstest_name": "Z-test p_value", "drift_score":
0.5579883407675066, "drift_detected": false}, "smoker": {"current_small_hist": [[30, 18], [0,
1]], "ref_small_hist": [[119, 75], [0, 1]], "feature_type": "cat", "statstest_name": "Z-test
p_value", "drift_score": 0.8824197605105635, "drift_detected": false}, "day":
{"current_small_hist": [[15, 2, 16, 15], [0, 1, 2, 3]], "ref_small_hist": [[47, 17, 69, 61], [0, 1, 2,
3]], "feature_type": "cat", "statstest_name": "chi-square p_value", "drift_score":
0.5314874902340183, "drift_detected": false}, "time": {"current_small_hist": [[15, 33], [0,
1]], "ref_small_hist": [[53, 141], [0, 1]], "feature_type": "cat", "statstest_name": "Z-test
p_value", "drift_score": 0.5875270459230384, "drift_detected": false}}}], "timestamp":
"2022-09-11 14:59:55.397867"}'

```

As you see, it has rendered us an output, something long as The Odyssey by Homer! We need the “drift score” (float value) and “whether there is a drift or not” (boolean value) for every numerical and categorical feature. We accomplish it by `evaluate_data_drift`:

```
import json
```

```
def evaluate_data_drift(ti: Any) -> List[tuple[str, float, Literal[True, False]]]:
```

```
    """
```

```
    Evaluates data drifts and checks how many of them exist if any.
```

```
    """
```

```
    # Create and save the data drift profile as json
```

```
    data_drift_profile = Profile(sections=[DataDriftProfileSection()])
```

```
    data_drift_profile.calculate(reference, current, column_mapping=column_mapping)
```

```
    report = data_drift_profile.json()
```

```
# Convert to python dictionary
```

```
report = json.loads(report)
```

```
logging.info("Data drift profile is created and stored as a python dictionary.")
```

```
# Store features and their drift scores, and flag any data drift
```

```
drifts = []
```

```
flag = 0
```

```
for feature in (
```

```
    column_mapping.numerical_features + column_mapping.categorical_features
```

```
):
```

```
    drift_score = report["data_drift"]["data"]["metrics"][feature]["drift_score"]
```

```
    drift_detected = report["data_drift"]["data"]["metrics"][feature][
```

```
        "drift_detected"
```

```
    ]
```

```
    drifts.append((feature, round(drift_score, 3), drift_detected))
```

```
    if drift_detected is True:
```

```
        flag += 1
```

```
logging.info("%s data drift(s) is/are detected.", flag)
```

```
# Send email about data drift by AWS SNS
```

```
send_email(flag, drifts)
```

```
# Push the results to XCom
```

```
ti.xcom_push(key="drifts", value=drifts)
```

```
logging.info("Data drift evaluation results '%s' are pushed to XCom.", drifts)
```

If drift occurs, `drift_detected` becomes `True`, or `False`. Once we get all information we need, we'll store it in a list of tuples as `[(feature, drift_score, drift_detected)]` to send to email (see Exhibit 14). We also keep counts of the drifts detected with the `flag` variable. Then, we push the list to Airflow XCom (Don't worry about this for now!).

The next step is to send the notification to the subscriber email:


```

def send_email(
    flag: int, drifts: list[tuple[str, float, Literal[True, False]]]
) -> None:

    """
    Sends data drift results via AWS SNS.
    """

    # AWS SNS arguments
    topic_arn = get_parameter("sns_topic_arn")
    subject = "Data drift results"

    # SNS message depending on whether data drift exists
    if flag == 0:
        message = f"No data drift has been detected. Results are: {drifts}"
    else:
        message = f"{flag} data drift(s) is/are detected. Results are: {drifts}"

    # Send the notification
    send_sns_topic_message(topic_arn, message, subject)

    logging.info(
        "Email about the data drift results are sent to AWS SNS topic 'WaiterTipTopic'"
    )

```

Are we done? Not quite yet!

Step 16: Tracking Evidently Metrics on MLflow

Our pipeline is programmed to run monthly (any period could be set though), so we will collect Evidently statistics monthly. As step 16 is concerned, these statistics consist of only data drift values. We won't cover the other reports for the reasons we mentioned in

the previous section. That is a personal preference, and of course, you may decide on the extent of distribution and production of reports at any stage.

For our case, what could be a better option than MLflow to track and store these statistic records? We only need to create a new experiment on MLflow and a new database in RDS. One might argue about launching another RDS for this purpose. However, as I have a free-tier and Scrooge McDuck mindset regarding using cloud resources, I'll have a single RDS serve all tracking operations for this project. We'll see how to create more databases on the same RDS later in the AWS part of the series.

Before running `record_metrics` below, we should first start the MLflow server for Evidently and only after that, execute the function.

```
mlflow server -h 0.0.0.0 -p 5500 --backend-store-uri
postgresql://DB_USER:DB_PASSWORD@DB_ENDPOINT:5432/DB_NAME --default-artifact-
root s3://S3_BUCKET_NAME
```

We define `DB_USER` and `DB_PASSWORD` during the creation of the new database. `DB_ENDPOINT:5432` should be the endpoint of the RDS that is currently running. We'll define the `DB_NAME` argument as well; it could be something like `evidently`. `default-artifact-root` refers to our S3 bucket and key, which we saw earlier: `s3://s3b-tip-predictor/evidently/`.

Now, we can run the experiment for Evidently:

```
def record_metrics(ti: Any, tag: str) -> None:
```

```
    """
```

```
    Records data drift evaluation results in MLFlow,
    allowing them to be displayed on Grafana.
```

```
    """
```

```
    from datetime import datetime
```

```
    # Retrieve variables
```

```
    _, _, _, _, experiment_name, _ = get_vars()
```

```
# We store variables that won't change often in AWS Parameter Store.

tracking_server_host = get_parameter(

    "tracking_server_host"

) # This can be local: 127.0.0.1 or EC2, e.g.: ec2-54-75-5-9.eu-west-
1.compute.amazonaws.com.


# Set the tracking server uri

evidently_port = 5500

evidently_tracking_uri = f"http://{tracking_server_host}:{evidently_port}"

mlflow.set_tracking_uri(evidently_tracking_uri)


# Create an mlflow experiment

mlflow.set_experiment(experiment_name)


# Get the data drift evaluation results

metrics = ti.xcom_pull(key="drifts", task_ids=["evaluate_data_drift"])

metrics = metrics[0]

logging.info("Data drift metrics '%s' are retrieved from XCom.", metrics)


now = datetime.now()

date_time = now.strftime("%Y-%m-%d / %H-%M-%S")

logging.info("Date/time '%s' is set.", date_time)


with mlflow.start_run() as run:


    # Set a tag

    mlflow.set_tag("model", tag)


    # Log parameters

    mlflow.log_param("record_date", date_time)
```

for feature in metrics:

```
mlflow.log_metric(feature[0], round(feature[1], 3), feature[2])
```

```
logging.info("MLflow run: %s", run.info)
```

We retrieve the experiment name from Airflow Variables and create a new experiment. We define the experiment name while running Airflow dag (we'll see Airflow later in the series), and that is evidently-experiment-1. While launching the MLflow server for XGBoost training, we set the port as 5000. So this time, we should use another port for Evidently records, for example, 5500. If you need to know more about how to start and run an experiment on MLflow, you can check out Part 1.

Recall that in the previous section, we stored the data drift results in a list of tuples named drifts and pushed it to Airflow XCom. Now, we bring it back from there with xcom_pull. It looks like in the following and will be assigned to themetrics variable:

```
[('total_bill', 0.152, False),
```

```
('size', 1.0, False),
```

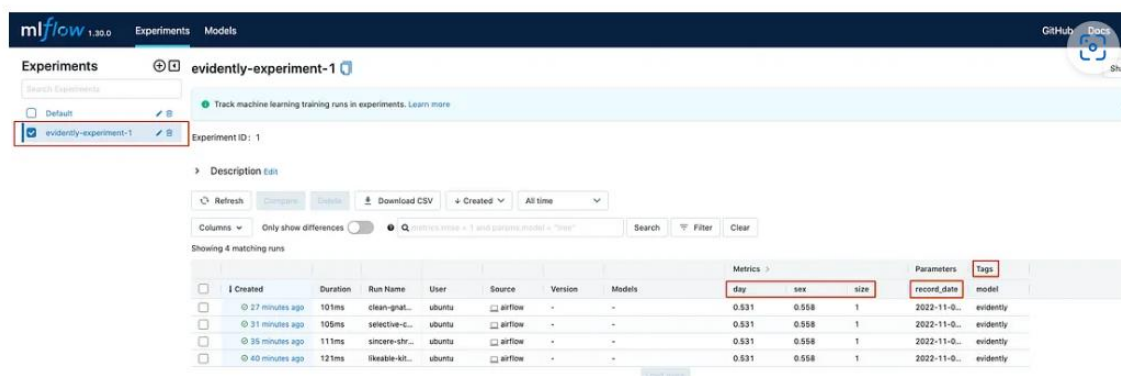
```
('sex', 0.558, False),
```

```
('smoker', 0.882, False),
```

```
('day', 0.531, False),
```

```
('time', 0.588, False)]
```

MLflow sets the tag (evidently) and logs the current datetime and the metrics: feature, drift_score, and drift_detected as in Exhibit 15.



| Created | Duration | Run Name | User | Source | Version | Models | Metrics | Parameters | Tags |
|----------------|----------|-----------------|--------|---------|---------|--------|---------------------------------|---------------------------|-----------|
| 27 minutes ago | 101ms | clean-gnat... | ubuntu | airflow | - | - | day: 0.531, sex: 0.558, size: 1 | record_date: 2022-11-0... | evidently |
| 31 minutes ago | 105ms | selective-c... | ubuntu | airflow | - | - | day: 0.531, sex: 0.558, size: 1 | record_date: 2022-11-0... | evidently |
| 35 minutes ago | 111ms | sincere-str... | ubuntu | airflow | - | - | day: 0.531, sex: 0.558, size: 1 | record_date: 2022-11-0... | evidently |
| 40 minutes ago | 121ms | likeable-kit... | ubuntu | airflow | - | - | day: 0.531, sex: 0.558, size: 1 | record_date: 2022-11-0... | evidently |

Exhibit-15: Evidently Data Drift Records on MLflow (Image by Author)

Another thing we might do is to build live dashboards and monitor the performance in real-time. For this, additional tools such as Prometheus and Grafana come into play.

Part 3: Collecting, Storing, and Visualizing ML Model and Data Performance Metrics with Prometheus and Grafana

We have seen different ways to produce and present reports for ML model quality and data quality metrics with *Evidently*. HTML dashboards, web page reports, and tracking and notifying metrics were the alternatives we saw. In addition, users may also want to monitor and visualize metrics in real time. To address the issue, we need to add two more tools to our MLOps arsenal: *Prometheus* and *Grafana*.

When we talk about real-time data, we usually mean streaming data, which frequently updates records. However, our data relies on a batch model and gets trained once a month. Not only after a few months can we interpret the visualization of data points, but perhaps we won't be much excited in doing so as the data plot doesn't change quite often. Anyway. This article can still provide us with some insights into these two tools we may plan to use for future projects. Our [project](#) that Exhibit-1 shows is to productionize the ML model that predicts waiter tips.

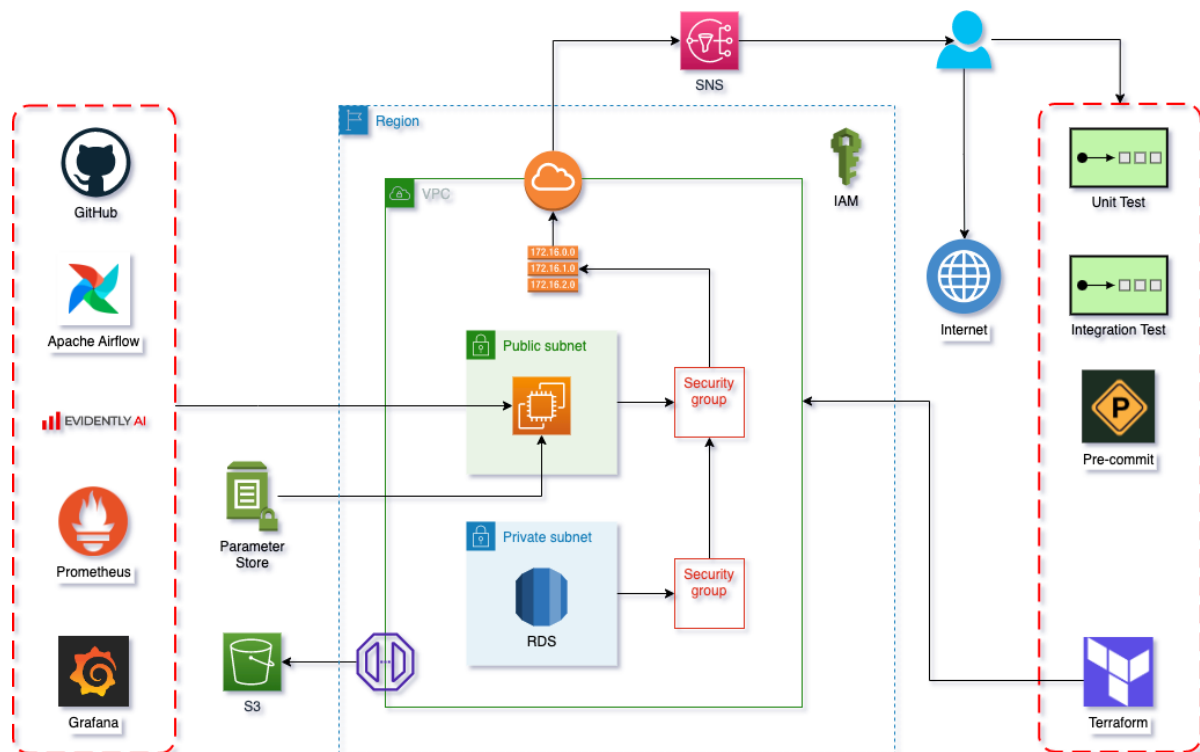


Exhibit-1: MLOps Project Diagram

we have finished all tasks, including 16, and thus will continue with real-time monitoring, as shown in Exhibit 2.

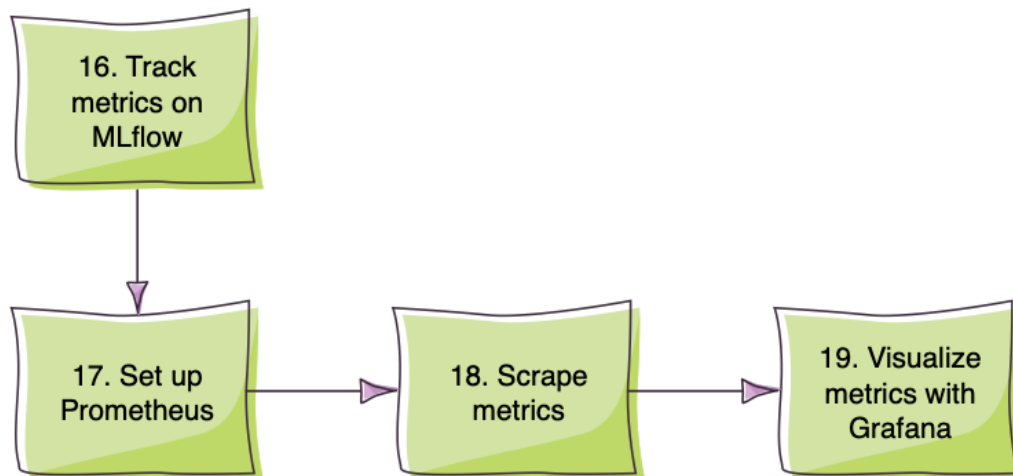


Exhibit-2: Prometheus and Grafana Parts of the Flow of Logic

We may start our journey with the fire of *Prometheus*. Let there be light! Metrics are any numeric measurements recorded over time and differ from application to application. In our case, the metrics are computed and provided by *Evidently*.

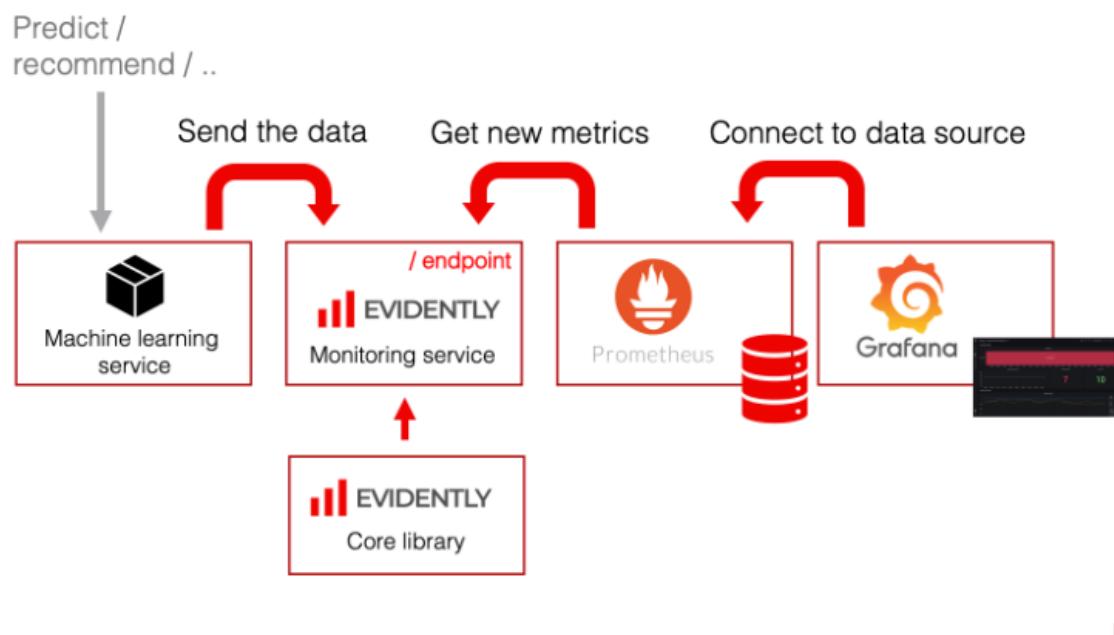


Exhibit-3: High-level Overview of the Integration Architecture

Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit that collects and stores its metrics as time series data. It stores metrics information with the name and the timestamp at which it was recorded, alongside optional key-value pairs, called labels[1]. In short, *Prometheus* is a time series database.

We saved train data as the “reference” and validation data as the “current” dataset. Reference dataset serves as a benchmark or baseline. The “current” dataset is the latest data we receive from the application.

We use reference and current datasets and treat them as model application logs. Then, we configure *Evidently*’s monitoring service (with the `model_monitoring` library) to read the data from the logs at a particular interval (to simulate production service where data appears sequentially). *Evidently* calculates the metrics for Data Drift, Target Drift, Data Quality, and Regression Performance, and sends them to *Prometheus*. `model_monitoring` library calculates the same metrics as other libraries, Dashboard and Profile. But, it is the right choice to make the metrics available to *Prometheus* for real-time monitoring. In turn, *Prometheus* records and stores them with timestamps. Then, *Grafana* displays the metrics on pre-built dashboards[2].

Prometheus performs the tasks through the architecture depicted in Exhibit 4.

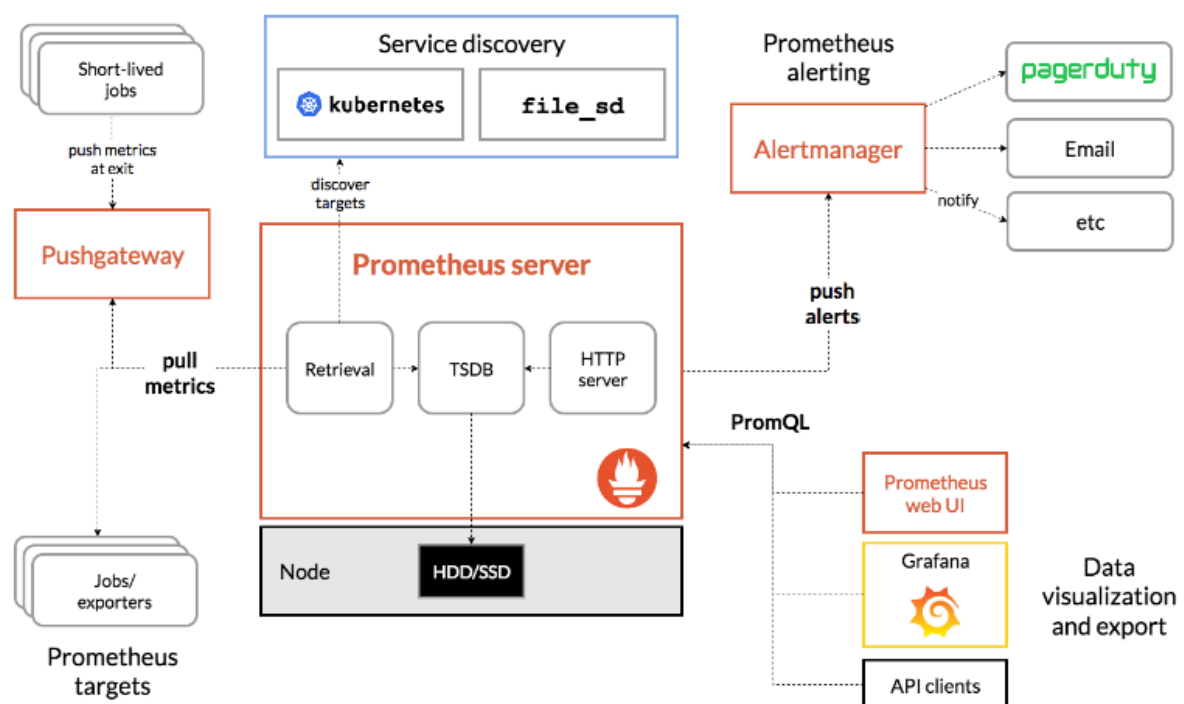


Exhibit 4: The Architecture of Prometheus and Some of its Ecosystem Components

Liberty of *Prometheus* allows monitoring of anything. It could be a database, server, or application, each of which we call a “target.” Each target has specific metrics to be monitored. For example, a server is the target, and CPU utilization can be a metric. If we target a database, latency can be one of the metrics. A prominent advantage of *Prometheus* is that it uses a pull system instead of a push system. That means we decide which targets, metrics, and intervals to employ and don’t have to allow all targets to push all their metrics.

In Exhibit 4, *Prometheus* targets consist of jobs/exporters. They provide the metrics to the *Prometheus* server. Users can find exporters on the *Prometheus* [website](#) to download them on their machines. Exporters know what metrics to follow and provide. You can customize them by enabling or disabling collectors. However, we are not going to use and install those packaged

exporters. Instead, we need *Evidently*, through its monitoring service (model_monitoring), to get the specified metrics from the target, which is our application.

Prometheus server's first component, the Retrieval unit, scrapes the metrics from *Evidently*'s monitoring service over the endpoint <http://127.0.0.1:9091/metrics>. *Evidently* exposes the endpoint with the port we defined (here, it is 9091, but any other port number is feasible as long as no conflict occurs). *Prometheus* natively uses port 9090 to monitor itself, and we dedicate another port number to our target. The Retrieval worker then pushes the metrics into and stores them in the TSDB, which stands for Time Series Data Base. *Prometheus* stores data locally or remotely. We'll go with the first option.

HTTP server accepts queries for the TSDB with PromQL, the native querying language, and displays the metrics on the Web UI. In other words, the Web UI asks the HTTP server via PromQL to show the data on the web. *Prometheus* also integrates with *Grafana* to benefit from its excellent data visualization and presentation capability. *Grafana* uses PromQL in the background to bring the data from *Prometheus*.

We can interact with *Prometheus* using their PromQL language or one of their client libraries. Python is one of the API clients that *Prometheus* support[3]. Python client library of *Prometheus* lets us define and expose metrics via an HTTP endpoint on our application's instance.

Prometheus alerting enables the firing of alerts based on pre-defined rules and notifying receivers. For example, an alert can be sent to a receiver when the web page request latency exceeds 0.5 seconds for 10 minutes. Setting up such an alert seems more suitable for streaming data metrics, but I do not think it would make an enormous difference for our application. Instead, notifications from *AWS SNS* can fulfill the alerting task for our batch model to a sufficient extent. If your application needs to be aware of metrics frequently, such as memory usage, CPU utilization, etc., then *Prometheus* alerting can be the right choice. CloudWatch is an alternative with the limitations such as consecration to AWS and cost.

Step 17: Setting Up and Starting Prometheus

From the information above, we need two main components to install: the *Prometheus* server and *Prometheus* Python Client. These two installations will take place on our EC2.

Install the server:

```
brew install prometheus
```

Start the service:

```
brew services restart prometheus
```

Install the *Prometheus* Python Client:

```
pip install prometheus-client
```

We can check if the *Prometheus* server is running:

```
lsof -i tcp:9090
```

We have said *Evidently*'s monitoring service is the target, and the Retrieval worker will scrape the metrics there over the endpoint, <http://127.0.0.1:9091/metrics>. That is the local host,

though. Therefore, we need to define the target, or *Prometheus* would not know from where and when to pull metrics over that endpoint. After the installation, we should open and edit the `prometheus.yml` file. We can access and open the file as follows:

```
nano /home/linuxbrew/.linuxbrew/etc/prometheus.yml
```

The file content is mainly as below[4]:

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  # - "first.rules"
  # - "second.rules"

scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ['localhost:9090']
```

The configuration file (`prometheus.yml`) has three blocks: `global`, `rule_files`, and `scrape_configs`.

The `global` block declares the configuration options which are applicable to all targets specified. The first, `scrape_interval`, sets how frequently *Prometheus* will pull metrics from targets. In the above case, it will scrape targets every 15 seconds. The second, `evaluation_interval`, states how often *Prometheus* will evaluate rules. We use rules to create new time series and condition-based alerts. We can override the global options for any job by restating them in the `scrape_configs` block under the `job_name`.

The `rule_files` block specifies the location of rules that the *Prometheus* server will load. For the moment, we haven't defined any rules.

The block, `scrape_configs`, states targets *Prometheus* should monitor. *Prometheus*, as said earlier, monitors its own health by exposing data about itself over an HTTP endpoint to scrape.

In the default configuration there is a single job, called prometheus, which scrapes the time series data exposed by the Prometheus server. The job contains a single, statically configured, target, the localhost on port 9090. Prometheus expects metrics to be available on targets on a path of /metrics. So this default job is scraping via the URL: <http://localhost:9090/metrics>[4].

Now, we need to add another HTTP endpoint as a target to scrape metrics from *Evidently*:

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: "prometheus"
    static_configs:
      - targets: ["localhost:9090"]
  - job_name: "waiter-tip-prediction"
```

```
static_configs:  
- targets: ["localhost:9091"]
```

The new target (waiter-tip-prediction) has the local host endpoint at port 9091. You can put any port number as long as it does not conflict with those of other services.

Since both *Evidently* and *Prometheus* are installed and will run on the same machine (EC2) instance, which we use as the local host, we should expect the metrics to be available at <http://localhost:9091/metrics>. If *Evidently* were running on a different machine, we would need to define the target in the configuration file as something like ["54.75.5.9:9091"] and scrape the metrics at <http://54.75.5.9:9091/metrics>. In the latter case, the security group of the *Evidently* server should allow the traffic on 9091. Let us keep things simple and stick to the original configuration for now.

Next, we restart the *Prometheus* server:

```
brew services restart prometheus
```

Any time we change the *prometheus.yml* file, we need to execute the above command.

On which machine the *Prometheus* server we installed, we can open its web UI on the browser at the URL http://IP_address:9090 where the IP address belongs to the *Prometheus* server. Installing *Prometheus* on EC2, using EC2 as the local host, and doing port forwarding, we can access its Web UI from our machine at home or work with <http://localhost:9090>.

Exhibit 5 shows the Web UI.

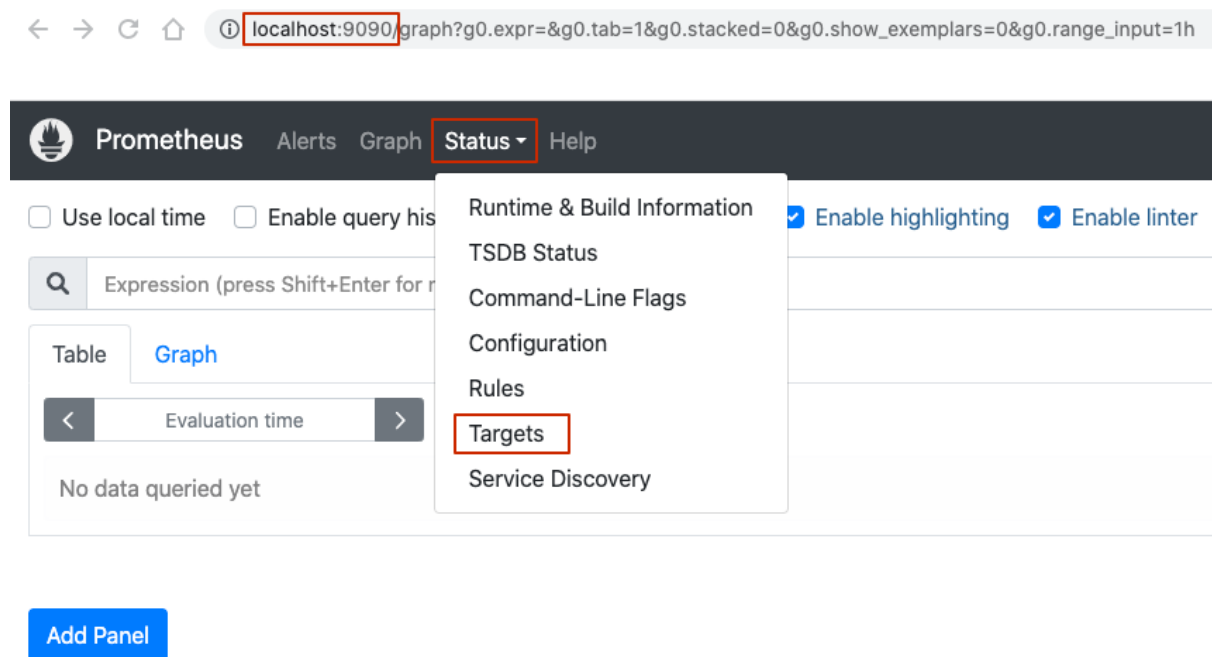


Exhibit-5: Prometheus Web UI (Image by Author)

On the opening page, we can navigate the targets, metrics, and other links of concern.

The screenshot shows the Prometheus web interface at localhost:9090/targets. It displays two target groups. The first group, 'prometheus (1/1 up)', has one target 'http://localhost:9090/metrics' which is 'UP'. The second group, 'waiter-tip-prediction (0/1 up)', has one target 'http://localhost:9091/metrics' which is 'DOWN' with an error message: 'Get "http://localhost:9091/metrics": context deadline exceeded'.

| Endpoint | State | Labels | Last Scrape | Scrape Duration | Error |
|-------------------------------|-------|---|-------------|-----------------|--|
| http://localhost:9090/metrics | UP | instance="localhost:9090" job="prometheus" | 1.712s ago | 3.513ms | |
| http://localhost:9091/metrics | DOWN | instance="localhost:9091" job="waiter-tip-prediction" | 22.593s ago | 10.35s | Get "http://localhost:9091/metrics": context deadline exceeded |

Exhibit-6: Prometheus Targets (Image by Author)

Our server at 9090 is up but down at 9091. Both endpoints worked on my Mac flawlessly, but the second endpoint didn't respond on Linux. Installing *Prometheus* in a container environment should solve this problem. Containerization of the application is the best solution, yet doing so for *Prometheus* would require me to change the whole architecture. It is a helpful note to keep in mind for the future.

Before deeper navigation, we need to populate the *Prometheus* storage with data. So, the next step is defining and pulling metrics from our target.

Step 18: Defining and Scraping Metrics

Now, we can start scraping data.

```
import logging
from typing import Any, Dict
```

First, we bring our reference and current datasets from the local disk:

```
import pandas as pd
```

```
local_path = "../../"
```

```
reference = pd.read_csv(f"{local_path}data/reference.csv")
current = pd.read_csv(f"{local_path}data/current.csv")
```

Second, we let *Evidently* monitor the metrics (these are the same metrics we defined in the previous article, there is no change):

```
from evidently.model_monitoring import (
    DataDriftMonitor,
    DataQualityMonitor,
    ModelMonitoring,
    NumTargetDriftMonitor,
    RegressionPerformanceMonitor,
)
from evidently.pipeline.column_mapping import ColumnMapping
```

```

# Monitoring program
evidently_monitoring = ModelMonitoring(
    monitors=[
        NumTargetDriftMonitor(),
        DataDriftMonitor(),
        RegressionPerformanceMonitor(),
        DataQualityMonitor(),
    ],
    options=None,
)

# Monitoring results
evidently_monitoring.execute(
    reference_data=reference, current_data=current, column_mapping=column_mapping
)
results = evidently_monitoring.metrics()
logging.info("Data metrics were found by Evidently.")

```

We saw the column mapping in Part 2. As you may notice, we get the same data as we did before. The truncated output shows the results produced by the above script:

```

num_target_drift:count / 194 / {'dataset': 'reference'}
num_target_drift:count / 48 / {'dataset': 'current'}
num_target_drift:drift / 0.3342798027210424 / {'kind': 'prediction'}
num_target_drift:reference_correlations / 0.6696378161594713 / {'feature': 'size',
'feature_type': 'num', 'kind': 'prediction'}
num_target_drift:reference_correlations / 0.9098400196021671 / {'feature': 'total_bill',
'feature_type': 'num', 'kind': 'prediction'}
num_target_drift:reference_correlations / 1.0 / {'feature': 'prediction', 'feature_type': 'num',
'kind': 'prediction'}
num_target_drift:current_correlations / 0.7781020941928385 / {'feature': 'size', 'feature_type':
'num', 'kind': 'prediction'}
num_target_drift:current_correlations / 0.927607262610768 / {'feature': 'total_bill',
'feature_type': 'num', 'kind': 'prediction'}
num_target_drift:current_correlations / 1.0 / {'feature': 'prediction', 'feature_type': 'num', 'kind':
'prediction'}
num_target_drift:drift / 0.0717575921409452 / {'kind': 'target'}
num_target_drift:reference_correlations / 0.48977362420792425 / {'feature': 'size',
'feature_type': 'num', 'kind': 'target'}
num_target_drift:reference_correlations / 0.656591531992239 / {'feature': 'total_bill',
'feature_type': 'num', 'kind': 'target'}
num_target_drift:reference_correlations / 1.0 / {'feature': 'target', 'feature_type': 'num', 'kind':
'target'}
num_target_drift:current_correlations / 0.5291799405888153 / {'feature': 'size', 'feature_type':
'num', 'kind': 'target'}
num_target_drift:current_correlations / 0.77115420517272 / {'feature': 'total_bill',
'feature_type': 'num', 'kind': 'target'}
num_target_drift:current_correlations / 1.0 / {'feature': 'target', 'feature_type': 'num', 'kind':
'target'}

```

All metrics in the output should look familiar, except that they are not in the HTML format. You might notice that the values differ from those in the previous article because I tried a different train-test split ratio before generating this report. The output format allows us to send the metrics to the *Prometheus* server.

The output tells us that there are 194 observations in the reference dataset and 48 observations in the current dataset. The prediction has a data drift score of 0.3342798027210424. Total_bill, a numerical type, has a correlation score of 0.927607262610768 with the prediction in the current data set. And so on.

However, *Prometheus* doesn't understand *Evidently's* output. We need to define and expose it via an HTTP endpoint so that *Prometheus* understands it. Python client library of *Prometheus* here helps us.

```
import prometheus_client

data_metrics = {}
registry = prometheus_client.CollectorRegistry()

for i, (metric, value, labels) in enumerate(results, start=1):

    if labels:
        label = "_".join(list(labels.values()))
    else:
        label = "na"

    metric_key = f"evidently:{metric.name}:{label}"
    prom_metric = prometheus_client.Gauge(metric_key, "", registry=registry)
    prom_metric.set(value)
    data_metrics[f"evidently_{i}"] = prom_metric

logging.info(
    "Evidently data metrics were translated to Prometheus for querying and \
    displaying them on Prometheus and Grafana."
)
```

CollectorRegistry helps return metrics in a format *Prometheus* supports. We first create a variable, metric_key , and fill it with the elements of *Evidently's* output. The metric_key has three components: metric, value, and label. For example, the first line in the output is:

```
num_target_drift:count / 194 / {'dataset': 'reference'}
```

The output follows the format of metric name/value/label. Here, the metric name is num_target_drift:count; the value is 194; and the label is {'dataset': 'reference'}.

We check if the label is not null. If it is null, we name it "na." Non-null labels are dictionaries. Some dictionaries have more than one key-value pair. We convert dictionary values into a list and create a single component by uniting the list elements with an underscore. That is how we compose the label part.

We concatenate the metric name and label, and prepend evidently to it. For example, the first output line will be re-defined as follows:

```
evidently:num_target_drift:count:reference
```

Every metric must be unique and follow a particular convention:

The metric name specifies the general feature of a system that is measured (e.g. `http_requests_total` — the total number of HTTP requests received). It may contain ASCII letters and digits, as well as underscores and colons. It must match the regex `[a-zA-Z_][a-zA-Z0-9_]`*

Labels enable Prometheus's dimensional data model: any given combination of labels for the same metric name identifies a particular dimensional instantiation of that metric (for example: all HTTP requests that used the method POST to the `/api/tracks` handler). The query language allows filtering and aggregation based on these dimensions. Label names may contain ASCII letters, numbers, as well as underscores. They must match the regex `[a-zA-Z_][a-zA-Z0-9_]` [5].*

We have many metrics that share the same name but have different labels. Using labels enables discrimination between the metrics and the prevention of collision. To satisfy the conditions for uniqueness and generality, we should concatenate them to create a new metric name.

We should be careful with metric names since *Prometheus* may render errors during runtime. We need to pay attention to how an application outputs metrics and preclude any conflict between the naming conventions of the application and *Prometheus*. Figuring out a correct renaming procedure will guard against possible naming errors.

Python client and CollectorRegistry convert the renamed metric into the format *Prometheus* supports. The metric that has the supported format is defined as `prom_metric` and is of gauge type. A gauge metric is a single numerical value that can go up and down. For example, temperatures, current memory usage, or the number of requests are typical gauges as they can go up or down. Other metric types are counter, histogram, and summary. You may refer to the documentation for more information[6]. Our application uses gauges only.

We set the metric's value (`prom_metric`) with the `set` method. That assigns a numerical value to the metric variable. The metric that *Prometheus* understands and uses has three pieces of information: metric name (after we renamed it), metric type (gauge), and value. It should look as follows:

```
gauge:evidently:num_target_drift:count:reference
```

After describing all metrics, we store them as values in the dictionary called `data_metrics`. Keys of the dictionary are defined as `evidently_{i}` where `i` is the index of the related metric:

```
evidently_1 : gauge:evidently:num_target_drift:count:reference
evidently_2 : gauge:evidently:num_target_drift:count:current
evidently_3 : gauge:evidently:num_target_drift:drift:prediction
evidently_4 : gauge:evidently:num_target_drift:reference_correlations:size_num_prediction
evidently_5 :
gauge:evidently:num_target_drift:reference_correlations:total_bill_num_prediction
evidently_6 :
gauge:evidently:num_target_drift:reference_correlations:prediction_num_prediction
```

```
evidently_7 : gauge:evidently:num_target_drift:current_correlations:size_num_prediction
evidently_8 : gauge:evidently:num_target_drift:current_correlations:total_bill_num_prediction
evidently_9 : gauge:evidently:num_target_drift:current_correlations:prediction_num_prediction
```

Now, we can put all snippets together:

```
import logging
from typing import Any, Dict

import pandas as pd
import prometheus_client
from evidently.model_monitoring import (
    DataDriftMonitor,
    DataQualityMonitor,
    ModelMonitoring,
    NumTargetDriftMonitor,
    RegressionPerformanceMonitor,
)
from evidently.pipeline.column_mapping import ColumnMapping

# Define the column mapping
column_mapping = ColumnMapping()
column_mapping.target = (
    "target" # 'target' is the name of the column with the target function
)
column_mapping.prediction = (
    "prediction" # 'prediction' is the name of the column(s) with model predictions
)
column_mapping.id = None # There is no ID column in the dataset

column_mapping.numerical_features = ["total_bill", "size"] # List of numerical features
column_mapping.categorical_features = [
    "sex",
    "smoker",
    "day",
    "time",
] # List of categorical features

def monitor_evidently() -> Dict[str, Any]:

    """
    Initiates Evidently monitoring, and computes the data metrics.
    """

    local_path = "../../"
```

```

reference = pd.read_csv(f"{local_path}data/reference.csv")
current = pd.read_csv(f"{local_path}data/current.csv")

data_metrics = {}
registry = prometheus_client.CollectorRegistry()

# Monitoring program
evidently_monitoring = ModelMonitoring(
    monitors=[
        NumTargetDriftMonitor(),
        DataDriftMonitor(),
        RegressionPerformanceMonitor(),
        DataQualityMonitor(),
    ],
    options=None,
)

# Monitoring results
evidently_monitoring.execute(
    reference_data=reference, current_data=current, column_mapping=column_mapping
)
results = evidently_monitoring.metrics()
logging.info("Data metrics were found by Evidently.")

for i, (metric, value, labels) in enumerate(results, start=1):

    if labels:
        label = "_".join(list(labels.values()))
    else:
        label = "na"

    metric_key = f"evidently:{metric.name}:{label}"
    prom_metric = prometheus_client.Gauge(metric_key, "", registry=registry)
    prom_metric.set(value)
    data_metrics[f"evidently_{i}"] = prom_metric

logging.info(
    "Evidently data metrics were translated to Prometheus for querying and \
    displaying them on Prometheus and Grafana."
)

return data_metrics

```

The next step is to see these metrics on the Web UI. The following code achieves this:

```
from flask import Flask, Response
```

```
app = Flask(__name__)
```



```
@app.route("/metrics")
def display_metrics():

    """
    Displays data metrics on the web page.
    """

    res = []
    for _, value in metrics.items():
        res.append(prometheus_client.generate_latest(value))
    return Response(res, mimetype="text/plain")


if __name__ == "__main__":

    metrics = monitor_evidently()
    app.run(host="0.0.0.0", port=9091)
```

The above code allows us to open the Web UI and see the metrics at <http://127.0.0.1:9091/metrics>.



```
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 4.3779e-05
go_gc_duration_seconds{quantile="0.25"} 8.9017e-05
go_gc_duration_seconds{quantile="0.5"} 0.000116579
go_gc_duration_seconds{quantile="0.75"} 0.000178293
go_gc_duration_seconds{quantile="1"} 0.007554839
go_gc_duration_seconds_sum 0.044295568
go_gc_duration_seconds_count 174
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 35
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="gol.19"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 2.0087208e+07
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 9.0770544e+08
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.532401e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 5.02607e+06
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 1.2155064e+07
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 2.0087208e+07
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 4.558848e+07
# HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use.
# TYPE go_memstats_heap_inuse_bytes gauge
go_memstats_heap_inuse_bytes 2.4567808e+07
# HELP go_memstats_heap_objects Number of allocated objects.
# TYPE go_memstats_heap_objects gauge
go_memstats_heap_objects 91711
# HELP go_memstats_heap_released_bytes Number of heap bytes released to OS.
# TYPE go_memstats_heap_released_bytes gauge
go_memstats_heap_released_bytes 2.8090368e+07
# HELP go_memstats_heap_sys_bytes Number of heap bytes obtained from system.
# TYPE go_memstats_heap_sys_bytes gauge
go_memstats_heap_sys_bytes 7.0156288e+07
# HELP go_memstats_last_gc_time_seconds Number of seconds since 1970 of last garbage collection.
# TYPE go_memstats_last_gc_time_seconds gauge
go_memstats_last_gc_time_seconds 1.675087008114043e+09
# HELP go_memstats_lookups_total Total number of pointer lookups.
# TYPE go_memstats_lookups_total counter
go_memstats_lookups_total 0
# HELP go_memstats_mallocs_total Total number of mallocs.
# TYPE go_memstats_mallocs_total counter
go_memstats_mallocs_total 5.117781e+06
```

Exhibit-7: Prometheus Metrics (Image by Author)

Exhibit 7 shows how we see the metrics at <http://127.0.0.1:9090/metrics> endpoint (self-monitoring). These are the metrics that *Prometheus* collects from the target by scraping metrics HTTP endpoints. Since *Prometheus* exposes data in the same manner about itself, it can also do scraping and monitor its health[7]. *Grafana* later displays these metrics in its user-defined dashboards.

Grafana

Grafana is a multi-platform open-source analytics and interactive visualization web application. It provides charts, graphs, and alerts for the web when connected to supported data sources. *Grafana* targeted time series databases as data sources, such as InfluxDB, OpenTSDB,

and *Prometheus*, and then it added relational databases such as MySQL, PostgreSQL, and Microsoft SQL Server[8].

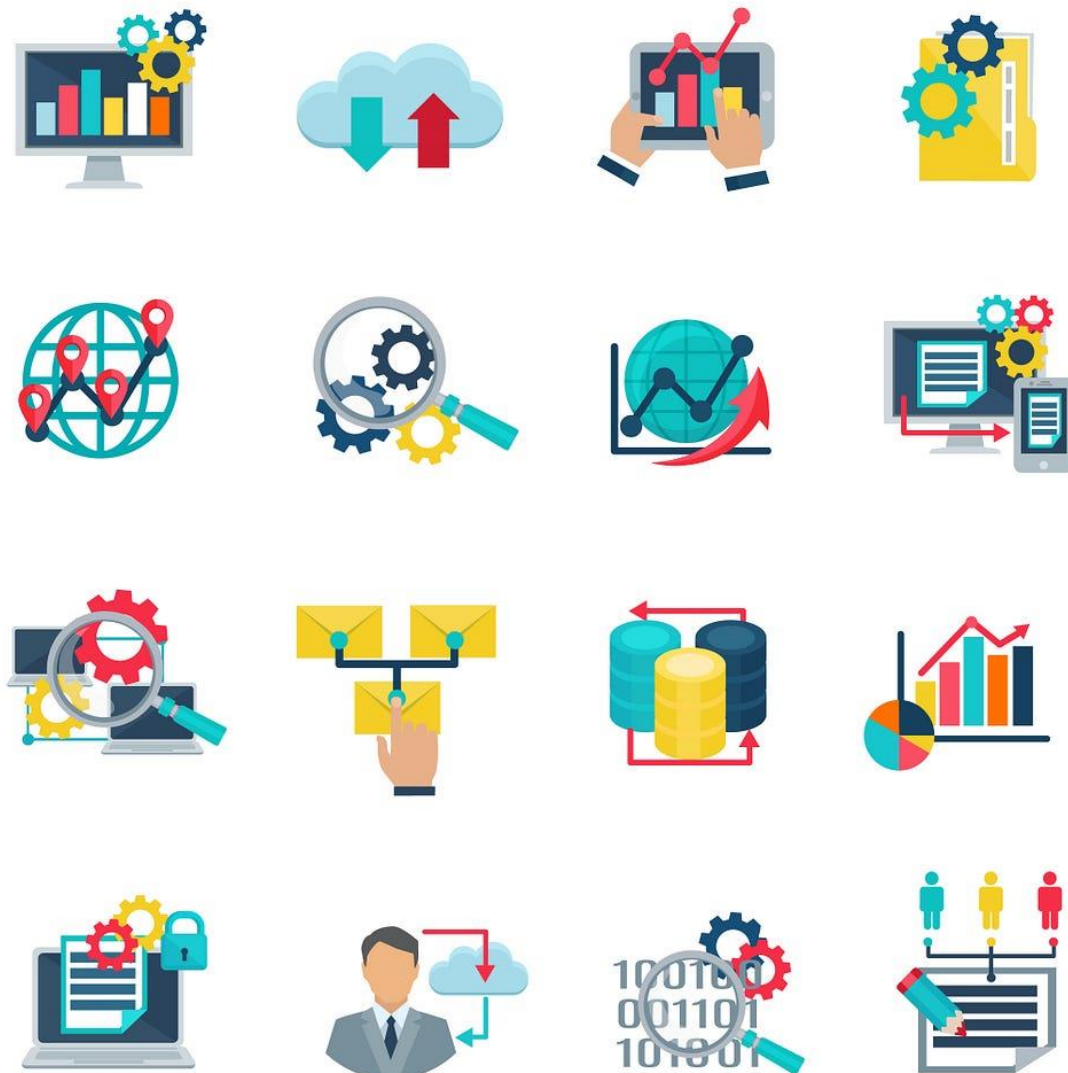


Image by [macrovector](#) on Freepik

When we use *Prometheus* as the data source, *Grafana* collects data from *Prometheus*' TSDB storage. Since we also store *Evidently* metrics in our RDS PostgreSQL database, we have one more option for connecting *Grafana* to RDS. We'll go with the second option. At this point, you might ask why we bypassed *Prometheus* after all setup we did above.

Prometheus is good for reliability and monitoring highly dynamic service-oriented architectures but not a good fit for situations where accuracy is crucial. Our application is a batch model. Furthermore, we are not using extra functionalities of *Prometheus*, for example, alerting. Using its alert manager could have made it more essential to our application. Therefore, neither *Prometheus* nor *Grafana* nor *Evidently* should find our app so exciting as they are rather cast for dynamic and streaming architectures. We had the chance to see and learn about these tools so we can use them for future projects, which are better fit for the tools' purposes.

However, we have no specific reason why we have decided to bypass *Prometheus* regarding our application. Also, note that there is no principal difference between using *Prometheus* and PostgreSQL as a data source as far as *Grafana* is concerned, and we'll see this a little later.

Installation

Install *Grafana*:

```
brew install grafana
```

Start *Grafana*:

```
brew services start grafana
```

and check if it's working:

```
lsof -i tcp:3000
```

While starting *Grafana* now or later, if you happen to receive the following message at your surprise:

Service `grafana` already started, use `brew services restart grafana` to restart.

Don't move and just do what it says!:

```
brew services restart grafana
```

I know, that was straightforward!

Grafana works at port 3000. Now, you can open the browser at <http://localhost:3000>. You may install *Grafana* on EC2 and do port forwarding to access it from your home or work machine. Alternatively, you may install it on your local machine to directly access *Grafana* by opening the browser. In the first case, you don't need to do much about the security group permission. Because the RDS security group should allow traffic from the security group attached to the EC2 instance following the convention and our deployment configuration. In the second case, we should add an inbound rule to the RDS security group allowing access from our local machine. And the source should be our IP address.

Step 19: Visualizing Metrics

Once we open the browser, we face the login page in Exhibit 8:

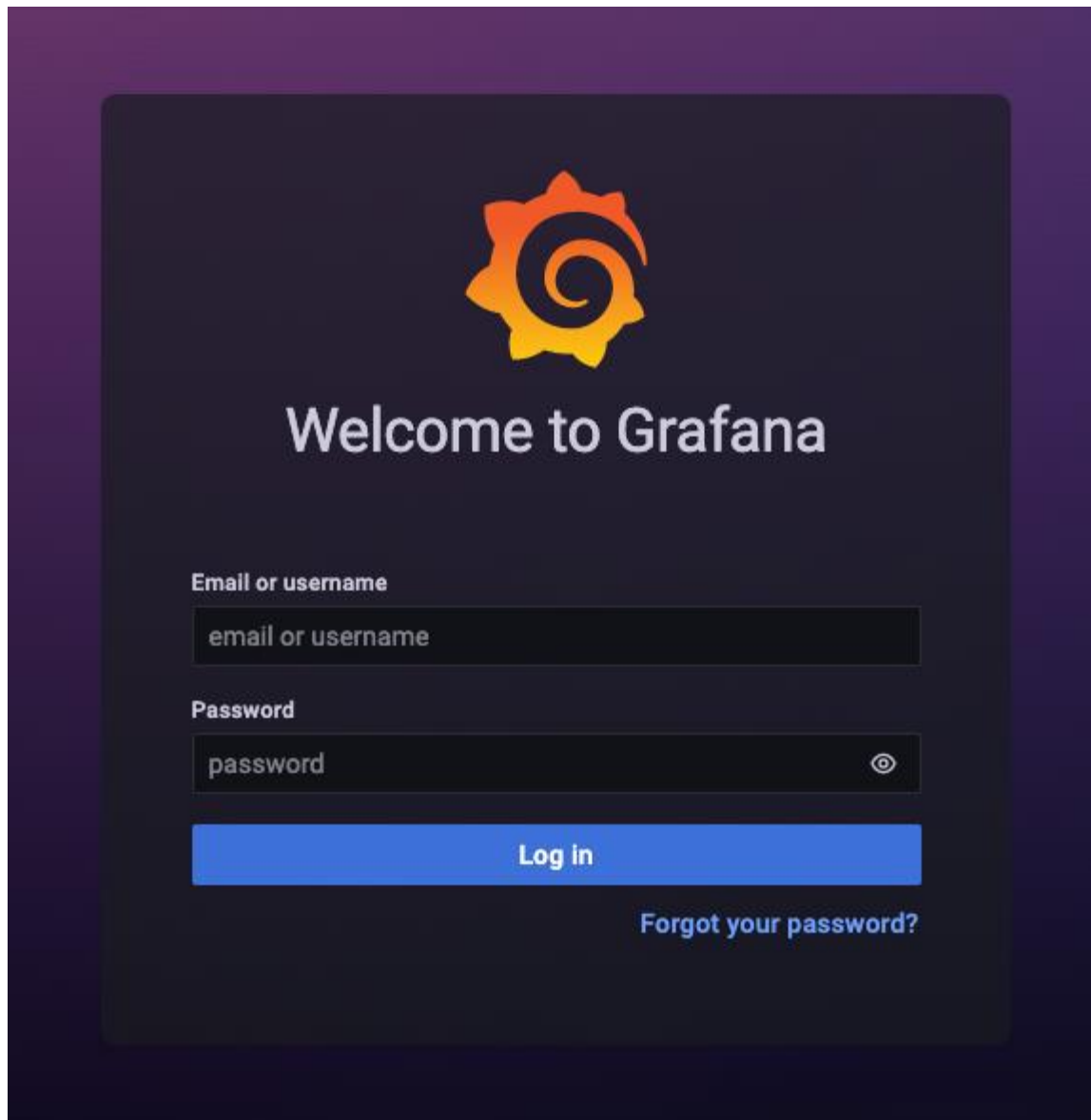


Exhibit-8: Grafana Login (Image by Author)

Here, we enter “admin” for username and password, then click Log in. *Grafana* will prompt us to define and redefine a new password. After doing so, we’ll get in. On the left panel, under “Configuration,” we choose “Data sources.”

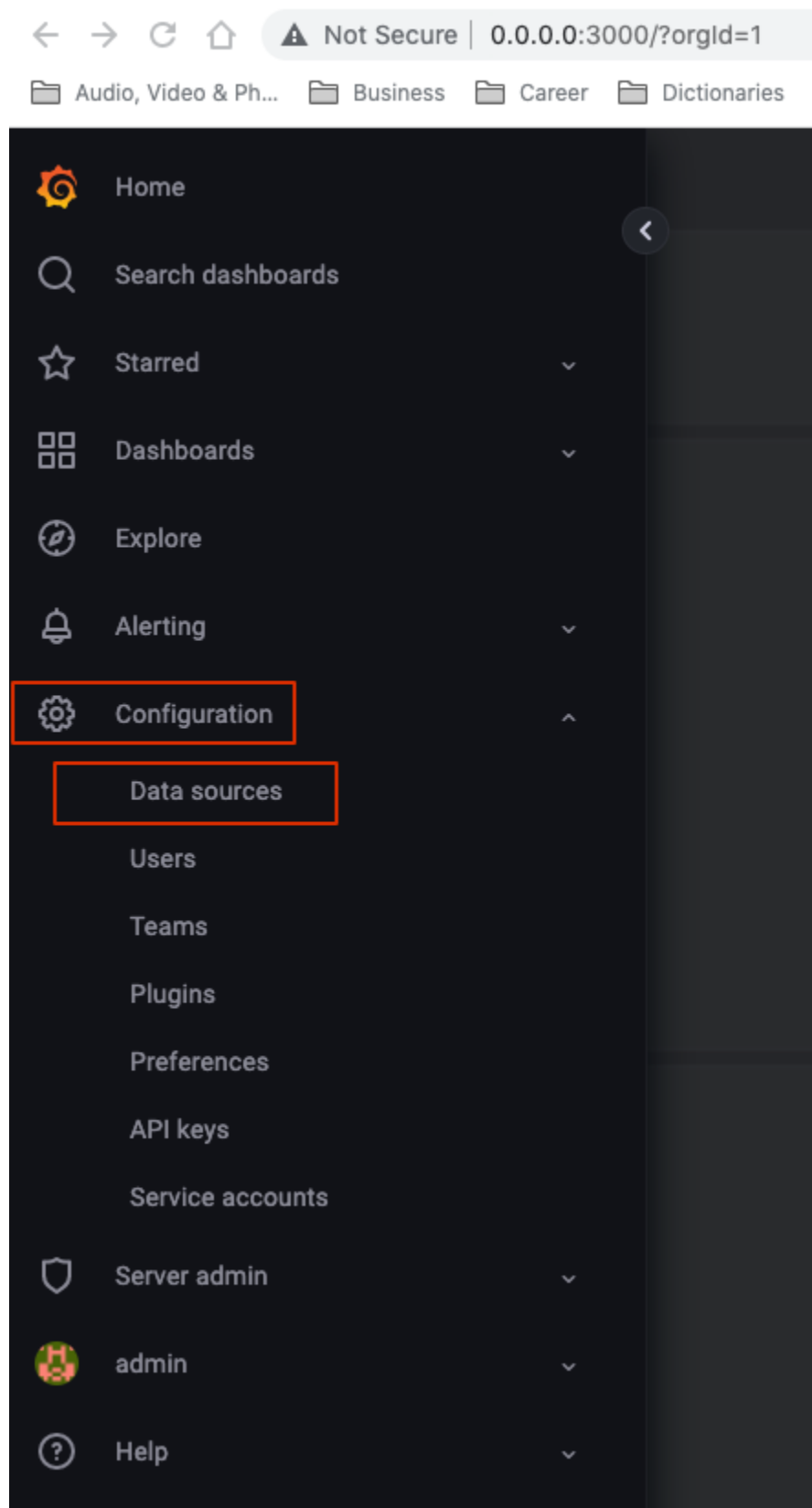


Exhibit-9: Grafana Data Sources (Image by Author)

We choose PostgreSQL as the data source. If we wanted to use *Prometheus* as the data source, we would select it from the list.

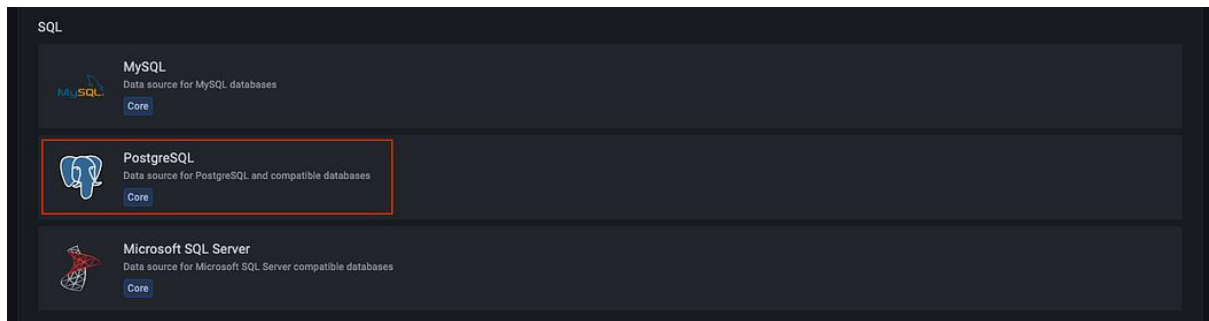


Exhibit-10: PostgreSQL as the Data Source (Image by Author)

Enter the endpoint information:

A screenshot of the 'Data Sources / PostgreSQL' settings page. The title is 'Data Sources / PostgreSQL' with a subtitle 'Type: PostgreSQL'. There is a 'Settings' tab with a list icon. Below the tab is a green box with a checkmark and the text 'Alerting supported'. Below that is a 'Name' field with a dropdown arrow, containing the text 'PostgreSQL', and a 'Default' toggle switch. Below this is the 'PostgreSQL Connection' section. It contains a table-like form with the following fields: 'Host' with the value 'wtp-rds-instance.cmpdlb9srhwd.eu-west-1.rds.ar', 'Database' with the value 'evidently', 'User' with the value 'postgres', 'Password' with a masked value '.....', and 'TLS/SSL Mode' with a dropdown menu showing 'disable'. The entire connection section is highlighted with a red rectangular box.

Exhibit-11: PostgreSQL Data Source Settings (Image by Author)

Any name (PostgreSQL) for the data source is welcome. The host is our RDS endpoint, something like wtp-rds-instance.cmpdlb9srhwd.eu-west-1.rds.amazonaws.com. We have already defined the user and password when creating the database. Keep the TLS/SSL Mode as disable. Scrolling down the page,

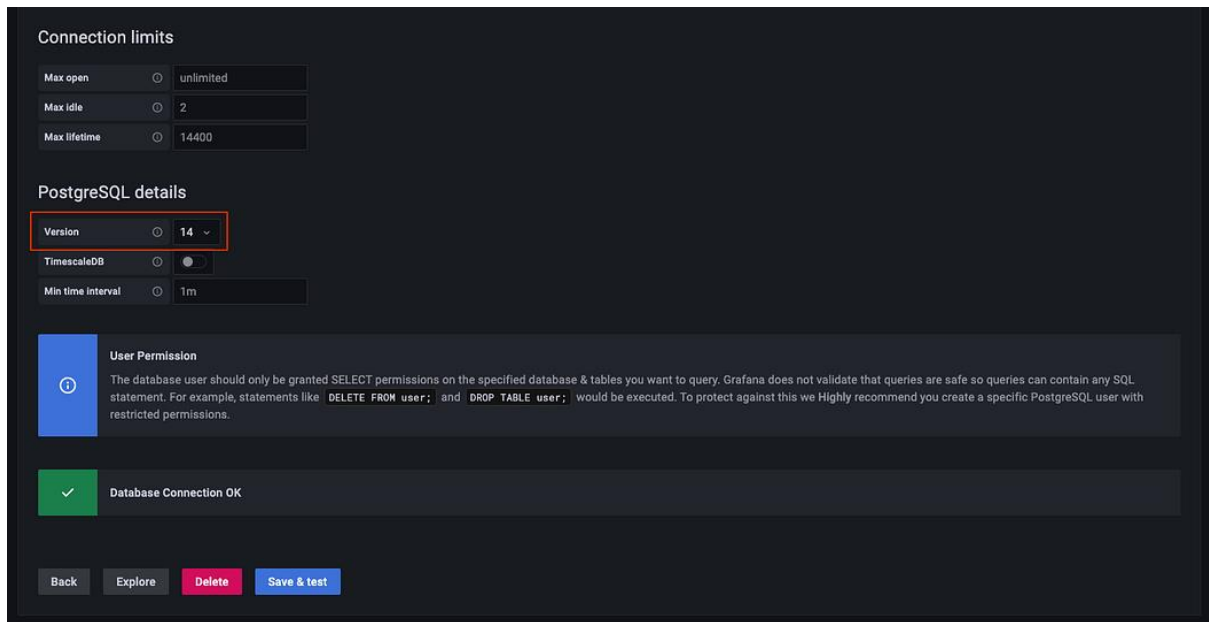


Exhibit-12: PostgreSQL Data Source Settings Continued (Image by Author)

we enter the version of PostgreSQL, which one can find on the AWS RDS database page. Ours is 14.3, so 14 here is okay. After clicking Save & Test, if we get the “Database Connection OK” message, we’re all set and get out of the settings by clicking Back.

Next, we create a dashboard

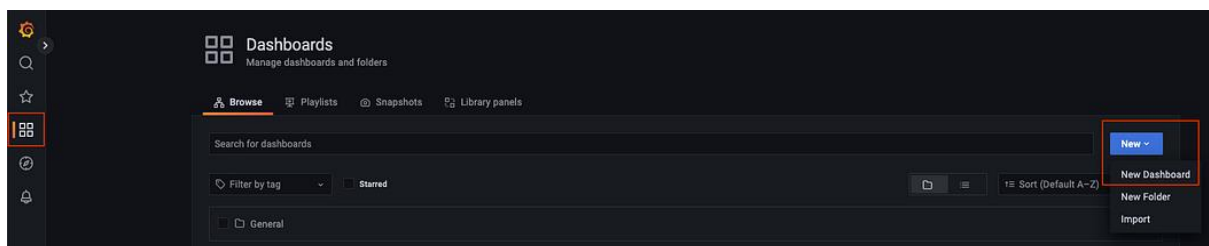


Exhibit-13: Creating a Grafana Dashboard (Image by Author)

by clicking on the left icon and New Dashboard. The new dashboard comes with a panel menu to enable the creation of panels suitable for our use case.

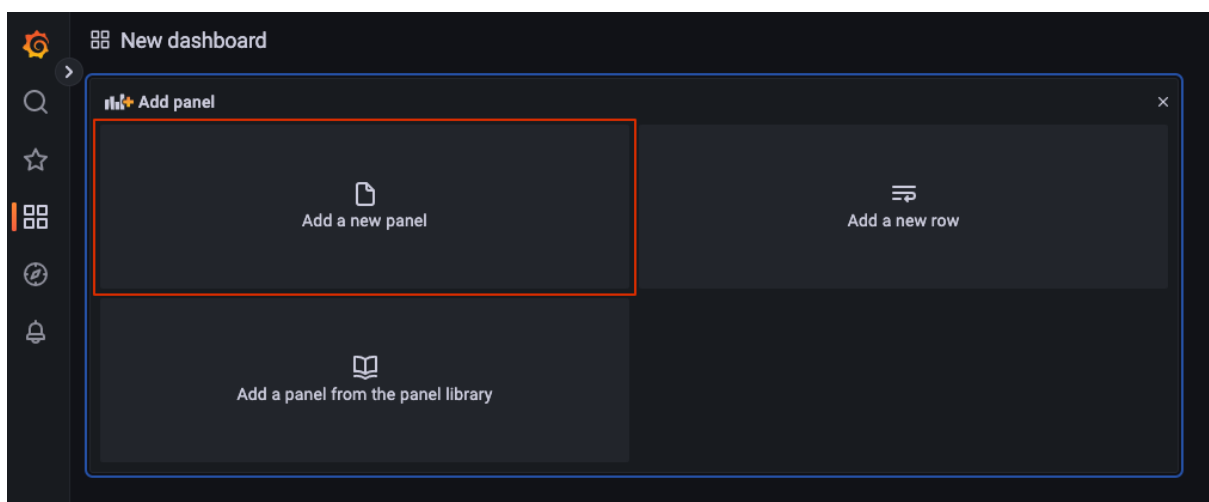


Exhibit-14: Adding a New Dashboard Panel (Image by Author)

Choosing “Add a new panel,” we enter the panel details page.

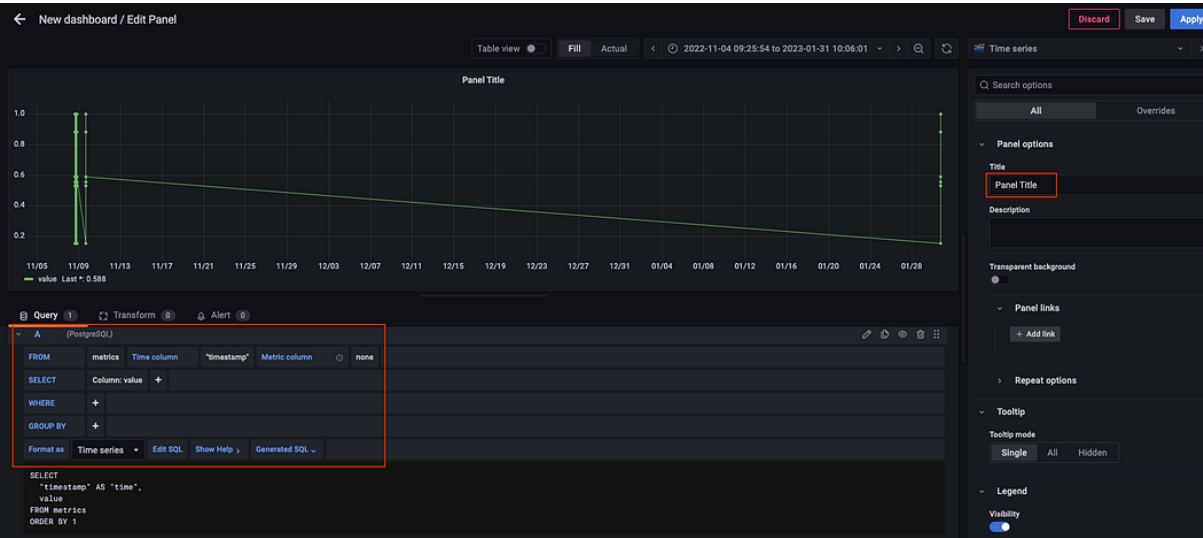


Exhibit-15: Query Statement for the Dashboard Panel (Image by Author)

We create our query as shown in the red box. Clicking Generated SQL translates it into the SQL statement in the below section. This SQL command displays the data on the upper section of the page and pulls the metrics from the RDS table shown below:

| | key [PK] character varying (250) | value [PK] double precision | timestamp [PK] bigint | run_uuid [PK] character varying (32) | step [PK] bigint | is_nan [PK] boolean |
|----|-------------------------------------|--------------------------------|--------------------------|---|---------------------|------------------------|
| 1 | day | 0.531 | 1667910938960 | 6a56e8c34dfe4cd9b73ab33736fdda70 | 0 | false |
| 2 | day | 0.531 | 1667911412483 | 2db30297db5841c2a1a4d712e5f50a7c | 0 | false |
| 3 | day | 0.531 | 1667911786321 | c1d527b2d891485bb0f242e47ed47880 | 0 | false |
| 4 | day | 0.531 | 1667912022135 | ecd558471a2f420d8809985665f667bf | 0 | false |
| 5 | day | 0.531 | 1667912274980 | 3acfe1445cc94d6cb1027828e76ea841 | 0 | false |
| 6 | day | 0.531 | 1667912511200 | 005f9619aeca4582bcac81e51477ccc5 | 0 | false |
| 7 | day | 0.531 | 1667912802614 | f9c1edbc7c9467d901446d0ce0364e7 | 0 | false |
| 8 | day | 0.531 | 1667919887198 | de61a8491dfc401a9e939e6119abac7b | 0 | false |
| 9 | day | 0.531 | 1667920230620 | aa9e94865664459ba47b8534ca6dc916 | 0 | false |
| 10 | day | 0.531 | 1667921786281 | b20e08f0c52d48c69fbf3ffca441abc5 | 0 | false |
| 11 | day | 0.531 | 1667924578902 | 7b0b8da43a3f422180317bfcc852662c | 0 | false |
| 12 | day | 0.531 | 1667996218223 | 8491eef5329b42cfad55d2e4bcbd844e | 0 | false |
| 13 | day | 0.531 | 1675087953643 | 845132a888954bec8b5496f48bd3d8e3 | 0 | false |
| 14 | sex | 0.558 | 1667910938936 | 6a56e8c34dfe4cd9b73ab33736fdda70 | 0 | false |
| 15 | sex | 0.558 | 1667911412448 | 2db30297db5841c2a1a4d712e5f50a7c | 0 | false |

Exhibit-16: RDS PostgreSQL Table Used in the Dashboard Panel (Image by Author)

Here, we use only two columns, timestamp and value, in the query to display. There are numerous options on the right pane of the *Grafana* panel window, where you can spend hours! We'll only set the panel title here, click Apply, and save the dashboard as below:

Save dashboard

New dashboard

Details

Dashboard name

evidently_metrics

Folder

General

Cancel Save

Exhibit-17: Saving the Grafana Dashboard (Image by Author)

The dashboard name is evidently_metrics and will be available on the page for interactive and real-time usage.



Exhibit-18: The Saved Dashboard (Image by Author)

A use case with PostgreSQL, which confirms *Grafana* as a great tool, is when one uses PostgreSQL as a database in the back end for an e-commerce application. A user can see the dynamic data such as sales, cost, and gross profit in real-time. *Grafana* also provides many ready-to-use dashboards. He can immediately put one of them into production.

We have seen how to record, store, monitor, and view metrics that show the data and model performance. *Evidently*, *Prometheus* and *Grafana* have helped achieve this task. The next challenge is to design all tasks as a process, organize their flow and schedule it.

